

Ultra-large-scale Sites <working title>

– Scalability, Availability and Performance
in Social Media Sites

(picture from social network visualization?)

Walter Kriha

With a foreword by << >>

Copyright

<<ISBN Number, Copyright, open access>>

©2010 Walter Kriha

This selection and arrangement of content is licensed under the Creative Commons Attribution License:

<http://creativecommons.org/licenses/by/3.0/>

online: [www.kriha.de/krihaorg/...](http://www.kriha.de/krihaorg/)

```
<a rel="license" href="http://creativecommons.org/licenses/by/3.0/de/"></a><br /><span xmlns:dc="http://purl.org/dc/elements/1.1/" href="http://purl.org/dc/dcmitype/Text" property="dc:title" rel="dc:type">Building Scalable Social Media Sites</span> by <a xmlns:cc="http://creativecommons.org/ns#" href="http://www.kriha.de/krihaorg/books/ultra.pdf" property="cc:attributionName" rel="cc:attributionURL">Walter Kriha</a> is licensed under a <a rel="license" href="http://creativecommons.org/licenses/by/3.0/de/">Creative Commons Attribution 3.0 Germany License</a>.<br />Permissions beyond the scope of this license may be available at <a xmlns:cc="http://creativecommons.org/ns#" href="http://www.kriha.org" rel="cc:morePermissions">www.kriha.org</a>.
```

Acknowledgements

<<master course, Todd Hoff/highscalability.com..>>

ToDo's

- The role of ultra fast networks (Infiniband) on distributed algorithms and behaviour with respect to failure models
- more on group behaviour from Clay Shirky etc. into the first part (also modelling of social groups and data)
- OpenSocial as a modelling example. Does it scale?
- finish chapter of popular sites and their architecture
- alternative architectures better explained (spaces, queues)
- cloud APIs (coming)
- consensus algs for the lowest parts explained
- failure models (empirical and theoretical, in connection with consensus algs)
- practical part: ideas for monitoring, experiments, extending a site into a community site as an example, darkstar/wonderland scalability
- feature management as a core technique (example: MMOGs)
- ..and so on...
- Time in virtual machines
- The effect of virtual machines on distributed algorithms, e.g. consensus
- Modelling performance with palladio
- Space based architecture alternative
- eventbasierte Frameworks (node.js / eventmachine) in I/O
- client side optimization hints
- queuing with data bases (<http://www.slideshare.net/postwait/postgresql-meet-your-queue>)
- spanner: googles next infrastructure, <http://www.royans.net/arch/spanner-googles-next-massive-storage-and-computation-infrastructure>
- CAP explanation:
<http://www.instepaper.com/text?u=http%3A%2F%2Fwww.julianbrowne.com%2Farticle%2Fviewer%2Fbrewers-cap-theorem>
- Puppet config management:
<http://bitfieldconsulting.com/puppet-vs-chef>
- Agile but extremely large systems configuration problems!

Foreword

<<by ?>>

Copyright	2
Acknowledgements	3
ToDo's	4
Foreword	5
Introduction	13
Part I: Media, People and Distributed Systems	17
Media	18
Meaning across Space and Time	18
Partitioning	18
Social Media	19
Being digital, distributed and social	19
Short Digression: The fragile concept of ownership in digital times	20
Superstructures	24
Social Media and their Price	24
People – communicating, participating, collaborating	25
Coordination	26
Where is the Money?	29
Findability	30
Epidemics	31
Group Behavior	31
Social Graphs	32
Superstructures	32
The API Web – the Sensor Web – the Open Web?	33
Supersize Me – on network effects and endless growth	33
Security	34
Federated Access Control to Private Data	36
De-Anonymization of Private Data	37
Identity Spoofing in Social Networks	38
Scams	39
Bootstrapping a large community	40
Part II: Distributed Systems	41
Basics of Distributed Computing Systems	42
Remoteness, Concurrency and Interactions	42
Functions of distributed systems	43
Manifestation: Middleware and Programming Models	45
Theoretical Underpinnings	47
Topologies and Communication Styles	49
Classic Client/Server Computing	49
The Web Success Model	49
REST Architecture of the Web	50
Web2.0 and beyond	53
Web-Services and SOA	56

Peer networks	59
Distributed Hashtable Approaches	60
Bittorrent Example	63
Special Hierarchies	64
Compute Grids	65
Event-Driven Application Architectures and Systems	66
Reliability, Availability, Scalability, Performance (RASP)	71
Resilience and Dependability	71
Scalability	72
Availability	75
Concepts and Replication Topologies	79
Failure Modes and Detection	85
J2EE Clustering for Scalability and Availability	89
Reliability	97
Deployment	97
Reliability and Scalability Tradeoff in Replication Groups	98
Performance	98
Monitoring and Logging	99
Distribution in Media Applications	99
Storage Subsystems for HDTV media	99
Audio Server for Interactive Rooms	103
Distributed Rendering in 3DSMAX	105
Understanding the Rendering Network Components of 3dsMax	105
Using partitioning to speed things up	107
Part III: Ultra Large Scale Distributed Media Architectures	109
Analysis Framework	110
Examples of Large Scale Social Sites	113
Wikipedia	113
Myspace	113
Flickr	115
Facebook	118
PlentyOfFish	118
Twitter – “A short messaging layer for the internet (A.Payne)”	118
Digg	119
Google	119
YouTube	119
Amazon	120
LiveJournal Architecture	120
LavaBit E-mail Provider	120
Stack Overflow	120
Massively Multiplayer Online Games (MMOGs)	122
On Shards, Shattering and Parallel Worlds	124
Shard Architecture and visible partitioning	125
Shardless Architecture and Dynamic Reconfiguration	127
Feature and Social Management	129

Security in MMOGs	131
Methodologies in Building Large-Scale Sites	131
Limits in Hardware and Software – on prices, performance etc.	131
A History of Large Scale Site Technology	133
Growing Pains – How to start small and grow big	133
Feature Management	134
Patterns and Anti-Patterns of Scalability	134
Test and Deployment Methodology	135
Client-Side Optimizations	136
A Model for RASP in Large Scale Distribution	138
Canonical or Classic Site Architecture	138
Classic Document-Oriented Large Site Architecture (Wikipedia)	140
Message Queuing System (Twitter)	140
Social Data Distributor (Facebook)	140
Space-Based Programming	141
Queuing theory, OR	141
Basic Concepts	141
Applications of QT concepts in multi-tier Systems	151
Service Demand Reduction: Batching and Caching	151
Service Demand Reduction: Data-in-Index	153
Service Demand Measurements	153
The n-tier funnel architecture	154
Cost of slow machines in mid- or end-tier	154
Queue length and Residence Time	156
Output traffic shaping	156
The realism of Queuing Theory based Models for distributed systems	157
Request Processing: Asynchronous and/or fixed service time	157
Heterogeneous hardware and self-balancing algorithms	158
Dispatch in Multi-Queue Servers	158
Unfair Dispatch: Shortest Remaining Processing Time First	158
Request Design Alternatives	159
Heijunka	160
Tools for QT-Analysis	161
Applicability of QT in large-scale multi-tier architectures	162
Combinatorial Reliability and Availability Analysis	162
Stochastic Availability Analysis	168
Guerilla Capacity Planning	168
Concurrency and Coherence	169
Calculation of contention and coherence parameters	172
Client Distribution over Day/Week/Year	175
Simulation	175
Tools for statistical analysis, queuing models and simulation	176
Architectural Principles and Metrics	177
Architectural Principles	178
Metrics	178
Changes in Perspective	178
Part IV: System Components	179
System Components for Distributed Media	179
Component Interaction and Hierarchy	179

Latency, Responsiveness and Distribution Architecture	179
Adaptations to media	184
Content Delivery Networks (CDN)	186
HA-Service Distributor	188
Distributed Load Balancers	189
Distributed Caching – not an Optimization	191
Caching and Application Architecture	191
Caching Strategies	192
When not to cache	192
Invalidation Events vs. Timeout	193
Operational Criticality	193
Pre-Loading of Caches	193
Local or distributed caches	193
Partitioning Schemes	194
Memory or Disk	194
Distribution of values	194
Granularity	194
Statistics	194
Size and Replacement Algorithms	195
Cache Hierarchies	195
Memcached	195
Fragment Architecture and Processor	197
Compression	201
Local or predictive processing	202
Search Engine Architecture and Integration	202
Special Web Servers (light-weight)	203
A pull based Web Server Design?	203
Scheduler and parallel Processor	204
High-availability failure detector	204
and lock service	204
Buffering and compensation for networked audio	204
Data Center Architecture	205
Geographically Dispersed Data Centers and Topology	205
Scale-out vs. Scale-up	206
Data Stores	208
Requirements and Criteria	209
virtualized storage:	209
External Storage Sub-Systems	210
Grid-Storage/Distributed File Systems	210
Distributed Clustered Storage	214
ZFS	215
Database Partitioning and Sharding	215
Cache concepts with shards and partitions	222
Why Sharding is Bad	223
Social data examples and modeling:	224
Partitioning concepts and consequences	224
Data Grids and their rules of usage	224
Database based Message Queues	226
Read Replication	226
Non-SQL Stores	226

Key/Value Stores	229
Semi-structured Databases	229
Scalaris	231
A new database architecture	232

Part V: Algorithms for Scalability 233

I/O Models 233

I/O Concepts and Terminology	235
Connections	235
The Asynchronous Web	236
The Keep-Alive Problem	237
I/O Processing Models Overview	238
Thread per Connection Model	238
Non-Blocking I/O Model	240
Synchronous Notification (Multiplexing) Model	240
Digression: API is UI or "Why API matters"	244
Asynchronous I/O Model	246
Java Asynchronous NIO	249
Virtual Machine Level Asynchronous I/O	249
Staged Event-Driven Architecture (SEDA)	251
Building Maintainable and Efficient Servers	253
Zero-Copy	254
Context-Switching Costs	254
Memory Allocation/De-Allocation	257
Locking Strategies	257
I/O Strategies and Programming Models	258
Libevent – an example event-notification library	260
Node.js – a new async. lib	260

Concurrency 260

Classic shared state	262
Consistency Failures	263
Performance Failures	263
coarse grain locking of top-level functions or data structures	263
pre-emption with locks held	264
thundering herd problems	264
False Sharing	265
Liveness Failures	265
Software Composition/Engineering Failures	265
Visible lock owners: mutex vs. semaphore use	266
composable systems in spite of shared state concurrency with locks?	266
Performance impact of test-and swap with lock-free synchronization	266
Provable correctness?	266
Classic Techniques and Methods to deal with shared state concurrency	266
Fighting Context-Switch Overhead: Spin-locks	267
lock breaking in time: generation lock and swap, memory retiring	267
lock breaking in space: per CPU locking	267
lock breaking by calling frequency: hot path/cold path	268
threading problem detection with postmortem debug	268
Transactional Memory and Lock-free techniques	268
Generational Techniques	273
Task vs. Data Parallelism	275

Java Concurrency	278
Active Objects	278
The Erlang Way	281
Multicore and large-scale sites	286
Scale agnostic algorithms and data structures	286
Partitioned Iteration: Map/Reduce	287
Incremental algorithms	289
Fragment algorithms	289
Long-tail optimization	289
consistent hashing	289
beyond transactions, large scale media processing	293
mostly consistent/correct approaches:	293
Failure Detection	293
algorithms dealing with heterogeneous hardware environments	293
Shortlived Information	294
Sharding Logic	294
Scheduling and Messaging	294
Task and processing Granularity with same block size, task time etc.	294
Collaborative Filtering and Classification	294
Clustering Algorithms	294
Number Crunching	294
Consensus: Group Communication for Availability and Consistency	295
Paxos: Quorum based totally ordered agreement	295
Paxos Implementation Aspects	297
Agreement based on virtual synchrony	300
Optimistic Replication	300
Failure Models	303
Time in virtually hosted distributed systems	303
Part VI: New Architectures	304
Cassandra and Co.	305
Adaptive, Self-Managed ULS Platforms	307
“Human-in-the-loop”	307
Self-management with interacting, hierarchical feedback loops	308
Emergent Systems Engineering	311
Scalability by Assumption Management	313
Cloud Computing: The Web as a platform and API	318
Canonical Cloud Architecture	321
Cloud-based Storage	322
Cloud-based Memory (In-Memory-Data-Grid)	322
Time in Virtualized Environments	323
The Media Grid	323
Peer-to-Peer Distribution of Content (bbc)	324
Virtual Worlds (Secondlife, DarkstartWonderland) – Architecture for Scalability	325
Immersive multi-media based collaboration (croquet)	326

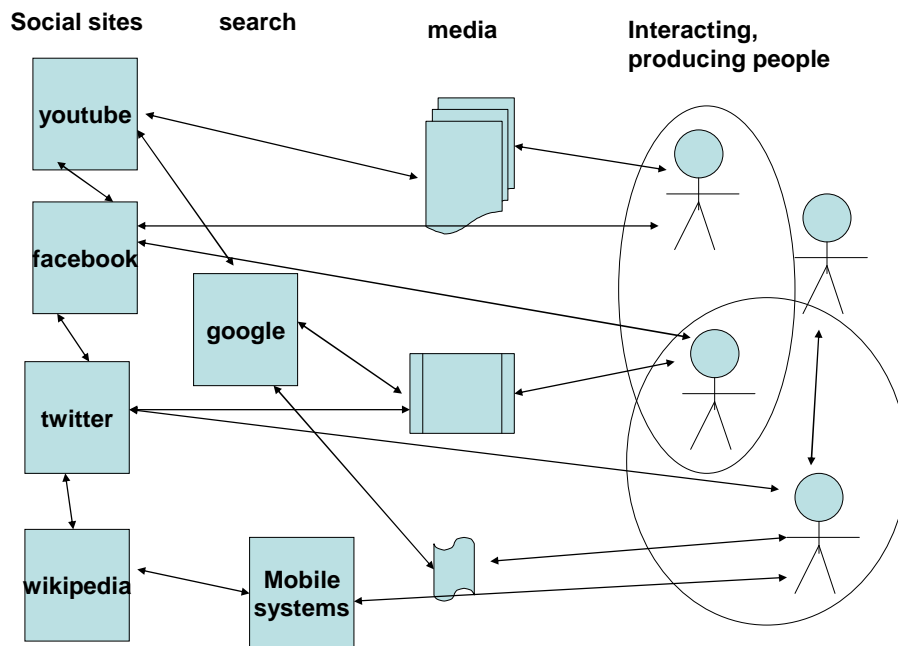
Part VII: Practice	328
A scalable bootstrap kernel	329
Exercises and Ideas	329
Data Storage	329
Modeling and Simulation	329
Performance Measurements and Profiling	329
Distributed Algorithms	329
Measurements	330
Going Social	330
Failure Statistics	330
Part VIII: Resources	331
Literature:	332

Introduction

This book has three major parts. The first part deals with the interdependent changes in media, people and distributed systems of the last 8-10 years. The second part explores large scale sites, their architectures and technologies down to the algorithm level. And it explains the specific adaptations for social media in those sites in all parts of the architecture. Modeling and visualization of distributed architectures is included as well. And the third part presents current developments e.g. in scalable MMOG design etc.

The drivers behind the first part are: The changes in distributed systems technology of the last 8-10 years which took those systems outside of companies and based them after fixed wire internet also on mobile and wireless networks. The change in media themselves which became digital and social and which are no longer the carrier of information only. And finally the change in people who walked away from passive consumption and turned to active communities and social networks.

The following diagram of participating people with numerous overlays and interacting media and communication systems displays the high degree of entanglement present today.



The three drivers are very much interdependent on each other – with the actively participating digital citizens perhaps being the new kid on the block.

Media and the technology they are based on have always been depending on each other. Changes in technology have brought new classes of media or new ways to use existing ones. Distribution too has been core to media ever since. Most visible when we are talking about broadcasting media but also in a much deeper way when we realize that media were always about bridging gaps between recipients

distributed across space and time. We will spend a little time thinking about media and distribution in this very basic sense because we will later see that some of the problems media face due to this separation will show up again on different, purely technical levels of distributed computer systems as well.

In other words we will discuss media and distributed systems on several levels: Media in the distributed system of producers, intermediates and consumers and media in the distributed computing infrastructures that have been a result of the internet. And of course we will investigate the connections and dependencies between those distributed systems because this is where lately new developments have appeared that seem to threaten existing businesses, political practices and individual rights. To name a few of these developments: File sharing and the question of digital rights, content creation and distribution as a job for specialists, ad hoc organization of groups, the question of privacy and anonymity in digital systems and last but not least the changing role of journalism due to blogging. We will discuss how technologies like Web2.0, community sites and social networks changed the way media are created, distributed and received. And we will see that the old slogan from the first years of the web: “content is king” is no longer true. It has been replaced by the social function of media: fostering collaboration and communication, group building and targeting. Rightly we call media now “social media” in this context.

The web has been an enabling technology for regular people who can now create, manipulate, distribute and receive media in previously unknown ways. The sheer quantity of the media constructed raises new problems: How do we store, find and distribute those media efficiently? It looks like we will rely on collaborative as well as computing technologies for solutions to those problems. We will take a closer look at technologies which can further enhance this ability to participate: Semantic tagging, microformats etc. But who are these people who live a life around community sites, blogging, RSS feeds, twitter messages, continuous tracking of friends, presence indications and much more? The question really matters because it has a deep impact on the technical systems supporting those lifestyles. These people can form ad-hoc organizations, their demands on infrastructure has epidemic qualities which regularly overwhelm technology and which has been answered e.g. by computing clouds with instant scalability. They might even go beyond the mere demand for fast and reliable services and ask for transparency of process and the company behind – again creating demand for new features in large scale community sites. The kind of data kept in these sites changed as well as the communication style going from n:1 (many users against one company) to m:n:k (many users to each other to some companies).

Then it is time to investigate the technological base of the distributed systems which created the new opportunities and which are driven by them. My approach is to give a short history of distributed systems and their core features (and mistakes) to give the reader a better understanding of the technical problems and challenges behind all the new web features and perhaps even to allow certain trends and developments to become visible.

Starting with the basic principles of distributed systems we will show the various answers that have been given in the form of “middleware” in the past. Classical distribution topologies like client-server, peer-to-peer and others and the

associated programming models explained. Architectural styles like REST or RPC are compared with respect to coupling and scalability.

Then comes a section on RASP: Reliability, Availability, Scalability and Performance. The move from company internal distributed systems to distribution on the internet caused the biggest problems and changes exactly in RASP: The ability to scale in an environment that is much less reliable than the company internal Intranets became a key success factor for large community sites and changed the way architects of distributed systems thought about certain algorithms and technologies.

The driver behind the second part is a rather practical one: Todd Hoff of www.highscalability.com created a portal for all things scalable and by browsing the sheer endless information on this site I realized a couple of things: First, there are many descriptions of large scale architectures like twitter, facebook, myspace etc. which are extremely interesting. They could be used to sum up the core features and methodologies behind scalable and reliable systems (e.g. do they all use a memory-caching layer?). Second, the reports are mostly written by the core architects and sometimes they are a bit dense. What might be a sufficient explanation for somebody working exactly in this area is probably just a bit too short an explanation to be understood by everybody (e.g. a comment on a certain cache system not being based on multicast and therefore scalable beyond 20 machines). In other words: an explanation of components, specializations and how they work together is needed. This includes modeling and visualization. Third, these architecture studies include specific technology (e.g. replication) which should be explained down to the algorithmic layer. Fourth, these large scale architectures created special solutions for their problems, sometimes by inventing new algorithms or by relaxing certain constraints. Optimistic replication, epidemic distribution and eventual consistency, functional partitioning and parallelization are just a couple of these new technologies.

The second part therefore presents some large-scale architectures and sites and investigates the distributed technologies and algorithms behind. Concurrency considerations, the handling of high speed I/O and database partitioning play a major role there as well.

Once the social and technological base of distributed systems is clear I will bring in the media. Media present very unique challenges to distributed systems which result from their size, realtime distribution needs etc. But they also relieve the technical base from some rather critical problems like transactional processing. We will therefore take a look at how distributed systems need to be adapted to support media properly. Concepts like partitioning of the information space are core to efficient treatment of media. Sometimes we will see that scalability and reliability of distributed systems forces us to adapt the higher “content” levels to fit into an efficient distribution strategy. Caching and replication are successful strategies to deal with media problems.

Finally in the third part I will investigate promising new applications of distribution principles to media. There are exciting new developments which try to go beyond current problems. Dynamically scalable, shardless Massively-

Multiplayer-Online Games (MMOGs), virtual worlds, P2P driven media distribution, self-managed distributed systems come to mind.

In the end the reader should have an understanding of current distributed systems technology motivated by the changes in media, people and technology over the last decade. The topic of large-scale social media sites seemed to be a good anchor for the explanation of distributed architectures and algorithms. Why is that so? The book has a little “hidden agenda” as well. It is the hypothesis that the size of the systems under investigation necessarily leads to a very different point of view towards system properties – especially the non-functional ones like stability, scalability, performance etc. And that the architecture as well as the development process experience major changes due to the changes in viewpoints. Let me give you some examples: Typically developers show a strong “functional fixation” towards interfaces for clients or customers. After looking at the way large scale sites deal with those functions we will realize that the “business function” part becomes somehow less important. This is probably not correct. It does not become less important: the other functions are becoming more important in comparison. It is not uncommon in ultra-large sites that business functions are designed to run in roughly the same time. They are split into smaller functions if this cannot be achieved otherwise. Sometimes business functions are turned off to keep the overall system stable. Amazon e.g. requires the 99.9 percentile of its services to complete within the defined service time. Application level code is suddenly forced to deal with system aspects violating transparency principles in a strong way. To me seeing and understanding those changes in perspective made writing this book big fun.

Part I: Media, People and Distributed Systems

Media

Meaning across Space and Time

Frequently media are seen as content within a container. This container can overcome distances in space and time and make the media available at the receiving end. When you add the capability of making copies of the container it looks like media are simply made for distributed systems. This point of view seems so natural that we tend to forget about the contextual requirements behind this process: Creator (let's assume this is the same person as the sender) and receiver need to share a lot of context to enable the distribution of media: a common language, a common social context etc. Otherwise what is shipped in the container turns into nonsense at the receiving end.

This possible “brittleness” in distributed media is a feature that media share with distributed computing systems. When a computer communicates with another computer they need to share certain contextual requirements as well and most people developing distributed computing systems had to learn this the hard way when small changes to protocols or structures on one side caused havoc on the other. We will see more of those structural or behavioral analogies between distributed systems on different levels.

Partitioning

Most distributed computing systems need to partition the content and its delivery across a population of receivers. Otherwise performance and connection complexity bring the system down. The same is true for media.

A classic view of media concerns the process of media creation and distribution. We can call it the one-way, top-down specialist viewpoint. In other words: media are created by media specialists (e.g. artists), they are published and distributed by specialists (e.g. publishers and networks) and finally they are consumed at the receiving end. This describes the so called broadcast process and it is a one-way street.

Production and Distribution are usually considered as two very different phases in media lifecycle. There are producers of media or content – few. And there are the masses of consumers of content. Even recent research ideas of the EU on media informatics (ERCIM) show this bias towards a client/server model of consumption and production. This hierarchical conceptual model of media production is now threatened by universal digital channels and machines. The digital content bits are shipped over distributed channels and systems to end users where they are again distributed to all kinds of players for consumption.

You need to compare this e.g. with John Borthwick, the CEO of Fotolog and his claim that both: production and distribution need to be seen together. [Borthwick]

Before we talk about how distributed computing challenged and changed this classic process we need to introduce the opposite of this process – the conversation. Clay Shirky in his book “here comes everybody” took a close look at social network sites like myspace, delicious, youtube, flickr, friendster etc. These sites allow everybody to publish whatever they want. Does this mean that all the media used in the context of these sites are broadcast media? Shirky explains that while the way of distribution looks like broadcast (everybody can watch) the media are much more geared towards conversation. They need context information and if groups form around those media then we see conversation and not top-down broadcast.

Conversation is peer-to-peer. It works only in small groups due to the complexity of n to n connections and it is two-way instead of one-way. Social network sites take the content produced in conversational contexts and “broadcast” them – but this is only a mix-up of different creation and distribution methods. An interesting question here is whether we can take broadcast content and bring it into a conversational context. “Remixing” content and discussing it in ones peergroup might be one example. The sharing of music in closed darknets another.

We will discuss Shirky’s core statement on how social software changes the limits to conversational groups and allows the formation of large ad-hoc mobs further down.

Media have some other qualities that are important for distributed computing systems. Media are – when used in the broadcast sense – not transactional. This means simply that there are not many different clients that might change one existing instance of a certain media concurrently. In most cases media are not changed at all, at least not concurrently. Of course once we enter the conversational style of media exchange (we could also call it the collaborative style) this assumption is no longer true. Virtual worlds and massively multiplayer games need to maintain the world state in a transactional fashion or experience some rather unhappy users and players.

Another important feature of many media with respect to distributed computing is due to human biology: Media reception requires in many cases realtime quality of service (QOS). Small delays in the playback of an audio stream are audible and destroy the experience. This is no small problem for loosely coupled, independently operating computers to guarantee the necessary quality of service at the receiving end.

Media are rather large in most cases. Only compression technology made it feasible to use media in IT systems at all. Media put a strain on operating systems and transport channels due to their size and the time based nature of media reception by human beings: A movie needs to deliver 24 or more frames per second or we will recognize gaps. Audio is worse still: even small interruptions become very audible.

When distributed systems need to bridge space to make media accessible they have to do so either by copying the media to the target system for consumption or they have to ship bits and pieces of the media towards the target system. In this case the distributed system needs to respect the realtime properties of media consumption if the media are continuous like movies.

We call the two cases the download case and the streaming media case. Both cases belong typically to the area of media consumption or distribution but this need not be.

Social Media

Being digital, distributed and social

The previously mentioned “classic” view on media as top-down delivery of content made by specialists and distributed by specialists received a couple of serious blows in the past, mostly due to technical changes in software and hardware. It started with content becoming digital and thereby reproducible at high quality, low cost and large quantities. Ed Felten of Princeton University describes the problems with digital media nicely in his famous talk "Rip, Mix, Burn, Sue: Technology, Politics, and the Fight to Control Digital Media".

What does it mean for content “being digital”? One can honestly say that many traditional publishers did not realize the disruptive nature of digital media and the responses to financial problems created by the digital nature were backward oriented in many cases. The answer to “cheap, high quality copies” was copy protection” and it became a key term for the entertainment and in parts also for the software industry. The industry tried – unsuccessfully – to change the digital nature back into an analog, non-reproducible nature. Software to prevent copying was installed (sometimes secretly), legal obstacles like making copy software illegal were tried and even advanced Digital Rights Management (DRM) systems were used. In many cases the regular and legal users had to pay a high price in usability for this protection of the content publishers.

Short Digression: The fragile concept of ownership in digital times

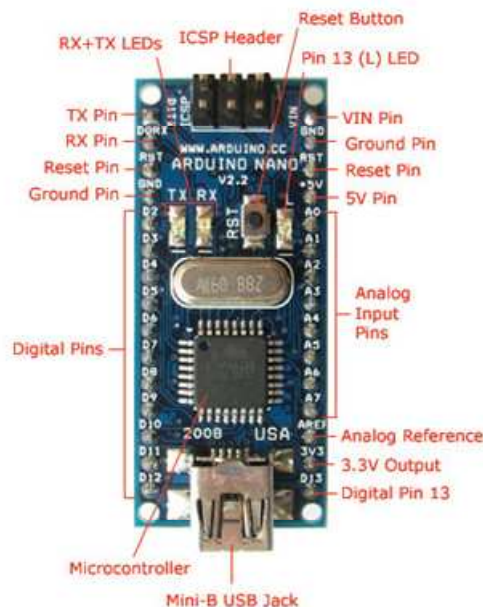
When copies are super abundant, they become worthless.

When copies are super abundant, stuff which can't be copied becomes scarce and valuable. Kevin Kelly, The Technium,
http://www.kk.org/thetechnium/archives/2008/01/better_than_fre.php

There is an undeniable tension between the digital world and traditional concepts of ownership. It becomes very visible in the struggle about intellectual property rights. The traditional economic theory puts ownership at the core of business and the reason is that resources are

considered scarce, not shareable because not-copyable. In the traditional economic theory things have value because of the scarceness and that explains why the music industry wants to turn the wheel back and make digital music again analog and thereby not copyable (at least not with the ease and quality and speed as is possible with the digital form).

But the matter of intellectual property rights is even more backward oriented. The proponents of software patents want to CREATE the scarceness in the first place. Turn something that is NOT scarce into something that becomes a value due to its artificial scarceness. Lawrence Lessig, Author of Code2.0 and other books on digital copyright claims that currently the law on intellectual property rights stifles creative use of materials. He created the Creative Commons Set of licenses (<http://creativecommons.org/>) as an alternative. On a worldwide scale the dominance of the western world with their immense pool of patents is a major handicap for developing nations. The situation becomes completely perverse when African nations are not allowed to reproduce AIDS drugs even though the population there can never afford the prices of western pharmacies. Intellectual property rights around hardware are an especially interesting topic. Hardware manufacturers do not Open Source the diagrams and construction materials used to build there systems. They fear that this would make copies trivial and they would be face cheaper copies made in china. But is this true? First: almost all hardware can be re-engineered by somebody. This happens on a daily base in this world. And second: Clive Tompson describes the Arduino microcontroller that was turned into Open Source by a small Italian company. [Tomp]



Arduino open source hardware

Everybody can use the wiring diagrams etc. to build an exact copy of the controller and it is done as a matter of fact. But strange things are happening: the Italian company is selling lots of controllers, still. They do

not generate a lot of money from those controllers – and they do not plan to do so. Their business model is about services around the controller and it seems to work. So they are really interested in others copying their design.

But there is something else that is vital to open source hardware: it creates a community around such products. And the value finally comes from the community. The community even improves the hardware design, the community discusses new features and fixes bugs. The community helps newcomers and manufacturers. And the original inventors are right in the middle of this community if they play it right. This is something that both fascinates and scares companies: they are slowly getting used to the fact that there is an outspoken community around their products with community sites, forums etc. But now they have to realize that some parts of this community will get involved in the future of products, product planning and finally in the way the company works. This kind of transparency is scary and also powerful. Stefan Bungart, philosopher and executive lead at IBM described these challenges in his excellent talk at the 10 year anniversary of the computer science and media faculty at HDM and the stream of his talk is definitely worth the time watching it [Bung]. But what happens if we really give up on the idea of intellectual property rights which can be used to exclude others from building the same, just better? One can assume that the same effect as with open source software will be seen: A ruthless, brutal Darwinism of ideas and concepts would result from this with a resource allocation and distribution that would be more optimal than the one that is usually claimed by capitalism to be the best.

The digital world has seen a decrease in value of most of its goods: CPU time, RAM size, disk space and communication costs have all come down to a level where one could claim that sharing those resources is basically free (and therefore a requirement that sharing happens at all as Andy Oram points out in his book on Peer-To-Peer networks). And the open source movement is a real slap in the face of traditional economy: Non-zero sum games instead of zero-sum games. Sharing instead of excluding. And the proponents of this movement have proof of the higher quality and faster reaction times this system can offer. Open Source Software, social networks like the one that supported Obama all show what communities can achieve without the sole interest of making profit – something that just does not happen in classic economic theory. And when these communities are given a chance through open, distributed computing systems and social software running there.

So the right answer for the content producing industry (or is it actually more of a content distributing industry anyway?) would be: forget about the media container and start concentrating on the real value for customers by embedding the container into the whole music experience. To have this experience music needs to be found, transferred and made accessible in high quality anytime and anywhere. And this becomes a service to the customer that the industry can charge for.

And this service has another advantage: it is not so easily reproducible. It depends on knowing what people like or dislike, on knowing about their communities, on offering fast and high quality access, on providing excellent usability for finding music etc.

Which leads over to the question of reproduction in general. Lets take a look at the list of non-copyable things from Kevin Kelly:

- Immediacy
- Personalization
- Interpretation
- Authenticity
- Accessibility
- Embodiment
- Patronage
- Findability

“Immediacy” is the difference between expecting the customer to visit a shop physically and having a browsing and streaming service available that allows immediate access to the music the customer likes.

Personalization allows simply a better service. Interpretation means helping somebody with something digital. Authenticity is a guarantee that the digital copy somebody is using really is correct and unchanged.

Accessibility can mean improved physical access (transport) or better user interfaces. Apple products shine in this respect over most others.

Embodiment is a band in a live-concert: the music is tied to the body of the band. Patronage is the willingness of customers to support somebody via donations. And findability makes it all possible by letting the customer find the things he wants.

Don't get me wrong: these things are also copyable in a way, e.g. by competing publishers. But in the first place they add to the digital copy in a way that can't be copied by the customer!

“being digital” does not end with physical things. Daniel H.Pink in his bestseller “A whole new Mind” raises questions about the future of working people and how it will look. He asks the readers three questions about the type of work they are performing:

- Can someone overseas do it cheaper?
- Can a computer do it faster (and being a computer science guy I would like to add: better)?
- Am I offering something that satisfies the monumental transcendent desires of an abundant age?

The last part points to the weak spot in the book: the author assumes a world of abundance. Goods are plenty (e.g. there are more cars in the US than people). On the other hand the book has one theme only: how to use the right part of your brain (which hosts creativity, gestalt perception etc.) for one purpose only: to make yourself still usable (in other words: paid) in this new society dominated by right brainers. Because people have plenty the scarce things (and therefore valuable things) are art, play, design etc. Lets just forget about the ideological nonsense behind the book (why would I have to sell myself in a world of “abundance”?) and concentrate

on the things that make copies impossible. And here the author may be on to something right: combining know-how from different areas to create something new is hard to copy. Combining different methods (like narrative methods from the arts and programmatic methods from IT also creates something new). And if the author finds enough followers we could enter a phase where millions of amateur (writers, poets, painters, designers etc.) create things that are unique. Again, looking at this from with the cold eyes of sociology would show us typical overhead phenomenons described by Pierre Bourdieu (the fine differences). Finally, virtual worlds add another angle to the copy and scarceness problem. In the position paper on “Virtual Worlds, Real Money” the European agency enisa takes a look at fraud in virtual worlds [enisa]. According to enisa many of those worlds implement a concept of “artificial scarceness” by restricting objects or services as it is done in the real world e.g. with currency. Users of the virtual worlds can then sell either virtual goods or services inside or outside of the virtual world. But we should not forget that we are at a very early phase in virtual world development where it is natural to copy existing procedures from the real world to allow users an intuitive access to the world – much like the office desktop concepts of some operating systems tried to mimic a real desktop (with some very questionable success raising the question on how far metaphors will carry us..). We will come back to the paper from enisa in the chapter on virtual worlds.

While the media industry was still grappling with the copy problem the next blow arrived: The digital nature of the media was now brought together with the distributed infrastructure of the internet, including different types of access, replication and findability options. Bringing digital media into the distributed infrastructure changed two things dramatically: the mode of operation or use of digital media and the way they are found and distributed. Before this development happened the media industry controlled how media were found, distributed and consumed (you need a xyz-player, a media disk or tape etc.). Now the media could be consumed in many different ways and many different places (streamed through the house, on a mp3 portable player, from a mobile phone, directly on a train via phone etc.). And the whole process of copying an audio disk for friends disappeared: file sharing networks allowed the distribution of content to millions for free. And of course they allowed easy findability of new content as well.

We need to compare this with the “vision” of the media industry. At that time this vision was still deeply rooted in the “disc” nature (analog) of the media container that was at the same time the focal point of the financial interests. “You need to sell CDs to make money”. Lets look at two examples: publishers of encyclopedias or other kinds of information and music publishers. An encyclopedia in the form of beautiful books bound in leather is certainly a nice thing to have – but is it really the most useful way of using one? The publishers reacted and started selling information on CDs. But do you really need more CDs? If you are a lawyer or a tax accountant you subscribe to information publishers who send you the latest updates on CDs. Is this really useful? You can’t make annotations on

those data graveyards and worse: you don't see annotations and comments others might have made about the same text parts. The correct way to use information services of course was online and social and wikipedia turned into the prototype of all those services.

The music industry did not fare better. They still shipped CDs to music shops but who has the time to go there? And how would I find music I don't know yet but might like? And what should I do with a CD or a CD archive? When you have a family you have learnt that CD archives don't work: the boxes are always empty, the discs somewhere in the kids rooms or in the cars. The music industry could not solve two basic problems: to let me find music I like easily and to let me consume this music whenever and wherever I have an opportunity.

Finally online music shops appeared like Itunes. They were still handicapped by copy protection rules at the cost of low usability at the end user side. Recommendation sites appeared, based either on content analysis of music (advanced) or simple collaborative filtering based on my music history, my friends or anybody's listening or buying patterns (amazon).

And still this is a far cry from how it could be when we consider realtime streaming. I should be able to browse music anywhere I am and at any time and then listen in realtime to the pieces I like. The system should record my preferences and maintain my "archive" online.

Before we deal with the next blow to the media industry a short recap of where we are is in order. We started with media becoming digital which allowed easy and cheap reproduction at low costs. Then we added the distributed infrastructure of the internet with millions of PCs in the homes of users. This allowed fast and easy distribution of digital media and at the same time provided meta-data services (recommendations etc.) that the music industry could not easily offer due to their social nature. And perhaps because the industry still considered "content as king" when media content started to become something very different: an enabling element in a distributed conversation across groups, social networks and communication channels. The way media were produced and consumed started to change and this was the third blow to the media industry.

Superstructures

What does "superstruct" mean?

Su`per`struct´ v. t. 1.To build over or upon another structure; to erect upon a foundation.

Superstructuring is what humans do. We build new structures on old structures. We build media on top of language and communication networks. We build communities on top of family structures. We build corporations on top of platforms for manufacturing, marketing, and distribution. Superstructuring has allowed us to survive in the past and it will help us survive the super-threats.

http://www.superstructgame.org/s/superstruct_FAQ

Social Media and their Price

<http://www.slideshare.net/wah17/social-media-35304>

Who creates content? In the eyes of the media industry the answer is clear: content is created by paid specialists. Flickr, YouTube, delicious, myspace, wikipedia and blogs have proven them wrong. Content can be created a) by you and me and b) collaboratively. But is it the same content? Is it as good as professionally created content? The quality question behind information bases like wikipedia or open source software led to heated discussions initially. Those discussions have calmed down considerably as the established content or software producers had to learn that in many cases they would not stand a chance against the driving forces behind open content or software production. A company just cannot compete with the large numbers of users adding or correcting wikipedia pages or testing and correcting software packages. And the lack of formal authority leads to a rather brutal selection process based on quality arguments instead of hierarchy.

But it is not only the content creation that changed. The content on social network sites is different from professional broadcast content. It is usually created independently of the media industry, it is sometimes conversational, oriented at small, private groups. It is discussed in instant messaging groups or chat forums. It is received, consumed, distributed and discussed in an interactive way that is simply impossible for regular broadcast media. IBM claims that people born after 1984 belong to a fundamentally different user group with respect to the use of interactive, always connected media technology. These people use chat, instant messaging and virtual worlds just as the older population might use e-mail. This active, conversational style of media use might be the biggest blow to the media industry after all.

Community sites feature a lot of social information created by user behaviour. One of the simplest being “who’s present?” at a certain moment. Social information can be more specific like “who is watching the same page right now?” and so on. This type of information is mostly real-time and semi-persistent. And it creates performance problems in web sites if one tries to use traditional databases to insert and query such information. There are simply too many updates or queries needed per second on a busy site to use a transactional datastore like a database for this purpose.

Currently it looks like the print-media industry, especially the newspapers, might pay the price of social media by going bankrupt. Even the mighty New York Times might not survive (<http://www.spiegel.de/netzwelt/web/0,1518,607889,00.html>). Online editions of newspapers seem to be unable to collect money for their services. And they fight with numerous problems like the micropayment difficulty (users do not buy pieces of media products because they do not know how to price them: there is no market for single articles). And paper editions are facing the competition of free micro-newspapers like “20 minutes”.

Social media killing off the traditional media because they are much more group oriented, active etc. is just one economic impact of social media. There are more if we look at the next chapter: interacting people do not only create social media. In principle they can organize many services that traditionally have been provided by the states (like currency, security) or companies (like hotels, pensions). Go ahead and read the quote from Kortina at the beginning of the next chapter: Through social media and sites people share many more things like bed and breakfast while travelling and visiting friends made through facebook, couchsurfing etc. These services used to be commercially available and are now put back into the private, non-economic space. Obviously beds and breakfast are NOT really scarce on this world. Organizing was hard and a professional service. But this is changing with social network sites and again some goods and services are no longer ruled exclusively by economic scarcity. The digital life now starts to determine the prices in the analog world as well.

People – communicating, participating, collaborating

Fotolog CEO John Borthwick,

<http://www.borthwick.com/weblog/2008/01/09/fotolog-lessons-learnt/>

By digging into usage data we concluded that the Fotolog experience was social, social media. Understanding this helped us orientate our positioning for our members, our advertisers and ourselves. The rituals associated with digital images are slowly taking form - and operating from within the perspective of a mature analog market (aka the US) tends to distort one's view of what how digital imagery is going to be used online. The web as a distinct medium is developing indigenous means of interactions.

URL: <http://essays.kortina.net/>

Couchsurfing is Beta Testing a City
June 27th, 2008 ·

Couchsurfing is my new favorite social net. I checked it out this week prior to my trip to Palo Alto, and now CC and I have connected with 2 people in the real world and have gained two new friends. Our hosts showed us around the area, gave us a feel for what life was like in Palo Alto, and told us about the cool stuff they're doing. We got a tour of Stanford Campus, went hiking, went to a cool place for dinner, got a homemade pancake breakfast, got some free rides, and had great conversations.

Although I've been to the Bay Area before, I don't think I've ever gotten a feel for what it would be like to live there until this past visit. It's tough to assess a city when you're just a visitor staying in hotels. Spending time in homes and apartments of people that actually live there and joining them in their nightly excursions is probably the best way to actually experience the city like a local. Thanks to Sasha and Vanae for hosting—good times, for sure.

I must admit, that Twitter & Facebook also came through. I tweeted about heading out to Cali, which got imported into my Facebook feed. My college buddy Wes saw this and mentioned that he had recently moved to San Fran and had some couches we could crash on. We spent two nights with Wes, ate fantastic Mexican food, and discovered two of the coolest bars I've been to in some time. Wes also introduced me to a pretty cool new band, Ghostland Observatory. Here's a good track: [Vibrate](#). I love using the internet to connect with people in the physical world.
URL: <http://essays.kortina.net/>

Coordination

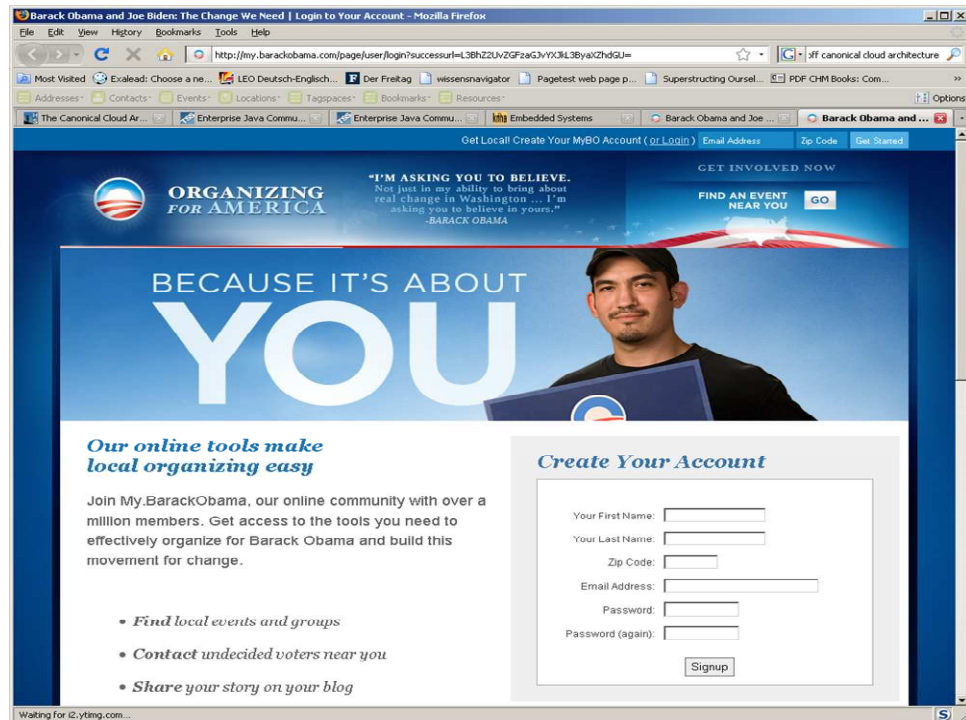
Getting members of a distributed system to collaborate and act in an organized way to achieve a common goal was and is a hard problem – no matter whether we are talking about people or computers. Interestingly, the bringing together of distributed human beings with the distributed organization of the internet seems to reduce exactly this problem considerably – at least for the human part.

Politically interested people might have noticed (and hoped) that the new social networks and sites allow easy and independent organization around specific topics. It was Clay Shirky who said in the subtitle of his book “The power of organizing without organizations”. He claims that social software and social networks reduce the organizational overhead needed to form active groups and therefore allow the creation of ad-hoc groups. The media created and distributed on those sites become actionable items, they support group behaviour.

<<Distribution now a network effect. Mixing hierarchical and democratic methods

(two technology rev. Articles) >>

Let us take a closer look at one of the most successful social networking strategies of ever: Barack Obamas fight for presidency. This fight was supported by several social networking sites, especially <http://my.barackobama.com>.



This site played a major role in the organization of events etc. In particular it allowed:

- small donations to be placed easily
- sharing of personal details like phone numbers to be later used for event organization
- learning about events
- learning about groups and interested others
- planning and organizing of local events and activities
- sharing of stories and blogs
- contact other potential voters
- use the blog
- buy fan articles
- personalized use of the site
- meet Obama and other prominent representatives (chat, webcast)
- get information on elections, social groups etc.
- watch videos from speeches or events
- send messages to election staff
- find connections to other social network sites with Obama content: flickr, youtube, digg, Twitter, eventful, LinkedIn, Facebook, MySpace etc.
- learn about all these features in a tour video

<http://www.youtube.com/v/uRY720HE0DE&hl=en&fs=1&rel=0>

A heise article mentions the following success factors of Obamas site: It brought Obama more than 500 Million Dollar in donations, 75.000 local events were organized using the sites data and participants which exceeded 1 million finally. A core problem for the site were the ever increasing numbers of users.

David Talbot describes the Web2.0 strategy used in a Technology Review 11/2008, Report. According to Talbot the team around the site understood

that users and visitors would automatically re-distribute content (speeches etc.) once they were available on social networking sites. So a lot of media content was placed on different sites like facebook or youtube as well. One of the biggest success factors resulted from the database with information on potential participants and supporters. This information was used to tie online-activities with real-world events outside.

“The Obama campaign has been praised—with good reason—for its incredible use of technology. Many organizations would love to replicate its ability to do outreach, its focus on data, and its ability both to coordinate the efforts of hundreds of thousands of volunteers in a single direction and to empower those individuals to take control of their own distinct parts of the campaign. The use of technology within the Obama campaign creates two seemingly contradictory points: the technology strategy was not a technology strategy—it was an overall strategy—yet it could not have been executed without technology. But this misses what programmers have always understood about software—a truth that has finally blossomed in the age of social networking: software itself is an organizing force that equips organizations to achieve their goals. The Obama campaign used technology as a front-end enabler rather than a back-end support, and this synchronization between mission and tools allowed for the amplification of both.” Benjamin Boer, The Obama Campaign – A programmers perspective [Boer]

Software and the distributed runtime systems as frontend enabler! Data centric and linking different platforms the Obama campaign showed the typical Web2.0 characteristics. But according to Boer there was one special additional ingredient that made it so successful: “grassroots experimentation”, the will to innovate and experiment with the live system, enabled by the use of open source software that provided both a means to changes and ubiquitous know-how by volunteers. It is this combination of software technology and social environment that is responsible for the success. We will take a look further down whether those characteristics also show up in the well known social sites like facebook, flickr etc.

Compared to Obama the competition (Clinton, McCain) led more traditional, hierarchical campaigns with less use of new media like social networks. In McCains case his social network site seemed to be unable to deal with a larger number of requests or users. Obama on the other site used different communication channels and media and therefore did not miss larger sections of the population. And perhaps the most significant difference was in the ways the candidates handled the “everybodys”: The Obama site allowed self-organization of supporters and created only a very flat hierarchy and control structure. This made the organization of local events extremely easy and efficient because it delegated power to those who needed it – the organizers themselves. This aspect of digital media in the context of distributed and social systems makes many companies extremely uncomfortable: What if the new forum is used to badmouth one of my products? What if consumers use my collaborative site to band up against the company? Social Networks are a far cry from the tightly controlled information handling policies of the classic marketing and PR departments or the classic broadcasters and that is why these classic

organizations frequently show little success in dealing with social networks. Those networks require due to their distributed, open nature a large degree of transparency and freedom and are always a bit “out-of-control”. (In a classic PR campaign you build some presence or presentation and when it is done you go public with it. In social networks and virtual worlds the phase of building the presence is the phase which attracts the most interest and you need to realize that “the way is the goal” here. Media in social networks need not be perfect – they need to be useful. Btw: the Internet-Philosopher Dr. Felix Weil mentioned in his talk on the occasion of the first Web2.0 day at HDM that transparency and presence are the two core requirements for all activities on the internet.

Blogging has become a standard procedure in journalism. It allows independent authors to voice their opinion and to connect it with others. This highly distributed, egalitarian way of creating content is in stark contrast to the highly concentrated and controlled media industry of a Berlusconi, Murdoch or Turner. People run personal diaries on web servers, link heavily to other sites and let others comment on their content. This creates a content networks between independent content producers. The content/blogs might be hosted on a large server or on individual computers.

Blogs have had a very important side-effect: Re-mixing. Re-mixing is taking existing content and modifying it, bringing it into a new form and than publish it again. The idea of re-mixing is rather radical for any media. It raises questions about authorship and ownership of content. But it has turned into a form of art for virtual communities.

Meta-services like <http://de.globalvoicesonline.org/> collect and present selected blogs to their audience and ensure that political voices will be heard. And even text based SMS messages can be used to form groups, conduct surveys etc. with the help of social software like <http://www.frontlinesms.com/>

So called Wiki’s – simple content management systems which allow everybody to create and edit pages which will be seen and/or edited by others have become a popular way to organize projects. Augmented with some project management and communication facilities they allow groups to plan and schedule events or they serve as the groups permanent memory. Wikis are simple applications running on web servers, sometimes backed by databases or source code control systems for the purpose of versioning and search. A good example of group planning social software is www.basecamp.org or just for distributed appointment scheduling www.doodle.ch.

And we haven’t even touched games, especially multi-player online games yet. There is lots of content created in those virtual worlds (characters, buildings, stories etc.). And sometimes the distribution infrastructure plays an important role even for the game content.

Something important to notice here is that the game content – the story – is heavily influenced by the fact of distribution (latency, bandwidth etc.) force authors to different game ideas which have a chance to work in such a distributed environment. But also the fact of independent players communicating with each other can drive the game into totally different directions: this is owed to the interaction property of distributed systems. Those online games have to solve very difficult problems from security to fast updates, replication of the game world and so on.

Very close the idea of multi-player online games is the idea of collaborative work environments where people can develop things together. This could be source code, music, videos. Or people could create environments for learning.

Where is the Money?

We have already touched the money question in the chapter on digital media and the fragile concept of ownership. It comes back again via social media and interacting people. The obvious question is: who is paying for those social network sites and services?

Looking at all the available social network and community sites, services and offerings one question comes to mind: who pays for the services rendered? Further down we will take a close look at the necessary computing infrastructures to support online communities. It is true that on the client side – what Andy Oram once dubbed “the edge of the internet” sharing is easy due to CPU, disk, broadband connectivity etc. of the private machines being essentially free. But this looks very different once we look at how the services and communities are hosted on the server side (yes, there is still a lot of good old Client/Server computing going on and that is why this distribution architecture is discussed below).

Financing community sites adds more superstructures of distribution to the game. And it is all about advertising, at least initially. Sites will probably buy advertisements (e.g. from google adwords) to attract visitors. But soon sites can sell their own page space to PR broker networks (like www.affili.net) and generate money per click, lead etc.

But banner based PR is only one method to generate money. A successful community site can use the special, targeted collection of individuals of the community with their very special interests and sometime even social characteristics to offer companies who operate in the area of the social community very interesting services. I am not talking about selling community data directly. Instead, the community site can offer personalized, targeted information about articles or services to community members – and sell this as a service to participating companies. E.g. when a community member is searching for specific parts the companies selling those can be shown to the member – at a price of course. This way the site helps members to find interesting products or services.

Findability

The term “findability” was coined by Peter Morville in his book “Ambient Findability”. He defines it as:

- the quality of being locatable or navigable
- the degree to which a particular object is easy to discover or locate
- the degree to which a system or environment supports navigation and retrieval

[Morv] pg. 4

Finding something is a core problem of all distributed systems, human or computer-based. Services which help in finding things, services, addresses etc. are essential for the functioning of any distributed system. The things to be found can be internal items of a distributed computing system (e.g. the IP address of a host) or they can be images, videos and papers targeted for human consumption. And those services add more superstructures to our distributed systems e.g. via the connections created by search engines: Services supporting findability do have a self-reflective quality: they live within the environment and extract meta-data about the same environment which are then fed back into the same environment.

Search engines use distributed algorithms to the max, e.g. google invented the famous map/reduce (now map//reduce/merge) pattern for the application of algorithms to data on a large scale and at the same time seem to offer a certain resistance to federation technologies trying to increase scalability. We will take a look at search engine architecture further down. Search engines can use behavioral data (e.g. the search query terms within a certain time period) to predict trends like a possible outbreak of the flu (<http://googleblog.blogspot.com/2008/11/tracking-flu-trends.html>)

Epidemics

Michael Jackson's death has once again shown how fragile our systems are in case of sudden, unexpected events with a high social value for many participants. Several large sites were brought to a stillstand and – according to some rumors – google thought they were suffering from a distributed Denial-of-Service attack when they saw the high numbers of requests for M.J.

Systems have a hard time to adjust to such epidemic behavior and we will take a look at algorithms and architectures which might be capable of more resilience against this problem. Cloud Computing is one of the keywords here, as well as sophisticated use of consistent hashing algorithms as well as – surprise – epidemic information distribution algorithms. Fighting social epidemics with epidemic communication protocols!

Group Behavior

Findability, media and social networks create the environment for user behavior, or should we say group behavior as no users can easily aggregate into various groups. According to a heise news article on research at the ETH Zurich. [Heise119014], Riley Crane and Didier Sornette are investigating the viewing lifecycle of YouTube videos from being almost

unknown over creating a hype and then finally ending in oblivion [Crane]. The social reception systems seem to follow physical laws in certain cases, like the waves of aftershocks after an earthquake. Mathematical formulas can describe this behaviour – but is it really a surprise? Becoming popular requires social and technical distribution networks which have characteristics with respect to connectivity, topology etc. which define the speed and type of distribution. And in this case several different distribution systems (e-mail, blogs, talks between colleagues and friends, mobile phones etc. all participate in generating a certain epidemic viewing pattern. The researchers intend to use the viewing patterns to predict trends and “blockbusters” early on.

For the owners of large scale community sites the user and group behaviour is essential as well. Not only to make sure that the sites attract many people but also as a technical challenge. Those sites need to show exceptional scalability due to the spikes or avalanches in user behaviour mentioned above.

Distribution is a general property and phenomenon that shows up on many levels of human or technical systems. And these systems can have a big impact on each other. We will discuss Clay Shirky's statement that computer based social networks have changed our ability to get organized – which is a requirement for successful political action. While the impact on political actions is perhaps still debatable, the different ways of technical and social distribution systems have had a clear impact on the development of source code, especially Open Source. The following quote is from the Drizzle development team and shows how interconnected the various systems already are. You need to add to this list of services used all the instant messaging, chat, e-mail, phone and live video channels used during development to get an idea of how social and technical systems today are connected. “Participation is easy and fun. The Drizzle project is run using open source software (like Bazaar) on open and public resources (like <http://launchpad.net/drizzle> and <irc://irc.freenode.net/#drizzle> and <http://www.mediawiki.org>) in a true open source fashion. The Drizzle project also has [clear guidelines for participation](#), as well as [simple guidelines for licensing and use](#)“. [AboutDrizzle]

Group behaviour is important for the implementation of social sites as well: Can users be clustered together according to some criteria? In this case keeping the users belonging to this cluster together e.g. in a data store makes lookups much faster. And changes to the data stay probably within the cluster of users.

Massively Multiplayer Online Games have a rather natural way to group users by geography: All users within a certain game location are “close” to each other which means notifications need not exceed the location borders. It might even pay off to organize the group of users just for the time they spend in one game location into the same computer representation, e.g. the same cache.

Notifications and group behaviour are key. Facebook tries to find friend networks within their user data and use this for improved site performance by organizing data sets differently. Here the user clusters are more static than in the game case. And group behaviour – either static or dynamic – presents large problems for scalability: Facebook is limiting notifications to groups smaller than 5000 participants. In other words once your group gets larger than 5000 members you can no longer send a message to all of them easily. (twenty minutes). MMOGs sometimes create copies of game locations and distribute users to those “shards”. We will talk more about these partitionings later.

Social Graphs

<<open social, db models of social graphs, messages and numbers>>
<http://www.infoq.com/presentations/josh-elman-glue-facebook-web>

what can be done with this information? Social networks driving Content Delivery Networks?

Superstructures

Clay Shirky gives a nice example of the extreme fan-out possible due to the interconnectedness of different social and technical systems:

*Let me tell you what happened to a friend of mine: a former student, a colleague and a good friend. Last December decided to break off her engagement. She also had to engage in the 21st century practice of changing the status of her relationship. Might as well buy a billboard. She has a lot of friends on Facebook, but she also has a lot of friends on Facebook. She doesn't want all these folks, especially fiance's friends, to find out about this. She goes on to Facebook and thinks she's going to fix this problem. She finds their privacy policy and the interface for managing her privacy. She checks the appropriate check boxes and she's able to go from engaged to single. Two seconds later every single friend in her network get the message. E-mails, IMs, phone is ringing off the hook. Total disasterous privacy meltdown.
Kris Yordan on Clay Shirky, Filter Failures Talk at Web Expo 2008*

Shirky notes that privacy sometimes used to rest on inefficient information distribution. Those days are over. Information distribution happens on many channels at the same time and this fan-out can in seconds lead to an extreme overload on single systems. The friends and the *friends* from above will turn around and take a look at her profile. And this turns a rather long tail personal profile into a hotspot which possibly needs a different system architecture to scale well, e.g. dynamic caching. You don't cache long tail information usually.

The API Web – the Sensor Web – the Open Web?

(Tim O'Reilly, Web Expo)

Twitter – a sensor web? Scalability for Billions of sensors, possible via IPV6. Is there an open pub-sub infrastructure for sensors and actors?

Facebook – a dispatcher of social information?

“The knowledge tidbit that stuck out more in my mind than any other was that Twitter gets 10 times the amount of traffic from its API than it does through its website. It makes sense, I’d just never acknowledged it explicitly. [Dion Hinchcliffe’s workshop](#) painted a similar story for many other Web 2.0 successes. The canonical example is YouTube with the embedded video. The decision to put html snippets plainly visible, right beside of the video, was perhaps their most genius move. Modern web applications and services are making themselves relevant by opening as many channels of distribution possible through feeds, widgets, badges, and programmable APIs.” Kris Jordan, <http://www.krisjordan.com/2008/09/25/10-high-order-bits-from-the-web-20-expo-in-ny/>

Joseph Smarr tied together a number of technologies that will create the open web and thereby further accelerate the growth of social sites: OpenID (who you are), OAuth (what you allow) and XRDS for a description of APIs and social graphs. They all belong to the open stack (with open social etc.)

Currently lots of social information is locked up in silos. Some users just give away their passwords to allow the use of their social information from another site but this is obviously very dangerous. Facebook uses a redirect mechanism between third party sites and itself – much like liberty alliance: Requests are bounced back and forth and Facebook adds a token after successfully authenticating a user. The third party site does not learn credentials from users. But all this is still not perfect as my list of friends from one silo may be completely useless within another silo. XRDS will allow the specification of detailed social information together with fine granular access, protected by OAuth technology.

Of course this open stack will again increase the load on social sites through the use of their APIs.

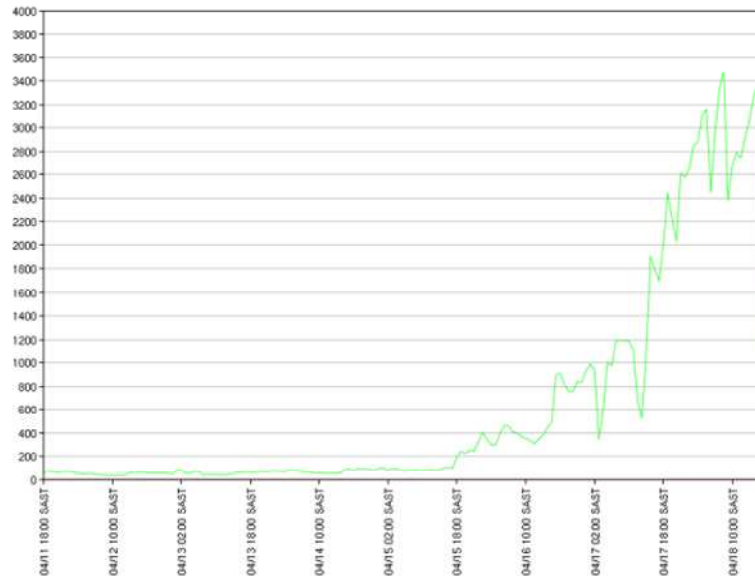
For OpenID see: <http://www.heise.de/newsticker/Identity-Management-Authentifizierungsdienste-mit-OpenID--/meldung/136589>

Supersize Me – on network effects and endless growth

Growth on the internet seems to follow a scale free pattern: many small sites, fewer mid-sized sites and very few supersized ones like Google. This seems to be the case also with social networks. We see strong competition between social sites currently – based on the recognition of the crucial start-up phase and its consequences for future growth. Why is it that the internet weeds out so many competitors and leaves only a small number of survivors? Communication platforms always show strong network effects: new participants increase the value of the platform even more for all participants. But this is only true within communication platforms, not between. A new myspace participant does not increase the value for facebook members and vice versa. These systems are technically isolated name- and rights spaces. This means in turn that every new participant in such a system has a rather high value for the platform – especially during the startup phase. And it means that the selection process will be brutal because members of platforms with a smaller growth rate will experience increasing isolation effects.

But there is an escape for this disadvantage: Just build on top of a successful community site which shows scale-free growth. Animoto is a good example. It is a facebook application which lets you supply pictures and audio data and creates videos from it. It supposedly grew from 50 to 5000 servers [NY Web Expo 2.0] in two days using amazon's computing cloud.

Animoto scalability on EC2, from Brandon Watsons blog



Looks like systems interfacing with those giant sites which show epidemic user behavior inherit this behavior. In the section on cloud computing we will discuss the ramifications of this fact.

The few supersites we see today are therefore also a consequence of social network applications. Growth, speed and the ability to provide new features quickly are what drives these super-sites. The rest of the book will take a closer look at the way these sites deal with their growth and speed requirements. And it is no real surprise that the first result is quite obvious: they are highly distributed systems. And that is why we start with a short presentation of distributed computing and how it developed into something that can support the super-sites of today.

Security

- federation of social applications
- private data (selling, de-anonymization)

Today's social applications receive, collect, store, analyze and re-distribute social data of their users. One of the biggest problems in this context comes from the fact that in many cases more than just two parties are involved: users want to allow other users or applications access to their private data. Marc Zuckerberg e.g. describes the way facebook allows this kind of access through a distributed authorization system.

Social sites also sell those data – albeit in an anonymized form – to PR agencies and interested parties. It is assumed that by leaving names and direct addresses out the users identity is protected. This is not true as has been shown in studies. <<de-anonymization>>
But above all is the danger of semantic attacks on users – digital analogies to the “art of deception” honed by Kevin Mitnick with his mostly telephone based spoofings and impersonations. Bruce Schneier describes nicely how e.g. identity theft and deception work in social networks.

Deception in Social Networks

Social Networking Identity Theft Scams

Clever:

I'm going to tell you exactly how someone can trick you into thinking they're your friend. Now, before you send me hate mail for revealing this deep, dark secret, let me assure you that the scammers, crooks, predators, stalkers and identity thieves are already aware of this trick. It works only because the public is not aware of it. If you're scamming someone, here's what you'd do:

Step 1: Request to be "friends" with a dozen strangers on MySpace. Let's say half of them accept. Collect a list of all their friends.

Step 2: Go to Facebook and search for those six people. Let's say you find four of them also on Facebook. Request to be their friends on Facebook. All accept because you're already an established friend.

Step 3: Now compare the MySpace friends against the Facebook friends. Generate a list of people that are on MySpace but are not on Facebook. Grab the photos and profile data on those people from MySpace and use it to create false but convincing profiles on Facebook. Send "friend" requests to your victims on Facebook.

As a bonus, others who are friends of both your victims and your fake self will contact you to be friends and, of course, you'll accept. In fact, Facebook itself will suggest you as a friend to those people.

(Think about the trust factor here. For these secondary victims, they not only feel they know you, but actually request "friend" status. They sought you out.)

Step 4: Now, you're in business. You can ask things of these people that only friends dare ask.

Like what? Lend me \$500. When are you going out of town? Etc.

The author has no evidence that anyone has actually done this, but certainly someone will do this sometime in the future.

We have seen attacks by people hijacking existing social networking

accounts:

Rutberg was the victim of a new, targeted version of a very old scam -- the "Nigerian," or "419," ploy. The first reports of such scams emerged back in November, part of a new trend in the computer underground -- rather than sending out millions of spam messages in the hopes of trapping a tiny fraction of recipients, Web criminals are getting much more personal in their attacks, using social networking sites and other databases to make their story lines much more believable.

In Rutberg's case, criminals managed to steal his Facebook login password, steal his Facebook identity, and change his page to make it appear he was in trouble. Next, the criminals sent e-mails to dozens of friends, begging them for help.

"Can you just get some money to us," the imposter implored to one of Rutberg's friends. "I tried Amex and it's not going through. ... I'll refund you as soon as I'm back home. Let me know please."

Posted on April 8, 2009 at 6:43 AM * 52 Comments * 14 Blog Reactions

To receive these entries once a month by e-mail, sign up for the Crypto-Gram Newsletter.

Comments

Federated Access Control to Private Data

The scenario is quite simple: A new application wants to use private user data in Facebook to allow a better service to its users, e.g. by showing to a user what his friends selected using the new application. To this avail the application needs to get access to the user's data within Facebook. A no-good solution of course is to ask the user for her Facebook login credentials (userid, password) and store them for later use. The new application "impersonates" the user in this case -- and could do so any time later without the user's consent because the credentials are no longer a secret between the user and just Facebook.

Recognizing that 3rd party applications would in the end fall back to such risky behavior most social sites realized that they need a way to federate security between sites without publishing secret credential information. Luckily such systems have been developed already for federated e-business on the web (see e.g. the Liberty Alliance proposal, SAML2 or the WS-Federation and WS-Trust standards) and can be used between social applications as well. The principle is rather simple: The original credential-keeping site (e.g. Facebook) is used to perform an initial authentication of the user and a token is generated for the third-party site. If the site needs access to user data it presents the token and thereby proves to Facebook that it acts as an agent for the user. Of course the tokens expire after a short time.

Technically so called federated security can be implemented in different ways <<slide from book one security..>> and it relies on a trust relation between the original site and the third party site. The third party site trusts the original site with respect to authentication, the original site accepts the third party as a user representative. The user herself trusts both sites with respect to proper use of the access right to private user data. The token generated during this process could further restrict access to parts of the user data only. Based on opened – an open standard for authentication on the web – a new standard called openauth has been proposed to allow the specification of access control rules in social sites.

A special case is where a user wants to authorize another user for access to her data or parts of them. Here the generated token is not handed over directly to an application of the same user but to a different user altogether who might want to use it in various applications. Again, the access rights behind such a token should be limited in power and time.

Problems with Oauth:

<http://blog.oauth.net/2009/04/22/acknowledgement-of-the-oauth-security-issue/>

De-Anonymization of Private Data

Social sites frequently sell anonymized user data. But it turned out that with the help of correlation techniques a users identity can be easily reconstructed from those anonymized data. <<example papers>>

Reality Mining: <http://www.heise.de/newsticker/Von-der-Idee-zum-Geschaef-Reality-Mining--/meldung/136644>

Geo-location used for de-anonymization: (from [Schneier]

Counterpane newsletter June 2009.

Philippe Golle and Kurt Partridge of PARC have a cute paper on the

anonymity of geo-location data. They analyze data from the U.S. Census

and show that for the average person, knowing their approximate home and

work locations -- to a block level -- identifies them uniquely.

Even if we look at the much coarser granularity of a census tract -- tracts correspond roughly to ZIP codes; there are on average 1,500 people per census tract -- for the average person, there are only around

20 other people who share the same home and work location.

There's more:

5% of people are uniquely identified by their home and work locations

even if it is known only at the census tract level. One reason for this is that people who live and work in very different areas (say, different counties) are much more easily identifiable, as one might expect.

"On the Anonymity of Home/Work Location Pairs," by Philippe Golle and Kurt Partridge:

Abstract:

Many applications benefit from user location data, but location data raises privacy concerns. Anonymization can protect privacy, but identities can sometimes be inferred from supposedly anonymous data.

This paper studies a new attack on the anonymity of location data.

We

show that if the approximate locations of an individual's home and workplace can both be deduced from a location trace, then the median

size of the individual's anonymity set in the U.S. working population is

1, 21 and 34,980, for locations known at the granularity of a census block, census tract and county respectively. The location data of people

who live and work in different regions can be re-identified even more

easily. Our results show that the threat of re-identification for location data is much greater when the individual's home and work locations can both be deduced from the data. To preserve anonymity, we

offer guidance for obfuscating location traces before they are disclosed.

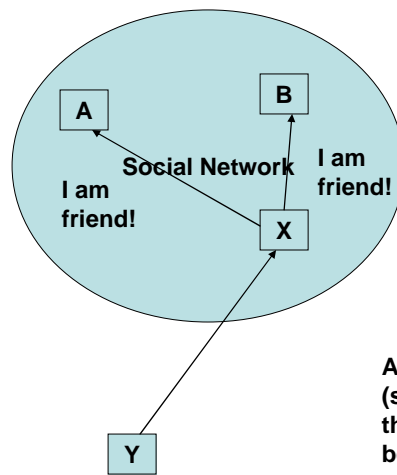
This is all very troubling, given the number of location-based services springing up and the number of databases that are collecting location data.

Identity Spoofing in Social Networks

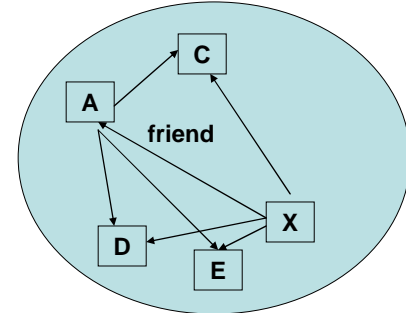
Recently some scenarios for the old "Nigerian attack" have been studied in social networks. In this attack an attacker impersonates a friend of the victim and tricks the victim into sending money e.g. via western union to some drop where it will be collected by the attacker.

The attack is made easier by the huge amount of private information that is made public in social networks. The first

diagram below shows an attacker Y creating fake accounts in a social network and sending friend requests to existing users there. Some will blindly accept those requests and thereby expose their social graph to the attacker. The attacker will record the graph and move over to a different social network.

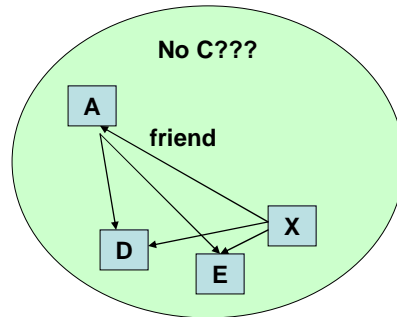


Y joins network as X and asks A, B for friend relation

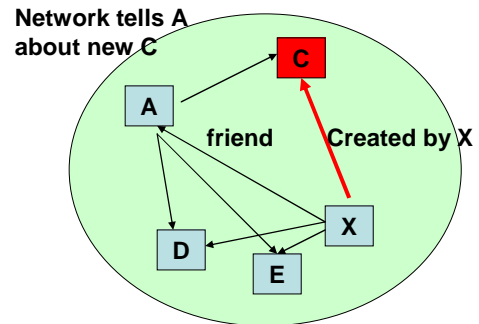


After being accepted by A as a friend (some users will accept anybody to bump their friend count), A's friend network becomes visible to X. X records the network and in the next step compares it with a different social network which he also joined and where A will most likely also accept him as a friend (he did it already once..).

On this second social network the attacker will also have a registration as X and he will send friend requests to A and A's friends which will most likely accept him as they did in the first social network already. X will again record the social graph around A and create a diff between both graphs. Users in one network but not in the other are now especially interesting to X. The attacker will create exactly those accounts in the network where they did not exist yet, copy real private data and pictures from those users in the other network over to the new accounts and create plausible identities by doing so. A and his friends will probably believe that those new accounts are also driven by their friends in the other network and not notice that they are really controlled by X.



In the new social network X also becomes friend with A and records the social network of A. He notices that C is missing. X creates C and uses C-private data from the other network to build a plausible persona (pictures, story, profile data).



The last step is for X impersonating C to send A a message about an emergency and A should send money to some western union spot somewhere („nigerian attack“). A thinks C is „his“ C from the first network but C is really a fake identity created by the attacker

Finally X will send an urgent message from one of the controlled accounts to A pretending an emergency and asking for money to be sent. In one case reported by a Microsoft employee there was a damage of \$1200 done.

Don't be too quick in dismissing this attack as being too far fetched. What would be the message that would make YOU act (perhaps with a bad feeling but still..). What if your other social network told you that C really is in London right now where you should send the money too because your dear American colleague has become a victim of European criminals? What if it involves family? What if it involves a technically challenged mother who just lost her husband and now needs help from her son? This is very specific but exactly this very specific type of information is sent by your social network to numerous people all over the world. In essence the social networks make the gathering of intelligence as a pre-requisite of trust establishment much easier. The mechanisms and patterns have been described by Kevin Mitnick in "The Art of Deception".

Scams

Is security of social networks really a technical problem? The post by Chris Walters about the impossibility of selling a laptop on ebay nowadays points to a very difficult relation between technical means and improved scams: does paypal make things really safer for buyers or sellers? Is the option to welch on a won auction really an improvement for ebay? (real auctions are non-revokable and cash-based).

<http://consumerist.com/5007790/its-now-completely-impossible-to-sell-a-laptop-on-ebay>

The post also shows some clever tactics by paypal to fight scams. What could ebay do to help people who had their account misused? What could they do to warn potential clients when e.g. suddenly addresses are changed? Does a changed address affect social reputation? What if the new address is in Nigeria?

Bootstrapping a large community

<<what is needed to build a large community? Patterns? Financials? Effect of chaotic influences on early starters == small wins turn into huge benefits. Small differences give a headstart with the network effect amplifying the wins.>>

Part II: Distributed Systems

Basics of Distributed Computing Systems

It is now about time to go one level deeper and take a look at the distributed computing technologies, infrastructures and applications that run all these social networks, communities and sites. We will do this in the form of a short history of distributed computing with its major achievements and mistakes. The goal is to allow the reader to understand the future possibilities but also the limitations of distributed computing systems.

Remoteness, Concurrency and Interactions

Distributed systems are characterized by two qualities: Concurrency and remoteness. Taken together they allow interactivity and are responsible for the decidedly non-deterministic, “alive” nature of distributed systems.

Concurrency leads to independent units communicating with each other. This interaction creates a distributed algorithm which comes to life only through the execution of local algorithms. In effect this means that distributed systems are of an emergent quality – difficult to develop and execute. But it also offers a positive quality: a chance to do more by using many execution units, a chance to have a more robust system due to the independence of the parts and possible redundancies. The price lies in increased synchronization costs and in increased costs for redundancy. The concurrency quality does not fit well with human programming abilities due to its complexity. Think about the sequential nature of human programs need to have to be understandable. A fundamental mismatch that special types of software called middleware want to mitigate (see below).

Remoteness implies a different quality of communication with respect to failure potential, speed, latency and throughput. Remoteness usually is seen as a problem due to the failure potential it implies. The other side of this coin is the possibility of several partners performing the same services and thereby providing a level of redundancy that can be higher than in non-distributed systems. It CAN be but usually will not because of the fact that this – intrinsically required redundancy has high costs associated with it. And this leads to the design of distributed applications without redundancy which gave distributed systems the general impression of low reliability associated with high costs.

Both, remoteness and concurrency form a third quality: computationally independent agents which can communicate and collaborate towards individual or common goals. It is this interactive quality that makes distributed systems rather special: difficult, surprising and sometimes creative.

Remoteness needs to be qualified even further: the topology of communication paths is of extreme importance in a distributed system. It decides whether the architecture is client-server, hierarchical or totally distributed in a peer-to-peer manor.

And within the frame built by remoteness and concurrency, topology has a major impact on performance, reliability and failures. And lastly upon the distributed application as well because we will see a tight dependency of application types and topology. A dependency probably much tighter than the one between applications and the distributed middleware chosen – which is itself dependent on the topology of communication.

An example: The last twenty years have seen the migration of transactional applications from mainframes to distributed mid-range systems. Only the database parts were frequently kept on mainframes. This turned out to be a major administration, performance and reliability problem because the midrange distributed systems could not really perform the transactions at the required rates and reliability – but turned out to be rather expensive.

The type of a transactional application requires a central point of storage and control: concurrently accessed shared data with high business value which are non-idempotent (cannot be repeated without creating logical application errors). Trying to distribute this central point of control across systems did not work (scale) well and today the largest companies in many cases try to migrate applications back to mainframes – which have turned into distributed systems themselves by now but with special technology to mitigate the effects of concurrency and remoteness.

And take a look at the architecture of google. It is highly distributed and seems to be doing well. But the different topology: a large number of clients sending requests to a large number of linux hosts with the individual host being selected at runtime and at random is made possible by the type of application: a search engine which distributes requests to different but roughly identical indexes. If a google machine dies (as many of the supposedly 80000 machines will be doing during a day) a request may be lost, run against a slightly outdated index etc. But so what? In the worst case a client will repeat an unsuccessful request.

Choosing the proper topology for a distributed application is arguably the most important step in its design. It requires an understanding of the application needs with respect to latency, concurrency, scalability and availability. This is true for transactional e-banking applications as well as community sites, media services or massively multiplayer online games.

Another important question for distributed systems is what level of quality should be achieved. In case of a system crash – should partners recognize the crash or even suffer from it by being forced to redo requests? Or is a transparent failover required? Choosing the wrong QOS gets really expensive or error prone. And is it even possible to transparently continue a request on a different machine independent of when and where exactly the original request failed? (see Java cluster article by Wu). This requires a completely transactional service implementation – a rare thing in web applications.

When applications or even the lower technical layers of distributed services called middleware do not match the characteristics of the problem to the requirements of a distributed system we usually end up with slow and unreliable applications.

Functions of distributed systems

The relation between distributed systems and media has not exactly been a love affair. Actually many algorithms, techniques and even programming models used in the distributed computing community do not fit at all to the transport or manipulation of media, perhaps over unreliable open public networks with unknown latencies etc. In the next chapter we will therefore show the adaptations needed to support media handling. Right now we give an overview of rather “classic” distributed computing and its technical baseline.

At the lowest level of a distributed system itself the most important function is to send and receive messages in a reliable way – with reliable meaning “at most once” semantics in most cases: a request will not be executed on the receiver side more than once, even if a sender did send it twice (perhaps because a response from the server got lost). Without such a failure detection logic which requires a message protocol with numbered requests, acknowledgements and state keeping at the receiver side, we would end up e.g. with orders executed several times and having goods shipped several times to our home.

These messages can be sent synchronously or asynchronously (server sends response some undetermined time later in a different request or no response at all is expected).

On a higher level – when the distributed systems needs to perform real application work – more functions are needed. The most popular ones are functions to find things (which includes names and directories, helper services like traders and brokers which mediate between requestors and providers). There is a host of “finding” services available in the distributed world, starting with the way hostnames are turned into real IP addresses via the domain name system (DNS) over centralized services called registries that keep information or objects (JNDI, X.500, LDAP) and finally the distributed indexes of peer-to-peer overlay networks. Taking this support for “findability” away from a distributed service has the same effect as shutting down google on the distributed media level or getting rid of white pages and phone registers in general.

Once things – which can be data or services (the ability to command something) – are found, they need to be accessed. This requires a protocol that allows transfer of data and or commands, including access control and concurrency control. The first should prevent illegal access, the second data corruption through concurrent modifications.

A sub-function of finding things is describing them so that they can be found and understood and used. Traditionally this has been the field of interface description languages which describe the data types and commands of messages that will be understood by receivers. Lately this

has been considerably extended. Description now includes all kinds of meta-data describing the provided services so that customers can decide whether and how to use the service. The role of meta-data, semantics and ontologies will only increase in the future of distributed systems. Most of these descriptions today are done in XML. On this level we see a major difference to the distributed media level: Unlike people distributed computing systems react very badly to slight changes to protocols or structures used in the transport of messages or content. Most systems cannot automatically adjust to changes in this area and this fact has led to two different attitudes towards those changes: either make the adjustments quick and either because changes will always happen – or try to avoid changes as much as possible using long term interface planning. No real winner has been decided with respect to this question.

We have mentioned “coordination” already above when we talked about the use of social network sites and communities to let people organize themselves (the Obama election fight e.g.). Within distributed computing systems we also have the need for coordination e.g. when a group of systems needs to work towards a goal or if a group of systems needs to learn the exact same outcome of something. In these cases we use voting algorithms like the famous two-phase-commit to achieve transactional qualities when we change data in several steps. Advanced algorithms use replication and multicast messages extensively to make progress even in case of individual failures in the group. [Birm].

There are many more functions needed in distributed systems like time service or a service that provides a global ordering of events within the system so that the causality of events can be respected. These functions are intrinsic requirements in distributed systems. Most of them are a must have for distributed applications (at different levels of quality of course). Unfortunately creating those functions in the context of concurrency and remoteness is hard and applications which try to implement those functions spend most of their time with system-level problems. When this mismatch between application programming and distributed functions was recognized the term “middleware” was born.

Manifestation: Middleware and Programming Models

Before we dive into a short history of middleware and the associated programming models we need to introduce two core terms: transparency and request granularity. Transparency means that certain ugly side-effects of distribution become invisible to the programmer – they become a “don’t bother” entity. Request granularity is how the message transport protocol in a distributed system is designed and especially used. Both concepts have led to horrible mistakes in the history of distributed computing. Overdoing transparency by promising that all effects of distribution are hidden by clever middleware led programmers to believe that things like latency, communication failures etc. do no longer exist. The result were slow and buggy applications because no matter how much middleware is put in place on communicating machines: it won’t bring Munich closer to Rome...

The same goes for request granularity. The decision about how big messages should be, how frequent and possibly asynchronous they should be is a function of the application design, the bandwidth available, the machine and network latencies, the reliability of all involved components etc. Traditionally distributed computing applications within organizations have tended to a rather fine-granular message structure and frequency – thereby mimicking the classical sequential and local computing model of programming languages. Internet-savvy distributed applications have on the other side always favored a more coarse grained message model. This can be seen in the ftp protocol and especially in the document centric design of the WWW and its http protocol. (see below REST architecture). If there is one lesson to be learned it is that no matter how clever middleware and programming model are, they cannot and probably should not hide the realities of distributed systems completely. Every middleware makes some assumptions and in most cases those assumptions cannot be circumvented e.g. by a different design of interfaces and messages by the application programmer: You can use a CORBA system for “data schlepping” but it will never be as efficient as e.g. ftp for that purpose.

Middleware is system-level software that was supposed to shield application programmers from the nitty-gritty details of distributed programming. But there is a large range of possibilities: from simple helper functions to send messages to completely hiding the fact that a function or method call was in fact a remote message to a remote system.

Over the time this transparency became more and more supported and developers of distributed system middleware decided to make concurrency and remoteness disappear completely from an application programmers list of programming constructs. Did regular programming languages contain special commands for distributed functions? No – so why should a programmer be forced to deal with these problems?

The concept of hiding remoteness and concurrency started with remote procedure calls. A regular function call got split into two parts: a client side proxy function which took the arguments, packaged them into a message and sent the message to some server. And a server side stub function which unpacked (un-marshalled) the arguments and called a local function to perform the requested processing. The necessary glue-code to package and ship command and arguments was mostly generated from a so called interface definition and hidden within a library that would be linked to client and server programs.

Programmers would no longer have to deal directly with concurrency or remoteness. A function call would simply wait until the server would send a response. The price being paid was that concurrency could no longer be leveraged because the program behaved like a local one and waited for the response to be ready. But this price was deemed acceptable.

The next steps were the introduction of OO technologies to even better hide remoteness and concurrency behind the OO concept of interface and implementation. Objects could also bundle functions better into

namespaces and avoid name clashes. The proxy pattern allowed nearly complete transparency of remote calls. Only in special exceptions a programmer became aware of the methods being remotely executable.

Already at that stage some architects (like Jim Waldo of SUN) saw problems behind the transparency dogma. He showed that the fact of concurrency and remoteness cannot be completely kept from application programmers and that the price to try this is too high. He showed e.g. the difference in calling semantics between local methods (by reference) and remote methods (by value) and that the respective functions should be clearly different to avoid programmer confusion (e.g. mixing by value and by reference semantics). He was surely right but may have missed to most important mismatch anyway: No matter how clever a middleware tried to hide the effects of concurrency and remoteness from programmers – it could never make these qualities of distributed systems disappear: Bad latency, confused servers etc. would still make a distributed system BEHAVE differently. The dogma of transparency and its realization in middleware caused many extremely slow distributed applications because the programmers no longer realized that the effects of network latency etc. would not disappear behind software interfaces.

But this was not the only problem that plagued distributed system middleware. The resulting applications also proved to be rather brittle with respect to changes in requirements which in turn caused frequent changes in the interfaces. The fine grained concept of objects having many methods turned out to be too fine grained for distributed systems. The consequences were rather brutal for many projects: Object models created by object experts had to be completely re-engineered for use in distributed systems. The resulting design featured components instead of objects as the centerpieces of architecture: coarse grained software entities featuring a stable interface for clients, hiding internal dependencies and relationships completely from clients. Whole design patterns were created to enforce this model of loose coupling, like façade and business delegate in the case of J2EE.

These components were still pretty much compile-time entities, focused at programmers to allow reuse and recombination into ever new applications. Enterprise Java Beans technology kind of represents the highest level of transparency and separation of context in this area. Programmers do no longer deal with concerns like transactions, security, concurrency and persistence. Components are customized through configuration information written in XML.

This distributed technology always had scalability problems – even in the protected and controlled environment of an intranet. The load on server machines was huge as they had the task of keeping up the transparency promise, e.g. by dynamically loading and unloading objects depending on use and system load. Cluster technology was introduced to mitigate the performance and reliability problems. Nevertheless – a globally visible entity representing a business data object and running within transactions always represents a bottleneck.

And a final example of mismatch between programming model and reality is the topic of distributed transactions. The objective of distributed transactions is to create the illusion of global serialization of actions within a distributed system. This is usually achieved by defining a quorum on the outcome of a global action – in other words a vote is taken by the participants and the global action is either accepted or rejected (sometimes all participants need to vote the same way, sometimes a majority is enough). The result is that a number of updates to data on different machines – which will necessarily take many messages and some time to do - can be done “in one go” or atomically and therefore consistent with a certain plan.

<<diagram dist.trans>>

But the performance costs and fragility of distributed transactions are considered very high. Blocking or not-responsive nodes can prevent the vote from terminating and a lot of bookkeeping is required. Some algorithms though specialize in making progress even in case of single node failures. Interested readers are pointed to the virtual synchrony approach of Birman and others [Birm]. According to Pat Holland most applications do not assume a mechanism for distributed transactions [Holl], especially if they are dealing with extremely large scalability, e.g. order items being spread across many machines due to their numbers. What can we do in this case? Distributed transactions are convenient but do not scale or lead to availability problems because of their locks. Holland shows a typical pattern to be used in this case: the application and the application programmer needs to take over some of the responsibility for global serialization. There is no longer a mechanism for global serialization available, instead, the items to be changed are explicitly represented as entities within the business logic and pushed to the application level. Now global consistency is a question of arranging the proper workflow to achieve it. We will present Hollands solution in more detail in the section on adaptations of distributed systems.

Theoretical Underpinnings

A few theoretical considerations have turned out to be of essential importance to large-scale system design. They are in no specific order:

- failure is the norm, membership detection critical
- consistency and availability are trade-offs (Brewers’s conjecture, CAP Theorem)
- forward processing is essential
- end-to-end argument
- ways to reach consensus, vector clocks
- adaptability

The large number of components used causes repeated failures, crashes and replacements of infrastructure. This raises a couple of questions like how we detect failures and how algorithms deal with them. We need to bootstrap new components quickly but without disruption to existing processes. We need to distribute load quickly if one path turns out to be

dead. But the hardest question of all is: when do we have a failure in a distributed infrastructure? The short answer to this question is: we can't detect it in a distributed system with purely asynchronous communication. There is no clock in those systems and therefore we cannot distinguish e.g. a network partition from a crashed server/process/application. This is what is meant by "The Impossibility of Asynchronous Consensus", the famous Fischer-Lynch-Paterson Theorem. A good explanation of its value and limitations can be found in [Birman] pg. 294ff. The longer answer is that real systems usually have real-time clocks and they use algorithms to keep clock-drift between nodes under control. This allows them to define a message as "lost" and take action, e.g. reconfiguring dynamically into a new group of nodes. This allows progress to be made even in the presence of a network partition or server crash.

Probably the one theorem with the biggest impact on large-scale systems is "Brewer's conjecture", also called the CAP Theorem [Gilbert]. It simply states that we can have only two of the following three: consistent data, available data, network partitions at the same time. The reasons for this leads straight back to the discussion of failure detection in the asynchronous computing model: consensus is based on membership detection and this is again based on failure detection. The practical consequences are nowadays reflected in architectures like Amazon's Dynamo eventually consistent key/value store. Here the designers have chosen to favor availability over consistency (within limits) and use algorithms that achieve eventual consistency (background updates, gossip distribution etc.) The effects of eventual consistency can be somehow limited and we will discuss techniques to achieve this in the chapter on scale-agnostic algorithms, specifically optimistic replication strategies. An interesting feature of such systems is to hand back data with a qualifier that says: watch out, possibly stale. Or the possibility to hand back several versions which were found during a read-request to the client and let it choose which one it will use.

In many cases traditional algorithms tend to stop working in the presence of failures. A two-phase commit based transaction needs to wait for the coordinator to come up again to make further progress. There are a number of algorithms available – especially from group communication based on virtual synchrony – which allow processing to go forward even in case of failures.

<<some examples from birman and fbcast, cbcast, abcast, dynamic uniformity discussion >>

The end-to-end argument in distributed systems leads back to our discussion on transparency. It deals with the question of where to put certain functionalities. If a designer puts them too low in a processing stack (network stack), all applications on top of it need to carry the burden. But of course they also get the benefits of a built-in service. Large-scale systems need to use very special algorithms like eventually consistent replication and therefore have a need to push some functions and decisions higher up towards the application logic. Partitioning of data stores is

another area which requires the application to know certain things about the partitioning concept. Another good example is the question of transparent replicas across a number of nodes. What if an application distributes several copies of critical data in the hope of guaranteeing high-availability but incidentally the storage system put all the replicas into different VMs but on one big server? The application wants a largely transparent view of the storage subsystem but there are other views which need to know about real machines, real distribution etc. (in p2p systems the so called “Sybil attack” shows exactly this problem).

Consensus is at the core of distributed processing. To achieve consistency we need to have a group of nodes agree on something. It could be as basic as the membership of this group. Or some arbitrary replicated data value. Many different consensus protocols exist. Paxos e.g. is a quorum based, static group communication protocol with totally ordered messages and a dynamically uniform update behavior. In other words it is very reliable but potentially rather slow as it is based on a request/reply pattern for accessing the quorum members. [Birman] pg. 380. We will discuss Paxos below. The google lock service “Chubby” is based on it. It is used to implement what is called the “State-machine approach to distributed systems”: The consensus protocol is used to build a replicated log on the participating nodes which all nodes agree on. This means that nodes who run the same software and receive the same commands in the same order will end up in the same state. The commands received can be input to a database which will be in the same state on all nodes after processing those messages. More on the state-machine approach can be found at [Turner].

<<vector clocks and merkle trees>>

<<adaptability>>

Topologies and Communication Styles

The way participants in a distributed system are ordered and connected has a major impact on the functions of the system. We will discuss a number of well-known topologies and how they work.

Classic Client/Server Computing

Sound outdated, doesn't it? Today we do Cloud Computing, not old Client/Server stuff. Fact is: most of the new Web2.0 applications, the Software-as-a-Service (SaaS) applications like the google office suite all work in the client-server paradigm of distributed computing. It pays to take a look at what this paradigm really means.

Client-server computing is deeply asymmetric because expectations, assumptions, services and financial interests etc. all differ between clients and servers. Let us sum up some of the differences. Traditionally clients use services from servers. They have expectations of availability therefore. Clients send information to servers which means that they have expectations of security and privacy as well. Clients in most cases wait for the results which means the server plays an integral part in the workload of the client. And when there is a human being behind her “user agent” on the client side it means a sharp limit for the response time on the server and what a server can do during this time. Servers on the other hand cannot be

run by everybody like clients. Running servers is more expensive and requires more money.

But some things must have changed even in client-server computing? When we use cloud computing as an example then we can say that the servers certainly have gotten bigger. They turned into data centers actually. Only data-centers where whole clusters of servers look like one big machine to the clients can handle the traffic from millions of users which use the cluster to store media, use services etc.

And something else changed which we will discuss in more detail in the Web2.0 section: The many to one relation of classical client server (with clients having individual relations to the server maintainer but not to other clients) has become a many-to-many relation, perhaps not directly connected like in certain peer-to-peer networks but mediated through the cluster running the social community.

The Web Success Model

There is little doubt that the success model of the web is deeply rooted in the client-server mode of its operation. This is documented in its transport protocol http which operates asymmetric: clients start requests, servers answer but cannot by themselves initiate a communication with a client. And the success model of the web is deeply document or resource centric: nouns instead of the uncountable verbs of fine-grained, local distributed computing. This architecture has gotten the name REST which stands for Representational State Transfer – a term coined by Roy Fielding, one of the inventors of http and the web architecture. This architecture proved extremely scalable. It is the architecture which distributes lots of media around the world.

REST Architecture of the Web

What are the core characteristics of this architecture? Readers interested in the details and historical context should read the dissertation of Roy Fielding or his excerpt on just the REST architecture. But we will use a short paper from Alex Rodriguez on RESTful Web Servers which covers the basics [Rodr] and the excellent article by [Sletten]. And a little hint: The difference between REST and other ways of communicating between clients and servers is more a question of style than of technological platform. But this is true for many cases like the difference in interfaces between a concurrent and a single-threaded application. And sometimes a specific style fits an application area very well and then becomes “best practice”.

Rodriguez defines four strands that make a service RESTful:

- explicit use of http protocol in a CRUD like manner
- stateless design between client and server
- meaningful URIs which represent objects and their relationships in the form of directory entries (mostly parent/child or general/specific entity relations)
- use of XML or JSON as a transfer format and use of content negotiation with mime types

But in the end there is one principal difference between RESTful architectures and e.g. RPC-like messages: REST is all about nouns, not verbs. What does this mean? It means that the application developers design the interfaces to their system using a concept of nouns, documents or resources, not actions. Most distributed applications that use Remote Procedure Call (RPC) technology define a lot of actions that are offered on the server side: `add(x,y)`, `calculateFrom(input1, input2)`, `doX`, `doY(parameter)` and so on. There is an endless number of actions (messages or commands) that can be defined.

RESTful applications define access to their systems around the concept of nouns or things and what can be done with them. If you think a little about this concept you will realize that the actions around things are frequently rather limited and computer science has given those actions a short name: CRUD. Create, Read, Update, Delete is what is needed in dealing with things like documents, records in databases etc. And the parameters to those few actions are the name of the thing that it concerns (the URI of the resource) and an optional body with additional information in case of an update or create action. Doesn't this look very much like the good old Unix file API? It will do the job in many situations nicely. But there are limits and to understand the limits of REST it might be useful to take a look at the limits of the file API. Everything is a file, or? While true in general Unix systems had one important escape in case of problems with the limits of the file API: the `ioctl` system call. It could be seen as another way to write to the resource – and it actually writes to it. But what it writes are special commands, not data. This interface has seen much use and abuse. It breaks compatibility with existing tools which do not know about the intricacies of `ioctl` (much like a generic client does not understand special RPC methods provided by a server). And it has been abused to provide additional writes of data etc. The more the `ioctl` interface is used the less of the generic file API is useful and there have been applications and driver software that just used `open`, `close` and `ioctl` to do the job. With an extremely complex RPC interface hidden within the numerous parameters of the `ioctl` system call for that device. This type of design is certainly not REST like.

The RESTful interface and communication style could be called more abstract. It concentrates on the “what” instead of the “how”. And it has some side-effects that make it extremely valuable in a context that requires scalability and the help of intermediates, in other words, the web.

How does this noun-centric style of communication fit to Rodriguez four strands? When he says that explicit http should be used for RESTlike services it is exactly the CRUD functionality

that he demands. And http itself has very few actions that basically map perfectly to a CRUD like communication style:
GET -> Read (idempotent, does not change server state)
POST -> Create resource on the server
PUT -> Update Resource on the server
DELETE -> Delete Resource on server

A RESTful application that is true to this type of architecture will not use GET for anything that changes state on the server. This is actually quite an important property because crawlers etc. can rely on GET requests being idempotent so that they do not accidentally change state on a server.

A POST request should mention the parent URI in the URL and add the information needed to create the child in the body. Many frameworks for web applications did not understand the importance of separating idempotent operations from state changing operations. Instead, they foolishly folded most http operations into a single service-method and thereby lost the semantic difference. These frameworks allowed the definition of endless numbers of actions per application. Struts is a good example for the more action oriented thinking instead of a RESTful architecture. The focus is on the actions, not on the resources. Assembling an integrated page for a portal requires the assembler to know lots of actions which will finally extract the bits and pieces needed. In a RESTful architecture the assembler would use the names of the resources needed (the URIs) directly. Again, a different level of abstraction.

Is this separation of updates and reads something new? Not by far. Bertrand Meyer of OO fame calls this a core principle of sound software design and made it a requirement for his Eiffel programming language. He calls it “command-query separation principle”:

“Commands do not return a result; queries may not change the state – in other words they satisfy referential transparency” B. Meyer, Software Architecture: Object Oriented Versus Functional [Meyer]

Especially in the case of multithreaded applications referential transparency – the ability to know exactly that a call does not change state – makes understanding the system much easier. A few more interesting words from Meyer:

”This rule excludes the all too common scheme of calling a function to obtain a result *and* modify the state, which we guess is the real source of dissatisfaction with imperative programming, far more disturbing than the case of explicitly requesting a change through a command and then requesting information through a (side-effect free) query. The principle can also be stated as “*Asking a question should not change the answer*”. [Meyer], pg. 328f. The big advantage of separating changes from queries is that queries now become the quality of mathematical functions – they

will return always the same output for the same input, just like functional languages work.

(Just a small thought on the side: is this really true? Let's say I have created a query for the price of a thing. This looks like a idempotent, stateless method call at first sight. But what if a shop receives many of those queries in a short time? Couldn't the shop be tempted to increase the price based on the interpretation of those queries and increased interest?)

The principle of separating queries from changes is useful in practice. Just imagine the fun when you find that during the processing of a request several calls to databases are made (transacted) and that you have to do an additional http/rpc like request (not transacted) to a foreign server. It turns out that this request is for looking whether a certain customer already exists within the foreign server. And that this server will AUTOMATICALLY add the user once it receives a query for a user that is not yet in its database. This makes the code within your request processor much more complicated as it forces you to do compensating function calls in case something turns out wrong later with this user or in case you just wanted to do a lookup. Related to the question which method to chose for an operation is the question of response codes, especially where the http protocol is used. Badly designed response codes can make it very hard for an application to figure out what went wrong. Image the following scenario taken from a large scale enterprise search project: The search engine's crawler repeatedly crawls a site for new or changed articles. The site itself has the following policy regarding deleted articles: A request for a deleted article is redirected to a page which tells the user that this article is no longer available. This information itself is returned with a 200 OK status code which tells the crawler that everything is OK. The crawler will not be able to learn that the original page has been deleted. Only a human being reading the content of the response will realize it.

Here is a short list of status codes and their use, taken from Kris Jordan, towards RESTful PHP – 5 basic tips
[Jordan_RESTfulPHP]

***201 Created** is used when a new resource has been created. It should include a Location header which specifies the URL for the resource (i.e. books/1). The inclusion of a location header does not automatically forward the client to the resource, rather, 201 Created responses should include an entity (message body) which lists the location of the resource.*

***202 Accepted** allows the server to tell the client “yeah, we heard your order, we'll get to it soon.” Think the [Twitter API](#) on a busy day. Where 201 Created implies the resource has been created before a response returns, 202 Accepted implies the request is ok and in a queue somewhere.*

304 Not Modified in conjunction with caching and conditional GET requests (requests with *If-Modified-Since* / *If-None-Match* headers) allows web applications to say “the content hasn’t changed, continue using the cached version” without having to re-render and send the cached content down the pipe.

401 Unauthorized should be used when attempting to access a resource which requires authentication credentials the request does not carry. This is used in conjunction with *www-authentication*.

500 Internal Server Error is better than *OK* when your PHP script dies or reaches an exception.

Kris Jordan, <http://queue.acm.org/detail.cfm?id=1508221> see also

Joe Gregorio, How to Create A REST Protocol,

<http://www.xml.com/pub/a/2004/12/01/restful-web.html>

The second strand is stateless design. From the beginning of distributed systems the question of state on the server has been discussed many times over and over. And it is clear: forcing the server to keep state (to remember things about clients between calls from the clients) put the server at risk of resource exhaustion and performance problems, not to mention the failover problems in case of server crashes. But the discussion about distributed communication protocols has shown that sometimes state on a server just can’t be avoided to prevent duplicate execution or to achieve transactional guarantees. But the most important thing to remember is that the question of state can be heavily influenced by the design of the communication between client and server.

Distributed object technology tried to put the handling of state right in the middle of the architecture: after all, what are objects without the ability to hold state? And they paid a heavy price for this transparency in terms of performance and reliability as Enterprise Java Beans are proof of.

RESTful applications try to design the interfaces independent from each other and make the client hold state in between. The client will then add this state to his next call so that the server has all the information needed to process the request. Cookies are an ideal mechanism for that. In case the cookie cannot hold the information anymore at least the authorization part should still be kept there which is according to Jordan the way Flickr works.

Bad interface: `server.next()`

Good interface: `server.next(page 3)`

And of course the server will generate a response page with links to the next couple of pages.

We could now talk days and weeks about the problems of state in distributed systems. State has been used to attack systems, state needs to be tracked for performance reasons, state needs to be replicated for failover reasons and so on. But it is best when you can avoid the problems already at the design phase of your distributed application.

“Speaking URIs” is the third strand of REST. This is not a simple as it may sound. There are people who defined a URI as being

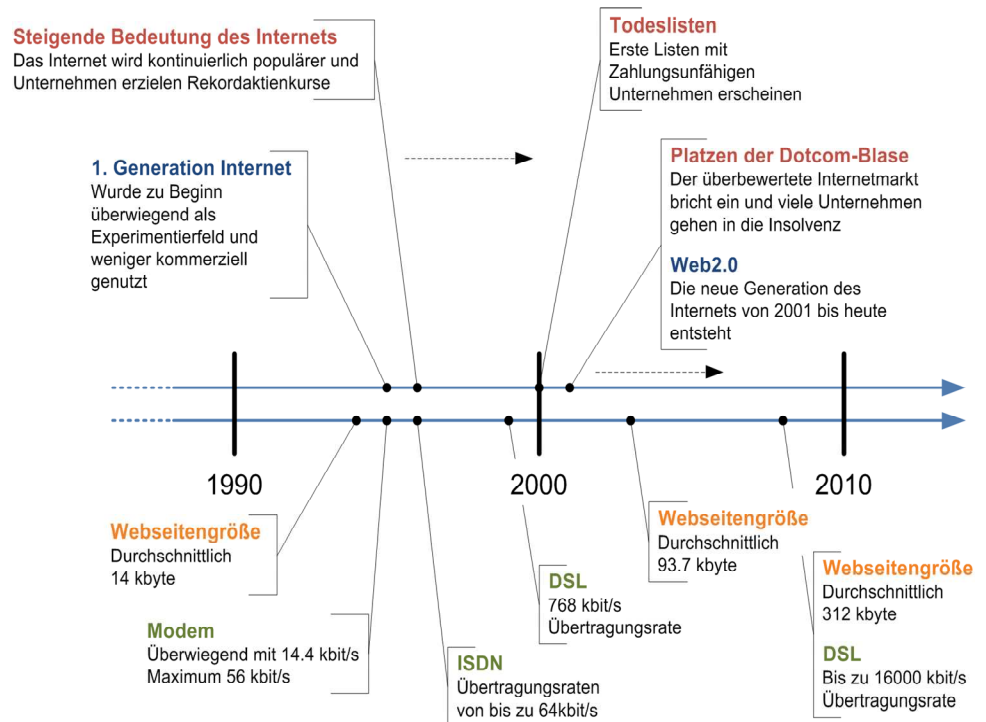
“opaque” in other words URIs should not encode any form of meaning. All they should be is unique. REST goes a very different way and asks you to encode your object model as a tree. Paths in the tree denote different objects at different levels of hierarchy and readers will be able to understand the path structure because it represents object relations in your application.

And last but not least RESTful applications should use XML or the JSON (Javascript object notation) format to transfer responses (and there should always be a response generated, even if the client asks for a partial URL only).

Today most web services offered follow the REST architectural style because it turned out to be the simplest one with the best performance. And by a happy coincidence RESTlike architectures seem to fit nicely into the new world of Web2.0 applications which we will investigate next. And afterwards we will look at the competition: Web-services based on XML, SOAP, WSDL and the SOA concept.

Web2.0 and beyond

There have been endless books and articles on Web2.0 and associated technologies like AJAX (e.g. “AJAX in der Praxis by Kai Jäger) and at the computer science and media faculty at HDM we have been early and strong adopters of this trend. Many community applications built with traditional or new languages (Ruby on Rails etc.) have been built during the last couple of years, accompanied by a stream of Web2.0 oriented special interest days. We will concentrate here only on some vital characteristics of Web2.0 as described by Till Issler [Issl]. The following diagram is taken with permission from his thesis:



It shows two important aspects of Web2.0. First it describes the development on the web after the crash of the dotcom bubble till today. And secondly it shows the first major characteristic of Web2.0: its dependence on bandwidth – in other words: broadband technology being available on a large scale at moderate prices. There would be no XouTube, no Flickr, no Facebook or StudiVZ, no iTunes etc. with only modem connections being available.

Lets list the major Web2.0 characteristics according to [Issl]:

- Availability of broadband connectivity
- The Web as a platform
- Web Services
- Users as active participants
- User generated content
- Collective intelligence

This list does not sound overly technical. Yes, the increasing bandwidth was necessary to carry media of all kinds in reasonable time and latency but the rest is more of a change in use and attitude than due to a breakthrough technology. One technology is frequently mentioned as THE Web2.0 technology. It is Asynchronous Javascript with XML or shortly AJAX. It consists of two major changes. The first change was to the communication protocol between client and server. Up to AJAX a client needing information did a request to some server and the result was a new page delivered by the server. There was no reasonable way to incrementally pull bits and pieces of information from a server and update the display accordingly. The famous XMLHttpRequest Object added to the browsers allowed Javascript code running on

the client to transparently and asynchronously pull information fragments from the server and update the screen in the background.

The result was a major increase in usability especially in web shops. Previously users had to input both zip code and city name because using the zip code to run a query on the server for the city name would have been a costly synchronous roundtrip resulting in a new page and thereby disrupting the user experience. Now even single keystrokes could be secretly sent to the server who used them to guess the word the user wanted to type (This has serious security implications because it changed the semantics of the page based communication style. Previously the user would have been forced to “submit” a page to send it to the server. Now client code running in the background could contact other servers (like google maps) to create so called mashups – mixes of information from different servers. Again in many cases a borderline or even clear violation of browser security but nevertheless extremely useful.

And this brings us to the second change caused by AJAX: the client platform (aka browser) became a powerful computing platform ready to run major source code (mostly javascript). This meant that some processing could be moved from the server back to the client. Remember, the good old client/server communication model always put a lot of strain on the server which could now be relieved a bit. On the other hand totally new functions were now possible e.g. the aggregation of information on the client and from different sources.

At the same time, and perhaps enabled by technologies like AJAX, the web turned into a computing platform itself. Things that required a fat client program previously are now being offered on the web. During this time the web also changed from an information gathering platform into an active application and service platform. This trend is far from being over: The webtop movement turned into things like Software-as-a-Service (SaaS) with google e.g. offering a complete office suite running on the web and Cloud Computing where more and more users store their data on some server on the web or use its services.

Web services in general became independent and composable, resulting in the previously mentioned “mashups”. Applications using different services from different providers. The typical example is a chain of stores enhancing their location finder with information from google maps.

It does not really matter whether those web services are implemented using RESTlike architectures or based on SOAP and XML/WSDL. It is only important that these services are available (round the clock) and that they are easily integrated into ones own applications.

User behavior changed considerably during those years. The number of internet users increased and we saw the birth of the “online family”. Families spending several hours a day connected and communicating with each other via instant messaging and chat. E-mail became the sign of the older generations. Users also became much more active (we talked about it in the first chapter) and this led to an increase in user generated content. Sites like LinkedIn or Xing basically provide a platform that is then filled by users. And so are many others like YouTube, Flickr etc. Users generate content and by doing so generate metadata, so called “attentional meta-data”. This is data derived from their behavior and it is the base of what we call “collective intelligence” today. It is intelligence derived from collective actions of users. One example is the tagging of things and thereby creating a classification automatically and for free. This classification has the additional advantage of being free from hierarchical control and authority as is usually the case with ontologies. It was Clay Shirky who coined the term “folksonomies” for this type of classification.

So far the list of Web2.0 features taken from [Issl] with some comments and add-ons. The WebX.0 trend is far from being over. We are now seeing more 3-D interfaces which we will discuss later in the chapter on virtual worlds.

Web-Services and SOA

While component models were at the height of the time, a separate development seemed to take distributed systems back into the past technologies: Webservices – a technology based on XML messages shipped mostly via http started to become popular. Their design wanted to follow the architecture of the web: loosely coupled services communicating via textual messages. No objects on the wire and therefore much less responsibility for the partners. And of course much less transparency as clients and servers were fully aware of the fact that they were communicating with remote systems using data copies as messages.

But the design did not completely follow the web principles: The web did not only operate stateless in many cases. The web uses http which provides only a few basic functions to send and receive documents – very much unlike traditional RPC models. It does not promise many other things as well: no transactions, no multi-party security, no guaranteed availability etc. And last but not least the web had a human being in its architecture as well: the person operating the user agent software. In other words: somebody bringing semantic understanding into the whole game – something webservices could not assume because they intended to provide collaboration between machines in the first place.

Initially webservices seemed to follow the Remote Procedure Call model of fine grained functions. Soon it became clear that this approach could work in a highly protected intranet with guaranteed response times but would raise a problem on the much less reliable internet. The communication style soon become more “REST”-like, using simple functions like http get and post to transfer document like data structures in XML. This scaled much better.

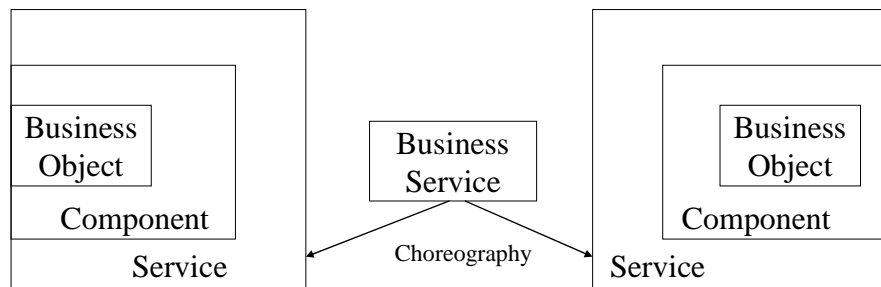
One of the most interesting concepts of web services was the automatic service discovery using a common repository called UDDI. Service providers would register their servcies in UDDI where clients would find them. Clients would use meta-data to understand and use those services. The services where described in WSDL – pretty much the same concept as an interface definition language (IDL) but written in XML.

UDDI was a major flop – simply because services described in XML does not imply that machine-requestors would UNDERSTAND those XML descriptions. UDDI ran into a major semantic problem of different terms and languages used to describe services.

The web services concept produced more and more specifications in the area of security, transactions and federation but it took an integrating concept to finally turn this soup of standards into an architecture: Service Oriented Architectures (SOA). Web services always raised the question of why they where needed. They did not really create any kind of new technology. Instead, they replicated old distributed computing concepts using a new terminology.

The SOA concept finally brought a breakthrough: It represents a top down architecture based on the notion of processes instead of objects or components. Processes use services to achieve their goals. The services are largely independent following the “loosely coupled” paradigm of web services. To be useful a service must be LIVE. This put the pressure no longer so much on development but on the runtime systems of distributed applications. A mission critical service must be available or a large number of business processes can be affected.

SOA Design



This diagram is modelled after O.Zimmermann et.al. „Elements of a Service-Oriented Analysis and Design“ (see resources). The paper also shows nicely how flow oriented a SOA really is and that a class diagram does not catch the essence of SOA. A state-diagram performs much better. The authors also note that SOA is process and not use-case driven design.

But what does “loosely coupled” really mean? Lets discuss this promise in terms of transactions, security and semantics. But first take a look at what the web does in those cases. There are no transactions on the web and especially no distributed transactions. By not providing these promises the web architecture becomes easy and scales well in an unreliable environment. Security is based on point-to-point relations with SSL as the mechanism of choice and does not allow multi-party relations easily. And semantics still rely on humans or specifications but largely escape machine interpretation.

In traditional distributed systems transactions, just like security, are context based. A context flows between calls and represents a transactional call or an authentication state. A distributed transaction would lock many objects in the distributed system. It was clear that the topology and QOS of the Internet made a different architecture necessary than on the intranet. Transactions e.g. could not use the locking based model of a two phase commit. Instead, partners cooperating on the internet need to rely on compensating functions in case of problems. But compensation is fundamentally a different business concept too: the system no longer tries to hide the fact that it is a distributed system. Instead, it makes some problems visible to the business to solve. Collaborating companies need to create a common security context, either through the use of central authorities (which does not fit well to the concept of loose coupling) or through federation. In any case intermediates may need to process those messages and add value to them. This means they have to sign their parts to make partners trust the information. SSL does not allow this kind of

collaborative document editing and webservices had to switch over to message based security (signatures and encryption of messages instead of using trusted channels)

The last problem: semantic, seems to be the hardest to solve. Cooperating services and organizations desperately need to be able to understand each other. But not in the way of the old wire protocol specifications which use tokens within the protocol whose meaning is caught in specifications. Instead, dynamically collaborating services need to discover meaning dynamically using e.g. ontology languages. Security assertions are one example where this would be needed.

Taken together “loosely coupled” can now be defined as:

- giving up on some transparency (like atomic distributed transactions) by bringing potential problems to the attention of higher instances (e.g. business with compensating functions)
- not using objects or object references on the wire
- keeping services largely independent of each other
- dynamically assemble services into larger processes through business process composition
- Specifying security requirements either in common languages (SAML) or using semantic technologies like ontologies to make partners understand each other
- Share live services instead of software components
- Give services the necessary environment to work through parameters (inversion of control)
- Model required and provided services for every service to allow reliable composition of larger processes.

The question of service resolution, i.e. how one service finds another one without creating a tight coupling is usually solved with the introduction of an Enterprise Service Bus (ESB) which takes over routing of requests.

<<ESB>>

But even with the introduction of an ESB, SOA can still mean a lot of hidden coupling. Services know when to call another service, what to call and especially what to expect from a service. Taken together this interaction mode is synchronous and stack oriented and a far cry from real de-coupling like in an event-based system. We will investigate different interaction modes below.

Never before SOA has distributed computing been closer to business concepts. Business thinks in process terms, not objects – a misunderstanding that took many years to get resolved.

This new software concept for distributed systems certainly takes into account the problems of services on unreliable and possibly slow internet connections – but it cannot completely mask them – nor does it try to do so. In a way this approach has existed for many years in distributed computing in the form of message oriented middleware which is closely related to the message oriented architecture of SOA. And it is still not without major problems as “The Generic SOA Failure Letter” by Mark Little demonstrates [Little].

But SOA may not even be the final answer to the problem of loose coupling. Simply knowing services and having them encoded into the control flow of applications and components makes them less suitable for arbitrary assembly into new designs. The answer here lies in the separation of another concern: interaction needs to be separated from computation. This is typically done in event-based distributed systems. Unlike the classic client/server paradigm of synchronous request and reply these systems separate participants to such a degree that they do no longer know about each other. Even worse: they do not expect other components to exist at all. In the best case they are written as autonomous components. Below we will discuss some of the qualities of event-driven systems.

Peer networks

We are leaving now the classic client/server topology which is not only the dominant model on the web but also in most other business related processes. These processes typically require central and atomic control over transactions. Once certain transaction numbers are exceeded or the processes are highly critical e.g. because of the money involved the server is frequently running on a mainframe type system. All the core business logic runs on the central system and the same goes for all transactions. This is not necessarily so but in most cases a central large server cluster (called a Sysplex in IBM lingo) is ideal for this type of communication. The concept of distributed transactions has been developed in the midrange system area but due to performance and reliability problems never became a dominant model.

When we now move to a different topology for distributed systems we will see that this will also change the kind of applications and processes we will run. Different topologies favour different kinds of applications.

Distributed systems with a business purpose mostly followed either client-server or hierarchical architectures but in academic research and now already in the consumer world a different distributed systems technology dominates today: Peer-To-Peer systems embrace millions of small home PCs and tie them into one distributed system. The P2P systems range from topologies which still use some kind of central server over hybrid systems where nodes can change roles dynamically to totally distributed systems without higher organisation.

<<slide on topologies of p2p>>

All these topologies have different characteristics and there are some dependencies between the degree of equality in P2P systems (e.g. all PCs run the same code and perform the same functions or there are some PCs which perform a dedicated function like keeping an index or other meta-data) and the way the system will perform or behave in case of crashes or attacks on single PCs. The higher the degree of equality, the more robust the P2P system will behave in case of attacks on single nodes. But at the same time its efficiency e.g. with respect to finding things will be lower and communication overhead will be higher.

Extensive research has been done on those systems and today the hybrid approach seems to be the most popular one. Some kind of special functions and services seems to be needed especially to locate services or documents with acceptable performance. Those functions will be performed by special nodes. But every node can theoretically change into one of those special function nodes if needed. This separates the important meta-data function from a specific host. Otherwise when a special, dedicated node is shut down (e.g. for legal reasons) the whole system stops working or becomes inefficient. The use of the meta-data e.g. to download a resource from a peer node usually happens in a direct peer-to-peer connection between requester and provider of a resource. Napster was a typical p2p system operating with a number of dedicated meta-data server who were finally taken down by the courts even if none of these servers served a single song directly to a client – ever. It is interesting to see that like with community sites (remember the Obama election site) there seems to be a balance necessary between hierarchy and anarchy.

The topology and communication style of distributed systems has a huge impact on the quality of service they can promise. There is less central control in p2p systems and therefore those systems can make less promises to their users. Routing tables are typically NOT updated in a guaranteed consistent way. Resources may exist in the p2p system but due to segmentation or request aging may not be found. There are no guarantees that a client will receive all parts of resources. Security is weak and billing e.g. almost impossible. And the single machine which typically lives “at the edge of the internet” as described by Andy Oram does not have a stable IP address and may be powered down at any minute. But despite of all these theoretical problems the sheer numbers of participants in P2P systems makes it likely that there is enough compute power, storage and probably even content available.

So these systems – Kazaa, Edonkey, Emule, Bittorrent just to name a few – have become the bane of the content producing industry. They work so well that they became a threat to content producers which – instead of considering these systems as new channels of distribution – see them as a threat and started legal actions against them.

But peer-to-peer systems need not be restricted to file copying and sharing applications. They can play a vital role in content production or distribution of content as real-time streaming data as well. Even game

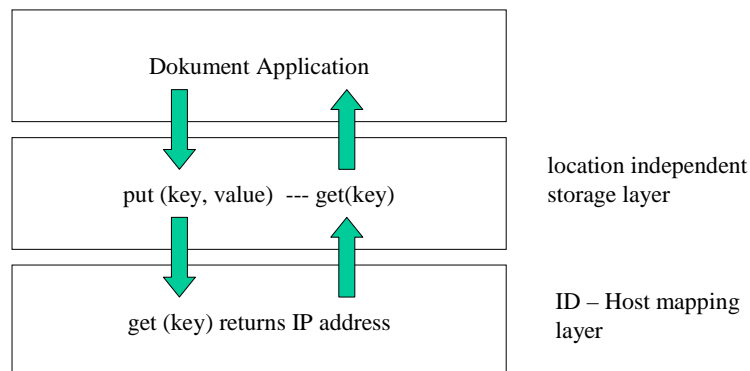
platforms exist which are based on P2P technology and many companies use these “overlay networks” to distribute updates to software, games etc. Let’s take a look at how such a P2P network works and why they are sometimes called “overlay networks”.

Distributed Hashtable Approaches

Many P2P systems use the concept of a distributed hashtable to assign content (documents, media etc.) to machines. This is done through a two layer API. One layer creates a storage layer which takes media and stores them on specific machines. Which machines are used for storage is decided on a lower layer which simply associates keys with machines. This can be done by creating a distance functions between the key of a document (which could be its hash value) and the hash value of an IP address or better a unique name of a peer node.

This sounds straight forward but P2P systems need to solve another problem: Their nodes typically change IP addresses at least every 24 hours which means that the regular way of finding machines using the Domain Name System does not work, at least not out of the box. P2P systems therefore create an “Overlay” network. They assign unique identities (stable) to machines and just assume that IP addresses are only temporary. A clever bootstrapping process then allows new machines to announce their presence and get integrated into the system.

Distributed Hash Tables (DHT)

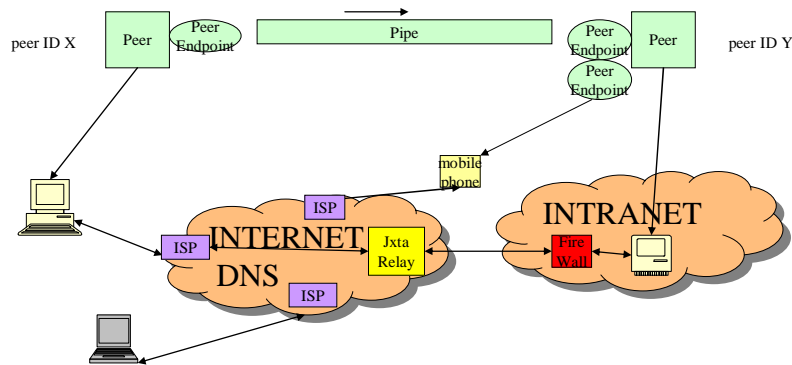


For an overview of different DHT approaches compare CAN, CHORD and e.g. KADEMLIA. Look at how the routing algorithms deal with high rates of peers leaving/entering the network. The advantage of a DHT lies in its simple interface and location independence.

DHT approaches differ vastly in the way they perform, react on changes to the network through nodes coming and going, security and reliability etc. Some p2p systems try to guarantee anonymity and protection from censorship, others try to optimize storage reliability. Some try to account

for optimizations for geography or speed (creating specialized sub-areas within the peer network). Many different communication channels can be used by peer clients and the P2P software therefore tries to create some form of transparent communication between peers – independent of the location and communication abilities of those peers. Because the requirements of p2p systems are the same in many cases frameworks have been developed to provide assistance. A very popular example is the JXTA framework from SUN (www.jxta.org) which provides an extensible software platform for the creation of p2p services and appliatoins. The framework does provide help in the case that clients behind firewalls need to communicate or need to create connections between loosely available partners.

Abstracting away the physical differences



Nodes on the edge use all kinds of identities, naming and addressing modes. They are disconnected frequently. They are behind firewalls with NAT. JXTA puts an abstraction layer above the physical infrastructure that allows programmers to program without worrying about the physical differences in latency etc.

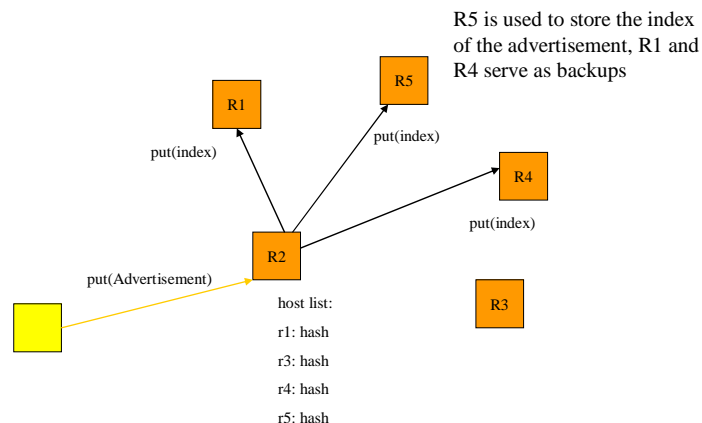
Peers communicate initially by advertising their own existence and features through so called advertisements – xml data – which are sent to so called rendezvous servers. These rendezvous servers a specialized peers which perform administrative task like storing advertisements or communicating with other rendezvous servers. This means that a superstructure is created on top of otherwise equal peers – A pattern that is seen in many distributed systems and that is responsible for more efficient communication and search between participants.

In general the working conditions of p2p systems are significantly less reliable than tightly controlled and administered intranet software. JXTA design reflects those problems and uses e.g. a loosely consistent tree walking algorithm to locate and place content on specific machines. The price to pay is the lack of guarantees that a specific content will be located during a search. Again, this is a pattern frequently found in distributed systems: reducing the service level guarantees makes some applications impossible but allows new types of applications to show up. Those applications would not have been possible under the heavy weight of

existing service level guarantees. A typical example in the web services world would be whether all actions need to be transactional (can be rolled back completely and automatically) or if it is OK to enter a second phase of compensating actions in case one part of a complex transaction did not go through. There are of course different business contracts behind the different approaches.

The next slide shows how content is distributed in a way that makes finding it more robust. Here the content is placed on several hosts which are somehow close to each other (defined by the distance function).

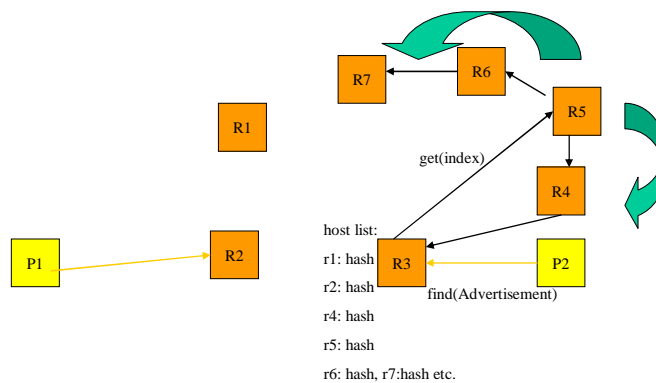
A loosely consistent tree-walker (Store)



The Rendezvous peer R2 calculates the hash of the advertisement, applies the distance function and finds R5 as best storage location for the indexed advertisement. It also stores the content at „nearby“ hosts (hosts which are close to R5 in R2's routing table. On a random base the rendezvous peers exchange routing tables and detect dead hosts. (See: „a loosely consistent DHT Rendezvous Walker, B. Traversat et.al.)

If the content needs to be found and retrieved a tree-walking algorithm is used. The closest node is calculated and the content retrieved from that node. In case that node is unavailable or does not have the content the search algorithm starts walking in both directions from the node and looks for the content on nearby nodes. Hopefully the content will be found somewhere in the neighborhood of the target node.

A loosely consistent tree-walker (Walking)

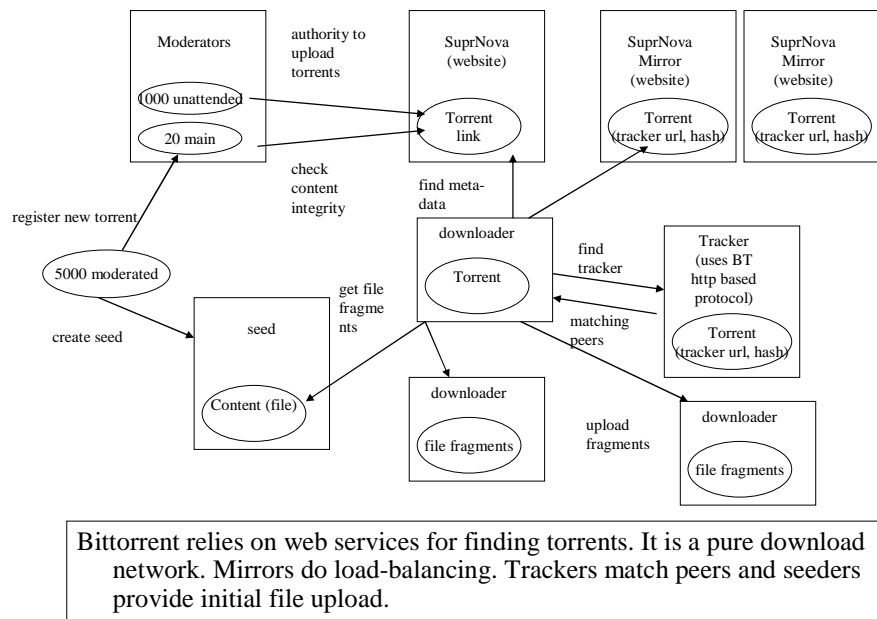


In case of a high churn rate the routing tables have changed a lot. In case a query fails at one host the host will start a tree-walk in both directions (up and down the ID space) and search for the requested content. This allows content lookup even if the rendezvous peer structure changed beyond our initial backup copies.

Bittorrent Example

The bane of the movie industry has one name: Bittorrent. This content distribution network allows fast sharing of rather big content (aka movies) in an efficient way. It uses a typical architecture of meta-data servers (so called trackers) and download servers (the peers). A person willing to share content creates a torrent description and places it on a tracker. Here interested parties can look for content and find the initial provider. Once the peers start downloading from the initial provider the protocol makes sure that machines that downloaded parts of the content are at the same time functioning as download servers for those parts. This allows extremely fast distribution of content. Critical phases are mostly the startup phase (when everybody wants to connect to the initial provider) and the end phase (when most requests have been satisfied and the peers with complete copies lose interest in serving it any more)

Bittorrent Architecture



Special Hierarchies

Anonymity, friends, location, speed, security...

A good introduction to different approaches and to the general concept of anonymity in P2P networks can be found in the thesis of Marc Seeger [Seeg]. There concepts like darknets, mixes etc. are discussed. For media people the concept behind so called brightnets is perhaps the most interesting one as it mixes different public media and distributes the result. At the receiving end the original media can be reconstructed but the bits distributed are no direct artwork and therefore not protected by copyright laws – at least this is how the proponents of brightnets argue.

Idea: friends join a distributed streaming platform and organize “evenings”. Each evening a different person of the group supplies the music which is streamed to the distribution network and finally to the friends. Is this a violation of copy right? The friends could just as well come together physically in ones living room and nobody could say anything about providing the music in this case. An architecture like the media grid (see below) could be used for p2p distribution of the streaming content.

Compute Grids

Within the lifecycle of digital media the point of creation and the point of distribution both require huge computing power – fortunately not permanently. The need for this computing power comes and goes, depending e.g. on the 3-D rendering process of a computer animation or the demand for specific media on various devices and locations.

Unfortunately getting the right amount of compute power when it is needed and only then (meaning we don't want to pay for excess compute power when we don't need it) is a rather tough problem. Energy providers have built a huge distributed computing and energy providing infrastructure to deal exactly with this type of problem – e.g. during the breaks of world championship competitions when literally billions of people suddenly use energy by cooking something.

Compute Grids are supposed to solve exactly this type of problem by providing on demand compute power when it is needed. Owners of data centers on the other hand can sell their excess resources which would otherwise just sit around and idle.

It is not easy to distinguish compute grids from peer networks as in both cases machines can play producer and consumer roles for information. The most important difference seems to be the Quality of Service that is provided, i.e. the promises that are made for users of those architectures. Compute GRIDS typically are well administered conglomerations of hard and software which provide a service for money. This implies several layers of software for administrative purposes and a strong security foundation. Billing is e.g. a concept that is not found in most peer-to-peer systems. Security based on reputation systems is on the other hand a typical feature of peer networks.

Virtual Organization Diagram.

Today's GRIDS are based on Web Services Standards for communication and security. A typical platform for GRID computing is the open GLOBUS project. GRIDS try to hide the complex internals and administration from the user who might want to process certain scientific data but who is probably not interested in where this computation really happens – as long as it is safe, fast and not expensive.

For the processing of media this view of a GRID holds true but when it comes to the distribution and delivery aspect of media the internal architecture of a GRID may become more visible. We will see a nice example of this during the discussion of the MediaGrid architecture below.

<<media grid >>.

The idea of compute grids is not new to media processing: Visualization software like 3DSMax is able to use pools of inexpensive hardware for rendering purposes. Agents installed on those machines receive processing requests and perform partial rendering of images. But this simple reversed client/server architecture (many servers, one client) is quite different to what GRIDS can provide. In the first case of simple pools high-speed networking and a controlled intranet environment make issues like security and performance rather easy (besides dealing with complaints from users of those rendering machines about bad performance because of the agents eating too many cycles). A GRID cannot accept a bad QoS for other participants, needs to keep audit data for billing and treat different tasks separately with respect to security.

Event-Driven Application Architectures and Systems

The final topology and communication style presented here is rather new and does not seem to be very relevant for media related processing. It is the event-driven or event-based architecture for distributed systems and it is used especially in upcoming areas of technology.

Applications of event-driven systems

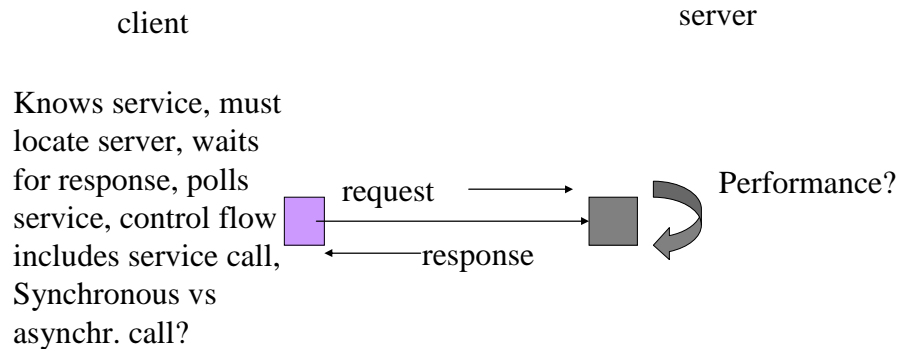
- **ambient intelligence, ubiquitous computing (asynchronous events from sensors)**
- **Information distribution from news producers to consumers (media-grid, bbc, stock brokers etc.)**
- **Monitoring (Systems, networks, intrusions) (complex event detection in realtime)**
- **mobile systems with permanent re-configuration and detection**
- **Enterprise Application Integration with ESB, MOM etc. to avoid programmed point-to-point connectivity and data transformations**

Characteristics:

asynchronous communication, independently evolving systems, dynamic re-configuration, many sources of information, different formats and standards used,

The diagram says it all: Event-driven systems do have a strong focus on asynchronous communication (senders do not wait for responses) that leads to rather independently operating subsystems. The architecture allows the connection between many sources and sinks of information without tying them together. So we can say that the two main points of event-driven systems are the de-coupling between participants and a very easy and powerful way to use concurrent computing power without the typical complexity associated e.g. with multithreaded systems. They promise dynamic reconfiguration in case of changes, adaptation to changes and a high scalability of the applications. And they are able to form data-driven architectures operating in de-coupled pipeline modes. We have already talked about the problems behind a classical client/server communication. Besides performance problems there is another thing that gives us headaches in those architectures: it is the high degree of coupling between components. Components are software entities which operate on nodes and which should be – at least in theory – composable to form new applications and solutions. Fact is that this does rarely happen in practice.

Architectural Cold-Spots in Request/Reply Systems



Control flow encoded in applications. Makes composition of application components very hard. Compare with separation of concerns in EJB. Calling a service becomes a (separate) concern! (see Mühl et.al.)

The reason for the lack of flexibility lies deep within the components and works on several layers, from communication style up to the semantics of the component itself. The list below mentions some of those problems. Please note that by simply calling a service a component ties itself in several ways to the service – an effect that became visible even in the early CORBA architecture which was a service architecture at its core. That was the reason later frameworks like EJB tried to hide the service calls within the framework itself and keep them out of the business logic of the components.

Coupling revisited: the causes

- Components have references to other components
- Components expect things from others (function call pattern) at a certain time
- Components know types of other components
- Components know services exist and when and how to call them
- Components use a call stack to track processing
- Components wait for other components to answer them

Coupling is deeply rooted in the architecture of languages and applications!

The event-based distributed architecture is radically different to synchronous client-server types. Components do not know each other and they do not share data in any way. Not sharing means that once a component works on data those data are local to the component and nobody

else has any access to them. Once done, the component can publish results and other components can start working on those. This is called a data-flow architecture because it is the availability of data itself that controls the processing. Because components do not work concurrently on the same data there is no need for locking or exclusion and the processing becomes simple and reliable.

Event-based architectural style

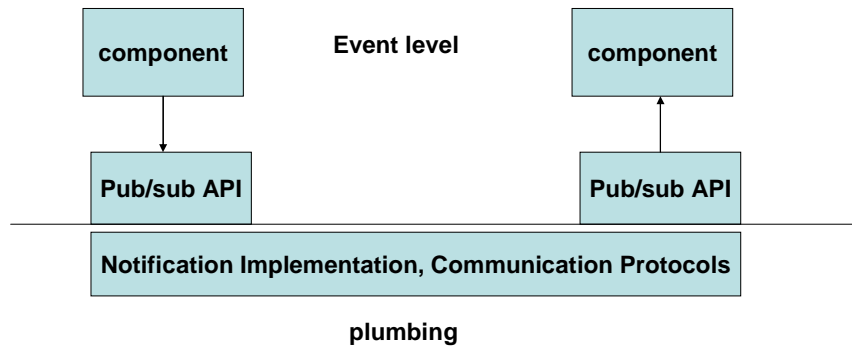
- **Components are designed to work autonomously**
- **Components do not know each other**
- **Components publish/receive events**
- **Components send/receive events asynchronously**
- **Some sort of middleware (bus, mom etc.) mediates the events between components**
- **Due to few mutual assumptions components can be assembled into larger designs**

Sounds a lot like integrated circuits!



It is a violation of event-driven architecture to encode sender/receiver information in messages. This adds coupling between components. And publishing a message with the expectation of getting a kind of response message simply tries to look like independent event processing but actually is simply a form of synchronous request/reply style with strong coupling: the sender of the requests NEEDS the response message which makes it clearly dependent on some other component. Security in those systems is problematic as well. PKI e.g. requires the sender to know the receiver so that messages could be encrypted using the public key of the receiver. But this is clearly a violation of the de-coupling principle. Even digital signatures of messages from senders break this principle. In many event-driven systems administrative overlay networks are then used to provide security e.g. by creating different scopes and connectivity between components for security reasons. But the components themselves are unaware of these restrictions. [Mühl]. How do event-driven systems work? There are many different technologies available, from a simple mash that connects every participant with each other and where every message is routed to every possible receiver to systems that use subscriptions, advertisements and content-based routing and the creation of so called scopes (topic areas) to optimize message flow. Middleware separates application components from the task of distributing and receiving the events.

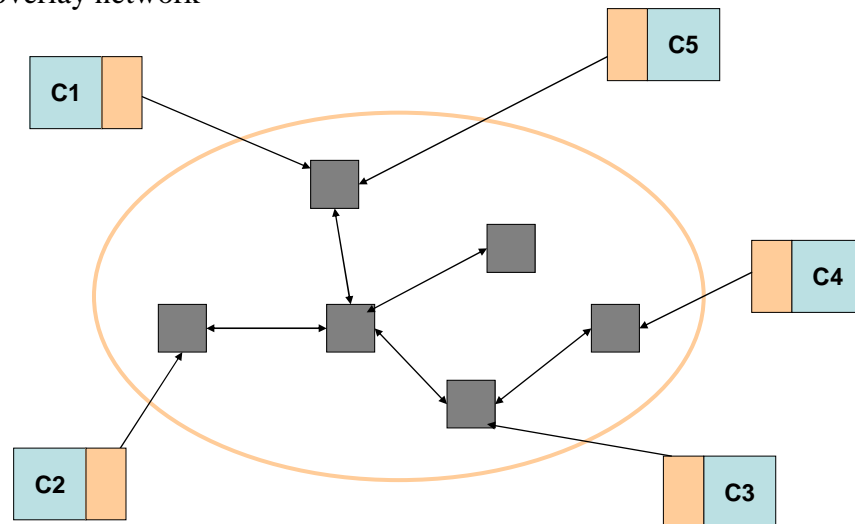
Event-Architecture and Notification Implementation



All combinations are possible: event architecture can rest on a weak, directly connected implementation (e.g. traditional observer implementation in MVC) or request-reply architecture can use true pub/sub notification mechanisms with full de-coupling)

The diagram below shows a middleware that connects several participating nodes. There is middleware logic within each node and optional control logic within the network itself. The core network members all route and filter events to and from the participating client nodes.

Rebecca distributed notification middleware through overlay network



See: Mühl et.al. Pg. 21

Relevant communication types are shown below:

Interaction Models according to Mühl et.al.

		Consumer initiated	Producer initiated
Adressee	Direct	Request/Reply	callback
	Indirect	Anonymous Request/Reply	Event-based

Expecting an immediate „reply“ makes interaction logically synchronous – NOT the fact that the implementation might be done through a synchronous mechanism. This makes an architecture synchronous by implementation (like with naive implementations of the observer pattern).

There is no doubt that event-based distributed systems do have a lot of potential for scalable, flexible and independent computing. But how relevant are they for media processing? The principle of data-flow processing in concurrently working units is e.g. used in graphic engines for shading and texturing. The reason the graphic pipelines in modern cards work so efficiently lies in the simplicity of the data-flow architecture.

Media distribution could become a domain of event-based systems with agents waiting for content to arrive, process (e.g. re-format) and republish the content again.

But the core domain of event-based systems within media could be the information aggregation area. Event-based systems can be used for so called Complex-Event-Processing (CEP). Messages from components are processed and turned into events. These events are then collected and aggregated in a CEP system and new, higher-level events are generated. These events can signify problems within the processing of an infrastructure. Or they could represent content analysis which was performed in real-time. Many data analysis systems work in an offline/after-the-fact mode: data warehouses collect data and then start an analytical process. Search engines collect data and create indexes and later run queries against the data collections. But CEP systems can detect things in real-time and also react on those in real-time.

For more on CEP see D. Luckham [Luck] or try the java CEP framework jesper.

Distributed Communication and Agreement Protocols

Wikipedia:

Gossip protocols are just one class among many classes of networking protocols. See also [virtual synchrony](#), distributed [state machines](#), [Paxos algorithm](#), database [transactions](#). Each class contains tens or even hundreds of protocols, differing in their details and performance properties but similar at the level of the guarantees offered to users.

- group communication
- transactions
- agreement and consensus

Reliability, Availability, Scalability, Performance (RASP)

Usually you only hear about these terms (sometimes called “-ilities”) when things go wrong. And they do go wrong on a daily base as news about crashed sites and services demonstrate. So turned the announcement of the new European digital library (www.europeana.org) into a disaster because the service was unable to cope with the flash crowd gathering after the news published the announcement. Web-shops are overrun and crash because a new product creates a high demand like the new Blackberry did. [HeiseNews119307].

Its not only the web applications and services which have a RASP problem: On 26 June 2008, right in the middle of a Euro 2008 football game most TV stations lost the signal due to a power failure at the IBC center at Vienna [Telegraph]. Broadcasters had paid around 800 Million Euro for the rights to UEFA and they were not pleased about the interruption that lasted up to 18 minutes in some countries. It looks like a failure in the uninterruptible power supply caused a reboot of the sending equipment after power was lost for milliseconds only. There was only one signal for all TV broadcast stations – a classic “Single-Point-Of-Failure (SPOF)” [ViennaOnline]. And UEFA will have to pay damages.

And we have to remember the core quality of all SOA, Web2.0, MashUps, Community Services and networks: They have to be up and running and available at all times to be called SOA, Web2.0 etc.

The RAS terms all mean some degradation of the quality of service promised. But this degradation need not be so spectacular as in server crashes due to flash crowds. Degradation can come very slowly and still deadly for a website: visitors still come to the site but they leave earlier than before. Why? Perhaps because the site got slower and slower over time and it is just no longer fun to use it to communicate with friends. This in turn means that an important aspect of the RAS terms lies in constant monitoring and reporting of system and applications status, from outside as well as inside.

Lets define the terms a little bit more detailed without becoming religious because they are of course tightly connected with each other and other aspects of system design like the overall architecture.

Resilience and Dependability

When we look at definitions of availability in the literature (e.g. the nice overview given by Morrill et.al.) we notice certain core elements. The Definitions are nowadays mostly based on ITIL terms [ITIL3] and they favor a rather integrated look at RAS. Resilience means business resilience and subsumes IT resilience which in turn subsumes IT infrastructure etc. ([Morrill] pg. 495.). The whole thinking about RAS has become very much top-down: Business requirements and a design phase concentrating on RAS issues guarantee e.g. continuous availability of the solutions.

As noted, any design for availability is not complete without consideration of how the system will be managed to achieve the necessary availability characteristics. In the past, availability management has often been an afterthought: organizations would determine how they were going to measure systems availability once the design was complete and the solution implemented. Today, with ITIL Version 3, the availability management process has moved from the service delivery phase to the service design phase. Early on, organizations determine how they will measure and control availability, capacity, and continuity management processes. ([Morrill] pg. 499)

While certainly a good approach it is in rather stark contrast to the way some of the ultra-large scale sites we will discuss below have been built. This also shows in the statement that “Mixing and matching components in an IT infrastructure can result in increased opportunities for failure.” ([Morrill] pg. 499). Most of our sites will be rather wild mixtures of technologies.

But this perspective also includes the conviction that applications need to be aware of availability techniques within the infrastructure to be able to use e.g. monitoring features, checkpointing or failure detection. And this might be true for all larger sites.

What we must take with us from the definitions of RAS is that availability today is a multi-dimensional feature. It comprises the ability to change the quality (or kind) of services rapidly to support business resilience. It also means to adjust to changes in use by quantitatively scaling up or down (do not forget down scaling to save costs). And it means being continuously available during various kinds of failure conditions on all kinds of scale and scope. Finally, the permanent monitoring of the integrity of the system despite changes for resilience is part of availability as well. Below we will discuss separate aspects of this overall notion of resilience but we keep in mind that this is just an artificial separation for analysis purposes.

Scalability

Why is scalability so hard? Because scalability cannot be an afterthought. It requires applications and platforms to be designed with scaling in mind, such that adding resources actually results in improving the performance or that if redundancy is introduced the system performance is not adversely affected. Many algorithms that perform reasonably well under low load and small datasets can explode in cost if either requests rates increase, the dataset grows or the number of nodes in the distributed system increases.

A second problem area is that growing a system through scale-out generally results in a system that has to come to terms with heterogeneity. Resources in the system increase in diversity as next generations of hardware come on line, as bigger or more powerful resources become more cost-effective or when some resources are placed further apart. Heterogeneity means that some nodes will be able to process faster or store more data than other nodes in a system and algorithms that rely on

uniformity either break down under these conditions or underutilize the newer resources. (Werner Vogels in "A Word on Scalability", http://www.allthingsdistributed.com/2006/03/a_word_on_scalability.html)

There are a number of problems that can be interpreted as scalability problems: A service shows sluggish behavior by responding very slowly to requests. Later the service might not answer at all or may not even accept a request or be visible at all. This gives us at least one end of scalability issues: complete loss of availability. We will discuss availability below and for now concentrate on scalability.

But what is scalability and when is a problem a scalability problem? Just responding very slowly need not be a scalability problem at all. It can be caused by a disk slowly disintegrating, by a network device becoming instable etc. We will call it a first order scalability problem (or just a scalability problem) if it is caused by an increase in the number of requests directed towards a system or – in case of constant requests – by a decrease in the number or size of resources of a system needed to process requests. We will call it a second order scalability problem if the problem is caused by the scalability architecture or mechanisms themselves: When the measures taken to scale a system need to be extended. This happens when a distributed cache needs more machines or when additional shards are needed to store user data. Frequently in those cases it turns out that the scalability mechanism used originally now poses an obstacle for further extension e.g. because the algorithm used to distribute cached data across machines would invalidate all keys when a new machine is added. Or when user distribution across database shards turns out to be ineffective but driven by a static algorithm that does not allow arbitrary distributions. Actually the dying disk example from above can be a second order scalability problem because it raises the problem of rebuild time needed to get the system fully functional again. Raid arrays e.g. are notoriously slow to rebuild a broken disk. Originally intended to provide performance and availability the array can now turn into a scalability problem itself.

(First order) Scalability has two very different aspects. The first one describes the ability of a running system to scale according to requests received or more general to an increase of load. The goal is to keep the Quality-of-Service either at the current level or to let it degenerate only slightly and in a controlled fashion. In this case the ability to scale must be already present in the running instance of the system. An analogy to the human body comes to mind. If I need to run faster my body reacts with an increased level of adrenaline, a higher heartbeat and so on. My body scales to the increased load and this ability is part of the core adaptability of the human body. But there is a downside associated with this ability: Both, the running system as well as the body need to be prepared for increasing load. This can mean that in both systems some parts have been running idle while the load was low. In terms of computing hardware it is possible that a gigabit network line has been installed at high costs, parallel running servers have been bought that run at 5% load each, more software licenses have been bought and so on. And all for only one reason: to be able to scale whenever it is needed.

The advance costs of scalability are especially dreadful in case of the famous flash crowds which hit sites that suddenly got popular (e.g by being “slashdotted” or by just being announced). In this case the costs of scalability need to be spent for a load that may only happen once in the lifetime of a site. Clearly this is not cost effective. We will take a look at edge caching infrastructures later that can be rented and allow a better distribution of content by using a separate infrastructure temporarily.

The first aspect of scalability is necessarily limited therefore because nobody spends huge amounts of money just in case some sudden increase in load or requests might happen.

I believe the second aspect of scalability is much more important for distributed systems: it is the potential of the architecture to be made scalable. You might say: but isn't every architecture scalable by adding either hardware or software or both? The sad truth is: no, not if it hasn't been built to scale. To understand this statement we need to look at two different ways architectures can scale: horizontally and vertically.

Database servers are a typical case of vertical scalable systems: the database runs on a big server machine, initially together with other services. Soon those services are removed to increase CPU and IO capability for the database. Later more CPUs and RAM are added on this machine until the final upgrade level has been reached. Now vertical scalability is at its end and the next step would be to add another database server machine. But suddenly we realize that in this case we would end with two different databases and not two servers working on one and the same data store. We cannot scale horizontally which is by adding more machines.

Sometimes very bad things happen and we cannot even scale vertically. Let's say we can run one application instance on a server machine only. The software does not allow multiple installations. It turns out that the software only uses user level threads, no kernel level threads. User level threads are within one process thread which means all of these user level threads are scheduled using one and the same process thread. We can add tons of additional CPUs in that case without the application being able to use any of those new CPUs.

More and more the solution to problems with vertical scalability is by using virtualization technology that is able to create separate virtual rooms for software on one machine. But it does not help us with the database problem..

Frequently a much nicer solution is using horizontal scalability by adding more machines. But this has some subtle consequences as well. Ideally it would not matter which server receives which request. As long as all requests are stateless this is no problem. But this requirement is clearly an architecture and design issue. If the requests are not stateless we need to make sure that the current state of the communication between client and servers is stored somewhere and all the servers can get to it with good performance. Or we make sure that requests from one client always end up on one and the same server. This requires so called sticky sessions and appropriate load-balancing equipment. The first solution with distributed

session state btw. Is an excellent choice for the problems of the next section: when your application needs to be available at all times and even if single instances of servers crash.

But also within the application software there is no end to scalability problems. Many applications need to protect shared data from access by multiple threads. This is done by using a lock or monitor on the respective object which realizes some form of exclusive access for one thread only. But if those locks are put at the wrong places (e.g. too high in the software architecture) a large part of the request path becomes completely serialized. In other words: while one thread is busy doing something with the locked data structures all the other threads have to wait. No number of CPUs added to this machine will show any improvement in this case. The new threads will simply also wait for the same lock held by one of them.

Besides fixing those bottlenecks in the software of the application the answer to both scalability and availability requirements today is to build a cluster of machines. We will discuss this approach in the section on availability.

But even with cluster technology becoming a household item for large web sites there is more to scalability and it again is associated with architecture. But this time it can be the architecture of the solution itself, e.g. the way a game is designed, that plays a major role in the overall scalability of the application. The magic word here is “partitioning” and it means the way application elements are designed to support parallelization or distribution across machines. And this is finally a question of how granular objects can be associated with processing. We will learn the trade-offs between adding CPU and paying the price for increased communication overhead in the chapter on Massively Multi-Player Online Games (MMOGs).

And a final word of warning: we have already discussed the limiting effect of scale on algorithms in the case of distributed transactions. Scale effects work this way almost always. There comes a size where most proven technologies and off-the-shelf solutions just do not work anymore and require special solutions. We will discuss some of those large scale site architectures later.

For a really extreme form of scalability and how it affects design – or should we say “re-define” design – take a look at Richard Gabriel’s paper “Design beyond human abilities” [Gabriel]. There he talks about systems that have grown for twenty or more years and which are so large that they can only be adjusted and extended, not remade from scratch.

Heterogeneity is natural in those systems.

A nice comparison of scale-up and scale-out techniques can be found in [Maged et.al.] “Scale-up x Scale-out: A Case Study using Nutch/Lucene”.

Availability

Intuitively availability means taking a close look at all components within your system (system components like hardware boxes as well as networks

and application instances or databases, directories etc. There shallst not be a single point of failure within your complete system to deserve the attribute “highly-available” which we will from now on simply call “HA”. This in turn means that a load-balancing concept alone is a far cry from being “HA”. It is a necessary concept as it can remove one specific Single-Point-Of-Failure (SPOF) but there are many other SPOFs left. Actually what can be considered a SPOF largely depends on your scope as we will see. (Btw: if you are having trouble understanding options in load-balancing or why you sometimes need to balance on MAC vs. IP level, when to choose a different route back to a client and how to do this – don’t despair: there is a short and beautiful book about “Load Balancing Servers, Firewalls, and Caches” by Chandra Kopparapu and it will explain all this on less than 200 pages [Kopparapu])

The opposite of availability is downtime, either scheduled (planned software upgrades, hardware maintenance, power savings etc.) or unplanned (crash, defect). Unplanned outages are rather rare within the infrastructure and seem to mostly come from application or user error. Availability can therefore be expressed like this:

$$\text{Availability (ratio)} = \frac{\text{agreed upon uptime} - \text{downtime (planned or unplanned)}}{\text{agreed upon uptime}}$$

Contnuous availability does not allow planned downtime

Examples of downtime causing events are shown in the list below:

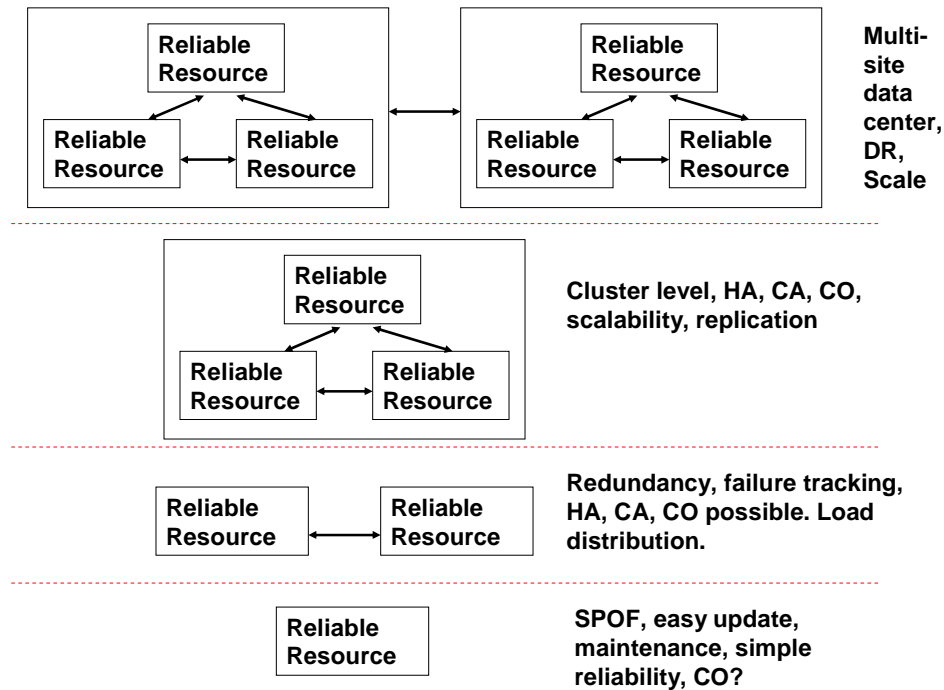
<p>Unplanned Outages</p> <ul style="list-style-type: none"> Physical breakage Design error in hardware or software Environmental events, such as loss of power or cooling Operator or user accident, inexperience, or malice Natural disasters and accidents, such as setting off sprinklers Human-caused disasters, such as terrorist activities <p>Planned Outages</p> <ul style="list-style-type: none"> Planned software or hardware upgrades Preventive or deferred maintenance Governmental or policy regulations

Morrill et.al, Achieving continuous availability of IBM systems infrastructures, IBM Systems Journal Vol. 47, Nr. 4, pg. 496, 2008

Today the answer to HA is usually some form of cluster technology as it is explained in [Yu]. But before you run off to buy the latest cluster from SUN or IBM or even try to assemble one on Linux by yourself you should answer the most important question about availability: what level of availability (understood as uninterrupted service) do you really need? The answer can be in a range from “application can be restarted several times a day and five hours downtime is ok” to “5 minutes scheduled downtime a year with backup datacenters for disaster recovery”. And the costs will therefore range between a few thousand dollar and many, many millions for worldwide distributed data-centers.

We have mentioned above that a core quality of SOA and Web2.0 sites is within the extreme availability that they provide. Continuous availability (CA) is much more than just HA because it reduces downtime to zero. And that means continuous operations (CO) as well – the ability to upgrade software without restart is an example. And finally 11th September 2001 has brought disaster recovery (DR) back into peoples mind. Geographically distributed data centers mean avoiding SPOFs on a very large scale. Let’s put the various concepts of availability into a diagram which shows the various dimensions involved (following the terminology developed in [Morrill]).

The diagram of availability scopes starts with basic reliability guaranteed by a high MTBF of single systems. Do not underestimate the role of simple reliability. Individual high reliability is still extremely important in the light of FLP and the impossibility of consensus in asynchronous systems. It is true as well for network connections across multiple nodes. Without individual reliability many of our distributed algorithms will not work properly anymore, e.g. they will not come to a consensus in reasonable time.



High availability (HA) starts with redundancy of nodes and a typical example can be found in load balancing sections of an architecture. But even on this level the multi-dimensional nature of availability shows: We can call it load balancing or high availability or both, depending on where our focus is. And with this first duplication of infrastructure we inherit the basic problems of distributed systems as well as its promises for better throughput or availability. We will take a closer look at redundancy and load balancing later when we discuss Theo Schlossnagel's ideas for availability and just mention here that even for such a simple architecture we will have a lot of questions to answer: will there be failover and what does it really mean? How will failures be detected? Do we need to duplicate all nodes? Do we use passive backups with switch-over capabilities or all-active architectures? How do we handle replicated data between nodes?

Before we look at clustering as a solution for HA we need to clarify two subtle points in distributed systems. The first point is about the role of redundancy in distributed systems. Even after many years of distributed systems and the ubiquity of multi-tier applications in intranets and internets few people seem to understand that distributing computing across several nodes, components etc. makes the whole processing much more unreliable, insecure and especially brittle. The likelihood of one of those nodes or components failing is much bigger and the only answer to this problem is called redundancy through replicas. Actually there are more problems behind a failing node even in case of redundant equipment: you need to detect the failure first which again is much more difficult in distributed systems than in a big local installation on one machine (see below: failure detection). But let's first concentrate on redundancy. Many companies were shocked when they had to learn this the hard way by ever increasing operating and maintenance costs of their distributed applications. Server farms with hundreds and thousands of servers pile up

huge costs for energy, cooling, software, monitoring and maintenance. And still, you will only get to the potential benefits of distributed systems if you accept the costs of redundant systems. You can build a distributed system without redundancy but it will expose all kinds of RAS problems due to overload, component failures etc. A typical case where redundancy is likely to be violated in architectures is the role of the data-store. In many applications there will be just one instance of a central database and it is both a SPOF and a bottleneck for performance. And last but not least we need to realize that introducing redundancy to fight distribution problems means at the same time to introduce more distribution problems between redundant components. We will discuss advanced consensus algorithms to secure common state between replicas later – and learn about an opportunity to save considerable costs.

The second subtle point is about failure detection. Redundant equipment won't help your system in case of failures if you cannot detect which nodes or components are at fault and also when they start showing problems. The good old fail-stop model assumes that a node that shows a problem simply fails at once and completely and on top of this that the other participants in the distributed system can detect this fact immediately. This is an extremely unrealistic assumption. The typical case is that an application receives a timeout error from one of the lower network or middleware layers and is then free to assume one of several things: a network failure (perhaps partial, perhaps total, perhaps persistent or temporary), a node failure (the own node, the partner node, the operating systems involved, the middleware layers involved, all of it either permanently or temporarily), a server application failure (server process is down, perhaps permanently, perhaps temporarily).

The next step after simple redundancy is clustering. Here the dimension of throughput enhancement and performance are much more clear and we are typically talking about business solutions which need continuous availability (CA). Monitoring with automatic restart of processes or machines is certainly a requirement as is the ability to update code for reasons of bug fixing or business change. A core feature of those clusters is the virtual IP concept which means that the whole cluster of machines will look like a single entity to outside clients and failures within the cluster will be transparently masked by the infrastructure. The most advanced examples of this technology is probably represented by the IBM parallel Sysplex architecture with its various options for scalability and availability across distances.

Caching is of core importance within such clusters and we will look a products like memcached. Also on the level of clusters database partitioning and replication becomes a requirement and we will discuss several solutions for this problem.

We have said that availability is a question of scope. One cluster serving a site to the whole world might both be a throughput problem as well as a disaster recovery problem. Soon the need for more data centers will show up and create problems with respect to replication of data. How do we

keep the replicas in sync? How do we guarantee that users will get the closest (meaning fastest) server access? Routing requests to and between geographically distributed data centers is part of our section on content delivery networks.

And the next important question is about the exact quality of service that is hidden behind pure “availability”. In other words: how transparent for the user is the implementation of HA? Here the answer can be in a range from “after a while the user will realize that his user agent does not show any progress and will deduce that the service might be down. She will then try to locate a different instance of our service running somewhere in the world, connect and login again. Then she has to redo all the things she has already done in the crashed session because those data were lost when the session crashed. When a service crashes the user is transparently routed to a different instance of the service. All session data were replicated and the service will simply continue where the old one stopped. The user did not lose any data from the old session.”

Clearly “availability” has a different quality in both cases. It depends on your business model which quality you will have to provide and hopefully you can afford it too. The second case of transparent switching to a new service is called “transparent fail-over” and has substantial consequences for your architecture (yes, it needs to be designed this way from the beginning) and your wallet.

More reasonable assumptions include nodes that show intermittent failures and recovery, leading to duplicate services or actions in case backup systems were already active because a more permanent failure was assumed. There are algorithms to deal with these cases – so called virtual synchrony and group communication technologies which try to organize a consistent view of an operating group in a very short time of reconfiguration [Birman] but those algorithms are rarely used in regular distributed applications as they require special middleware. Birman correctly points out that in many cases the concepts of availability by redundancy and failure detection can be simulated with regular distributed system technology, e.g. by using wrappers in front of SPOF components.

The worst case assumptions of failure modes includes completely sporadic operation of nodes which finally leads to a state where a consistent view of a group of communicating nodes is no longer possible. If we add bad intent as a specific failure we end up with so called “Byzantine failure models” where subgroups of nodes try to actively disrupt the building of consensus within the system.

Werner Vogels and the CAP Theory of consistency, availability and network partitions. Eventually consistent data. What are the implications for data (data with TTL, probability factor?) Amazon's Dynamo makes these options explicit (against transparency).

Read replication (slaves) and consistency: problem. Better with memcaches? But what if single source memcached data are overrun?

Modeling availability with failure tree models will be part of our modeling chapter.

Concepts and Replication Topologies

High-Availability can be divided into application availability (runtime) or data availability [LSHMLBP]. Only application availability of course knows the difference between stateless and stateful architectures: stateless applications can be made highly available rather easily: Just run several instances of these applications! The problems lie in routing clients to a working instance of such an application and track existing instances to make sure that enough are available.

Once applications hold state the problems start. In order to move processing to a different instance the state must be available to the new instance. Various ways have been found to transport state over: state on disk storage, state in a database, state in shared memory, state replicated over networks etc. (<<how does virtualization today change state management e.g. network state, memory etc.??>>

The way application handled state has always had a big influence on performance and failover capabilities and we will take a close look at how our example site architectures deal with this problem. Do they use “sticky” sessions? Where do they hold state? J2EE applications use replicated stateful session beans to hold client session state across machines and use an external database to serialize requests. [Lumpp] et.al. page 609.

Communication state is also critical for modern multi-threaded applications: requests from one client need to be serialized, e.g by using transactions. No amount of CPUs and threads allows us to process these requests in parallel because then inconsistencies would materialize.

In case of a crashed server, how is a new application attached to the current state? There are a number of options available:

Associating a new instance with current state during failover

-Cold standby (server and application need to be started when primary hardware dies

-Warm standby (failover server is already running but failover application needs to be started first. Both share one SAN e.g.)

-Hot standby (both failover server and application are running but application acts as a secondary only – i.e. does not control requests. Data is possibly replicated)

-Active-active configuration (both servers and apps are running and processing requests. Needs coordination between apps in case serialization of requests is needed. Load can be shared but room must be left for one machine to take over the load from the other. Every application holds its own data which make data replication a requirement as well).

From: Chiterow et.al, Combining high availability and disaster recovery solutions for critical IT environments, IBM Systems Journal 47, Nr. 4/2008

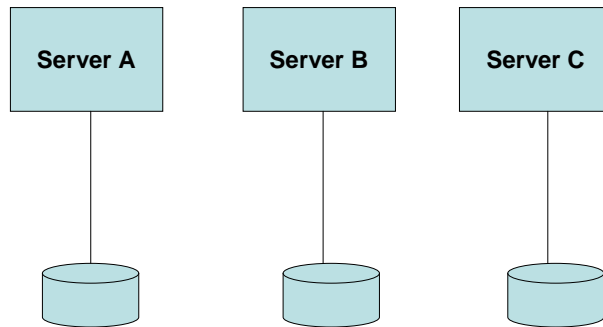
Obviously there are big differences between those approaches with respect to failover time and visibility to clients due to delays. And at least in the case of cold standby an external arbiter is required who decides that the primary is down, starts the backup and routes all requests to the new instance. All the other configurations can be driven with external arbiters as well but could also use some form of group communication protocol to decide by themselves who is going to run the show. Financially the differences are probably not so big as in any case the backup machine needs to be able to take the same load as the primary. The only exception could be made in case of dynamically increasing capacity e.g. due to additional CPUs made available as is done by IBM mainframe systems. Here an active-active configuration could run with 50% mips on both machines which are changed to 100% mips in case of failover. Midrange systems usually do not have this capability and you will be charged for all the CPUs built in independently of the current use.

<<clarify the concept of lock holding time during failover!!>>

A typical high-availability configuration today is called a cluster. A cluster is a number of nodes who work together and present themselves to the outside world as one logical machine. In other words: clients should not realize that they are dealing with a number of nodes instead of just one but they should be able to get the benefit of better availability and scalability.

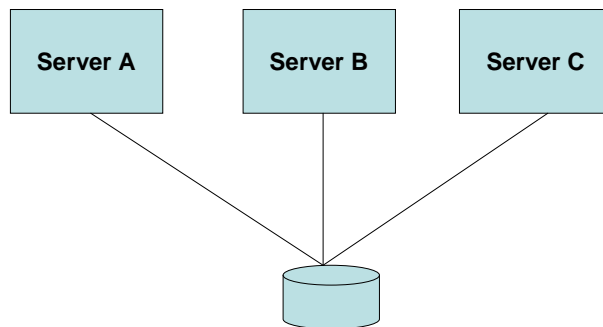
An important distinction in cluster solutions is between shared-nothing clusters and shared data clusters [Lumpp] et.al. page 610ff. A shared nothing cluster partitions its data across server machines.

Shared Nothing Cluster



While this is a typical architecture of ultra-large scale sites as we will see shortly, without additional redundancy built into the architecture it leads to very poor availability. If one server dies a whole data partition will be unavailable. A better architecture is provided by shared data clusters as shown below:

Shared Data Cluster



Here every server can access all data and it does not matter when one server does not function. Of course the storage should not be designed as a single point of failure as well.

Can a cluster span across different locations? The answer is yes, within reason. A very popular form of clustering according to Lampp et.al. is the stretched cluster which works across locations. In a stretched cluster it is assumed that there is no difference with respect to nodes. All nodes can be reached equally fast and with the same reliability. This is of course only true within limits once we span the cluster across different locations. But it is a cluster form that is easy to administrate. Once the distance between locations becomes an issue due to latency and network failures or bandwidth we need to go for a global cluster and by doing so enter the area of disaster recovery which we will discuss below. A global cluster has one primary and one secondary cluster and a special management component decides which cluster does processing of requests and which one is the backup.

Hot standby already requires some form of data replication. Several solutions exist which work on different levels: Operating System replication via IP (e.g. Linux DRBD), disk/storage system (block) level replication (intelligent storage subsystems performing the replication), DB Level replication (commands or data are sent to the replica), application level replication. An important question about replication mechanisms is about the level of consistency they provide. It is usually either block consistency (possibly across volumes) or application consistency. Using this classification on the above technologies it turns out that operating system replication via IP and disk/storage replication offer only block level consistency. The atomic unit of work is basically a block of data, much like or exactly like a disk block. The sequence of block writes will usually be respected (in a fbcast like manor), even across volumes which are frozen/paused in that case. This way so called “consistency groups” are created. What these methods cannot provide is an application unit of work consistency because they do not know which operations form one atomic, all-or-nothing group of writes. This is only known at the application or DB level.

We have just described the consistency aspect of replication. There is another aspect in replication and it regards the atomicity of replication: Either have both primary and replica updated or none of them. This is an extremely important feature and depends on the replication protocol used. A synchronous replication protocol will guarantee the atomicity of replication because it always waits for the acknowledgement of the replica as well. It will not allow a case where the primary got updated but the replica didn't due to a crash or network problem. Or vice versa. And it pays the price in round-trips needed to achieve this. Usually there are two roundtrips necessary at least. And due to this reason there are distance limits for synchronous replication, currently around 300 kilometers between primary and replica.

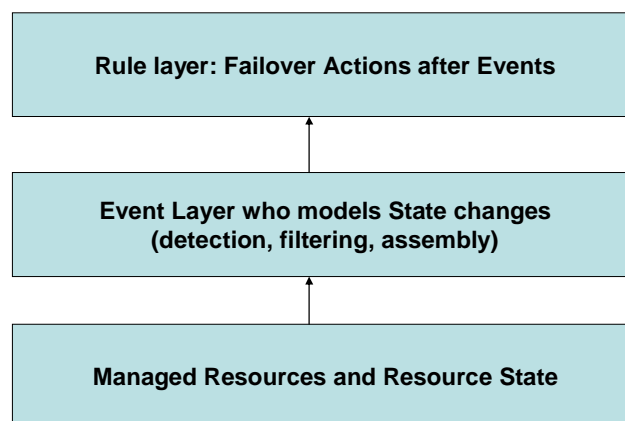
Asynchronous replication does not need to wait for acknowledgements and allows both higher throughput and longer distances. The price is paid in a potential loss of data resulting in an inconsistency between primary storage and replica. And in case of a failover this can result in wrong business data or processes.

Before we tackle the problem of disaster recovery we need to talk about one very important and difficult aspect of HA clusters: The question of when and how to fail over. We have said that in a HA solution failover needs to be automatic. But how is this done? Via scripts? According to the authors the way this is done today is via a three level correlation engine as it is used e.g. by Tivoli software from IBM.

A description of such an engine can be found in Stojanovic et.al., The role of ontologies in autonomic computing systems. The diagram below shows the architecture of a correlation engine:

Automation: correlation engine diagram

Correlation Engine Architecture



Stojanovic et.al., The role of ontologies in autonomic computing systems.

How close is this concept to Complex-Event-Processing languages and architectures?

The concept of high or continuously available systems (HA, CA) has been extended with the concept of disaster recovery (DR) over the last decade. Actually DR has always been an important concept in the largest of financial companies. But due to the growing importance of internet services and presences the fear of disaster is now present in many large websites.

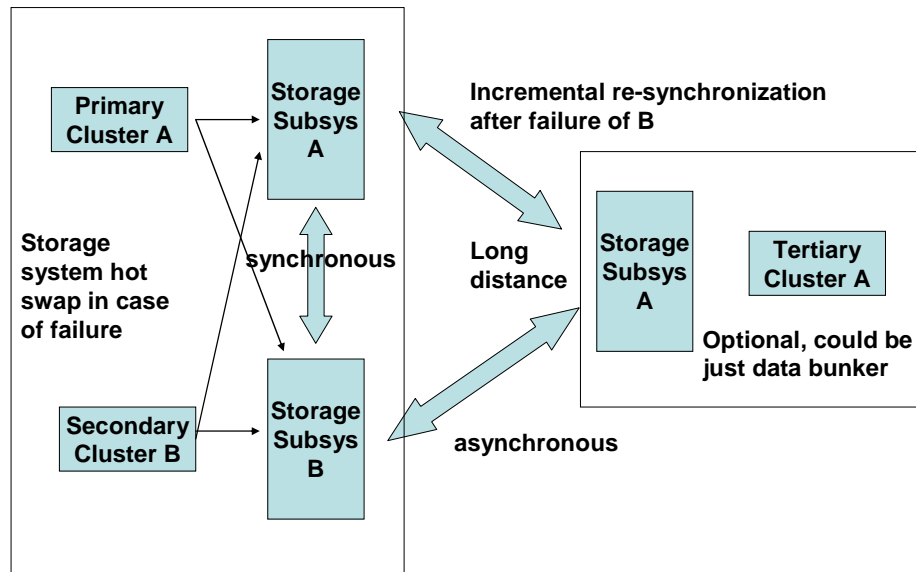
Let's start with some definitions of HA and DR and the differences between them, taken from Chiterow's et.al. paper on combinations of HA and DR technologies for critical environments [CBCS]. The following table presents the main properties and differences according to the authors:

High-Availability vs. Disaster Recovery

<ul style="list-style-type: none"> -Single component failure assumption -Local infrastructure -No data loss allowed -Automatic failover -Synchronous replication mechanisms used -Sometimes co-located with failover infrastructure -Short distance to failover infrastructure 	<ul style="list-style-type: none"> -Multi-component or complete site destruction assumes -Long distance infrastructure -Some data loss possible -Human decision to use backup facility due to costs -Replication mostly asynchronous -Share nothing between sites (net, power, computing...) -Long distance to failover infrastructure (several hundred kilometers)
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

With DR we are obviously talking multi-site data centers, geographically distributed data centers possibly on different continents. There can be many reasons for such architectures: performance, closeness to customers etc. but as Clitherow et.al. mention frequently it is because of regulatory requirements (e.g. that there need to be x miles between primary and secondary site) that a multi-site configuration is chosen. A 3-copy architecture seems to be a rather popular choice in those cases and here we are discussing the architectures described in [Clitherow] et.al. The role of the third site can be just as a data bunker with no processing facility attached. It could take days to get processing up on the third site or there could be a complete hot standby processing facility in place just waiting to take over. Due to the asynchronous communication protocols used between the primary sites and the tertiary site there is usually no active-active model used for the third site. Some ultra-large scale sites solve the problem by using the third sites actively but only for read requests while all changes are routed to a master cluster (or a two site active-active cluster located close to each other).

3-copy Disaster Recovery Solution



After: Clitherow et.al.

The failure model in disaster recovery with a 3-copy solution is usually like this: no data loss if either Cluster A or B fail. If both fail there can be some data loss between the two main clusters and the tertiary site due to asynchronous replication used.

Failure Modes and Detection

[Caffrey] J.M. Caffrey, The resilience challenge presented by soft failure incidents,

[Google] Chubby/Paxos Implementation paper

The role of ontologies in autonomous systems

Selfman.org

Availability is based on redundancy. Redundancy is based of failover – the ability to move a request to a new processing or data infrastructure, possibly without the client noticing the problem. We will discuss failover in more detail using J2EE clustering as an example later. For now we will concentrate on one essential pre-requisite for failover: the ability to detect an error.

And this is where all our efforts to achieve availability through avoiding single points of failure and by replicating as much as possible turn against us. Techniques to achieve fault tolerant behavior tend to mask errors – sometimes over a longer period of time until it is too late to use preventive measures.

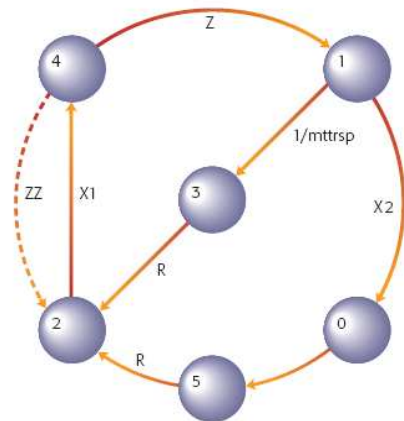
A beautiful example for this effect has been described by the Google engineers Tushar Chandra, Robert Griesemer and Joshua Redstone in their paper “Paxos Made Live - An Engineering Perspective” [CGR]. It describes the use of the Paxos consensus algorithm (we will talk about it later when we deal with consensus protocols for replication) to implement a replicated, fault-tolerant database based on a distributed log system. The database is then

used e.g. to implement large-scale distributed locking. The protocol needs to make sure that all replicas contain the same entries. The system is used to implement the Chubby distributed event mechanism further described in [Burrows]

The paper by Chandra et.al. is especially important from an engineering point of view. It describes the effort needed to transform an academic algorithm (Paxos) into a fault-tolerant and correct working implementation. The team noticed certain deficiencies in the development of distributed systems, notable in the area of testing and correctness. They developed advanced failure injection techniques and implemented injection points within their protocol which led to the discovery of several problems and bugs. And they made the following experience:

In closing we point out a challenge that we faced in testing our system for which we have no systematic solution. By their very nature, fault-tolerant systems try to mask problems. Thus they can mask bugs or configuration problems while insidiously lowering their own fault-tolerance. For example, we have observed the following scenario. We once started a system with five replicas, but misspelled the name of one of the replicas in the initial group. The system appeared to run correctly as the four correctly configured replicas were able to make progress. Further, the fifth replica continuously ran in catch-up mode and therefore appeared to run correctly as well. However in this configuration the system only tolerates one faulty replica instead of the expected two. We now have processes in place to detect this particular type of problem. We have no way of knowing if there are other bugs/misconfigurations that are masked by fault-tolerance. [CGR] page 12

So the 2/5 availability system had secretly turned into a 1/4 system. What do we learn from this experience? Without state (or history) we cannot detect this error because catch-up is a legal phase within the state model of the protocol. The state model with transitions and their respective likelihood is another requirement. The modeling can be done with Markov models and associated probabilities for transitions. The diagram below shows the Markov model for blade-processor CPU plane, taken from Smith et.al, and their availability analysis of blade server systems [STTA].



Transition rate	Memory	Processor
X1	2/mttfmem	2/mttfcpu
X2	1/mttfmem	1/mttfcpu
R	1/mttfmem	1/mttfcpu
Z	1/mttbootl	(1-cpt)/mttbootl
ZZ	n/a	cpt/mttbootl

Figure 4
Markov model for BladeCenter memory and processor subsystems

Steady-State availability of bladecenter CPU and memory subsystems:

$$A_{processor} = \frac{((mttfcpu)/(2 \times mttbootl + mttfcpu + 2 \times (1 - cpt) \times (mttrcpu + mttrsp))) + ((2 \times (1 - cpt) \times mttfcpu \times mttrsp) \div ((mttfcpu + mttrsp) \times (2 \times mttbootl + mttfcpu + 2 \times (1 - cpt) \times (mttrcpu + mttrsp))))}{1}$$

Smith et.al. Page 627

For us two transitions in this diagram are important: the transition X1 leads over to an error state with associated reboot. If a hard error is found within the failing CPU the transition Z is taken which leads to a stable one CPU server. If the CPU problem turned out to be spurious the reboot will transition via ZZ into the old state of two CPUs working correctly. Z and ZZ have associated probabilities but are legal transitions.

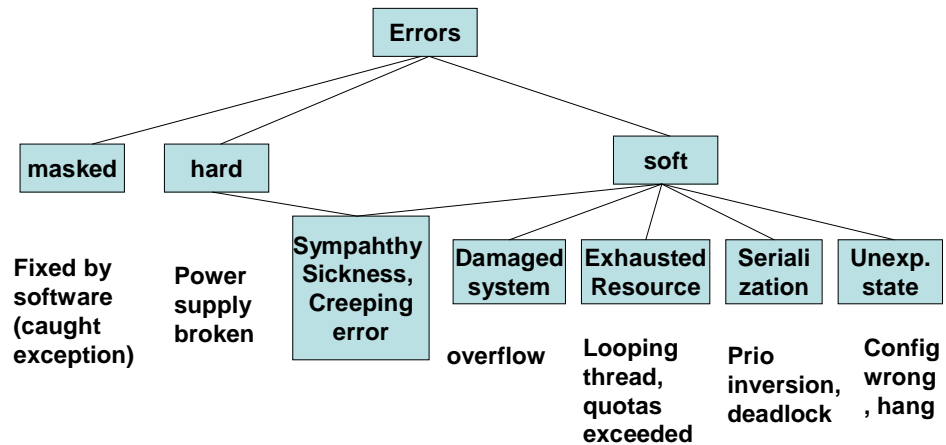
Let's assume the spurious problem happens again and again due to some unknown failure? Only when we observe the state changes (history) of this reliable system we will notice that there is a problem.

How do we notice the problem? From outside we might notice a decrease in throughput or performance, depending on the workload and its parallelism. But what if we do not have two but 20 processors? There is almost no chance to detect the problem via workload measurements – the remaining 19 CPUs will distribute the work and the only real error is a decrease in availability – with 19 CPUs still working this is a theoretical situation, not yet a real performance problem. We learn that we need to separate availability strictly from observed performance and throughput. Both are independent concepts.

What we need to detect the problem of a CPU permanently cycling between down and up is an event logging system which counts those transitions and knows about the probabilities of such events happening. In case those probabilities are exceeded (we will shortly see how this can be calculated for more complex behavior like transaction runtimes) the event system will raise an alarm and provide a causal reason for the alarm: too many cycles in CPU X.

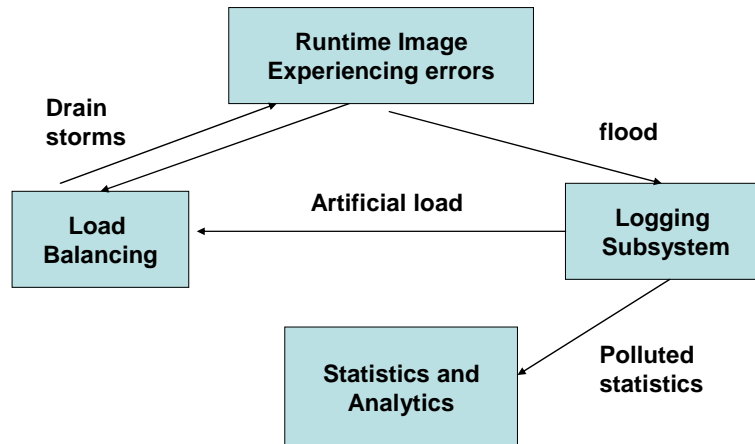
By this we will get an analytic explanation which we could not derive from observed performance or throughput data.

Time to give some further terminology developed by the availability people, here especially [Caffrey]: The diagram below gives a short classification of error types and examples. The focus obviously is on soft failures as the one described above.



Caffrey, pg. 641ff.

Creeping errors are long term consequences of other errors. Today most failures seem to be soft failures with damaged systems and exhausted resources being the most prominent ones. Soft failures usually occur over a longer period of time until finally a dramatic loss in availability occurs. This makes them especially hard to find. Sometimes combinations of soft errors further complicate the picture. They generally tend to be associated with the liveness of an application, i.e. the ability to make progress. Between the real error event and further consequences can be quite some time. Caffrey e.g. describes a case where a wrongfully terminated management process left locks on resources behind and prevented the start of an application a week later. This behavior makes it especially hard to define when exactly a component is in error. Interdependencies between the runtime and error logging and analysis components further complicate the data about possible soft failures as is shown in the diagram below (see also [Hosking] pg. 655f.



Drain storms btw are false interpretations by a client regarding the ability of a server to accept more requests. Certain error cases within the processing of the server look like decreased response times and are interpreted as “server is idle” by the client. Thus more requests are sent down to the broken server. Domas Mituzas of Wikipedia describes such effects in this paper on “Wikipedia: Site internals, configuration, code examples and management issues [Mituzas]. Especially load balancers are affected by drain storms.

I call a related phenomenon “thread hole” and it works like this: a backend service is unavailable but a multithreaded client does not realize this fact. Instead, every request that the client receives will be also directed towards the non-functional backend and results in another thread being stuck. This depletes the VM quickly of threads and – without a limiting thread-pool size – will cause havoc to the application.

<< dependencies between loosely coupled layers: fourthsquare incident>>

We haven’t really solved the question of detecting errors as a pre-requisite for failover yet. Even without the requirement of automatic failover the situation is bad and described beautifully by J.R.M Hosking:

In the 1970s, the most common IBM mainframe was the System/370 Model 158, a 1 million instruction per second (MIPS) machine with one processing unit and a maximum of four megabytes of main memory. The current IBM mainframe is the System z10* EC, which is a roughly 27,000 MIPS machine with up to 64 processors and one terabyte of main memory per logical*

partition (LPAR). The current z/OS operating system is a direct, lineal descendant of the MVS* operating system that ran on the Model 158. In the intervening years, many new types of work have been developed and now run side by side with programs that could have run on the Model 158. The fundamental error-logging processing in the operating system (OS), however, remains unchanged, as does the official IBM service recommendation that customers look at these logs and resolve problems by doing searches in problem databases or opening incident reports with IBM service. [Hosking] pg. 653f.*

Hosking developed two different methodologies to detect mostly soft failures: An analytical method called failure scoring and a statistics based method called adaptive thresholding.

Failure scoring tries to identify problems before they can lead to unavailability. One way to do so is to properly tag the priority of error messages. “Noise” through tons of uncritical messages need to be filtered out to make a possible chain of critical events visible. A clear theoretical understanding of the nature of error events is necessary as well: when are we talking about a regular error event like “file not found” with little chance of damaging the system or wasting resources? And when could an error event potentially disrupt system functions like perhaps an error event describing the attempt to overwrite illegal memory? A special feature of failure scoring describes Hosking as “symptom search” where a database of past events and their consequences is used to find out whether a certain type of event has led to severe problems in the past. Interestingly for this method to work it is necessary to develop a special taxonomy of “severity” of errors. Usually people have very different ideas about severity of an error and this turns out to be a bad indicator for soft failures.

A mathematically more involved method to detect critical errors is “adaptive thresholding” where – based on a large number of statistical events – a machine learning algorithm tries to decide whether a certain “tail” of measurements simply represents especially long running transactions or erroneously looping transactions. Technically, the algorithm tries to find a good “cut-off” value where the long tail begins. Then a generalized Pareto distribution is fitted to the tail. In case of a bad fit the tail is interpreted as being in error. While failure scoring includes the risk to miss some critical error events, the adaptive thresholding method (adaptive because the method adjusts for changes in the system data over time) runs into the danger of falsely declaring something as an error which is simply a long running task.

In a follow up procedure the calculated threshold values e.g. for certain transactions can then be used for comparison with actual performance data. Transactions beyond the threshold (e.g. slow

transactions) are then automatically investigated by machine learning algorithms to find critical properties (attributes) which could be responsible for them being slow. [Hosking] pg. 664 We end this section with the statement that both methodologies presented are not able to drive a fully automated failover. We will come back to the problem of error detection in our discussion of group communication and replication protocols and the CAP-theorem.

J2EE Clustering for Scalability and Availability

For the concepts behind clustering see Lumpp et.al, From high availability and disaster recovery to business continuity solutions, [LSHMLBP]. The authors describe HA approaches (stateless, stateful, cold, warm, hot, active-active). For the use of hardware saving group communication solutions (e.g. to achieve loadbalancing or failover) see Theo Schlossnagels paper on “Backhand” [Schlossnagle]

This chapter will describe the implementation of cluster solutions using the J2EE platform. The goal is to create a platform that does support both availability and scalability. Three concepts are essential in this context: First the concept of contention between parallel requests caused by locking all except one request to avoid inconsistencies. Wang Yu describes the negative effect of contention (hot locks) on scalability in the first part of his series on Java EE application scaling which deals with vertical scalability. This type of scalability is further influenced by memory consumption and the type of I/O handling (blocking or non-blocking).

Horizontal scalability, described in the second part of the series [Yu] has one big problem for throughput: holding session state to achieve fault-tolerance. With respect to fault-tolerance or availability in general we will need to discuss the problems of Single-Point-Of-Failure in Java EE architectures. Here the concept of “unit of failure” is helpful in deciding where to integrate failover options into an architecture. Finally some cluster management issues need to be discussed where we will use the state machine approach in distributed systems to get a better understanding.

Vertical Scalability means to grow a Java virtual machine as a response to increasing service demand (requests). This e.g. can mean to run more threads to service more requests. As we will see in the modeling chapter later this will soon lead to contention between the threads due to locking. Finally the serial part of our code – the part that needs to run behind an exclusive lock – will totally determine the maximum number of requests that can be handled. Adding ore CPUs or more threads will have no positive effect after this. Adding threads also has the ugly side-effect of increasing response times for all users.

Yu mentions the typical solutions to the contention problem:

- use fine-grained locking
- keep locking periods short
- use “cheap” locking mechanisms, not synchronized
- use “test and swap” for wait free locking to avoid context switches
- avoid class level locks

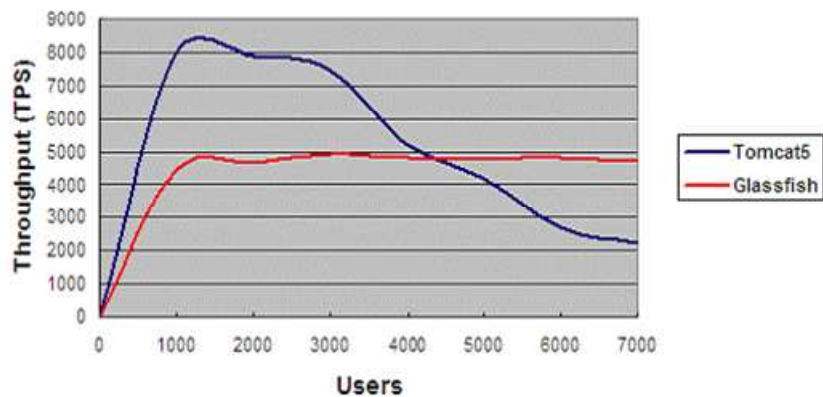
This basically means tracing your code and searching for bottlenecks like synchronized class level (static) methods. If you see 9 out of 10 threads waiting at the same type of lock you have probably discovered a serious bottleneck.

Two other important causes of scalability problems are memory consumption and I/O. Memory should not be a problem anymore – we’ve got 64-bit processors after all and can stuff in RAM almost as much as we want. The limiting factor turns out to be the garbage collection caused by excessive memory use within the VM. Yu mentions an application which simply stopped for 30 minutes doing GC and nothing else. It is not only the use of a large number of threads that can cause excessive memory use. Connection buffers can also have the same effect. Facebook architects had to re-engineer the way memcached used connection buffers to free gigabytes of memory bound to separate connection buffers. [Hoff] in “Facebook tweaks to handle 6 times as many memcached requests”. “

Blocking I/O – also called “thread per connection or request” has two painful side-effects due to the large number of threads required: each thread needs a fixed and large piece of memory at startup which considerably increases VM memory consumption. And a large number of threads cause a huge number of context switches which take away CPU from the workload until nothing is left for the requests. We will talk about alternative I/O models later in a special section so for now we simply state that non-blocking I/O works with only a small number of threads and does not show the above mentioned problems. It is albeit able to serve thousands of requests per second.

The diagram below shows the much better scalability of the non-blocking architecture. The blocking I/O solution on the other hand closely follows the universal scaling algorithm by Gunther which we will discuss in the modeling section.

Blocking vs Non-blocking IO tested in 4CPUs server



From: Wang Yu, *Scaling your Java EE Applications*

What about availability in the context of vertical scaling? The unit-of-failure here clearly is the whole VM. It does not make sense to think about failover within a VM as typically the isolation mechanism within a VM are too weak to effectively separate applications or application server components from each other. Availability is therefore defined by the overall MTBF of the hardware and software combination. Hardware should not be a problem to estimate – all vendors have to deliver those numbers – but software certainly is. It might be your last resort to calculate availability as follows: take the time for a complete hardware replacement and the time needed to perform a complete installation and boot of the software from scratch and multiply each value with the probability per year. The sum of the result will be your estimated yearly downtime and it also defines your average, expected availability. There is no transparency of failures for the clients which will have to accept the downtimes and also no failover.

Let's move to the second type of scalability: horizontal scalability. It means adding more machines instead of growing a single instance. Suddenly other external systems like databases, directory services etc. need to be duplicated as well to avoid SPOFs. The easiest cluster solution according to Yu is the “shared-nothing” cluster where individual application servers serve requests and use their own backend stores. These servers know nothing about each other and a load balancer in front of the array can simply distribute events to any server.

If there is session state involved and it is kept on a server the loadbalancer needs to send all requests of this client to the same server (sticky sessions). In case of a server crash the shared nothing

cluster does not support fault-tolerance or failover and the client will lose the session state. Frequently one can read that sticky sessions are therefore a bad design feature and should be avoided. This argument needs some clarification:

- Sticky sessions do have a negative impact on load balancer freedom to assign the next request.
- Sticky sessions have the advantage that a server does not need to read the session state at the begin of every request
- Sticky sessions are bad for failover if only one server (session owner) or his replication peer (see replication pairing below) hold a copy of the state. This forces the loadbalancer to know about the servers that hold a specific state AND prevent the load from a crashed server from being equally distributed across all servers.
- Sticky sessions avoid the disadvantages (except for LB freedom) and keep the advantages if the session storage architecture allows every server to get to a certain session state if it needs to, e.g. if it has to cover for a crashed server. This supports equal distribution.
- Load Balancer freedom might be possible even with sticky sessions in case of a pull mechanism used by application servers. (See chapter on special web servers).

There exist several mechanisms to keep the session state within a cluster. The determining factors are: size, frequency of storage and number of targets. In other words: how big is the session state? How many times will it have to be stored somewhere? And on how many machines will it be stored? The chicken way out is simple: Try to keep the session state inside of a cookie and let the client take care of it. This sounds rather outdated today – after all there are databases and distributed caches to store session data into. But the fact is that pushing the session storage problem to the client has huge advantages with respect to availability: Load balancers can send a request to any server available and the session state will always be available.

If for whatever reasons client side session state is not an option the worst possible alternative seems to be to store it within a database and update it frequently, e.g. per request. Pushing large numbers of bytes into the database on every request is putting a lot of load on it. Those data need to be serialized as well – another rather slow mechanism involved. And finally those large numbers of writes can change your typical read to write ratio of your web application considerably and have a negative effect on your database-replication setup.

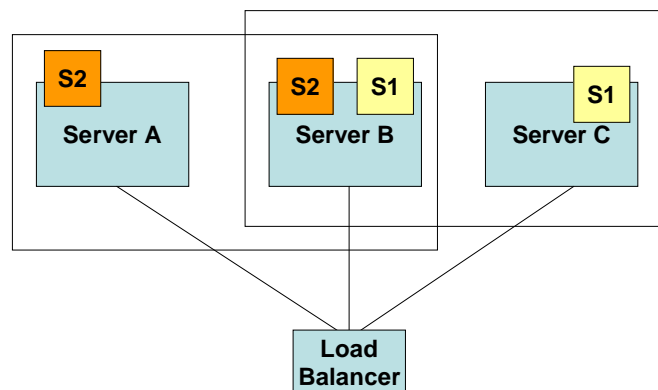
Making sure that only those data that were changed are really written is useful but forces the application to use special session state methods to notify the storage mechanism about granular updates. Btw: instrumenting the code that deals with session

storage is a necessary method to detect abuse e.g. through excessive session sizes.

Another alternative is replicating the session state between application servers and keeping it in memory. While certainly faster than the database solution this architecture forces a crucial trade-off on you: How many machines will participate in the replication? You can decide to simply replicate session state for a certain client to all machines. This makes the life of a load balancer much easier as it can now route a new request from this client to any machine available. But it also forces all machines to participate in every replication and even multicast based protocols will not scale beyond a small number of machines. (We will discuss group communication and replication algorithms later). Pairing machines to replicate a certain session state reduces replication overhead considerably but raises two other problems: the load balancer needs to know about the pairs and in case of a server crash there is only ONE machine which can take over the processing of the current request or session.

<<pairing diagram>>

Session Replication Pairs



Pairing requests means we have coupled session storage with processing location. We can no longer route the request to any server. And this has dire consequences: All the clients from the crashed server will suddenly show up at the one server which hosts the session replicas of the crashed server – in effect doubling the processing load of this server. And this means that, to make our fail-over mechanism work this backup server needs to run at a capacity that will allow doubling it without causing new problems, e.g. regarding the stability of this backup server. We are paying

literally a high price due to the low capacity this server needs to run in everyday business.

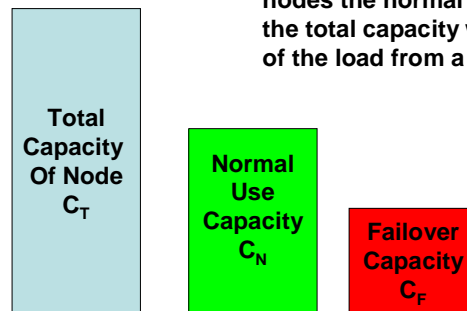
This pairing problem actually points to a rather generic problem for failover: the bigger the machines involved are the more important is to make sure that the load of a failed server can be equally distributed across the remaining machines. Not doing so results in a rather low average capacity limit for those servers as the following diagram shows:

**Effect of number of nodes on wasted capacity
(assuming homogeneous hardware and no sticky
sessions bound to special hosts aka session pairing)**

$$C_N + C_F = C_T$$

$$C_F = C_T / (n - 1) \quad (n = \text{number of nodes})$$

$C_N = C_T - (C_T / (n-1))$ with growing number of nodes the normal use capacity gets closer to the total capacity while still allowing failover of the load from a crashed host



This makes a central session storage server as used by IBM or Sun and others much more attractive again [Yu]. The solution seems to be a dedicated server with high availability and specialized software for reading and writing session state efficiently. There are no fancy SQL queries or locking needed and a specialized in memory store could easily outperform a regular RDBMS here. Yu claims that we will save on memory with a central solution compared to storing session state on all servers. This is right but we don't save any memory compared to the server pairing described above because to avoid SPOFs we will need two of those dedicated session storage servers anyway.

<<Raisin example with timeout feature for sessions>>

Given the costs and complexity associated with distributed session storage Yu suggests to re-evaluate the need for fault-tolerance and fail-over, especially transparent fail-over again. His argument is based on the fact that contrary to popular opinion many requests cannot even use an automated fail-over mechanism in case of a server crash. Because a load-balancer cannot know exactly WHERE a request was when the server crashed only those requests

that are idempotent (cause no server state change) can be automatically restarted. Otherwise there is the danger of performing the request twice.

Perhaps a lesser quality of fail-over might be acceptable after all? A fail-over mechanism that makes sure that clients will find a new server after a server crash and this server will be able to deal with additional load caused by the crash. But the clients will have lost some work which they have to redo now on the new server. This even allows the option of later reconciliation between the state before the crash and the new state created after the crash on the new server.

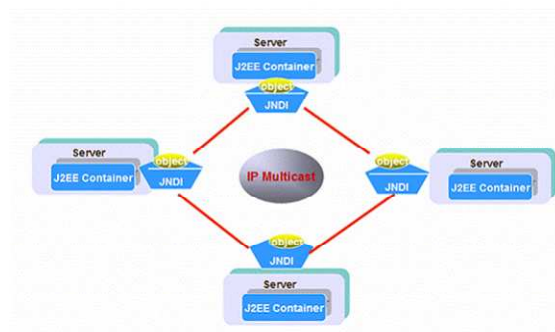
To close the discussion of clustering we need to talk about three technical aspects: how failover is done, specifics of EJB clustering and how SPOFs in supporting services are avoided. Let's start with the last point: avoiding SPOFs in mission critical services. A cluster is not much fun when it contains a single point of failure that makes the whole cluster inoperable. Such a component e.g. would be the JNDI directory service where critical public objects have been registered by system administration. If applications cannot get to their directory information, no processing whatsoever will happen in this cluster.

Vendors seem to have chosen rather similar solutions, basically consisting of replicated JNDI services at every application server/machine. This leads to the question how those services are kept in sync. Some vendors seem to simply propagate a change on one service to all the others which obviously know about each other to make this work.

<<jndi replication >>

A fault-tolerant JNDI name service

I

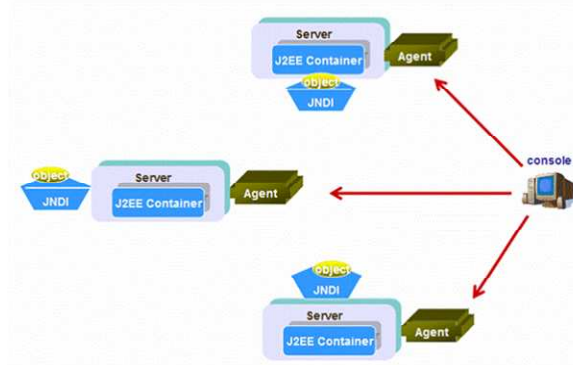


From: Wang Yu, uncover the hood of J2EE Clustering,
<http://www.theserverside.com/tt/articles/article.tss?t=J2EEClustering>

Other vendors keep the services independent and ignorant of each other and use a state machine approach for replication. As the state machine approach can explain some additional restrictions when using clusters we will give a short introduction by example. Lets assume there are system management agents running on all application servers. These agents accept commands from a common management station. System administration now sends initialization commands to all agents which perform those commands against the local JNDI service. After all those commands have been executed the JNDI services will all have the same content but are completely independent of each other.

<<independent JNDI services >>

Independent fault-tolerant JNDI



From: Wang Yu, uncover the hood of J2EE Clustering,
<http://www.theserverside.com/tt/articles/article.tss?!=J2EEClustering>

Clearly this state machine approach requires the same software on all machines to be present. And it is only valid for the system management aspect. Regular client requests coming from the load balancer are non-deterministic and do not follow the state machine idea. Software operating in lock-step on every machine is nice for achieving replicated content across servers. But it has some complicated side-effects on applications within a cluster, especially those who need to run only ONCE within an infrastructure. In other words those applications or objects that need singleton behavior. When all software is the same it is rather hard to establish a singleton. Yu suggests for those cases (e.g. collecting counts of requests) to use the database to collect the data from all cluster machines. Other solutions are to implement a group communication protocol that achieves consensus on who within a cluster needs to perform what.

The second topic to discuss is the question of how exactly a failover is performed. It turns out that there are several possible solutions, ranging from strictly client side decisions over code in the infrastructure levels to server side behavior where new machines start to respond for a crashed server. While all these mechanisms can achieve failover the big difference sometimes is in the system management and configuration overhead associated with them. Transparent but optimal server selection will be handled again in the chapter on load balancing and geographically distributed data centers.

Finally the EJB and J2EE architecture shows some specific problems with respect to failover and scaling. The two core concepts here are enterprise beans and remote objects. Stateless Session Beans are harmless and due to the fact that they contain no state they can be replicated across machines without any problems. Here the only problem lies in routing the client to a new instance, e.g. by providing the bean stub with additional server addresses. This way the client who downloads the stub dynamically does not even know about the other potential server locations running bean replicas.

Stateful session beans follow the same mechanism as replicated session state and can be located through client side code. And finally entity beans are stateless because they store their state transactionally inside databases and could be replicated as well across servers. But they do expose a different problem: They are no longer used remotely because usually there is a session facade in front of them which does a local call to the entity bean. This puts the facade and the bean into a single unit-of-failure and removes the remote call to the entity bean as a possibility for fail-over. Here the importance of a reasonable definition of those units-of-failure becomes obvious: Bundling facade and bean might reduce failover and availability because facade and bean cannot be replicated independently. But at the same time bundling those two into a local unit-of-failure prevents excessive remote calls and the terrible costs of potentially distributed transactions. The last point nicely shows that availability and scalability can be somewhat orthogonal concepts even if they seem to go along well in case of horizontal scaling.

More aspects of clustering like the use of distributed caches etc. will be discussed in separate chapters later.

<<diagram of EJB failover façade-entity bean local concept>>

<<pull concept of web application server to load balancer>>

Reliability

- idea: integrate CEP as an explanation system
- reliability and scalability tradeoff in networks (Birman pg. 459ff)
- self-stabilizing protocols
- epidemic, gossip protocols
- the role of randomness to overcome deterministic failures in state machine protocols

Dan,

listening to you (or similarly Dan Pritchett talking about eBay architecture)
I have some questions in my mind:

how do you **test** new features before you roll them out? how do you test them scale? you don't have test lab with same level of scale as your production farm, do you? that makes me think you can't guarantee or knowingly predict exact level of performance until real users hit the new feature in real time. how do you deal with that and what did you have to build into your system to support new features deployment, as well as rolling features back quickly if apparently they did not work out as you had expected?

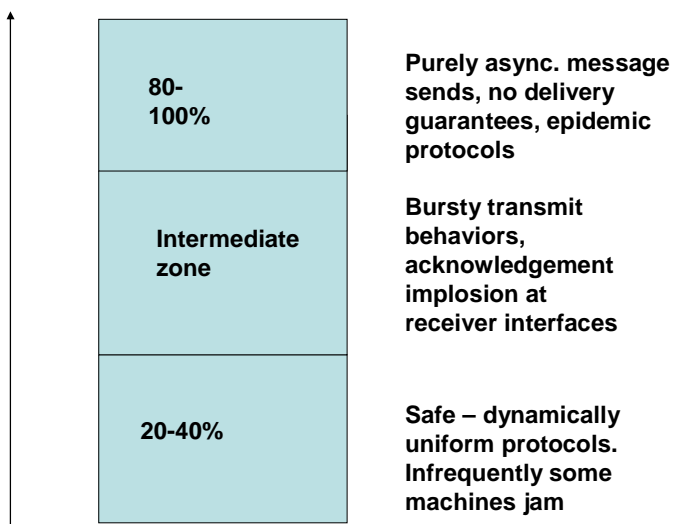
and another question is what did you have to do to support existence of **"practical" development environments** that behave as your production system but do not require each developer to work on dozens of servers, partitioned databases, and cache instances. How did this change your system's architecture?

Deployment

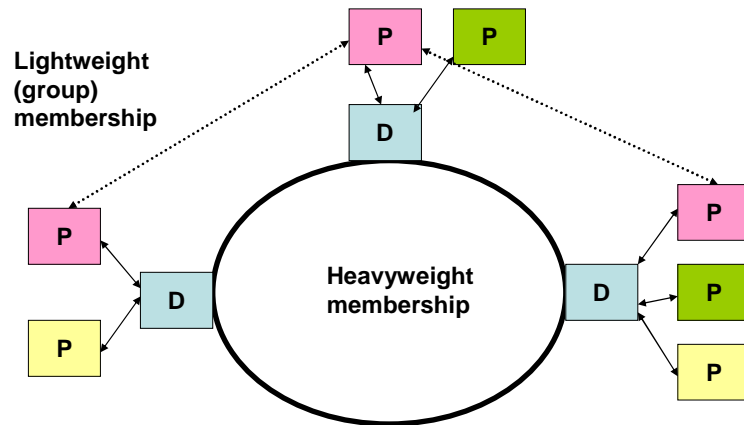
- transacted, incremental, available, see the Resin paper. GIT as a repository which avoids overwrites, partial writes.

Reliability and Scalability Tradeoff in Replication Groups

Load and participants



See Birman, pg. 459ff.



In the Spread group communication framework daemons control membership and protocol behavior (order, flow control). Messages are packed for throughput.

Performance

- the 4 horseman plus remote collection
- event processing modes
- alternative (ERLANG)
- Latency
- Operations vs. analytics: don't mix TAs with OLAP, keep queries simple and use separate complex background analytical processing. Keep search from destroying your operational throughput
- Concurrency: contention and coherence

Monitoring and Logging

CPE, Astrolabe

Schlossnagle on Spread-based logging

Distribution in Media Applications

- o Large Scale Community Site
- o Storage Subsystems for video,
- o Audio-Server for interactive rooms, clever adaptations, new uses
- o Distributed Rendering in media production
- o Massively Multi-Player Online Games (gamestar architecture sony everquest)
- o Search Engine Architecture and Integration

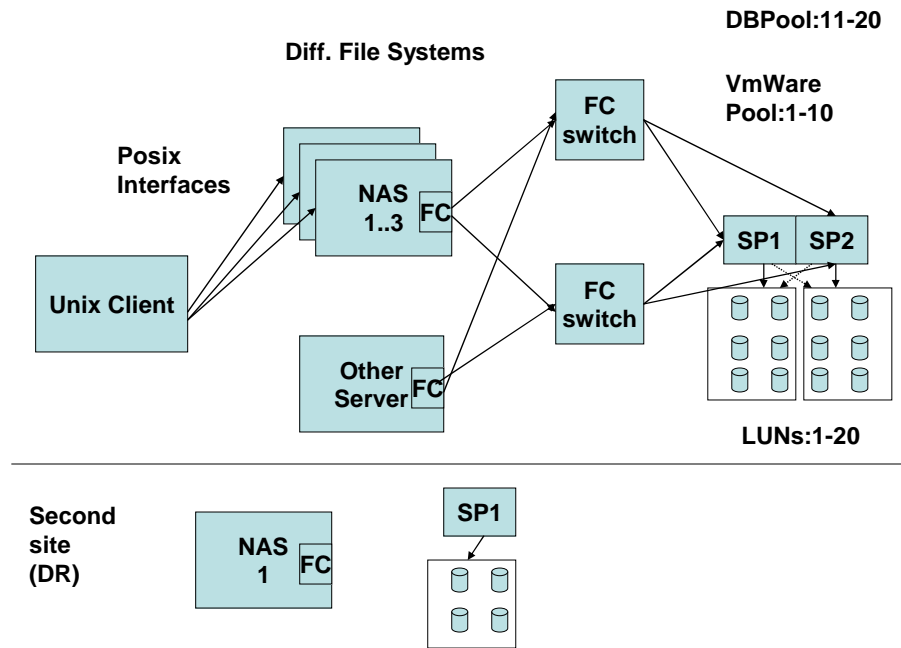
Storage Subsystems for HDTV media

In a recent workshop with a large german broadcast organization we have been discussing several options for large scale storage subsystems. They should be able to support around 20 non-linear editing stations with approximately 50 concurrent streams of HDTV content stored in those subsystems. We are talking between 50Mbit/sec and 100Mbit/sec for each stream and bandwidth as well as latency need to allow uninterrupted editing. The move toward HDTV was combined with going from tape

based editing with its distribution and copy latencies to disk based, concurrent editing of video material. The change in storage size and speed required a new infrastructure and the broadcast organization was worried about the scalability and usability of the solutions proposed by the industry. The worries were not in the least caused by the fact that all the proposals fell into two completely different camps: classic NAS/SAN based storage subsystems using fiber channel switches etc. and the newer architecture of grid-based storage, also called active storage.

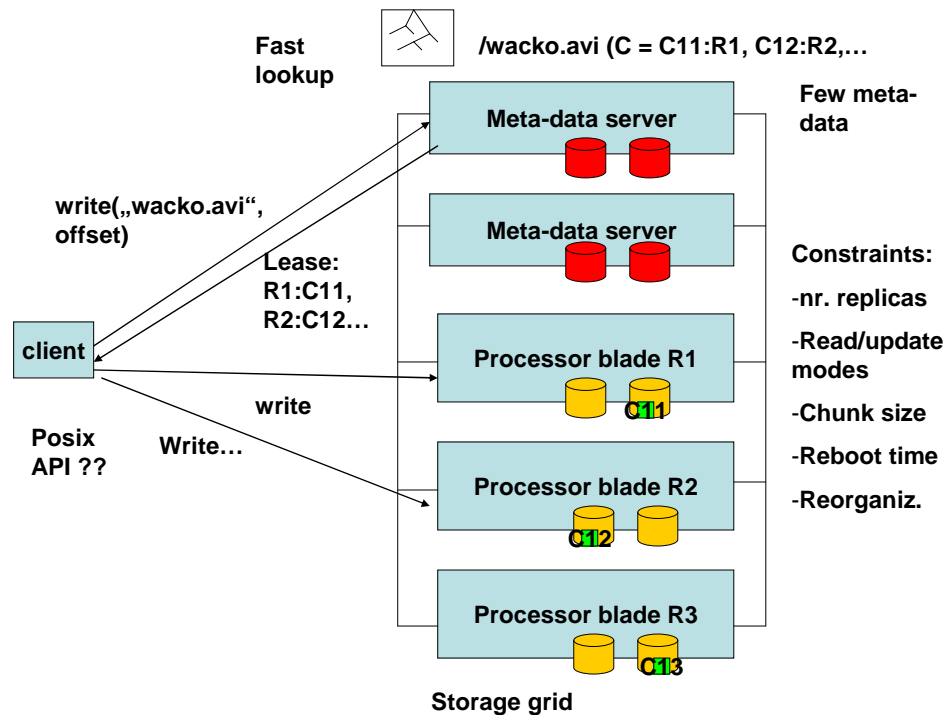
The organization had a midrange NAS/SAN combination already in production and the strength and weaknesses of this architecture are fairly well known: while file-systems can grow with little maintenance and organizational effort there are some limitations in the systems where bottlenecks restrict scalability: there can be several filesystems running on several NAS front-end machines but if there are hot-spots within one filesystem few things can be done to relieve the stress on the NAS carrying the filesystem as well as the SAN controlling the associated disks (see the problems myspace engineers reported about non-virtualized SANs). Storage processors used in the SAN also put some limit on scalability. Internal re-organization of disks within the SAN can be used to improve performance on critical filesystems. There are proven management tools to support server-free backup. Disk errors will require the reconstruction of disks which is a lengthy process due to RAID. One can say that the subsystem performs well but with a high management cost. It is used for several types of content like database content or media and there is little doubt that a new system based on NAS/SAN would be able to offer 500 or more Terabyte of storage with the proper access and throughput rates. Another big advantage of the classic architecture is its support for POSIX APIs on the client side which allows standard and unmodified applications to be used.

The diagram below shows a typical solution of a NAS/SAN combination.



On the right side of the diagram Storage Processors (SP) partition disks into different LUNs and allow partial shutdown of the storage e.g. for maintenance. Several NAS frontend servers exist and carry different filesystems. The subsystem also stores data from database servers. LUNs have to be put into pools for filesystems and allow transparent growth. If a NAS server crashes the filesystems it carries are unavailable. In case of disaster there is a switch to a passive system which is smaller than the master storage center. Still, it is a rather expensive solution for disaster recovery and the possibility of active-active processing should be investigated as the distances are small.

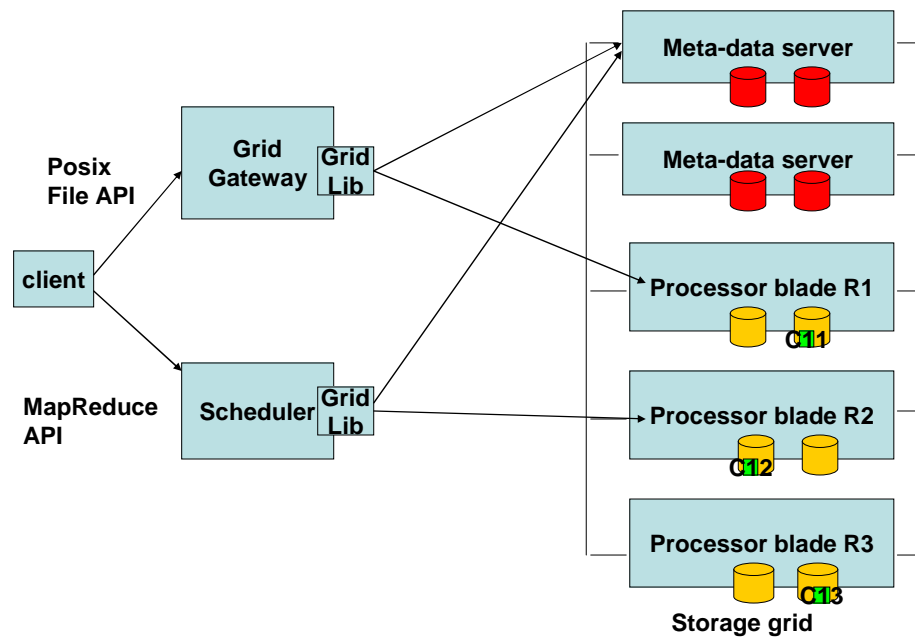
The situation on the grid storage side is much more complicated. The technology is rather new, few vendors use it and of those vendors most created proprietary solutions consisting of applications and the grid storage system (e.g. AVID). To get a technical handle on grid storage we compared it with a well known grid storage system: the google filesystem (see the section on storage where a detailed description is given). We were also careful to note the differences in requirements because most grid storage systems tend to be specialized for certain use cases. The diagram shows only the major components of a grid storage system. And the question was whether the promises of unlimited scalability were justified. Those promises were based on the fact that the overall bandwidth and storage capacity would increase with every blade installed in the system.



The claim of unlimited scalability seems to be in conflict with the obvious bottleneck in the system: the master servers. Would they put a limit on scalability? We did not have a running system where we could take the measurements needed for regression analysis and later processing with Gunthers “universal scalability formula” (see chapter on analysis and modeling). The solution was to check for scalability problems with master/slaves architectures e.g. in GoogleFS. As we will see in the chapter on algorithms below Google uses quite a number of master/slaves architectures without real scalability problems. The core requirement here is that the master is only allowed to server meta-data. And those have to be kept small. This is different to the NAS/SAN solution where a NAS server plays the role of master for its filesystem (doing lookups and keeping file/block associations) AND has to collect and serve the data to clients.

So with some architectural validations we could put the worries about master bottlenecks to rest. And bandwidth as well as parallel access from clients should be excellent due to the direct connection to the blades. In case of disk crashes or bit-rot the new disk or chunk could be easily re-created from replicas and in a much shorter time than in the classic solution.

But other worries became much more visible: The API of GoogleFS e.g. is non-standard, meaning Non-Posix. Typically in storage grids there is a tight coupling between applications and the grid. And a big question: what should be done with the huge number of CPUs running in the grid? What kind of work should they do in addition to serving data? How would programming work? It became clear that some components were missing in the picture and the diagram below shows a gateway and scheduler service added:



The gateway is needed to attach Posix based clients to the grid – for applications which have no customizable storage interface. And the scheduler needs to accept parallelizable jobs and distribute the tasks over the blade CPUs using the classic map/reduce pattern that made Google famous (see chapter on scalable algorithms below for an explanation).

In the end the following results were written down:

Grid Storage vs. NAS/SAN

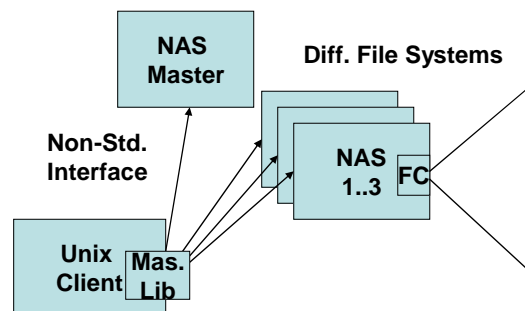
- | | |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> • Posix-Grid gateway needed • Special caching possible but not needed for video (read-ahead needed?) • Huge bandwidth and scalable • Maintenance special? • Proprietary? • Parallel Processing possible • Special Applications needed • Questionable compatibility with existing apps. • Disaster recovery across sites? • Standard Lustre use possible? (framstore?) • More electric power and space needed for grids | <ul style="list-style-type: none"> • Posix compatible • Special caching difficult to implement in standard products • Hard limit in SPxx storage interface but plannable and limited lifetime anyway • Simple upgrades • Standard filesystem support • Dynamic growth of file systems via lun-organization • Maintenance effort to balance space/use • Proven, fast technology • Expensive disaster recovery via smaller replicas • Several different filesystem configuration possible • Without virtual SAN hot-spots possible on one drive • Longer drive-rebuild times |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Key points with grid storage: watch out for proprietary lock-in with grid storage and applications. Watch out for compatibility problems with existing apps. Without real parallel processing applications there is no use for the CPUs, they just eat lots of power (atom?). You should be able to program your solutions (map/reduce with Hadoop). Definitely more prog. Skills needed with grids. NAS/SAN won't go away with grid storage (which is specialized).

Some of the points were converted into possible student projects and the list can be found at the end of the book. Of especial interest would be a

Lustre implementation on our own grid (framestore seems to run such a system successfully), a ZFS implementation on NAS and the proxy/gateway servers and using Hadoop for transcoding and indexing video content.

But not only the grid storage solution can be further optimized: the master/slaves concept of the grid can be used just as well with the classic NAS/SAN solution as can be seen here:



It comes as little surprise that the API problems of the grid solution show up here as well.

Audio Server for Interactive Rooms

- concept of ubiquitous media, mobile devices, interactive rooms.
- Blog upload of media
- RWTH Aachen reference

<<picture interactive room>> (Stanford or RWTH)

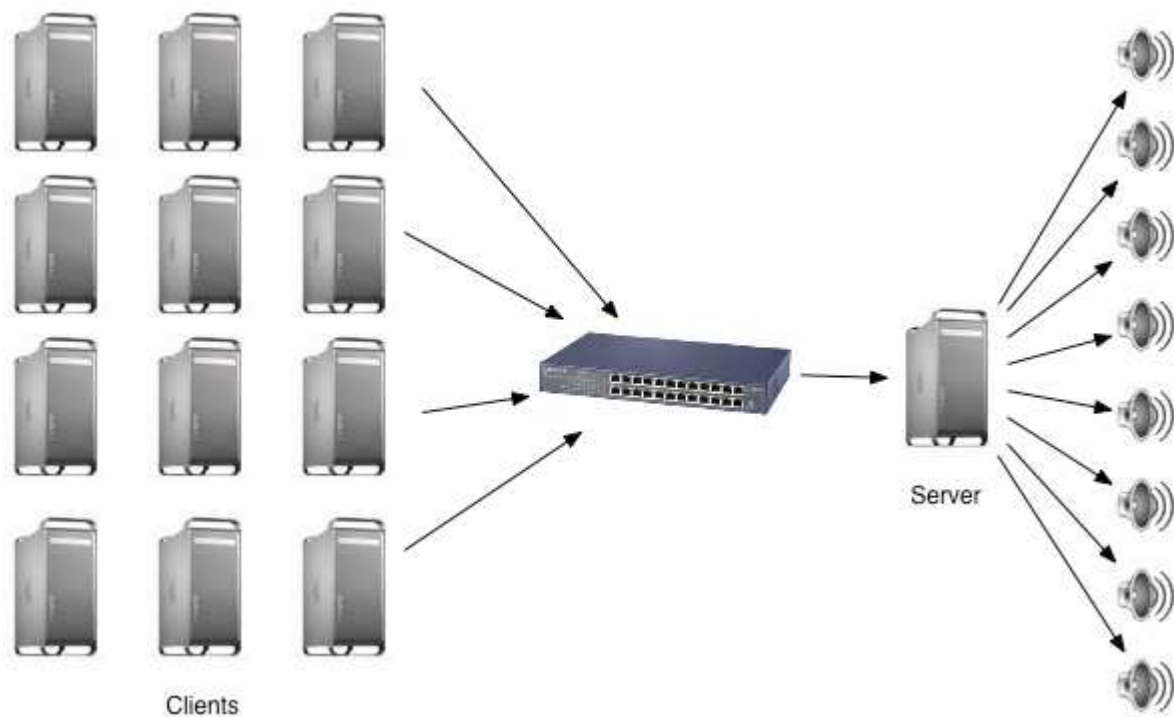
Interactive rooms are facilities where users equipped with mobile computing environment find infrastructure which enables collaboration and communication, e.g. through the use of large TFT panels and beamers. Users can interactively zoom in and out of presentations and have their equipment present information on wall displays.

Everybody who has done software design in groups has notices the difference laptops with wireless communications can make on group performance. The presentation of visual or audio information at the correct place in an interactive room is a problem that needs to be solved in this context.

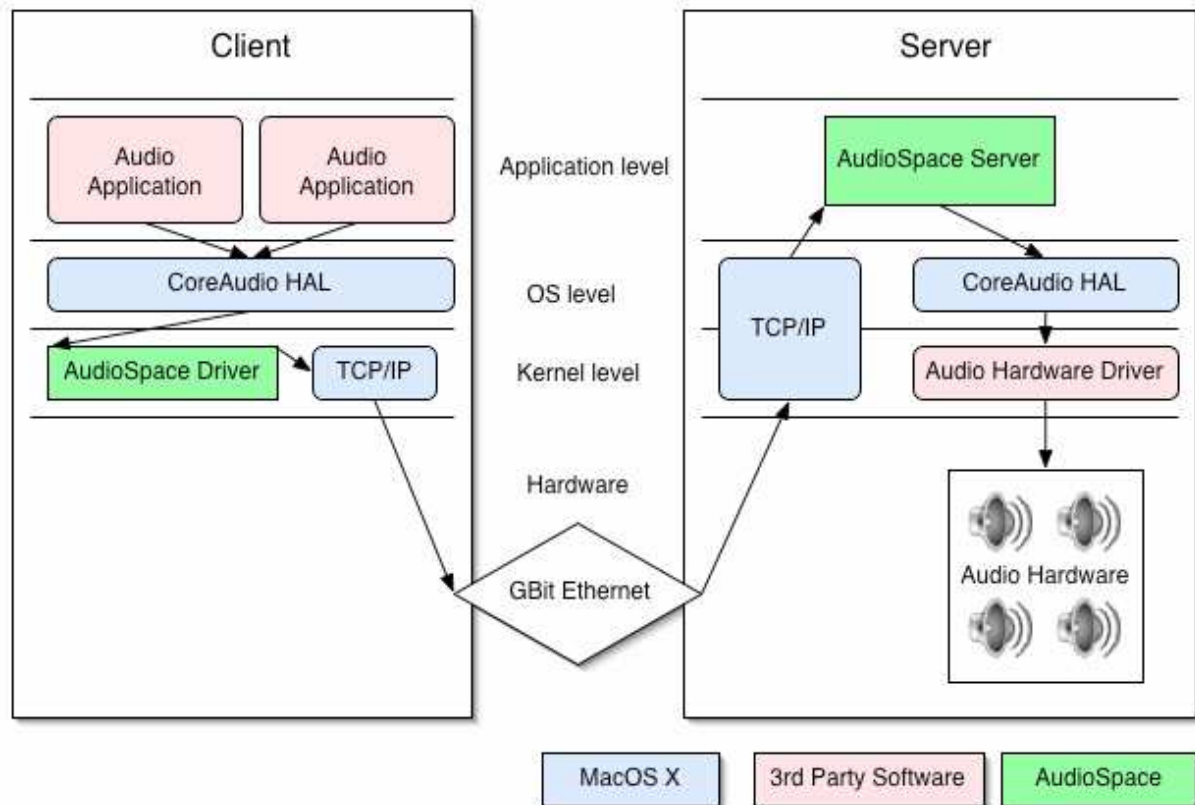
The audio servers developed by Stefan Werner at Hochschule der Medien Stuttgart is part of an interactive room concept of the RWTH Aachen. Its goal was to allow a large number of audio creation machines (without speakers or audio hardware) to create audio content and control the playback of the audio sound through a central audio server which controlled a 7.1 audio system.

This design allowed audio content to be played in front of several different flat panel screens in the interactive room, depending on the configuration of the clients.

The software developed consisted of client and server parts and included a kernel component for the MAC OS based clients. PMC encoded audio was sent between clients and server. Distribution problems like near-realtime requirement for playback (10ms response time), Jitter and clock skew between machines had to be compensated for and new algorithms to compensate for the effects of distribution had to be developed.



Software architecture of distributed audio server:



Distributed Rendering in 3DSMAX

Rendering is the process of media data creation from raw data using matrix, differential and integral calculations. It is no surprise that these calculations put a heavy burden on a single CPU.

Just to make the importance of distributed rendering clear I have taken some numbers from Markus Graf's thesis on "Workflow and Production Aspects of Computer-Animations in Student-Projects" [Graf].

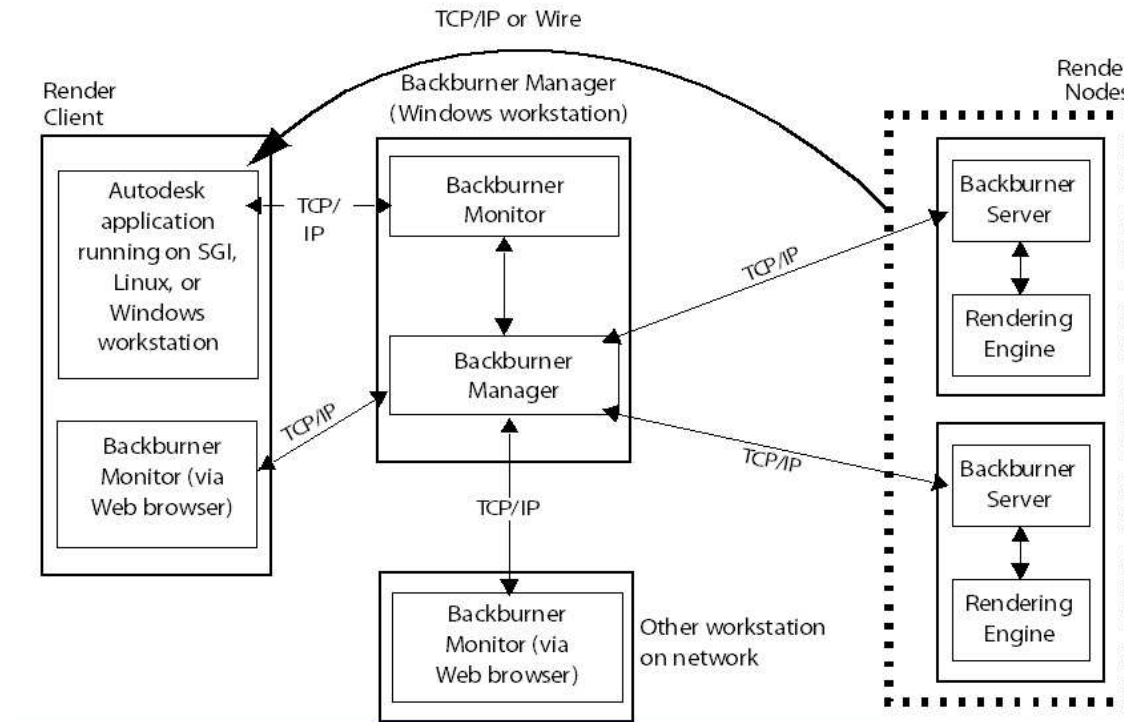
Creating a computer animation requires rendering the raw movie data (animation data, light information, surface properties etc.) into frames. In professional productions each frame takes between one half of an hour and ninety hours to create. Some numbers: let's say we want to make a movie five minutes long. This means 5min. x 60sec. x 30frames and results in 9000 frames. Let's assume only 10 minutes rendering time per frame we end up with 1500 hours rendering time. Usually a frame consists of multiple layers which can be rendered independently. This reduces the individual rendering time per layer but adds to the frame rendering time. At 5 layers per frame and only 5 minutes rendering time per layer we need a whopping 3750 hours of rendering time.

This is when distributed rendering becomes an issue. The following pages describe distributed rendering in 3dsMax. They have been written by Valentin Schwind.

Understanding the Rendering Network Components of 3dsMax

The following components are common to all rendering networks:

- An Autodesk application that sends jobs to the rendering network (the render client).
- At least one Windows or Linux computer that does the rendering (the render node).
- A workstation that distributes and manages the jobs running on the rendering network (the Backburner Manager).
- At least one workstation that monitors the jobs running on the rendering network (the Backburner Monitor).



Note: Rendering networks for Discreet Inferno® , Flame, Discreet Flint® , Discreet Fire® , Smoke, Autodesk Backdraft® Conform, and Lustre require additional components in addition to those shown. For more details about these rendering networks, see the latest user's guide for Autodesk Burn™ and/or the latest installation guide for Lustre.

The following list provides more detail about each component.

Render client—This is the Autodesk application running on an SGI®, a Linux, or a Windows workstation. From here, you create and send rendering jobs (such as a Flame Batch setup or a 3ds Max scene) to be processed by the Backburner rendering network.

Backburner Manager—This is the hub of the background rendering network running on a Windows 2000, XP, or higher workstation. Jobs are submitted from the render client to Backburner Manager, which then redistributes them among the rendering nodes on the network. To view the progress of the tasks, use Backburner Monitor.

You can either run Backburner Manager manually or run it as a Windows service. Running the Manager as a Windows service starts it automatically when the system is booted. Backburner Manager then runs continuously until either the workstation is shut down or the service is stopped.

Render node—This is a Windows or Linux workstation on the rendering network that processes jobs sent by the render client and assigned by Backburner Manager. Each render node runs Backburner Server to allow it to communicate with the Backburner Manager. Render nodes use common network protocols like TCP/IP and/or Autodesk Wire® to obtain frames and then transfer resulting rendered frames back to the render client.

Backburner Server—This is an application that runs on each render node in the rendering network. Backburner Server accepts commands from Backburner Manager to start and stop the rendering engine that processes the frames or tasks on the render node.

Rendering Engine—This is the Windows or Linux rendering engine that renders frames from jobs submitted from Autodesk applications. Many applications (such as 3ds Max) have their own rendering engine; Inferno, Flame, Flint, Fire, Smoke, and Backdraft Conform share a single rendering engine called Burn. Cleaner is both its own rendering engine and a rendering engine for Inferno, Flame, Flint, Fire, Smoke, and Backdraft Conform jobs requiring transcoding between video formats. The rendering engine is installed with Backburner Server on each render node. You can install multiple rendering engines on a render node. This allows the render node to render jobs from different applications.

Backburner Monitor—This is the user interface for the Backburner rendering network. It allows you to view and control jobs currently being processed. Jobs in the rendering network can be stopped, restarted, reordered, or removed entirely using the Monitor. You also use Backburner Monitor to identify any render nodes that are not working and check the overall health of the rendering network.

Backburner Monitor runs natively on a Windows workstation but can also be run through a Web browser from any workstation on the network.

Using partitioning to speed things up

The above architecture allows the distribution of individual layers and frames to rendering servers. This problem is “embarrassingly” parallel which means it lends itself easily to parallelization because the components (frames, layers) are independent of each other and can be rendered separately.

It comes as no surprise that this method puts a limit on the overall performance improvement that can be achieved: it is the time that a layer or frame needs to be rendered because this task is done sequentially on a server. Partitioning the movie into frames or layers is a rather coarse grained way to distribute the workload. A fine grained version would be to partition each frame or layer further into smaller parts. Those parts could then again be distributed to several servers and the overall time in the best case reduced to the time needed to render a complete frame or layer

divided by the number of fragments (if we assume no communication or synchronization costs).

We have now increased the granularity of the partitioning of the workload and ended up with better parallelization. This pattern is frequently used in distributed systems and applies e.g. also to the case of locking and synchronization: The more fine grained locks are set, the better the parallelization of the task. We will discuss the downsides like increased software complexity and danger of deadlocks in the chapter on concurrency and synchronization).

Distributed rendering in computer animation is conceptually rather simple but can still offer some surprises. When distributing workloads to several server machines the managing software expects identical interfaces on those servers, e.g. to accept certain frame sizes etc. While the interfaces of the remote procedure calls are all the same on those machines this does not mean that the resulting rendering is correct. In case 32bit and 64bit machines are used together rendering artefacts due to different rendering precision can be seen. In case of coarse-grain partitioning frames will show differences, in the fine-grained case the differences will be seen between rectangles of the same frame or layer: Interface is therefore not everything!

Part III: Ultra Large Scale Distributed Media Architectures

```
while (true)
{
  identify_and_fix_bottlenecks();
  drink();
  sleep();
  notice_new_bottleneck();
}
```

This loop runs many times a day.

(Todd Hoff, youtube article)

This recipe for handling rapid growth is probably very common. But the question is whether this is all we can do? The following chapters are trying to answer the following questions:

- What are the concepts used in large scale sites, patterns and anti-patterns?
- Can we model such sites and use the model to predict bottlenecks?
- Are there systematic ways to avoid scalability surprises?
- Certain statements show up repeatedly, e.g. “keep it very simple”. Can we find parameters for simplicity in such sites?
- Do large sites favor certain types of software, e.g. open source?
- How do business models and architecture interact?
- What is the development methodology behind ultra-large sites? How do they deal with extremely fast growth?

Analysis Framework

In this second part of the book we will look at many large scale sites like wikipedia, myspace, google, flickr, facebook etc. Our goal is to find the core principles and architectures used in scaling to such a size. And from there we will extract essential components like distributed caching, replication and drill down to the algorithms used in implementing them.

What kind of questions are we going to ask those architectures? The following list names core categories for scalability:

- The role of hardware – when to invest in bigger irons instead of more complicated software. Is it true that switching to 64bit hardware with its much bigger memory support is what made MySQL scalable in the end?
- What are the core areas for scalability? Global distribution, CDN, Loadbalancing, application servers, distributed caching, database partitioning, storage system.
- What is the role of programming languages? Are there certain specialized areas where on language dominates?
- What kind of software is used? Open Source or proprietary? Windows possible?
- How do we minimize hardware costs for fault-tolerance?
- How is monitoring done?
- Is there a certain path to scalability that is mirrored by all sites? Where are the main bottlenecks?

Last but not least we will try to describe the history of those sites as well. How they started, what the major inventions were and finally where they might end up in the near future.

An excellent starting point for site analysis is provided by Todd Hoff. He used a list of questions for the architects of lavabit to describe their site, its architecture and scalability solutions as well as the problems they had [Levison]. The core parts of the questionnaire are listed below:

<<questionnaire Hoff >>

- * What is the name of your system and where can we find out more about it?
- * What is your system for?
- * Why did you decide to build this system?
- * How is your project financed?
- * What is your revenue model?
- * How do you market your product?
- * How long have you been working on it?
- * How big is your system? Try to give a feel for how much work your system does.
- * Number of unique visitors?
- * Number of monthly page views?
- * What is your in/out bandwidth usage?
- * How many documents, do you serve? How many images? How much data?
- * How fast are you growing?
- * What is your ratio of free to paying users?
- * What is your user churn?
- * How many accounts have been active in the past month?

How is your system architected?

* What is the architecture of your system? Talk about how your system works in as much detail as you feel comfortable with.

* What particular design/architecture/implementation challenges does your system have?

* What did you do to meet these challenges?

* How did your system evolve to meet new scaling challenges?

* Do you use any particularly cool technologies or algorithms?

* What did you do that is unique and different that people could best learn from?

* What lessons have you learned?

* Why have you succeeded?

* What do you wish you would have done differently?

* What wouldn't you change?

* How much up front design should you do?

* How are you thinking of changing your architecture in the future?

What infrastructure do you use?

* Which programming languages does your system use?

* How many servers do you have?

* How is functionality allocated to the servers?

* How are the servers provisioned?

* What operating systems do you use?

* Which web server do you use?

* Which database do you use?

* Do you use a reverse proxy?

* Do you collocate, use a grid service, use a hosting service, etc?

* What is your storage strategy?

* How much capacity do you have?

* How do you grow capacity?

* How do you handle session management?

* How is your database/datatier architected?

* Which web framework/AJAX Library do you use?

* How do you handle ad serving?

* What is your object and content caching strategy?

* Which third party services did you use to help build your system?

* How do you health check your server and networks?

* How you do graph network and server statistics and trends?

* How do you test your system?

* How do you analyze performance?

* How do you handle security?

* How do you handle customer support?

* How do you decide what features to add/keep?

* Do you implement web analytics?

* Do you do A/B testing?

* How many data centers do you run in?

* How do you handle fail over and load balancing?

* Which DNS service do you use?

* Which routers do you use?

* Which switches do you use?

* Which email system do you use?

* How do you handle spam?

- * How do you handle virus checking of email and uploads?
- * How do you backup and restore your system?
- * How are software and hardware upgrades rolled out?
- * How do you handle major changes in database schemas on upgrades?
- * What is your fault tolerance and business continuity plan?
- * Do you have a separate operations team managing your website?
- * Do you use a content delivery network? If so, which one and what for?
- * How much do you pay monthly for your setup?

Miscellaneous

- * Who do you admire?
- * Have you patterned your company/approach on someone else?
- * Are there any questions you would add/remove/change in this list?

Added:

- did you use or change to a certain programming language for certain areas and why?

Examples of Large Scale Social Sites

Large sites eventually turn to a distributed queuing and scheduling mechanism to distribute large work loads across a grid. (Todd Hoff, highscalability.com)

*Architects are falling from their towers and starting to use common-sense technology (like HTTP, RSS, ATOM, REST) more and more and are abandoning 'enterprise' patterns and tools (think JEE, Portals, SOAP etc).
<http://log4p.com/2009/03/12/qcon-2009-2/>*

We will begin with a presentation and discussion of some hopefully prototypical sites. Most of the papers or talks can be found at the excellent site from Todd Hoff on scalability. www.highscalability.com. Todd Hoff collected numerous articles and presentations and frequently creates abstracts which we are going to use here heavily.

Wikipedia

This site has been chosen for several reasons: first of all information about its architecture is public. Second because of its size and focus on content it seems to represent a certain – older – type of Web2.0 site.

The discussion mostly uses information from [Mituzas] and [Bergsma].

Interesting aspects:

- content delivery network and geographical distribution
- mysql partitionings and clusters
- hardware choices
- monitoring and tracking
- application architecture and scalability
- load balancing technology used
- media related optimizations (storage, compression)

Myspace

This is one of the few Microsoft-based large scale sites. We can use a short wrap-up of a Dan Farino talk by Todd Hoff [Hoff] which highlights some very interesting aspects:

[Farino] Dan Farino, Behind the Scenes at MySpace.com,

<http://www.infoq.com/presentations/MySpace-Dan-Farino;jsessionid=3219699000EB763C9778865D84096897>

- Correlation of registered users and technology changes needed.

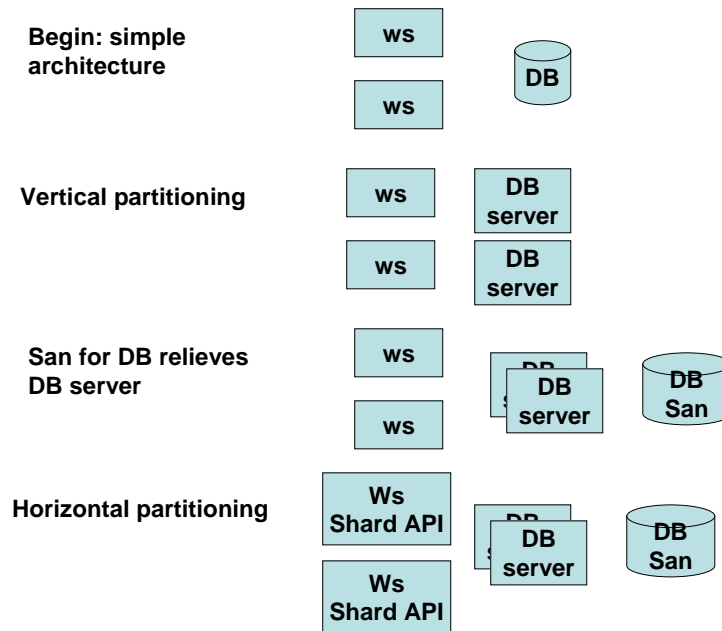
This really is a nice list.

- Database partitioning used (vertical, horizontal)
- The role of caching
- Tooling on Windows platforms

The diagram below shows the first phases of Myspace evolution. It started as a rather simple architecture with two web servers and one db server with direct attached storage (disks). After reaching 500000 users this architecture stopped working. Vertical partitioning was used to split the

main database into several databases with different topics. Soon updating the now distributed instances became a major problem.

As a first scaling measure the disks were put into a San and the database server relieved. Later came a switch to horizontal partitioning of the database.

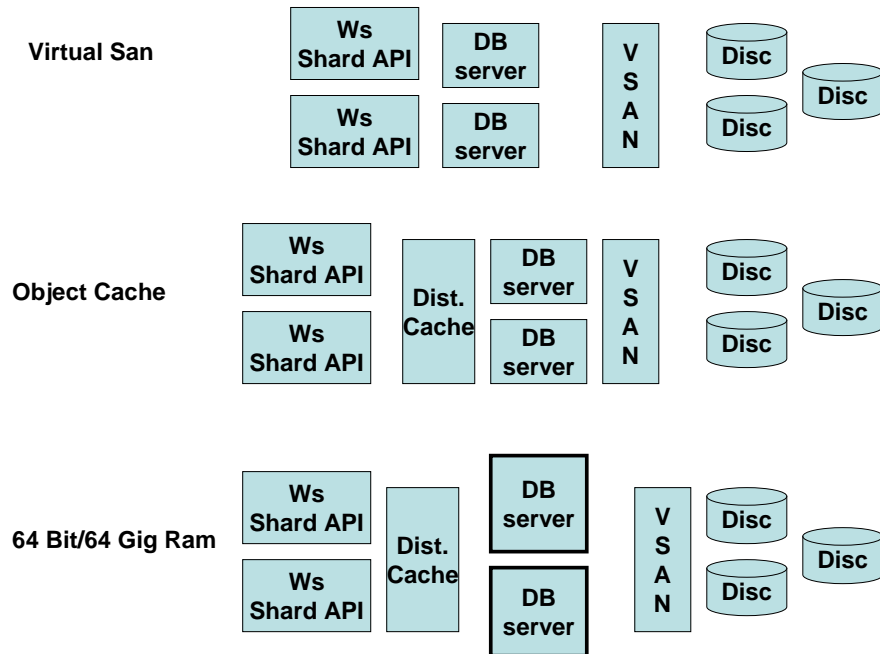


Further growth brought a new bottleneck – the SAN. Some applications were hitting certain discs within the SAN very hard and caused excessive load. Other discs in the SAN were idling. This was solved by moving to a virtual SAN which can internally re-organize data blocks transparently and thereby removing hot-spots on certain discs.

Then came a rather big change: the introduction of an object cache. The creators of Myspace mention that they should have introduced a cache much earlier (actually, most large-scale architectures use memcached or something like it). As can be seen the Myspace team had a focus on the database and storage layer for a long time, optimizing the hell out of it. The new object cache did probably change read/write ratio and overall traffic numbers considerably and it is questionable how the database and storage layer might have evolved with an earlier introduction. Also, the fine dependencies between cache and database organization and query behavior (see chapter on database partitioning later) made an optimal integration of the cache rather hard.

Finally the database servers were migrated to 64-bit architectures (again rather late but windows OS was not ready earlier) and equipped with a whopping 64 gigabyte of RAM. Again, not really a surprise given the database centric scalability of myspace which was used for a long time. The experience of moving towards a 64 bit architecture was very good and it looks like databases can really use the advanced hardware now possible.

This is not the case e.g. for a huge java VM running on such a box which would spend hours in garbage collection.



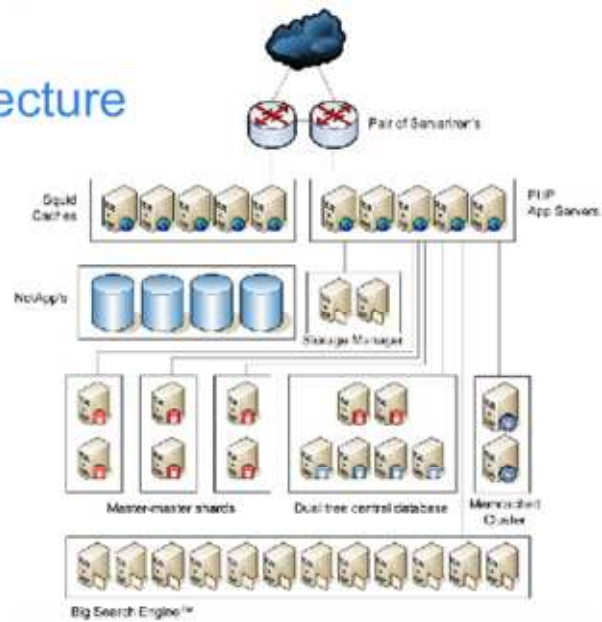
Flickr

[Hoff], [Henderson]

Again picked for its huge multi-media load together with community functions. A rather detailed list of features on highscalability.com.

- API
- Search
- Load Balancing
- Database Org.
- Master-master shards

Flickr Architecture



From Henderson, Scalable Web Architectures,

Throughput numbers: 40000 pictures, 100000 cache operations, 130000 database queries
Per second!

Flickr is based on the LAMP stack with the main focus on the data store (“push problems down the stack” approach). This principle is used in sessions as well: no state is best. Keep session state in cookies. Do NOT store local sessions on disk (even memory is bad). Henderson suggests centralized sessions (what is the difference?) Pull additional information from DB but avoid per page queries. But Henderson claims that scaling the DB is hard.

Loadbalancing: hardware expensive, software solutions tricky but nice with group communication providing virtual IPs with fail-over. Also layer 7 dispatching on hashed URLs, e.g. to cached pages in different cache servers (CARP)

Asynchronous queuing: some tasks take time and need to be done asynchronously. Image resizing e.g.

Relational data: best is to buy bigger hardware. Due to 90/10 ratio of reads to writes master – slave replication is OK. Flickr does 6 reads per write. But writes do not scale!

Caching: watch out for invalidation problems with shared memory. MySQL query cache gives bad performance (not the only ones who say so). Every write flushes the cache. With 10 reads per write there is no chance that cache values can be re-used.

Write-through and write back problems (mention Birman's discussion of distributed filesystems, NFS does have consistency problems with

caching). We need a caching strategy list. Sideline caching with memcached where application writes to DB directly and then updates cache.

High-availability. Master-Master replication, collision problems, schema design to avoid collisions. Replication lag, auto-increment problem (some are saying don't ever use it..)

Data Federation: vertical partitioning of tables which do not need to be joined. Split tables e.g. into primary objects (users) and store a reference to those primary objects in a central lookup table which says in which cluster the user data are stored. Migration problem between shards, locked data structures needed. (bucket-split approach used by wordpress does not need central lookup – the association between cluster-server and user data can be calculated from the splits. Needs good numbering scheme for users and application logic gets complicated in case bucket size varies due to machine differences.

And it raises the number of db connections needed per page creation. Facebook approach: try to keep a user and his friends together on one shard – but how do you know where to split? Dual tables create joins across separate tables? Simply duplicate the data and let the application logic deal with the two updates needed in case of changes... Do not use distributed transactions, accept inconsistencies and catch them over time with repair tasks running in the background.

Multi-site HA: most use master for write and backup sites for read only. SPOF for writes but hot/hot or master/master is very hard. Master/Master trees for central cluster? AKADNS like service with Akamai managing your domain (latency, load split). Are more smaller datacenters really cheaper than two big ones? (compare with Theo Schlossnagle's local DNS solutions for short latency requests) (Amazon is opening a CDN in 3 continents where static data will be available from S3)

File Serving: easy, many spindles needed. In memory, not on disk. Limits. Invalidation logic: use new URL after changes, avoids stale cache entry. CDN: cache invalidation problem. Push content to them or they reverse proxy you. Virtual versioning: Requests contain version and are stored with version information within a cache. Mod_rewrite converts versioned URL to path.

Authentication: permission URLs (waterken?) embedded tokens, self-signed hash which can be checked without going to DB. Invalidation of permission URLs is tricky – needs automatic expiration.

File Storage: stateful == bad. Move the problem up the stack again – do you need RAID with collocated data? File size – does it make a difference? Flickr filesystem without meta-data. Apps hold meta-data in special servers.

<<presentation on redmine>>

master-master shards
meta-directory für php
indirection, transparency lost but cachable
dual-tree central DB (master/slave)

master-master plus read slaves
db schema questions: where do comments go?
de-normalization
per features assessment

filesystem: picture
read/write separation, much more reads, parallel?
meta-data and write in master only, apps must take care of meta-data
special filesystem

session state in cookie, signed
good loadbalancing and availability
load spikes: first day of year plus 20-40 percent

solution: dynamically turn off features (195)
user visible transparency break (diagram)
needs monitoring
cal henderson: no math, no pre-calculations, measure and monitoring,
calculate trend,
peaks due to catastrophes
lessons learned:
feature, query awareness,
money?
backup: 1,2,10,30 days

Facebook

PlentyOfFish

From [Hoff]

- supposedly run by only one person??
- Business model and technology
- Storage and growth
- Database strategy
- Click Through Rates and advertising
-

Twitter – “A short messaging layer for the internet (A.Payne)”

[Hoff], [Blaine]

A Rails application!

Stores images for more than one million users on Amazon S3.

*Twitter's approach to solving their performance and scalability issues is a great example of thinking big while taking small steps. The team set about **iterative removal of bottlenecks**. Firstly they tackled multi-level caching (**do less work**), then the message queuing that decouples API requests from the critical request path (**spread the work**), then the distributed cache*

(memcached) client (do what you need to do quickly). Evan was asked about strategic work to take them to the next 10x growth. His responded that they were so resource constrained (can you believe there are only 9 in the services engineering team) and so under water with volume that they have to work on stuff that gives them most value (take small steps). But crucially, they create solutions to the bottlenecks making sure that whatever they fix will not appear on the top 3 problem list again (which is thinking big - well, as big as you can when you're growing like a hockey stick). <http://apsblog.burtongroup.com/2009/03/thinking-big-and-taking-tweet-sized-steps.html>

<http://qconlondon.com/london-2009/presentation/Improving+Running+Components+at+Twitter>

The following is an excerpt from an interview by Bill Venners with Twitter engineers [Venners]:

The concept of iterative removal of bottlenecks also applies to the way languages were handled at Twitter. Ruby was used both in the web front-end as well as the backend of Twitter. While the flexibility of Ruby was appreciated in the front-end it showed certain deficits in the backend:

- Stability problems with long-lived processes like daemons
- Excessive memory use per Ruby process and bugs in the garbage collector
- Missing optional types in the language (like soft-guards in E). The developers noticed that they were about to write their own type system in Ruby. This is the opposite to what developers using statically typed languages notice: that they are writing their own dynamic mechanisms over time. Probably a good argument for both mechanism in a language.
- No kernel threads in Ruby and therefore no way to leverage multi-CPU or multi-core architectures besides running several Ruby runtimes in parallel (which is the typical advice given for developers in Erlang runtimes, E vats and all other single-threaded runtimes but did not work due to excessive memory use by Ruby). The developers were willing to sacrifice some consistency and comfort for more threads. Scala with its shared nothing architecture.

What does this tell us about language use in ultra-large scale architectures? Language does both: it does not matter (lots of different languages used in those sites) and it matters a lot (with growth some language concepts are no longer optimal and need to be replaced by different concepts). Ideally the same language would be able to offer alternative programming concepts. And finally: the stability of an “old” virtual machine like the JVM is not to be scoffed at.

Interestingly the Twitter developers reported also that adding functional concepts provided to be very useful. They learned to appreciate “immutability” when later on they changed some functions back to shared state multithreading because they noticed that not all problems were well suited for the use of functional actors. And finally they learned that it is beneficial to test new technologies in important but perhaps not mission critical areas first.

Digg

Google

- Sorting with MapReduce
- Storing with BigTable
- Will discuss both later in algorithms, together with the API to the engine
-

YouTube

Picked for its huge multi-media load due to video serving.

Again a short wrap-up by Todd Hoff on the architecture [Hoff], based on a google video.

- sharding
- dealing with thumbnails
- caching strategy
- Video Serving
- CDN use
- Replication solution

Amazon

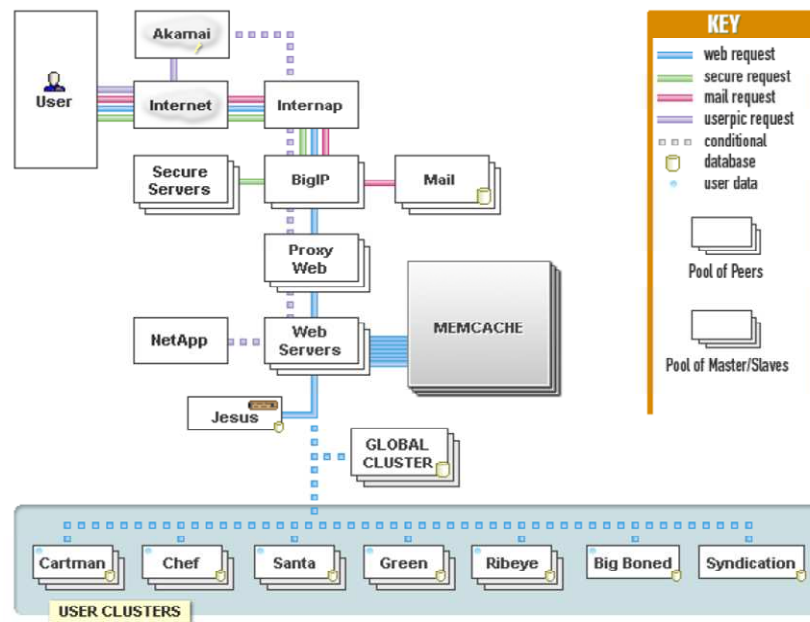
[Hoff]

- service architecture
- framework haters
- shared nothing
- eventually consistent

we will discuss the EC2 architecture later.

LiveJournal Architecture

Probably one of the top influential sites (memcached etc.). Good presentations available by Brad Fitzpatrick of Danga.com



LavaBit E-mail Provider

(for the excellent questionnaire, their DB scaling approach and the difficulty to scale-out a single server application like e-mail, the problems of web-mail with IMAP and no client caching, wrong read granularity etc.)
 [Xue], [Levison]

Stack Overflow

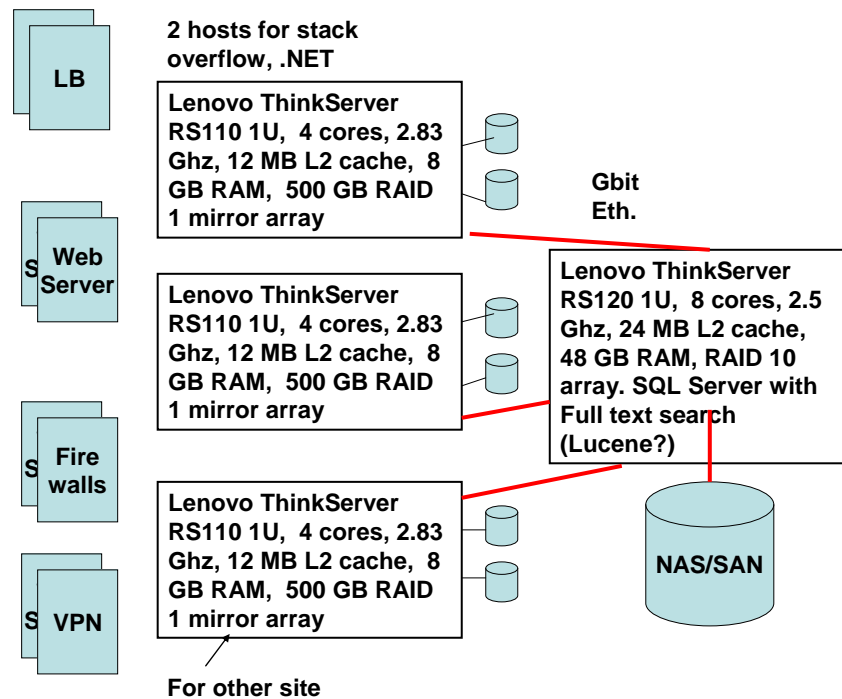
[Hoff] Stack Overflow Architecture,
<http://highscalability.com/stack-overflow-architecture>
<http://blog.stackoverflow.com/2008/09/what-was-stack-overflow-built-with/>

[Atwood] Jeff Atwood, Scaling Up vs. Scaling Out: Hidden Costs
<http://www.codinghorror.com/blog/archives/001279.html>

If you need to Google scale then you really have no choice but to go the NoSQL direction. But Stack Overflow is not Google and neither are most sites. When thinking about your design options keep Stack Overflow in mind. In this era of multi-core, large RAM machines and advances in parallel programming techniques, scale up is still a viable strategy and shouldn't be tossed aside just because it's not cool anymore. Maybe someday we'll have the best of both worlds, but for now there's a big painful choice to be made and that choice decides your fate. [Hoff] Stack Overflow Architecture

The quote by Todd Hoff on the architecture of Stack Overflow shows why I am discussing it: Stack Overflow is a medium sized site, built on the Microsoft stack (like PlentyOfFish) and it does use scale-up instead of scale-out like most of the other sites here. There are some subtle

dependencies between hardware solution, software and finally administration costs which become clear when we take a look at the components used by Stack Overflow [Atwood] to achieve 16 million page views a month with 3 million unique visitors a month (Facebook reaches 77 million unique visitors a month) [Hoff]



Jeff Atwood mentions a couple of very interesting lessons learned in the context of a scale-up solution based on mostly commercial software (ASP.NET MVC, SQL Server 2008, C#, Visual Studio 2008 Team Suite, JQuery, LINQ to SQL, Subversion, Beyond Compare, VisualSVN 1.5):

- Scale out is only possible with open source software, otherwise the license costs are just too high.
- Administrating your own servers is necessary because providers are unable to do so
- Go for maximum RAM size because it is the cheapest way to scale
- High-speed network equipment in the context of fault-tolerance is a huge cost factor (load balancers, firewalls etc.)
- DB Design done wrongly (copied from wikipedia) needs refactoring due to the large number of joins needed. Even a DB that is mostly in memory cannot do many joins. Go for a joinless design (BigTable approach).

We will take a look at the DB design of Stack Overflow in the section on DB partitioning to see what went wrong.

(http://sqlserverpedia.com/wiki/Understanding_the_StackOverflow_Database_Schema)

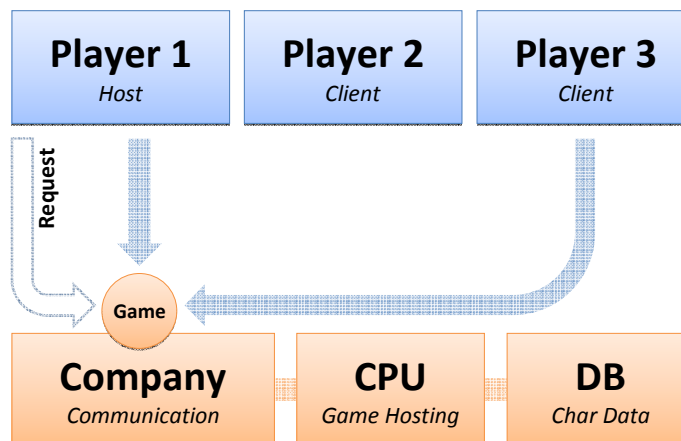
There is much more to learn from Stack Overflow. Jeff Attwood compares scale out architectures with the decision of PlentyOfFish to buy a monster

HP Proliant server and comes to some surprising conclusions when ALL costs (power, licenses etc.) are calculated. We will discuss this in the chapter on datacenter design where we will take a look at extra costs incurring due to centralization as well.

Massively Multiplayer Online Games (MMOGs)

“You have gained a level” was the title of a recent special edition of the gamestar magazine focussing on the evolution of MMOGs. And there is no doubt that MMOGs have grown up quite a bit. This is not only reflected in the quality of the graphical representations e.g when groups of 50 participants get together to perform a raid against some common enemy monster. The sheer numbers of participants which go into the hundreds of thousands concurrent users and several millions of subscribers show the social acceptance of online gaming as a hobby. How serious in online gaming? There is real money made from selling characters or equipment through auctions for example. This has rather large ramifications for the technological base of those online games: once serious money depends on the correct storage of game state transactional features become very important. Gamers hate to lose anything due to server crashes. And having said the “S-word” already it is clear that most MMOGs today are client/server based due to security reasons (cheating is a major concern for gamers) and also because it allows for a rather attractive business model: montly payments from gamers.

<<stiegler: c/s model>>



This means the architecture is a rather traditional client server model. Peer-to-peer approaches to gaming are discussed as well but there seems to be currently no way to run the same numbers of users on those architectures – not to mention the fact that with a client server architecture the company running the servers has an

easy way to bill clients for the services. Also, game companies fear cheating in peer-to-peer systems.

Even features like teamtalk – realtime communication between game participants seem to require servers at this moment.

Support for collaboration between participants depends on each game. Some games allow the development of new worlds by players. All of the games allow collaborative actions like raids. Some games nowadays use peer-to-peer technologies like bittorrent to download patches and upgrades to player machines.

The data exchanged during a game are surprisingly small. Sometimes only 50 to 60 k are needed to synchronize actions – a good fit to modem connection lines.

Cheating is a problem. Every bug will be exploited by players as well. Sometimes the game servers can detect manipulation or the use of illegal client side software and a player might get punished (put into jail or in the worst case lose the game figure that had been created spending countless hours with the game)

- social superstructures (money, love, guilds, cheating)
- collaboration (raids, development of worlds by users/groups), hotspots, flash-crowds
- extra channel communication (direct talk)
- patches and downloads via p2p
- serverfarms, clusters, proxies, worldserver, db-server, user splitting
- communication data
- large numbers (players, servers, money)
- distribution on content level through worlds.
- Transactions and availability

What is the current architecture of large online games like Everquest from Sony? The Game is divided into so called worlds which provide an easy way to split workload. More than 1500 servers worldwide run the game, split into cluster of 30 Machines per world. There are 3 types of server machines: proxy servers (doing fast calculations), world servers (holding world wide state and database servers which store the persistent state of each player and world.

But let's hold on a bit before diving into the technical details and take a look at something that it perhaps even more important for MMOGs than the vast technical arrays of server machines: A clear concept of "content mapping" onto the available hardware. This is basically simply a form of partitioning only that in the MMOG areas this partitioning is an intricate play between game story and content and physical hardware and its networking. An according to [Scheurer] if we talk about database sharding today we are using a core concept from game design.

On Shards, Shattering and Parallel Worlds

The evil wizard Mondain had attempted to gain control over Sosari a by trapping its essence in a crystal. When the Stranger at the end of Ultima I defea

ted Mondain and shattered the crystal, the crystal shards each held a refracted copy of Sosaria.

[http://www.ralphkoster.com/2009/01/08/database_sharding_came_from_uo/] (found in [Scheurer])

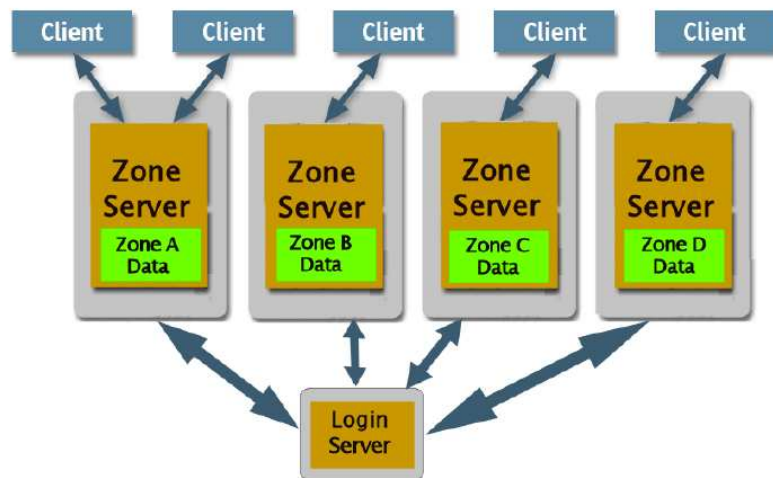
A shard is a copy of the game world. Players belong to a certain copy only and this shatters the world into different copies. Because of the binding of users to shards there is no single-world (or continuous world) illusion. Users are aware of the split nature of the world, especially if they can even experience some IT-related evidence like the need for server transfers. So why do MMOGs like WoW use shards at all? Sharding or shattering allows for very efficient mapping between game content and server infrastructure. It can happen on different levels of the game world: copies of huge parts of the world are called realms, copies of small sections are called instances. The latter shows that not only content but also actions can be mapped to separate infrastructure elements.. The smaller the section the better it is suited for high-performance action like PvP (player vs. player) because the number of participants is limited.

But as Scheurer points out, shards need not only be considered a game deficit due to technical necessities. Sometimes having different copies of a world allows players to change the world, e.g. after running into social problems within a certain shard. Different play modes of a game can be represented with shards as well e.g. fighting vs. non-fighting. And finally, games can cover the sharding on the content level e.g. by embedding the shards as kind of parallel worlds within the game story.

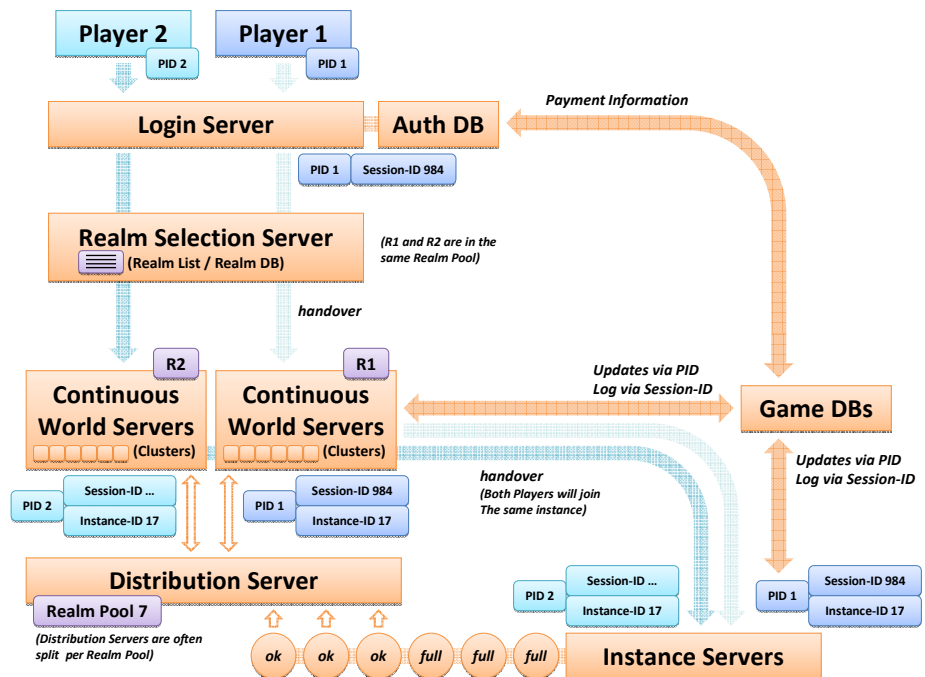
So how does a sharded architecture look like? The following gives some examples.

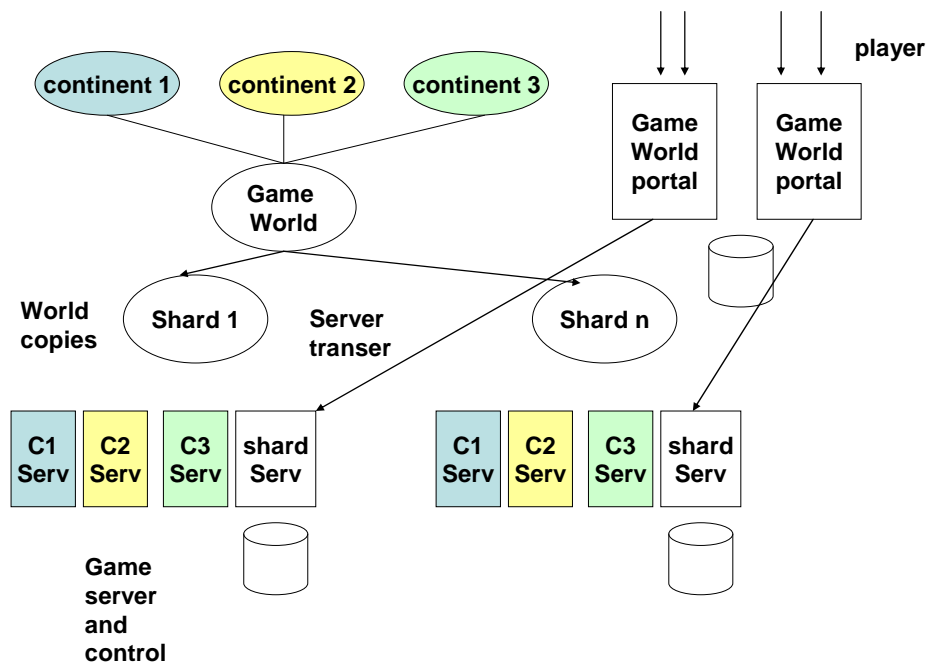
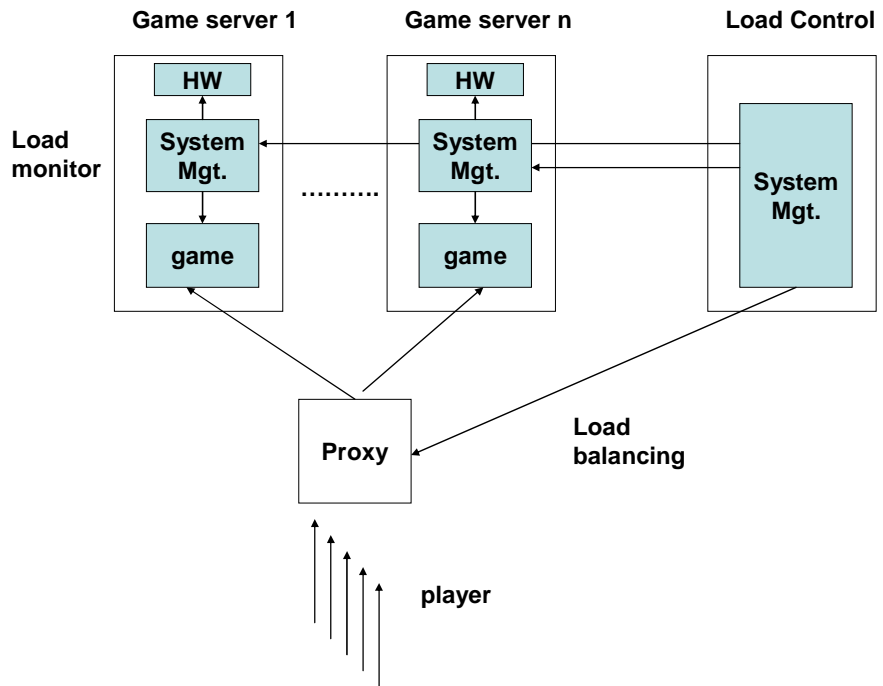
Shard Architecture and visible partitioning

Sharded Architecture

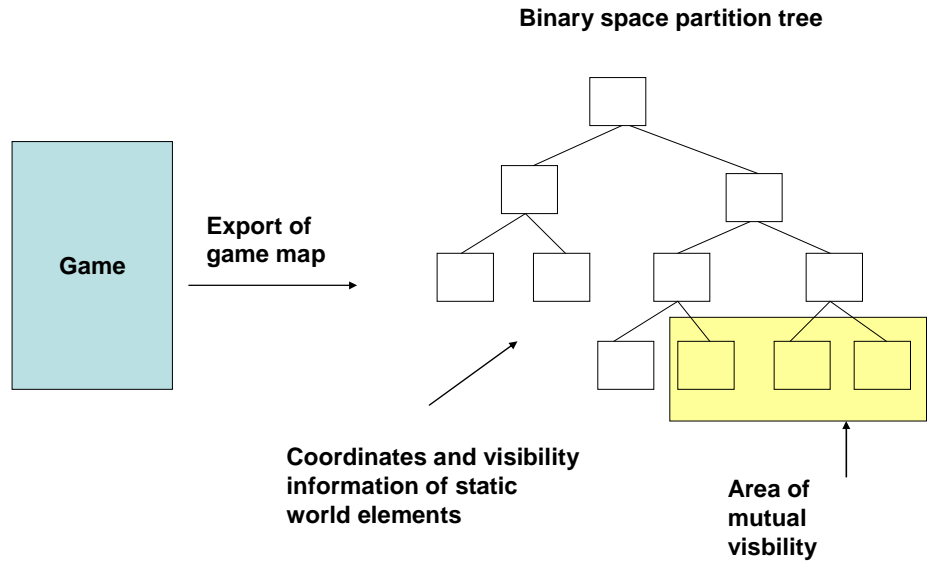


From: Project Darkstar, Sun

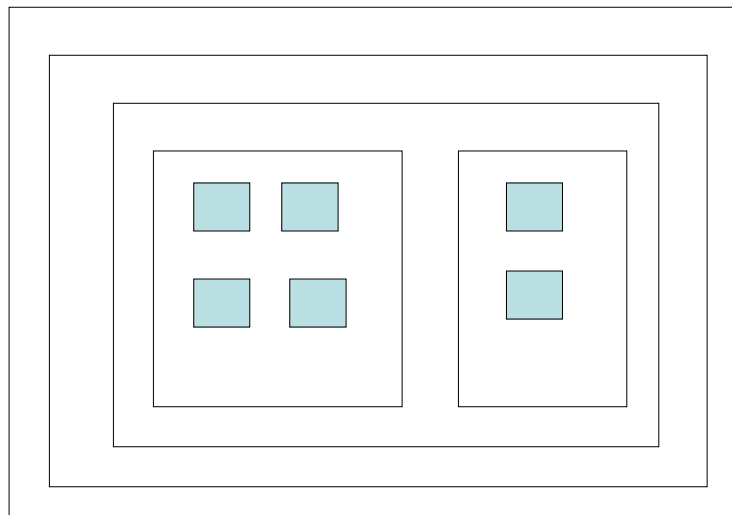




Shardless Architecture and Dynamic Reconfiguration

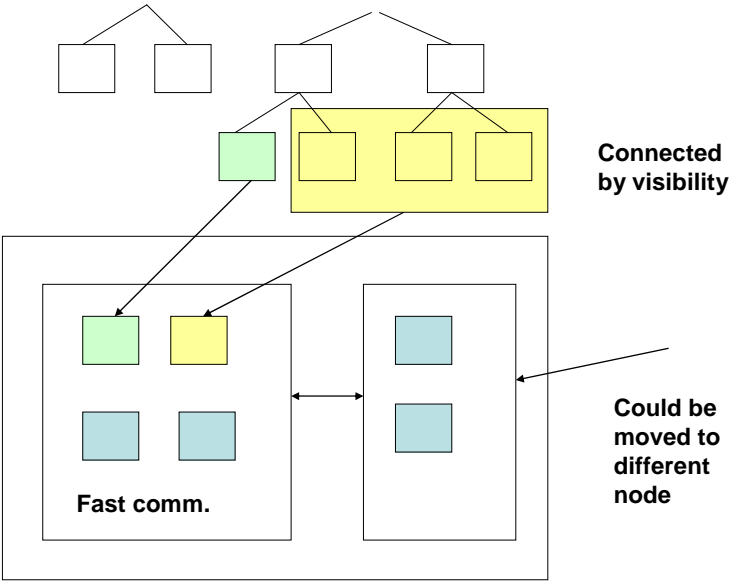


Grid node computing and administrative elements

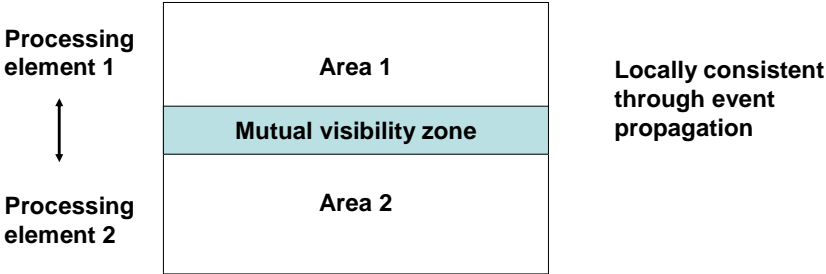


Borders: in-process, inter-process, inter-virtual-node, inter node

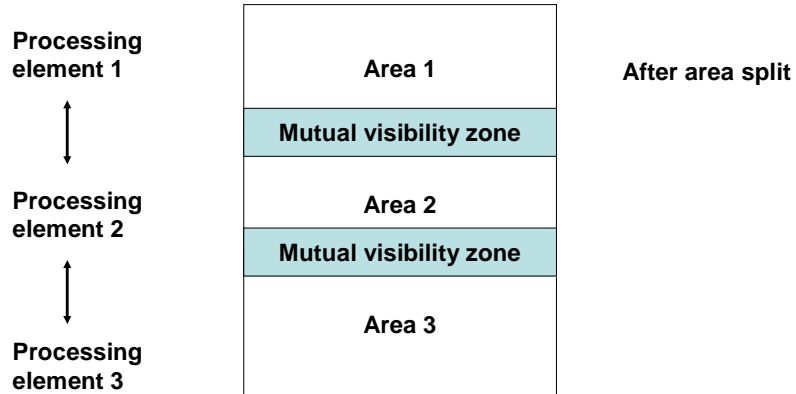
Static bsp to compute grid mapping



Dynamic reconfiguration of partitioning based on local inconsistency and static visibility regions



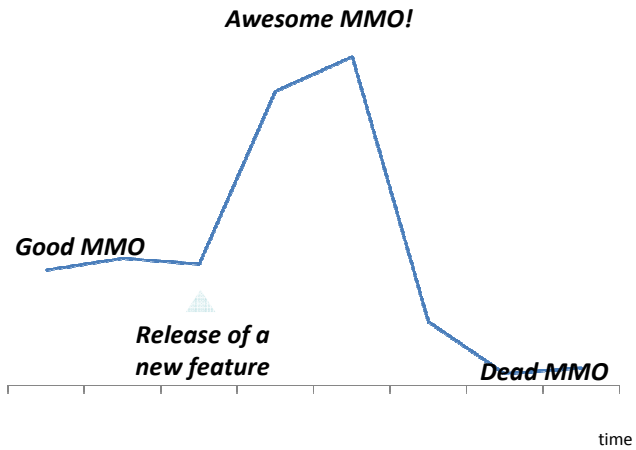
Dynamic reconfiguration of partitioning based on local inconsistency and static visibility regions



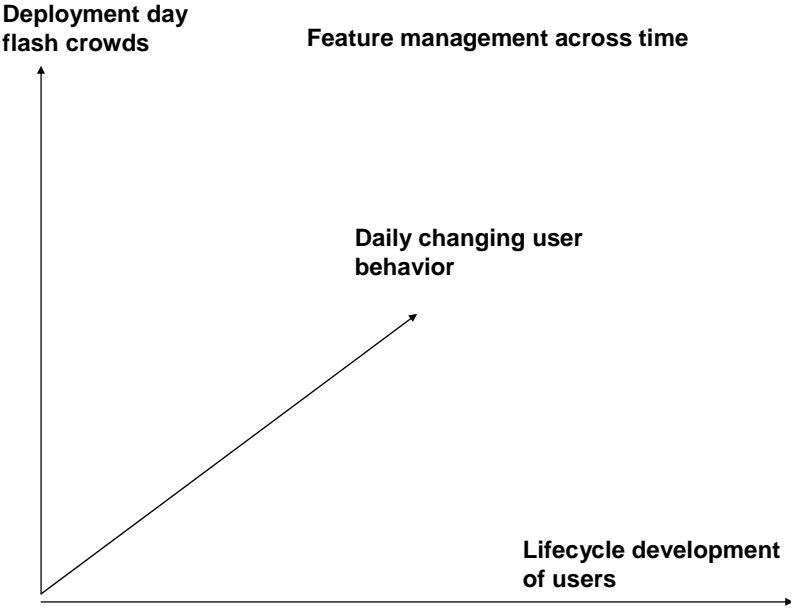
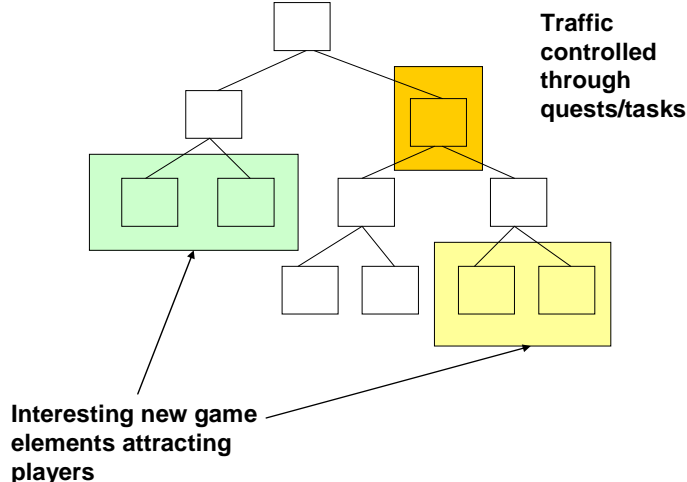
Feature and Social Management

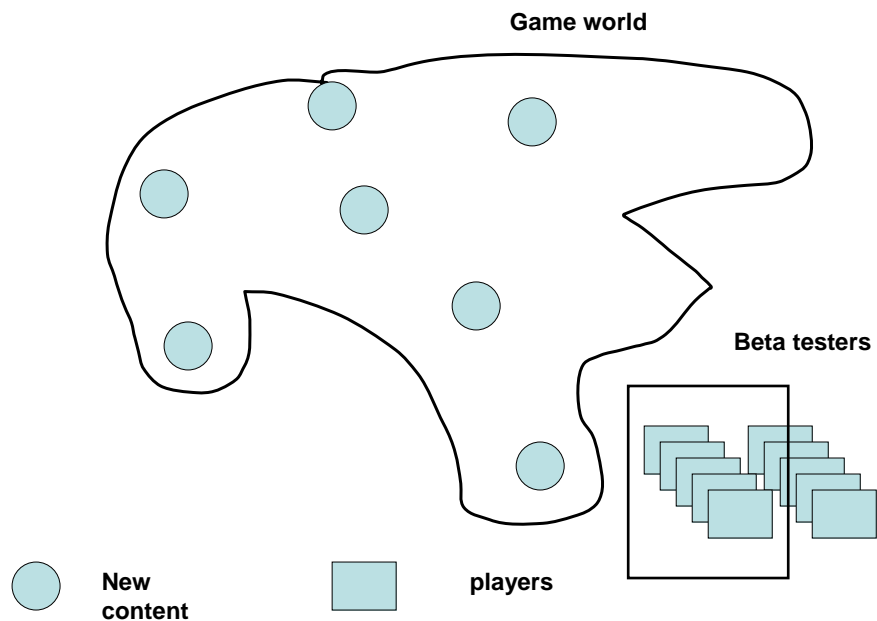
[Stiegler] people spreading, (e.g. instances)

“dead MMO slide”: temporal content mapping needed



Content level: game balancing to avoid flash crowds and hotspots





Security in MMOGs

<<rene Schneider on tainting in WOW, Kriha on attacking games, Secondlife copy bot etc., EU study>>

Methodologies in Building Large-Scale Sites

- open source
- instrumentation
- customization
- multi-component
- tracing and logging
- no special language
- permanent changes
- second order scalability (extension, rebuild in case of crash)
- cheap hardware
- no licenses

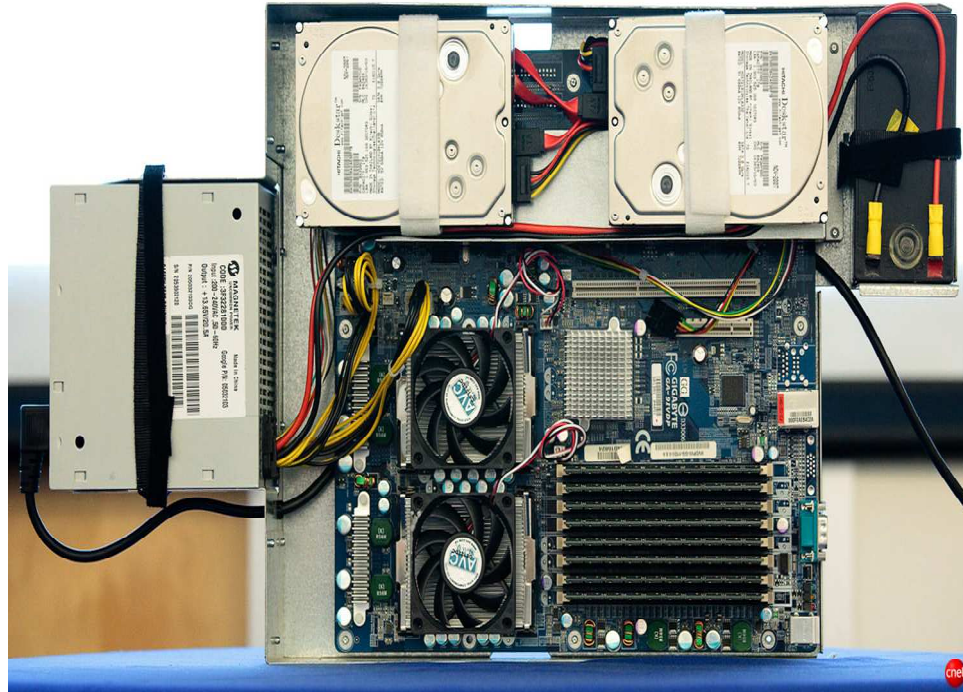
You are not going to build an ultra-large scale site? Does this mean the following does not matter to you? Think again. The goal of this exercise is to create awareness for possible scalability problems and the concepts for solving them. Even in smaller applications you will then be able to identify possible bottlenecks quickly and design a scalability path up front.

Limits in Hardware and Software – on prices, performance etc.

- DB table sizes possible? Connection numbers and multiplexing options?
- server failure rate of 3.83%
- Google query results are now served in under an astonishingly fast 200ms, down from 1000ms in the olden days
- [100s of millions of events per day](#)

- servers crammed with 384 GB of RAM, fast processors, and blazingly fast processor interconnects.
- 1TB of RAM across 40 servers at 24 GB per server would cost an additional \$40,000.
- 1U and 2U rack-mounted servers will soon support a terabyte or more of memory.
- RAM = High Bandwidth and Low [Latency](#). Latency always underestimated.
- a cluster of about 50 disks has the same bandwidth of RAM, so the bandwidth problem is taken care of by adding more disks.
- bandwidth of RAM is 5 GB/s. The bandwidth of disk is about 100 MB/s.
- Modern hard drives have [latencies under 13 milliseconds](#). When many applications are queued for disk reads latencies can easily be in the many second range. Memory latency is in the 5 nanosecond range. Memory latency is 2,000 times faster.
- while application processing can be easily scaled, the limiting factor is the database system.
- Even the cheapest of servers have two gigabit ethernet channels and switch.
- I'd much rather have a pair of quad-core processors running as independent servers than contending for memory on a dual socket server.
- MySQL scales with read replication which requires a full database copy to start up. For any cloud relevant application, that's probably hundreds of gigabytes. That makes it a mighty poor candidate for on-demand virtual servers.
- Max. 250 disk writes per second and disk.
- 15000 writes/sec against memcachedDB
- Kevin rose has 40,000 followers. You can't drop something into 40,000 buckets synchronously. 300,000 to 320,000 diggs a day. If the average person has 100 followers that's 300,000,000 Diggs day. The most active Diggers are the most followed Diggers. The idea of averages skews way out. "Not going to be 300 queries per second, 3,000 queries per second. 7gb of storage per day. 5tb of data across 50 to 60 servers so MySQL wasn't going to work for us.
- Queries per second?
- SATA drives: problem of "silent read error" where a read returns less data than requested. [Webster]
- HDTV means a seven-fold increase in bandwidth and system storage required
- Streaming video delivery: data rates up to 1.2 gigabytes/sec in 4k Non-linear-editing formats [Coughlin].
- Disk cache: 32Mb-64Mb
- (<http://www.heise.de/newsticker/Serverfestplatte-mit-64-MByte-Cache--/meldung/136501>)
- Cloud Storage numbers (comparison Rackspace vs. EC2)
- IOPS numbers

See also the Google hardware description on CNet and Heise (the h2 unit)



From: Stephen Shankland, CNet, http://news.cnet.com/8301-1001_3-10209580-92.html

A History of Large Scale Site Technology

Todd Hoff started a list of technological breakthroughs in the development of large scale sites in his article on cloud based memory. [Hoff]

- *It's 1993: Yahoo runs on FreeBSD, [Apache](#), [Perl](#) scripts and a SQL database*
- *It's 1995: Scale-up the database.*
- *It's 1998: [LAMP](#)*
- *It's 1999: Stateless + Load Balanced + [Database](#) + [SAN](#)*
- *It's 2001: In-memory data-grid.*
- *It's 2003: Add a caching layer.*
- *It's 2004: Add scale-out and partitioning.*
- *It's 2005: Add asynchronous job scheduling and maybe a distributed file system.*
- *It's 2007: Move it all into the cloud.*
- *It's 2008: Cloud + web scalable database.*
- *It's 20???: Cloud + Memory Based Architectures*

Growing Pains – How to start small and grow big

Top down planning of scalable solution needs a clear goal and lots of money in the first place. Typical social sites start small. How exactly interact users, site-management, technology and infrastructure services provided by others to allow growth? Does a computing cloud where you can rent computing power and services really help a startup company? Option: visit a local social community site, e.g. from a radio/TV station?

Feature Management

Avoid flash crowds, distribute functions, spread your population.
Examples in the section on virtual worlds and MMOGs

Patterns and Anti-Patterns of Scalability

Think end-to-end first

Before you start with punctual scalability measures (e.g. sharding the database) you should have an overall architecture with the major tiers including caching layers

Meta is your friend

- using a meta-data directory up front for flexibility and virtualization (table of shards, director in media grid) Problem: as soon as clients directly connect to sub-level components the meta directory can no longer virtualize the connection.

Divide and Conquer

- sharding and partitioning. Problem: as soon as the shards scale no longer or the partitions do not fit anymore. Mostly caused by one variable exceeding all scales (e.g. power users on one shard, one big app on one disc makes the disk subsystem unbalanced)

- copy and replicated to allow many concurrent channels. Problem: how to keep the copies synchronized
- caching: Problem: how to invalidate copies, e.g. by generating new references instead of deleting old ones.
- Breaking transparency (from SAN to DB sharding to feature shutdown in case of overload)<<diagram>>

Parallel does it better

Request resources in parallel to avoid sequential access times – but remember that this will also increase your communication traffic and bandwidth needs and put load on many machines. Use a scheduler framework for this.

Same size same time

Build requests with roughly the same size and timing behavior following the RISC pattern in CPUs. Scheduling is better for equally sized requests.

Build Batches

Collect requests and send larger units. But respect the “same size same time” pattern as well.

- no harmless function: Have an architect look at every function that uses resources or crosses tiers. This includes especially also every form of query against the database. Even better is to calculate the effects of a new feature on the overall architecture across all layers and tiers (this again requires the existence of a canonical architecture diagram for your application). Make features switchable so you can turn them off in case of problems or overload.

Async is your friend

Do whatever is possible outside of a request. Per-process, parallel-process or post-process but do not use request time to calculate expensive things. Use a queue mechanism for safe deposit of asynchronous requests.

Profiling is your friend

Measure everything (requires instrumentation which requires open source in most cases)

Only 100% will do

This is an anti-pattern for scalability. Think about where you can cut corners by relaxing certain rules for consistency. The road is the goal (use eventually consistent algorithms wherever possible within the business goals.)

Performance problems are sand dunes – they wander

This is a lesson that is sometimes hard to accept: when you have fixed a specific performance bottleneck or problem, the problem immediately shifts to a different spot in your architecture. When you think twice about this effect it is a rather direct derivative of amdahls law: removing the bottleneck with the biggest impact simply turns the next bottleneck into the biggest one.

The following paper from facebook is a good example:

Real-World Web Application Benchmarking by Jonathan Heiliger explains why Facebook uses a custom testbed and approach and not e.g. SPEC [Heiliger]. An important statement of the article is that Facebook saw a major effect of the memory architecture of their platform. It shows how careful one has to be with statements on what gives good performance and throughput: it is very context dependent:

As a social network the Facebook architecture is far from common - even though it may look like a regular 3-tier architecture in the beginning. They keep almost everything in RAM using huge clusters of memcached and use many cheap UDP requests to get to those data. This means that their access paths are already highly optimized and different to e.g. Google with its big distributed file system. Only then will memory access time become the next big bottleneck. And it is a reminder that all things said about performance are relative to platforms and architectures and what fits the one need not fit the other.

Finally the paper shows that performance/watt is a critical value for datacenter use.

Integrate lessons learned at eBay: Randy Shoup, [Shoup]

Test and Deployment Methodology

- how to test concurrent systems
- how to generate load
- where to test: rapid deployment, production tests
- start with a single server in production

- quality aspects?
- A/B testing with split user groups
- dynamic feature enablement or shutdown
- tool development
- monitoring and profiling
- external testing (e.g. Gomez)

Client-Side Optimizations

(with Jakob Schröter, <http://www.slideshare.net/jakob.schroeter/clientside-performance-optimizations>)

Probably the best information you can currently get on this subject.

wir hatten vor zwei Wochen nach Ihrer Veranstaltung „Ultra-large scale sites“ schon mal über das Thema Frontend bzw. client-side performance optimization gesprochen. Also was man bei großen Webseiten beachten sollte, damit die Webseite nicht nur schnell auf den Servern generiert wird, sondern auch schnell im Browser dargestellt und ausgeführt wird. Gerade durch immer anspruchsvollere Layouts und mehr Logik auf den Clients (JavaScript, AJAX, Flash, ...) sehe ich dieses Thema als wichtigen Bestandteil der Performance-Optimierung an, welches leider zu häufig auch vernachlässigt wird, da sich viele Entwickler sehr auf die Serverseite konzentrieren.

Important Keywords:

order and position of loading files

- file references in <head> are loaded before page is rendered (so watch out, which files really need to be loaded)
- if possible, load additional files after DOM rendering
- load css before js files

optimize images

- PNG often smaller than GIF
- remove unneeded meta data in image files

avoid and optimize http requests (which also helps unloading servers)

- combine js/css files
- use image css slices (combine images to save http requests)
- use more than one host for serving files (CDN) due to 2-parallel-request-limit in most browsers
- avoid http redirects

shrink data

- gzip compression for html, xml, css, js...
- minify js/css (e.g. YUIcompressor, Dojo ShrinkSafe, ...)

intelligent browser caching

- use etag header
- use expire header
- use http 304 not modified

js performance

- reduce onload actions
- js best practices
- choose the right AJAX library

tools

- Yahoo's YSlow Firefox extension
- Yahoo's smushit.com (image compressor without quality loss)
- speed limiter for testing site performance (e.g. webscarab)

Soweit meine ersten Ideen. Yahoo ist sehr aktiv in dem Bereich, unter Anderem gibt es hier eine interessante Präsentation bzgl. Bildoptimierung:

<http://video.yahoo.com/watch/4156174/11192533>

Zum Beispiel wird auch genannt, dass bei eine Verzögerung von 500ms beim Laden der Google-Suchseite die Anfragen um 20% zurück gingen, oder bei Amazon 1% der Käufe aus blieben als die Seite 100ms langsamer geladen wurde. Dies zeigt, dass minimale Performanceunterschiede durchaus auch Auswirkungen auf das Geschäft haben.

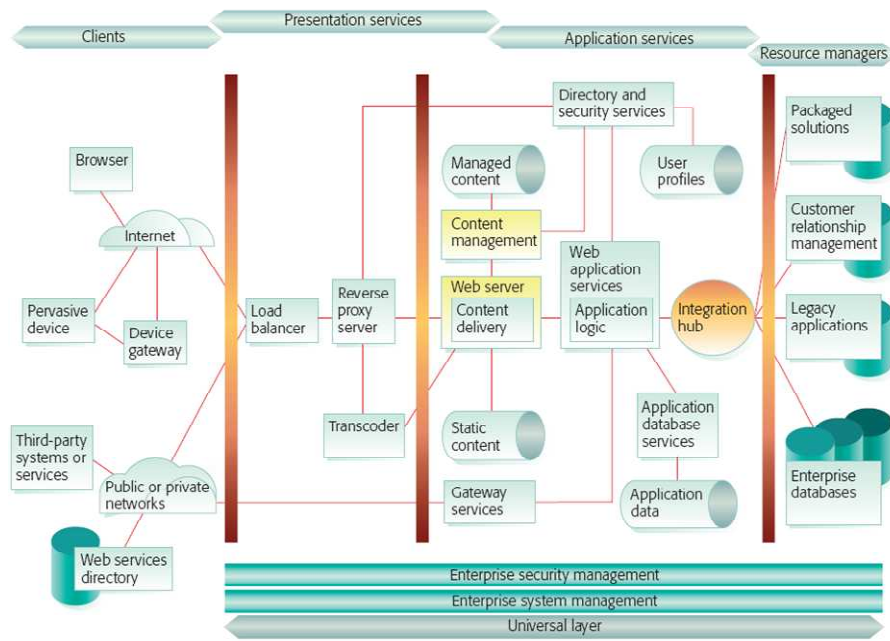
A Model for RASP in Large Scale Distribution

- SPIN/Promela
- canonical architecture
- queuing theory
- simulations
- failure tree models

After looking at the various large scale sites above we need to ask ourselves whether we are able to define some core architectures used by those sites. Let us start with the classic web site architecture to have something to compare to.

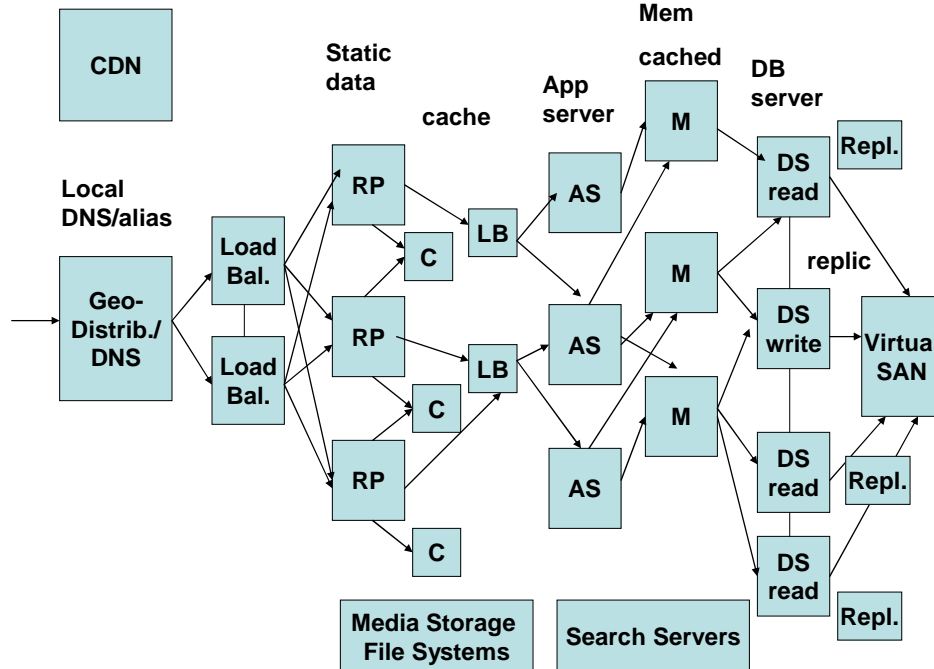
Canonical or Classic Site Architecture

Reception, user agent, distribution, processing, aggregation, retrieval, storage, global distribution, DNS aliasing, load balancing equipment, media storage, database setup and replication

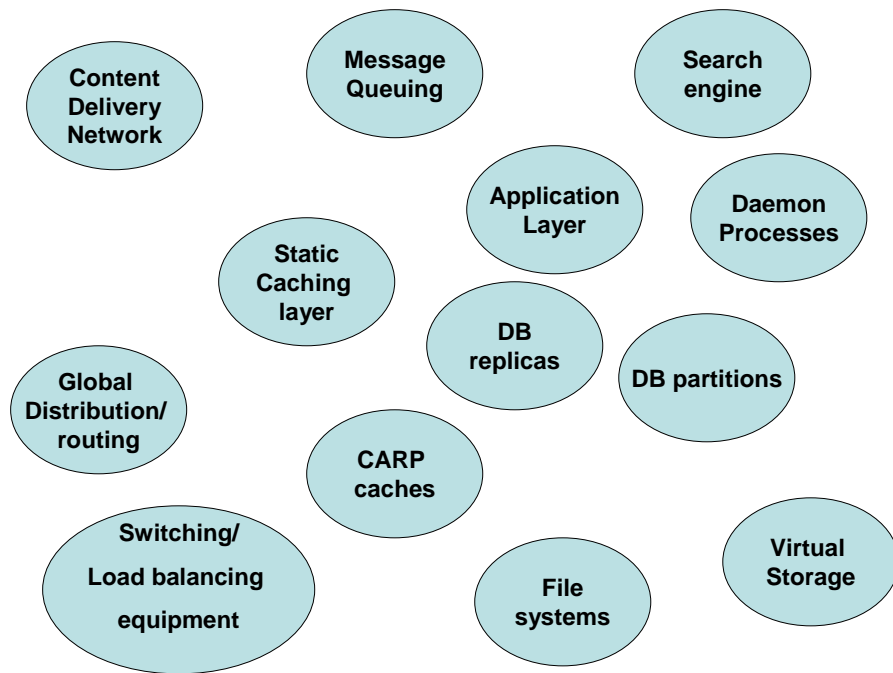


From: McLaughlin et.al., IBM Power Systems platform

For availability and scalability reasons lots of replicated components are needed as shown in the diagram below. Almost every layer needs load-balancing, switching and component replication.

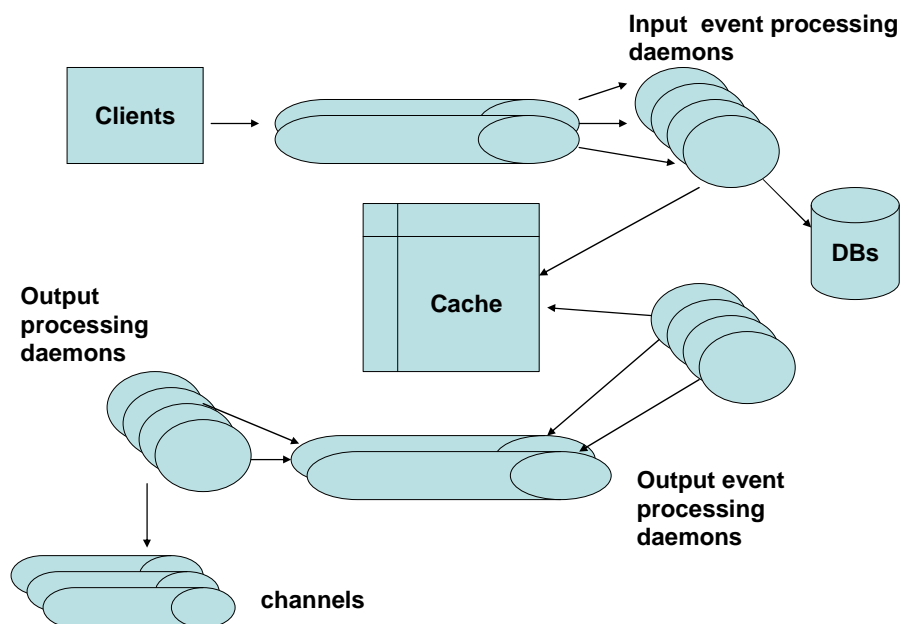


What are the conceptual building blocks for such sites? The diagram below lists some components used by ultra-large scale sites.



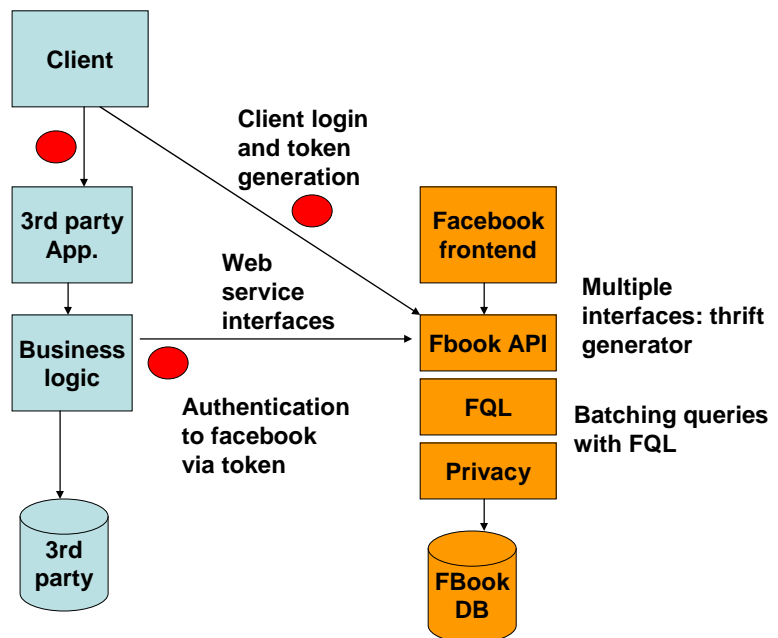
Classic Document-Oriented Large Site Architecture (Wikipedia)
Message Queuing System (Twitter)

Twitter seems to be ideally suited to be based on a message queuing paradigm with background daemons processing requests asynchronously and a huge cache holding messages.



Social Data Distributor (Facebook)

From Mark Zuckerbergs paper in “Beatuiful Architecture”.
 According to him Facbook went through several evolutionary steps which required new software technology. First came the realization that the social data within facebook needed to be shared with other applications. This meant opening up the business layer of facebook as a web services API through which 3rd party apps could access the social data in a secure and privacy respecting way. The service interfaces needed for different languages and protocols were generated from meta-data of the interfaces.
 To avoid having users offering their facebook credentials directly to 3rd party apps facebook developed a token based federated authentication system much like it was done by liberty alliance and others. Users still authenticated against facebook and received a token which could be presented to 3rd party applications for use in web service calls against the facebook API:



FQL was invented to reduce the number of requests from 3rd party applications for social data kept within facebook. Authentication is done like in liberty alliance with facebook acting as an Identity Provider. Lately facebook seems to allow OpenID authentication through other providers as well.
 Finally, with the Facebook Markup Language it is now possible to closely integrate 3rd party applications within the facebook portal. This allows excellent but controlled access to a users social data.

<<need to look at the open facebook system and ist data model>>
 <<OAuth now used for social plug-ins, see Heise article>>

Space-Based Programming [Adzic]

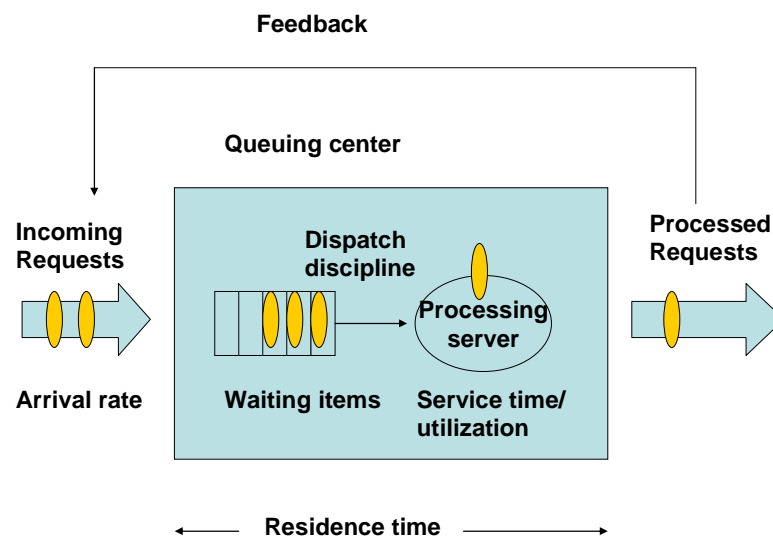
Queuing theory, OR

To guarantee the availability of a business solution the architecture needs to provide performance, throughput, decent response times etc. Those are all quantitative entities: the time it takes to service a request, the number of requests per second arriving or being serviced etc. Queuing theory is a way to analytically calculate request processing within systems as pipelines of queues and processing units. Queuing theory describes the interplay of two distributions of events: arrival rate and service time which are both in most cases assumed as exponential (random) distributions. The kind of distribution (random, constant etc.), their mean and standard deviation are the most important input values for queuing formulas.

Basic Concepts

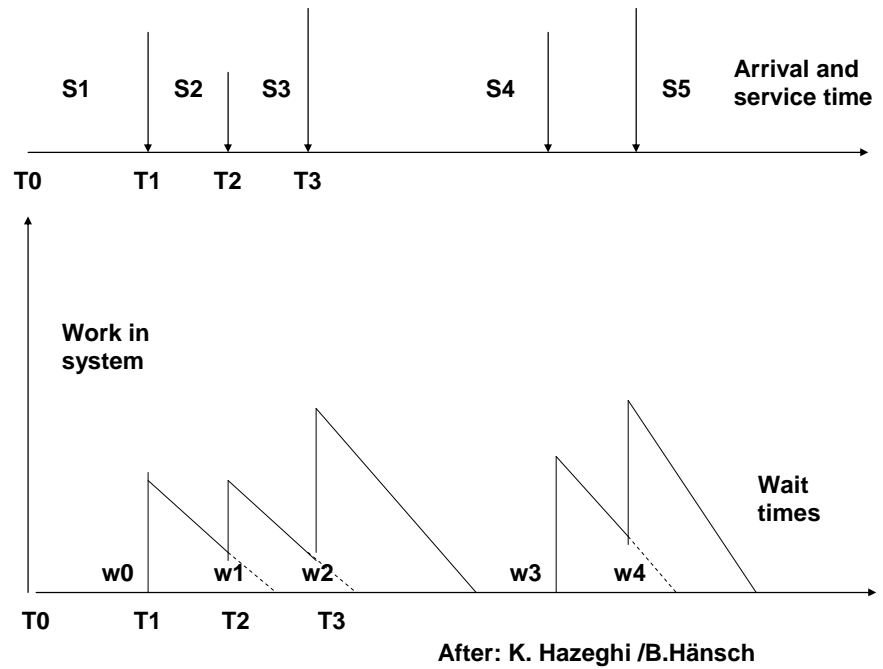
It is not the intention to provide a complete overview of queuing theory here. What we should look at whether this instrument is helpful in designing ultra-large scale sites. To do so we will first take a look at basic terms and laws of queuing theory and then think about their applicability in large scale design. The introduction is based on two papers: The application of queuing theory to performance optimization in enterprise applications by Henry H. Liu [Liu] and a paper on queuing analysis by William Stallings [Stallings]. We will also take a look at the “guerilla capacity planning” by Gunther.

The queue processing abstraction looks like in the diagram below:

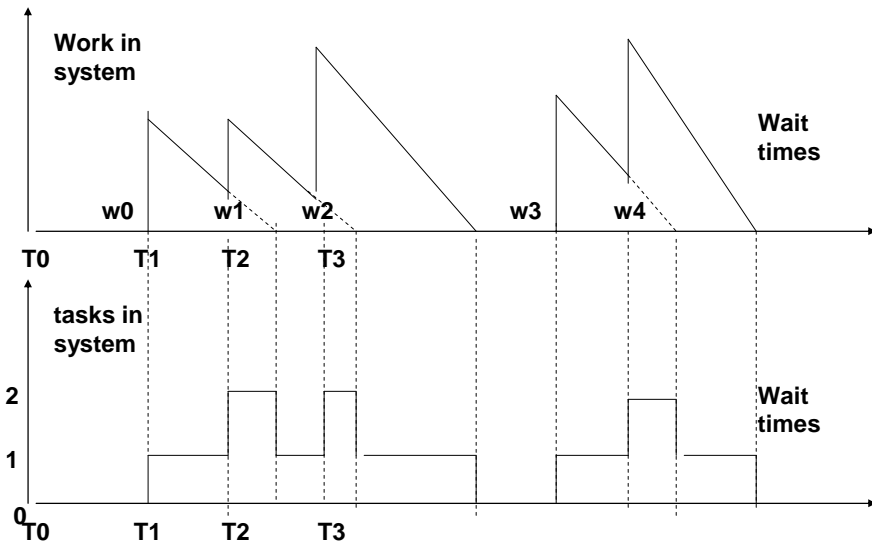


These processing elements can be connected to form process pipelines.

A different way to visualize queuing concepts is shown in the diagram below. Here new tasks arrive at times $T_1..T_n$ in the system. The tasks need different service times which is shown as a difference in length of $S_1...S_n$. The buildup of work in the system can be seen as the addition of lines in the lower half of the diagram. The dotted line crosses the x-axis exactly at the wait-time of the newly arrived service.



Task enter and exit behavior defines the overall number of tasks in the system at any moment:



After: K. Hazeghi /B.Hänsch

The terminology of queuing theory is very useful to describe the request flow through such architectures. The following list is taken from [Liu].

<<list of queuing theory terms>>

- **Server/Node** – combination of wait queue and processing element
- **Initiator** – initiator of service requests
- **Wait time** – time duration a request or initiator has to spend waiting in line
- **Service time** – time duration the processing element has to spend in order to complete the request
- **Arrival rate** – rate at which requests arrive for service
- **Utilization** – portion of a processing element's time actually servicing the request rather than idling
- **Queue length** – total number of requests both waiting and being serviced
- **Response time** – the sum of wait time and service time for one visit to the processing element
- **Residence time** – total time if the processing element is visited multiple times for one transaction.
- **Throughput** – rate at which requests are serviced. A server certainly is interested in knowing how fast requests can be serviced without losing them because of long wait time.

Generalized Queuing Theory terms after (Henry Liu)

“time” in this context always means an average value as all values here are of stochastic nature. Queuing Theory uses the so called

Kendall Notation to express the core qualities of queuing centers mathematically.

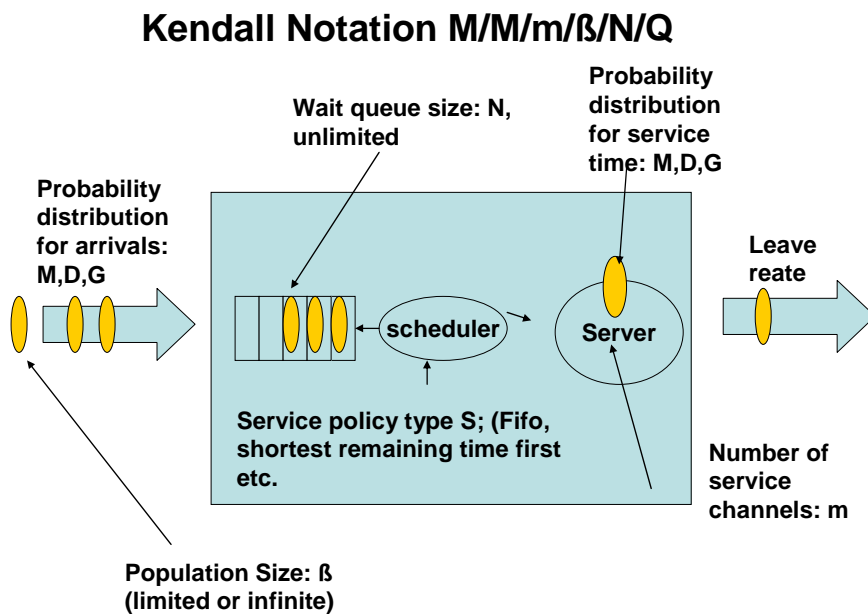


Table 1 Kendall notation ($\alpha/\sigma/m/\beta/N/Q$)

α	The type of probability distribution for the arrival process, e.g., <i>Markovian, General, etc.</i>
σ	The type of probability distribution for service time.
m	Number of servers at the queuing center
β	Buffer size or storage capacity at the queuing center
N	The allowed population size, which may be finite or infinite
Q	The type of service policy, e.g., FIFO.

After H.Liu

Specific versions of queuing models are expressed using Kendall notation like this:
M/M/m/N/N/FiFo which denotes: Markovian distribution of arrival and exponential service process distribution, the number of servers in the center, the wait queue size at the center and the population

size. The last parameter is the type of service policy. Buffer size and population size can be infinite in which case we are talking about an open queuing model usually denoted as M/M/1 model. Please note that the service distribution is assumed to be exponential which means that the system will show exponential degradation of service time in case of increased load. This is based on empirical observations and confirmed by many queuing specialists [Stallings].

Scheduling can either be fair (round robin, FCFS, FIFO) or unfair (shortest remaining processing time first, priority based) and pre-empted or run-to-completion. Pre-emption typically causes higher overhead due to cohesion costs (storing state, swapping images etc.).

Two important laws from queuing theory are statements about the performance of queue processing centers. The first one is Jackson's Law which states that it's really the *Service Demand*, not the *Service Time*, which is most fundamental to the performance of a queuing system. [Liu]. Service Demand is the average number of trips to the queuing node times the service time. Without feedback Service Demand is equal to service time.

The second, Little's law states that the number of requests both waiting and in service is equal to the product of throughput and response time.

<<simple questions and formulas>>

(b) Exponential Service Times (M/M/1)

$$r = \frac{\rho}{1 - \rho} \quad w = \frac{\rho^2}{1 - \rho}$$

$$T_r = \frac{T_s}{1 - \rho} \quad T_w = \frac{\rho T_s}{1 - \rho}$$

$$\sigma_r = \frac{\sqrt{\rho}}{1 - \rho} \quad \sigma_{T_r} = \frac{T_s}{1 - \rho}$$

$$\Pr[R = N] = (1 - \rho)\rho^N$$

$$\Pr[R \leq N] = \sum_{i=0}^N (1 - \rho)\rho^i$$

$$\Pr[T_R \leq T] = 1 - e^{-(1-\rho)T/T_s}$$

$$m_{T_r}(y) = T_r \times \ln\left(\frac{100}{100 - y}\right)$$

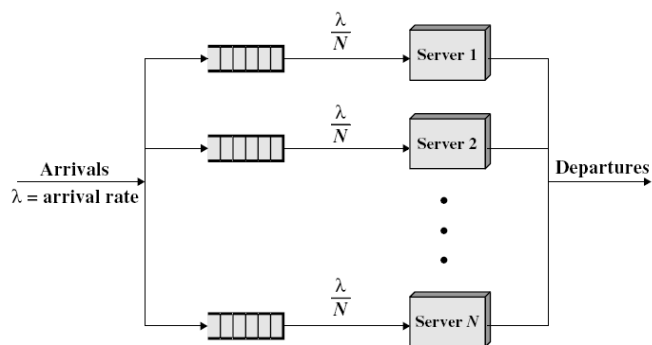
$$m_{T_w}(y) = \frac{T_w}{\rho} \times \ln\left(\frac{100\rho}{100 - y}\right)$$

After:
Stallings

Typical questions about queues are: what is the utilization of the processing element? (arrival rate x service time). How many items

are in the system at any time? (r) What is the response time? (Tr)
 Advanced questions are: How big must a queue be to not lose requests? (Is increasing the buffer size really a clever way to control your queuing system?)

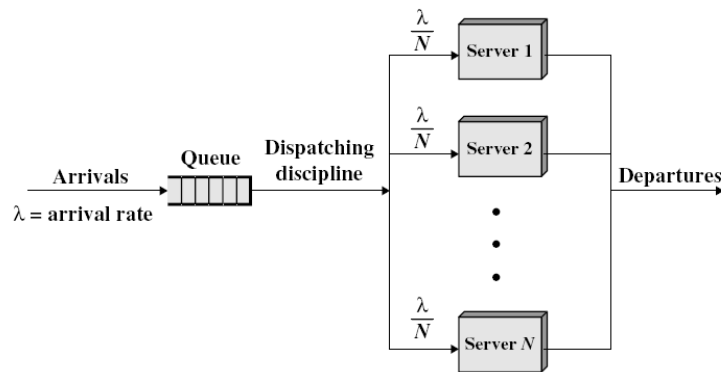
Queuing Theory can shed some light on everyday phenomena as well. Instinctively we do not like multiple single-server queues e.g. in banks or shops. Such queues force us to choose one and stick to it even if processing is rather slow in the queue we have chosen (aren't the other queues always faster?).



(b) Multiple Single-server queues

**After:
Stallings**

The global arrival rate λ is divided by the number of servers. Unfortunately this division is static and does not adjust for the situation within a server or between servers. In the worst case server 1 could be busy and server 2 could be idle without the chance for an item in server 1 to take advantage of this fact. Now let's compare this with a real multi-server queue:



(a) Multiserver queue

**After:
Stallings**

Here all servers get items from a single queue. Using queuing theory one can show that the multi-server queue allows a decrease in residence time by a factor of three and a decrease in waiting time by a factor of seven! [Stallings] The multi-server queue allows variations in the arrival rate to be compensated by variations in service time by different servers. It does not only avoid idle servers, it also distributes extreme services times more equally across all waiting items. We all know the ugly effect on a single server queue when one item causes a very long service time. This reduces the variation in response time.

<<single queue server or servers>>

While the difficulties of multi-queue server designs are obvious due to the independence of the queues (this does not mean that this kind of architecture is less important: it is heavily used in priority-scheduling service stations, see below) it is much harder to decide whether one queue with one fast server is better than one queue with multiple but slower servers). The following is based on B.Hänsch, Introduction to Queuing Theory [Hänsch]

First we need to calculate the utilization of a service system according to Little's law:

$$\rho = \lambda \cdot \frac{t_{BS}}{c}$$

$$\rho = \frac{\lambda}{\mu \cdot c}$$

Utilization calculated from arrival and service rate times number of channels. From [Hänsch]

This formula can be further refined to cover the effects of differences in variance of arrival and service rates.:

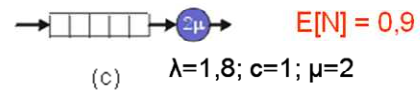
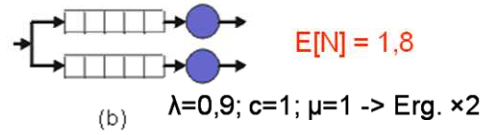
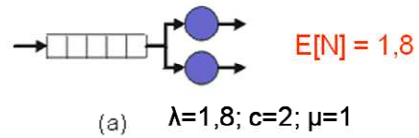
$$E[N] \approx \frac{\rho}{(1-\rho)} \cdot \sqrt{\rho^{(c+1)}} \cdot \frac{(v_A^2 + v_S^2)}{2} + \rho c$$

Expected number of jobs in the system. Rho means utilization. Variation in arrival and service rate is relevant.

From [Hänsch]

Now we distinguish three different cases of service stations and use arrival rate, service rate, variance of arrivals and services to calculate the expected number of tasks in the system according to the above formula.

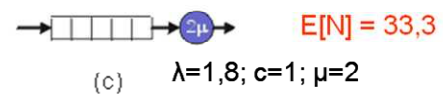
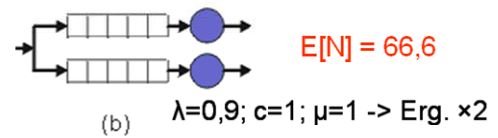
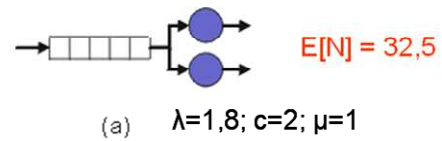
Mittlere gesamt Ankunfrate = 1,8,
 Mittlere einzelne Bedienrate = 1,0,
 Variationskoeffizient des Ankunftstroms $v_a = 0$
 Variationskoeffizient des Bedienprozesses $v_s = 0$



From [Hänsch]

A further increase in variance of arrivals and services turns the results around:

Mittlere gesamt Ankunfrate = 1,8,
 Mittlere einzelne Bedienrate = 1,0,
 Variationskoeffizient des Ankunftstroms $v_a = 2$
 Variationskoeffizient des Bedienprozesses $v_s = 2$



From [Hänsch]

Now the service station with more service units is slightly better than the single but faster server.

Are these results “physical” and what do they mean? Some observations and thoughts:

What needs to be explained is the big difference between two service units compared to one faster unit. What can really hurt a

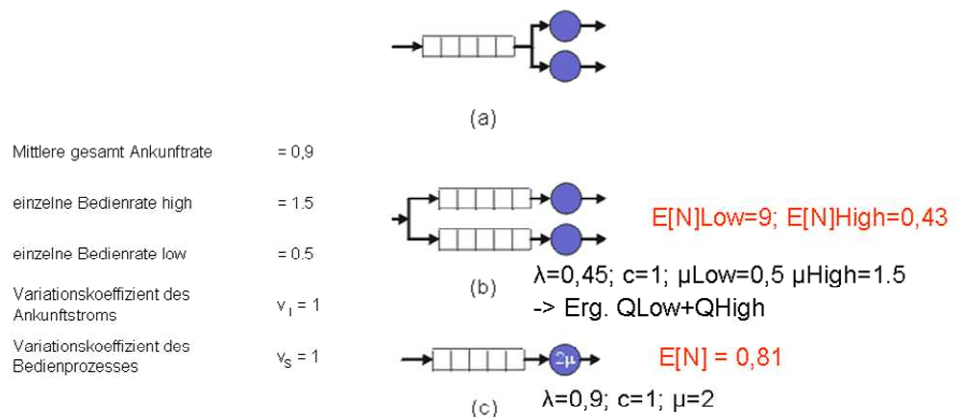
multi-service station? Probably the worst case that causes serious ineffectivity is when the two units are not fully utilized. Because we assume that the granularity of service is currently the complete task an empty unit cannot “help” the other which is busy. This is a well known anti-pattern in parallel systems: the granularity of job schedulings decide about utilization.

In our case the utilization is dependent on proper input being available – in other words the arrival rate is critical for the supply of tasks. With an arrival rate constantly below 2 and a service rate constantly at 1 per unit we see that both the dual service unit as well as the faster single service unit are constantly unterutilized. But why does this hurt the dual-service unit more? Only when the variance of this (negative) input and service behavior changes can we get a better utilization as is shown in the second diagram.

<<need to calculate $E[N]$ with different/higher arrival rate.>>. This observation fits e.g. to the design principles behind the Google Application Engine (GAE) which kills requests that take longer than 30 seconds to complete: if a task is the unit of dispatch its processing needs to be standardized to ensure utilization. Having many processing units and just one huge task does not increase efficiency.

The dual-service shows two critical phases: no input and not enough input to fill both units. But the second case (not enough input) is dependent on the granularity of requests: if we can make the requests small enough then both units should be able to run concurrently. Is the difference to the single service unit merely an artefact based on the assumption that the queue effectivity is determined by the number of concurrent requests in the service station? In any case it is important to realize that the variance in arrival rates is an important factor in multi-service unit designs. And this automatically leads to the idea of somehow turning the arrival rate into an optimum rate for such systems (see below: haijunka). And don't forget that the service distribution is exponential, leading to a sudden increase in service time in case of overload.

The second observation is that the differences between two slower vs. one faster unit are rather small. Even for the case with two queues with different priorities the difference between the high priority queue and both single queue models is very small but the decrease in effectivity for the slow queue is considerable!



From [Hänsch]

This raises the question whether prioritization really is a useful method, especially in the context of large-scale systems.

<<design question: is priority worth the complexity? Take a look at the alternative web server concept based on priority scheduling of responses below>>

And finally: changes in service rates are hard to achieve, both for the single server station as well as the dual unit. Serial parts in algorithms as well as the overhead due to multiple units (cohesion) will put limits to scalability.

Applications of QT concepts in multi-tier Systems

Instead of trying to calculate complex wait and service scenarios we will use some lessons learned from simple queuing models and apply them to large-scale multi-tier architectures. Further down we will also look at simulations of queuing models e.g. the Palladio System of KIT [Reussner <<check bib>>].

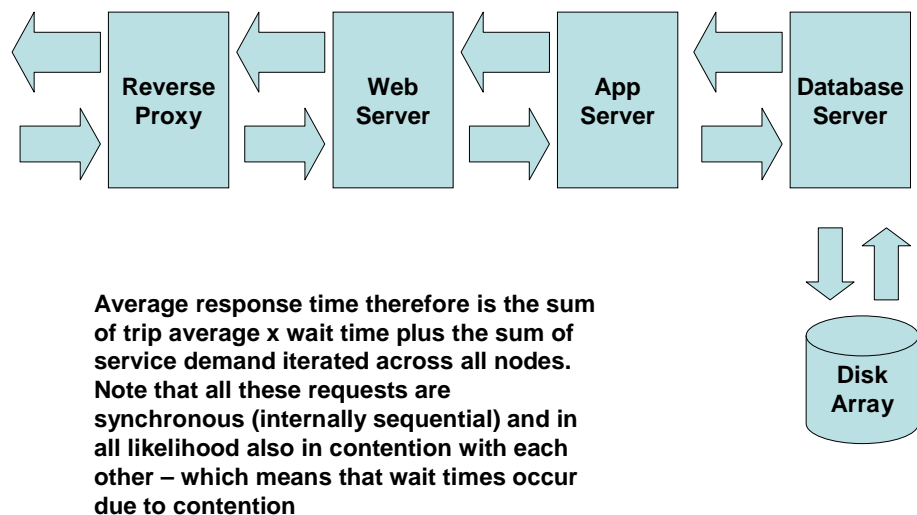
The following topics are important from an architecture point of view:

- Service-wait pattern in multi-tier systems
- Index in data
- Service Demand Measurements
- Cost of slow machines in mid-tier (cohesion at least, sometimes contention as well) : does queuing theory really apply? Requests go back instead of leaving the system through the final queue.

- Queue length (timeout, of client : whole residence time important) output queues? Buffering? Asynchronous output?
- Funnel architecture of multi-tier systems
- Heterogeneous hardware and self-balancing algorithms
- Dispatch policies in multi-queue server designs
- Unfair Dispatch disciplines
- Request Design Alternatives
- Finally: QT applicable to multi-tier systems due to requests not leaving at the end?

Service Demand Reduction: Batching and Caching

Liu describes a rather important quality of modern multi-tier enterprise application: it's "service-wait-service-wait" behaviour. The diagram below makes this rather obvious:



Reducing wait events and service demand (number of requests) will therefore considerably increase throughput or reduce response times in enterprise applications. Liu mentions several strategies (which turned out to be completely agnostic of programming languages or runtime platforms):

- array (batch) processing requests in groups. (reduction of service demand). This reduces the average number of trips to the queue processing center and is the same as some large scale sites describe as their "multi-get" feature for accessing caches or services.
- caching at high levels to avoid requests altogether

Liu claims that "Because of this significant improvement on performance, every enterprise software application should adopt and implement array processing even during the early stages of

product development life cycle before performance assurance and acceptance tests begin.”

But let us first ponder over this claim a bit. Demanding that batch processing (request bundling and avoidance strategy) should be used from the very beginning turns it into an architectural quality of enterprise systems. We are no longer really talking “optimization” here, even if Liu calls it this way. Given the sequential, synchronous nature of multi-tier architectures this is a reasonable thing to do. But what are we actually doing when we introduce batch processing? Exactly where do we win the time and throughput?

Clearly we have fewer requests at a specific queue processing center when we start batching requests. But individual service time should increase because of batched requests take longer to be processed. We will save on protocol overhead (sending and receiving the requests) and interrupt processing time and possibly also on context switching time if our individual requests would be sent by different threads otherwise. But if we cannot use internal parallelism during the batch request processing it is not really obvious where we make our wins. And if we can use parallelism internally at the receiving queue we could use this also to process more requests and would not have to use batching at all. Batching does not change the fundamentally synchronous way of processing either: initiators will still have to wait for the remote requests to finish and in case of batched requests they ALL need to be finished. Below we will take a look at the use of unfair dispatch in a web server and we will learn that getting rid of requests within a service station is extremely beneficial to throughput. Alternatively we could return requests on an individual basis, thereby reducing wait times within the upstream queue processing center at the cost of increased transport protocol effort. This is something that we will have to investigate further when we talk about I/O processing options later. I have a feeling that we need to express queue behaviour in terms of service time and contention only. I guess I am simply questioning Jackson's law here.

Liu's next optimization regards caching which reduces wait time. (I would have guessed that it reduces trips to the queue and by doing so indirectly also wait time). Introducing caching at the application server level obviously has the biggest effect as it reduces a whole number of requests later. When frequently used objects are no longer cached applications can experience severe performance and throughput degradation.

Service Demand Reduction: Data-in-Index

An interesting case of service demand reduction is “data-in-index” technology which can be used to avoid going to large data tables.

```
Select C3 from T1 where C1=<value> and C2=<value> order by 1 ASC;  
Select C3 from T2 where C1=<value> and C2=<value> order by 1 ASC;
```


With T1 and T2 being huge tables and C3 being the only column returned it pays off to add C3 to the indexes of C1 and C2. The reduction of unnecessary logic is just another case of service demand reduction while increasing the storage speed e.g. reduces wait time. The proposed duplication to avoid joins is a very effective technique used in large-scale storage systems. It simply takes some time to get used to this trade-off between lookup-time and storage utilization. <<link to the well known article on “how I learned to love data duplication...”>>

Intuitively we feel that pipelines of processing nodes work best if all nodes experience the same service demand. Another way according to Liu is to express this using utilization (being equal to throughput times service demand). A processing pipeline performs best if all nodes show the same utilization. As service demand is expressed as trips times service time it explains why equal service times are seen as a way to achieve equal utilization e.g. in CPU internal pipelines. Longer response times at one pipeline stage cause longer wait times upstream. (And now we know why web requests should be short when synchronous). And most importantly utilization is easily measured and compared between systems and equal utilization stands for a balanced and therefore stable overall system. [Liu]

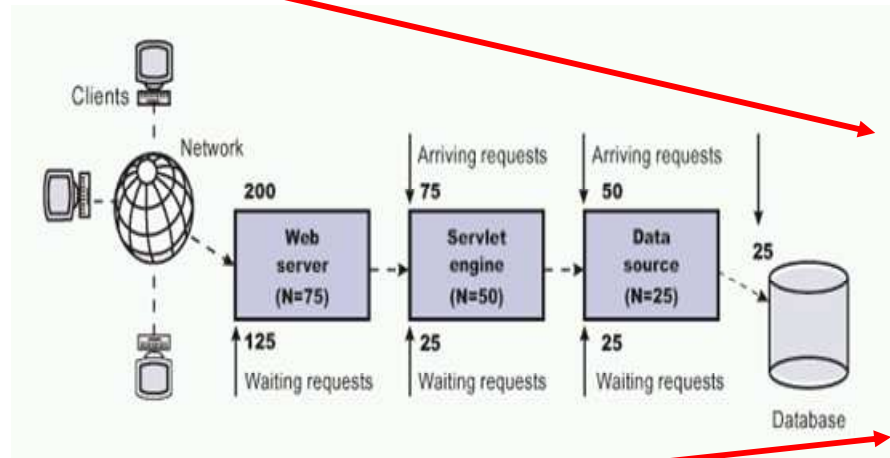
Service Demand Measurements

As we are going to use our analytical results and heuristics to define measurement points in our architecture one very important point needs to be discussed: the measurements of service demand. Just measuring the service rate in the processing units and keeping them close to 100% is a dangerous way to judge the performance of our system: Once we reach 100% utilization of our processing units we do not know how much additional work actually resides within our system waiting to be processed. We need to make the trade-off between customers waiting and optimal utilization of our processing units visible and measurable at all times to avoid creating long wait times which in turn cause timeout problems and dead request processing.

The n-tier funnel architecture

In concrete web applications many architects do an even more conservative interpretation of balance by demanding a funnel shaped request pattern from the beginning to the end of the processing pipeline:

<<diagram of request funnel for web applications>>

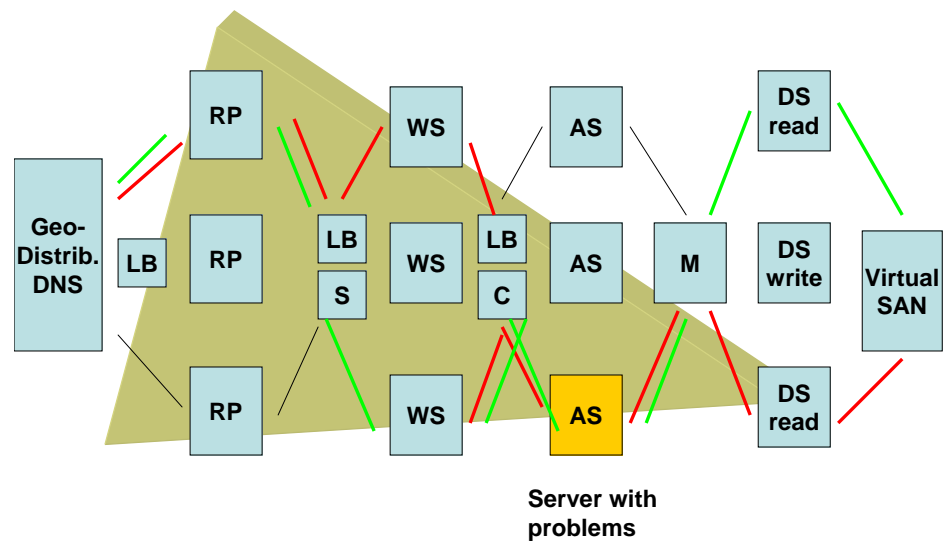


This is based on the experience that short increases of wait times or service demand at some point in the pipeline can have disastrous effects on overall throughput due to upstream effects. And that by simply adding more threads the service times for all requests increase as well. <<what is the physical explanation for this?>>

Cost of slow machines in mid- or end-tier

(cohesion at least, sometimes contention as well)

Service access layer, cloud of resources allocated and processing stalled, how does each tier allocate and schedule requests?



Multi-tier architectures typically use layers of load-balancing/switching between horizontally arranged servers. A request can therefore use many different paths from entry to backend system. We need to look at two scenarios: first, what happens to one request when a certain server in the path experiences performance problems and second, what does this mean for all the requests currently in the system?

To reach a certain server a request needs to pass many previous servers. By doing so the request typically allocates per server resources which are kept until the request comes back from a downstream component. In other words: a request blocks resources on upstream servers. Both contention and coherence in upstream servers get worse!

It depends on the implementation of those resources how severely other requests will be affected: a Thread-per-request model means that a thread blocked will not be able to service other requests and soon the available threads in a thread-pool will be depleted. Does this mean we should use a non-blocking I/O strategy anyway where we just save the request stack somewhere and use the thread to handle a new request? It will not help us in case of a serious problem with a server: all it does is to fill up our machines (upstream) with requests that cannot continue. Load balancing would soon route more and more requests to the remaining machines and – if their limits are reached – start to no longer accept new requests.

This example shows several important things:

- input queue limits are important. We should not accept more requests upstream just because we cannot continue some of

them downstream: we would just overload the remaining downstream servers.

- A slow or broken server affects many requests across a cloud of servers upstream.
- A slow or broken server reduces the number of input connections for upstream servers because they cannot forward those requests downstream
- A slow or broken server should be detected as quickly as possible to avoid sending requests against it and to reduce request acceptance upstream
- Non-blocking resource allocation does not help. Without strict resource management it can even blow up our servers.
-

Queue length and Residence Time

(timeout, of client : whole residence time important) output queues? Buffering? Asynchronous output?

In queuing theory the queue length as is usually a rather uninteresting parameter and in many cases it is assumed to grow infinitely. If a fixed queue size is assumed the most important question is usually: when do we start to lose customers because the queue is full?

Reality is much different here because a full queue is not the only reason for losing customers: customers can implement timeouts, in other words they can cancel requests if they take too long. Queue size is therefore just one parameter and we are really concerned about Residence Time, not queue wait time.

Some considerations:

Residence Time == Waittime in queue plus time spent in service
Residence Time < Customer Timeout

Queue size needs to be calculated so that for a given arrival and service rate the residence time is smaller than the customer timeout. <<give formula>>

What happens if we reject the request due to a full queue? The client is free to issue the request again, perhaps polling for a free slot. This is communication overhead for sure but it does not affect our internal servers in any negative way. The client is of course free to chose a different service station. In this case we lose a potential customer.

But what happens if we do not restrict queue size and end up with a residence time bigger than the client timeout? Then something really ugly happens: we end up processing dead requests. A typical example is when a client always uses the reload button on her browser faster than we can respond with the proper page. The previous request is already dead when we want to send its response.

How can we protect us from such a behavior? Caching our complete responses is certainly a good idea. Other instruments are asking the client during request processing whether she is still interested (e.g. by checking the connection or by asking for a computation token). If we know that clients use an aggressive form of timeout handling we can even offer them the average current residence time as a base for their decision.

In every case we need to track the number of closed connections or timeouts we experience throughout the processing in our multi-tier architecture: they can be signs for dead request processing. And we have to decrease queue size (or service time). And this has to happen in real-time because we want to avoid processing potentially dead requests.

Output traffic shaping

Many queuing theory algorithms assume independence between incoming and outgoing requests meaning that there is no connection between requests. This is of course only an assumption made for mathematical simplification. In reality user sessions consist of more than one request and therefore requests are not really independent. But what if we could use those dependencies? One idea could be to shape the incoming traffic through modulation of outgoing (processed) requests. Usually a user spends some time between requests to think about results of a request. And then she will issue a new request. By slightly delaying responses we can influence the time before a new request will be issued. This may sound like a rather small achievement but given thousands of requests per second it might make a difference in our machine. I do not know of any system that currently uses this approach though.

The realism of Queuing Theory based Models for distributed systems

We will later on discuss a certain type of load balancer: a pull based system. This system consists of service nodes which – on being idle – request new jobs from a central queue. A central queue model avoids the problems of multi-server queues where one queue is still full while others are idling. It should therefore lead to a better throughput. But implementations of this model have shown the adverse effect. What might be the reason for this?

One explanation could be that there are some important variables missing in the model, or perhaps in most QT models once it comes to distributed systems. QT seems to assume absolutely no latency between queues and processing nodes. After processing a node there is no time lost until a new request is being processed. This assumption is obviously not true in the case of real pull servers. If they wait for a request to finish throughput will suffer due to the time it takes to send a request to the central queue and getting a new request back. An alternative of course would be to slightly

overlap request processing at each pull server but this is essentially only the re-introduction of multi-server queues. A pull server would have to know exactly when to requests a new client request for processing without losing time. If a request has to wait at the node it could have possibly been processed at another node in the mean time.

The example only shows that important variables of real distributed processes are not modelled in QT and that this can lead to wrong assumptions.

Request Processing: Asynchronous and/or fixed service time

The overview of large scale sites has shown an increasing use of asynchronous request processing. Other examples like Google Application Engine API and the Darkstar Game Engine architecture enforce fixed or at least limited service times. The reason is well expressed by Neil Gunther in his Guerilla Capacity Planning book (see below) where he discusses the connection between Amdahls law (the fatal consequences of serialization) with the classic repair man queue model (a wait-based synchronous service model).

“Conversely, I have shown elsewhere (Gunther 2005b) that both Amdahl’s law (4.15) and Gustafson’s law (4.30) are unified by the same queueing model; the repairman model. Theorem 6.2 tells us that Amdahl’s law corresponds identically to synchronous throughput of the repairman. Synchronous throughput is worst case because it causes maximal queueing at the repairman (Fig. A.1) or bus. In that sense, Theorem 6.2 represents a lower bound on throughput and therefore is worse than the mean throughput. Once this interpretation understood, it follows immediately that Amdahl’s law can be defeated, much more easily than proposed in (Nelson 1996), by simply requiring that all requests be issued asynchronously!” [Gunther] pg. 218

Interestingly asynchronous requests have been also added to the new servlet API 3.0, see [Bartel]. While mostly geared towards Comet style AJAX communication, this would also allow parking a request, issuing parallel asynchronous subrequests and – once a fixed timespan has expired – to collect the data, skip missing data and return after a constant service time to the user. Clever request

design could then achieve an effect close to Hajunka (see below): a partitioning of service effort and time into same sized blocks.

Heterogeneous hardware and self-balancing algorithms

<<to-do>>

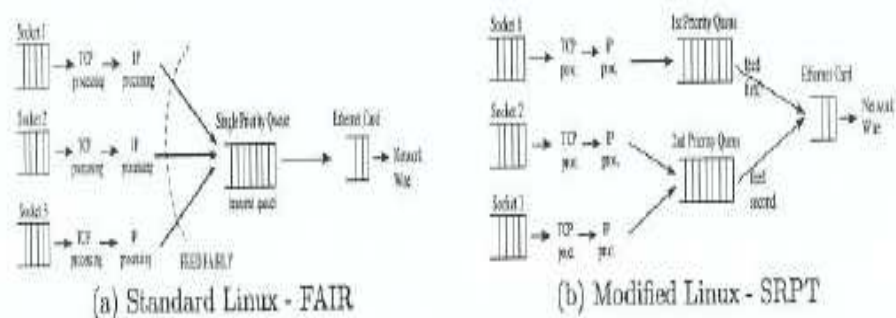
Dispatch in Multi-Queue Servers

The role of the dispatch discipline in multi-queue designs is quite interesting: What would be an optimal dispatch of incoming items? Load balancers have to find an answer to this question e.g. by tracking the load on servers. An alternative dispatch strategy would be to use pull instead of push: Let the servers pull new items when they are done with the previous item. But how would we implement priority queues in that case?

Unfair Dispatch: Shortest Remaining Processing Time First

We have not changed the way the system performs scheduling yet and simply assumed it would be FCFS – in other words a fair schema. Experiments have shown that a fair schema need not be the most effective. If the number of requests currently in the system is used as a measure for effectiveness of a processing station and if only a certain variance of arrival and service rate are assumed it turns out that the most effective strategy is to pick those tasks with the least processing time left before completion first. In the case discussed below the file size requested is used as an indicator for the processing time needed.

<<web server SRPT example>> [Schroeder]



Surprisingly long requests (for large files) were not starved to death under unfair scheduling. Under the constraint that the system experiences also situations of low load the large requests were then serviced mostly uninterrupted and this made more than up for the short delays by servicing short requests first.

What is the “physical” explanation for this? It is in the fact that multiple connections and tasks all require a certain overhead for switching and multiplexing which is reduced by an SRPT strategy. But we cannot only look at slight improvements of throughput in our large scale architectures: we also need to calculate the effects of our optimizations in case of a different input distribution. In other words: how does our system behave if we do not see the expected variance in the arrival rate for some time? We might detect a rather disturbing behaviour in this case, namely that the system might become much less effective or even instable due to the fact that not all the big requests are unable to complete. Fair scheduling might have been less effective in the case of an optimal input variance but it might deal better with less optimal input distributions. This is also a lesson learned from building operating systems: frequently a less effective but more general algorithm (for memory allocation or sorting e.g.) will show a more benign behaviour across a wider range of input situations.

Request Design Alternatives

It looks like a common and rather short request size and latency allows better throughput. But what if there is a big difference between some requests with respect to service time? The answer given till now was: use asynchronous processing. Are there alternatives to asynchronous processing? Surprisingly there are a number of design alternatives and they start at design time: In many cases it is a matter of request architecture whether the service times will differ largely or show a rather common service time. Every request needs to be checked at design time for unnecessary bundling of functionality which creates overly long service times. Requests can be configured towards a common time.

What else is possible? The section above used unfair scheduling to improve throughput. This is OK as long as arrival time distribution allows for low traffic times where long requests are handled effectively. Otherwise they are starved. Content based routing and partitioning can prove a viable alternative in this case: Route requests of a common service time towards one server only and use other servers for different requests. And even more optimization is possible: Once the requests are partitioned along service time the whole further processing chain can be optimized for the specific request requirements: block sizes, network parameters etc.

A very interesting alternative is descriptive batching of requests as is done in the case (like FQL) for internal optimization. At the first glance it seems to create larger request service times and achieve exactly the opposite of the intended effect. But due to its descriptive nature it allows internal partitioning and optimizations as most SQL processors e.g. do.

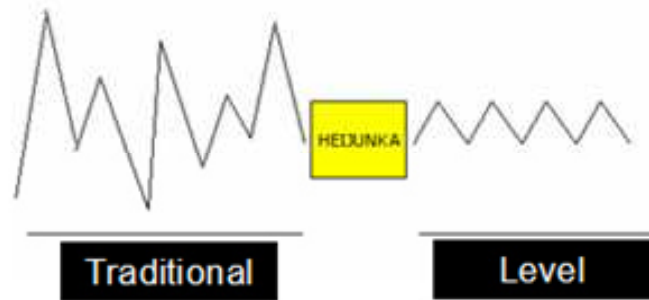
An easy alternative which also works in case of failures in recently deployed API functions is to dynamically turn API functions on and off if a the system experiences overload.

Finally there is the question why multi-tier architectures don't come with a feature that is e.g. very common in networks: the sliding window feature of TCP allows throttling of requests back to the client. And interestingly: we find exactly this feature in the Darkstar game platform architecture discussed below. Overloaded servers can send exceptions to clients and prevent further requests. In any case the worst design is to allow too many requests into your system. Or requests of very different service times. If one could kind of chop requests into a common size at runtime this would turn out very beneficial to throughput. The next section discusses the "heijunka" method used in Japanese automotive plants to achieve exactly this.

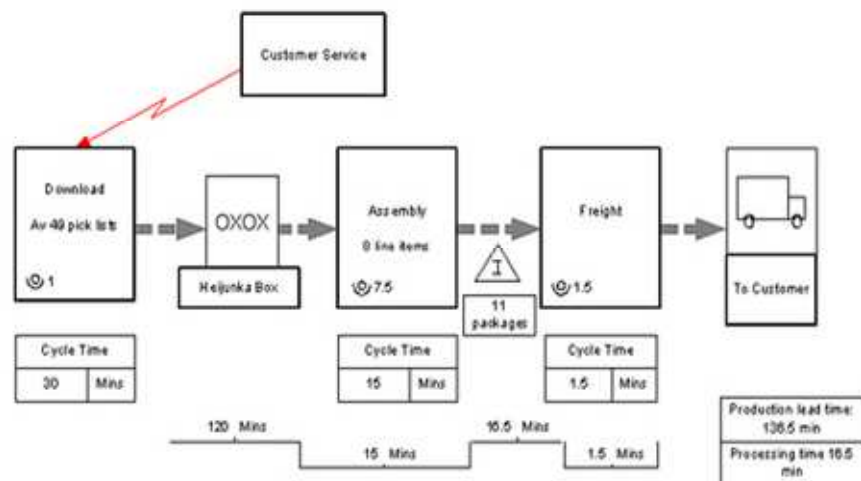
Heijunka

The last concept we are going to discuss here is "leveling" or "Heijunka" as it is called by Toyota. Leveling tries to avoid spikes in demand or production as these have been found rather cumbersome and ineffective. The opposite of those spikes is a balanced system. But the core assumption behind levelling or heijunka is that you need balance first before you can get velocity. The following diagrams and concepts are taken from the queuing theory pages of Peta Abilla [Shmula] who did supply chain management with various large companies like amazon.

The diagram below shows the levelling effects of heijunka:



Heijunka seems to chop incoming items into equally sized pieces. This could happen in the spatial as well as the temporal domain: either blocks of equal size are created (same sized compound messages, same size memory or disk blocks etc.) or the frequency of requests is fixed at a certain rate (x requests per time unit). A whole supply chain with levelling element might look like this:



To better understand the concept of levelling and balance we can take a look at car engines. A car engine from an engineering point of view can be considered as a standing wave: From the carburetor

and air filter elements through the intake manifold, valve system, cylinder area and throughout the cylinder exit and exhaust system a standing wave forms when the engine is running. The frequency of this wave can change e.g. to get more power but this change must lead to a new frequency throughout the whole system. You can't get more power by simply adding more fuel without regard to the other elements in the system. Is this balance tied to a low utilization of resources as some statements from Birman on the behaviour of protocols in relation to hardware utilization might suggest? Intuitively the answer seems to be yes. Network protocols break down when utilization reaches high levels (CSMA protocols especially), operating systems break down when disk utilization (both spatial and temporal) gets very high. Most servers in distributed systems run at rather low utilization rates (frequently as low as 20%) and system admins get rather nervous when these numbers are exceeded. On the other hand IBM mainframes are meant to run close to 100% utilization. This can only be achieved with an extreme form of workload measurements, prediction and balance in those systems. Does this have something to do with synchronous vs. asynchronous processing or the treatment of failures and exceptions?

For more information on queuing theory see Myron Hlynka's Queuing Theory pages at:

<http://web2.uwindsor.ca/math/hlynka/queue.html>

Tools for QT-Analysis

In QT-Analysis a point where the complexity of the calculations exceeds our abilities is quickly achieved. Calculators for QT exist which make life a bit easier <<example QT calculator

Clausthal/Hänsch>>

<http://www.integrierte-simulation.de/>

or: [http://www.stochastik.tu-](http://www.stochastik.tu-clausthal.de/index.php?id1=Presse&id2=Schulen)

[clausthal.de/index.php?id1=Presse&id2=Schulen](http://www.stochastik.tu-clausthal.de/index.php?id1=Presse&id2=Schulen)

With multiple queues, heterogeneous processing units and non-standard distributions only simulation can be done.

Applicability of QT in large-scale multi-tier architectures

Finally a word on the applicability of QT. QT makes several big assumptions about the queuing network under analysis which are probably not very realistic. The assumptions are made to make the math calculable. Considering the chain of nodes in a multi-tier architecture as a markov chain requires the nodes to be independent. In this case a single node can be treated as a simple queuing node with most likely M/M/1 characteristics. But are the nodes really independent? QT usually models production systems where request leave the queuing network at the end and (hopefully to allow easy distribution assumptions) do not come back (no

feedback). But real requests in multi-tier architectures leave the system at the entry point instead at the backend nodes. A request gets “smeared” across all nodes which are visited during its processing and that makes nodes far from being independent. Event-driven architectures using asynchronous I/O – if not programmed in a subrouting calling style – do not expect a request coming back from a downstream component. We will look at the Staged Event-Driven Architecture (SEDA) in the chapter on I/O as it represents a rather typical form of this architecture.

So QT models may lack quantitative applicability in our systems. They nevertheless let us explore very interesting heuristics about connected nodes and requests and are very important for a qualitative analysis. Still, a model that brings all the discussed “lessons learned” and constraints into one consistent model would be very nice to have.

Combinatorial Reliability and Availability Analysis

Systems are getting more complex every day. Multi-tier systems are notoriously hard to debug and cause enormous costs for servers and software units. But how much availability do we really get for the money or how much redundancy will we need to achieve a certain degree of availability?

Before we delve into questions about reliability and availability we need to think about the structure of the problem zone a bit. No matter what we want to track: bugs, performance, availability, events or steps – we will always realize that the system under analysis is composed of low-level components which interact with each other. On top of those components we find higher-order components which represent activities and even higher ones which finally represent business processes. Most companies have a hard time to associate business processes with certain server configurations, networks etc. We will discuss this type of architecture again in the section on logging and tracing where we will take a look at complex event processing approaches. Here we need to mention a rather new feature of today’s systems: their dependency on external services. Architectures which use external services (perhaps following a Service Oriented Architecture –SOA) can no longer just look at the availability of components. They need to find new ways to express reliability and availability guarantees for external services which then become core parts of the architecture.

Formal, perhaps even pre-implementation analysis of availability is not in widespread use, due to efforts or skills involved. This is rather unfortunate because we will see shortly that adding reliability and availability to existing components or tiers is rather difficult and expensive. For real-world projects we need analysis models that are both easy to learn and easy to use. Bailey et.al. present three reliability engineering techniques which show those properties [BSLT]:

- Failure Modes,
- Events and Criticality Analysis (FMECA),
- Reliability Block Diagrams (RBD) and
- Failure Tree Analysis (FTA).

We will take a look at those methods and how they work and also speculate about dependencies between capacity, utilization and availability.

The simplest method to start a reliability/availability analysis is FMECA. It is a risk assessment method and it mirrors analogous methods from other areas. The First Cut Risk Analysis (FCRA) in security analysis comes to mind and they are virtually identical:

Failure Modes, Effects and Consequences (FMECA) Analysis

Nr	Business Function	Mode	Reason	Prob.	Effect	Severity
1	Cash dep.	Crash	ATM	Low	No money	low
		Hang	Line	Med	ATM block	medium
2	Credit	Funct	ATM	low	ATM block	low
3						

The method starts with the most important business functions and creates scenarios where those functions fail. The “mode” expresses different types of failure. It is a heuristic methodology and allows early identification of potential problems. You can easily extend it with properties that deem important in your case, e.g. the way to detect the failure.

Bailey et al. mention some deficits as well: it is a static analysis only and it cannot represent multi-component failures and redundant components properly. But to me the biggest disadvantage is that the analysis is largely de-coupled from system architecture. It requires a very good implicit understanding of the system architecture to come up with critical functions and their possible causes. It is therefore clearly geared towards the business/financial effects of failures and does not tell you how to build a reliable system.

A more expressive modelling method are Reliability Block Diagrams (RBD's) like the following taken from Bailey et al.:

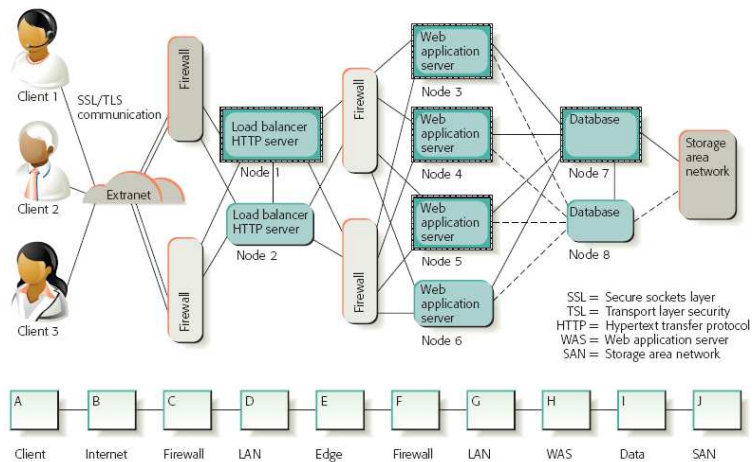
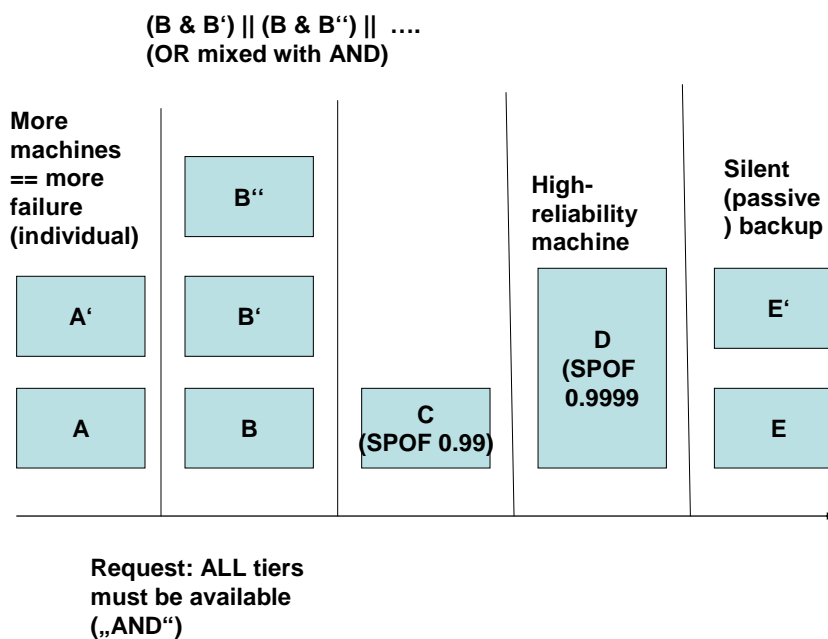


Figure 3
A three-tier server configuration for a Web-based application and its RBD

That the connection to system architecture is a bit closer can be seen at the bottom part of the diagram where the components involved in a request are drawn in sequential order. It is important to realize that ALL of these components need to be available TOGETHER to achieve some defined degree of availability. This in turn means that the availability of each component (or tier) needs to be higher than the targeted overall availability.

Above the request chain the diagram shows how components are redundantly implemented and organized as tiers. The various ways of doing so are represented in the diagram below:



Instead of talking about “and” and “or” we can also use the terms “serial availability” and “parallel availability” according to [Scadden]

Serial chain of ALL needed components: multiplying availabilities gives less overall availability or: the more chain members the higher individual availability needs to be

$$SerialAvailability = \prod_{i=1}^N ComponentAvailability_{(i)},$$

Redundant, parallel components where only ONE needs to be up: multiply unavailabilities and subtract from 1.

$$ParallelAvailability = 1 - \left[\prod_{i=1}^N (1 - ComponentAvailability_{(i)}) \right].$$

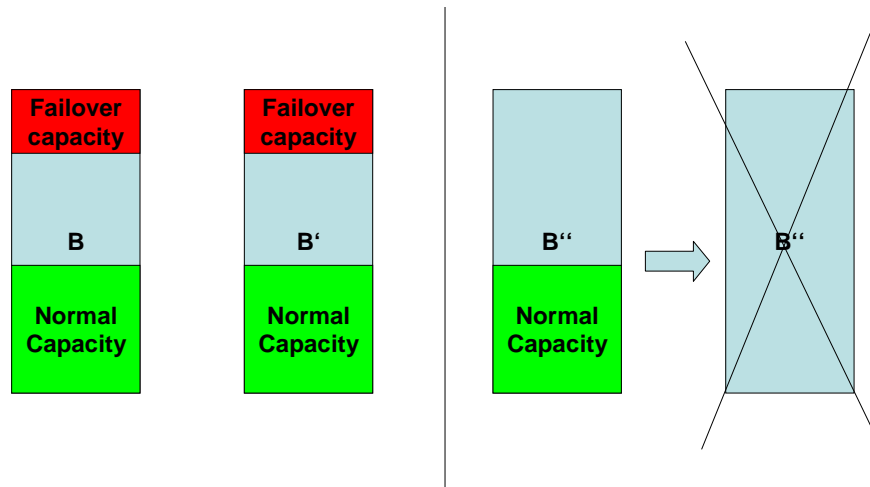
**From: Scadden et.al.,
pg. 537**

A number of observations follow:

1. If we add more machines we will experience more failures One big machine with an MTBF of 0.9999 is better than two smaller machines with the same MTBF because we now have twice the chance for failure.
2. In the architecture above component C with a reliability of 0.99 will limit the overall availability of the whole request processing chain to 0.99 * Prest. And even if we could optimize the other components to zero failure probability Amdahls law states that our limit would be 0.99. In other words: the weakest link determines overall availability
3. Sometimes a single big and highly reliable machine might be beneficial. We are using vertical scalability in this case, based on a highly reliable platform. Database Tiers frequently follow this pattern.
4. Passive standby can be used to achieve failover as well. Watch out for manual steps needed.
5. The redundant tier with B machines presents a 1/3 availability solution which means that one out of three machines can fail without disrupting service guarantees. The formula calculates the probability of more machines failing. Often some wrong assumptions are made in this case: it is assumed that the failure of one machine does not have an impact on other machines. This is frequently not true because of the effects of utilization on reliability: Most components show a more unreliable behaviour beyond a certain utilization level. The diagram below shows the increase in utilization that is caused by the crash of machine B''. The load is then distributed to machines B and B' but if these machines are now pushed beyond a certain utilization level our overall availability will go down.

$$P = \sum_{k=n+1}^N \binom{N}{k} p^k (1-p)^{N-k} \quad (1)$$

1/3 tier redundancy with p being a function of k due to capacity increase



This is the reason why 1/2 redundant configurations of midrange machines frequently show a very low utilization of no more than 20% to avoid getting into the “red” zone in case of crashes.

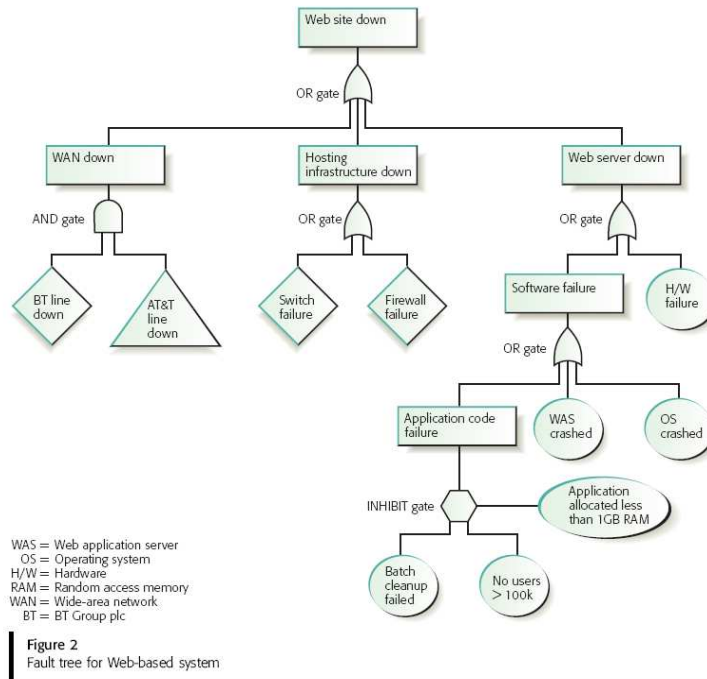
Bailey et. Al. also discuss the concept of fail over and define it as follows:

- switch over to redundant system
- preserve lock state
- roll back lost work.

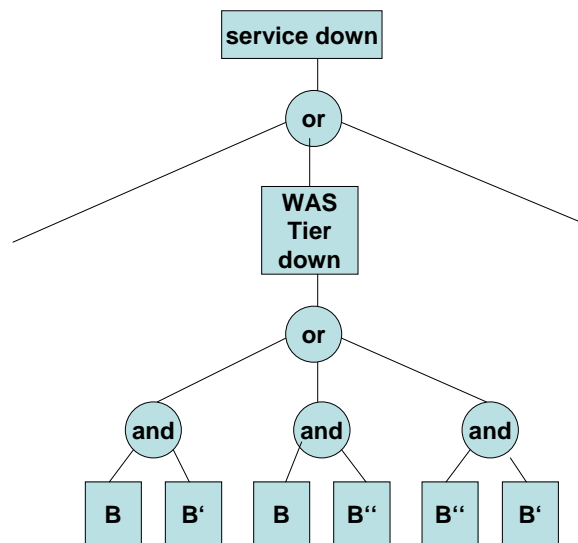
“The difference in solution availability comes down to the shape of the probability distribution of the lock holding time. It turns out that on failover there is no perceived outage most of the time. However, because there is always a finite probability of a long failover due to long lock holding time in all of these schemes, individual node availability is key to keeping down the probability of a long failover. A tighter distribution around a fast average failover will also drive availability higher.” [BSLT] pg. 587

Failover will be discussed in detail in the section on J2EE clustering.

With Failure Tree Analysis (FTA) the third methodology presented gets even closer and deeper into the architecture of the system: FTA assigns Boolean symbols to all connections between components which express AND or OR effects on availability.



We can reproduce part of our RBD model from above as a FTM as shown in the diagram below:



The value of FTA and FTM is not without questions according to the literature. Models seem to depend on individual authors and can become quite complex and crowded. Automatic analysis is possible but rather demanding with respect to space.

<<link to critiques of FTM >>

Let us finish reliability engineering with a short discussion of reliability and software. Much of the above comes from hardware development and relies heavily on reliability estimates. In case of hardware those number

can be gathered from statistics and are therefore rather reliable. But what about the software part of system architectures? This has always been the weak link, even in mainframe architectures as has been documented e.g. by K.Klink. Let's assume we are using a state-machine based replication technology between our server machines in the application server tier (see below the discussion of PAXOS consensus protocol within a replicated log in Chubby [Google]). What happens if the software algorithm has a bug? The state-machine approach will replicate the bug on all machines involved and crash each and every one of them. A similar effect once led to the destruction of an Ariane V rocket because the buggy software killed the backup hardware component just as quickly. [Meyer]. Multi-language, multi-hardware designs were considered long as a sure means to fight individual software bugs. The theory was that different languages and designs used would prevent all replicated instances of a solution from failing at the same time. There seems to be quite a bit of folklore involved as has been demonstrated by Tichy who performed empirical analysis of software development: He was able to show that most of these replicated software components still had the problems at the same spots in the source code and could not prevent crashes of the whole system. [Tichy].

Stochastic Availability Analysis

[STTA] W.E.Smith, Availability analysis of blade server systems (Markov Models, Semi Markov Processes, Generative Markov Models) state-space modeling approach
<<self management, fitting of long tail function>>

Guerrilla Capacity Planning

<<see also John Allspaw, Capacity Planning, oreilly Integrate slide set>>

One unique Guerrilla tool is Virtual Load Testing, based on Dr. Gunther's "Universal Law of Computational Scaling", which provides a highly cost-effective method for assessing application scalability. Neil Gunther, M.Sc., Ph.D. is an internationally recognized computer system performance consultant who founded Performance Dynamics Company in 1994.

Some reasons why you should understand this law:

1. A lot of people use the term "scalability" without clearly defining it, let alone defining it quantitatively. Computer system scalability must be quantified. If you can't quantify it, you can't guarantee it. The universal law of computational scaling provides that quantification.

2. One the greatest impediments to applying queueing theory models (whether analytic or simulation) is the inscrutability of service times within an application. Every queueing facility in a performance model requires a service time as an input parameter. No service time, no queue. Without the appropriate queues in the model, system performance metrics like

throughput and response time, cannot be predicted. The universal law of computational scaling leapfrogs this entire problem by NOT requiring ANY low-level service time measurements as inputs.

The universal scalability model is a single equation expressed in terms of two parameters α and β . The relative capacity $C(N)$ is a normalized throughput given by:

$$C(N) = N / (1 + \alpha N + \beta N (N - 1))$$

where N represents either:

1. (Software Scalability) the number of users or load generators on a fixed hardware configuration. In this case, the number of users acts as the independent variable while the CPU configuration remains constant for the range of user load measurements.

2. (Hardware Scalability) the number of physical processors or nodes in the hardware configuration. In this case, the number of user processes executing per CPU (say 10) is assumed to be the same for every added CPU. Therefore, on a 4 CPU platform you would run 40 virtual users.

with ' α ' (alpha) the contention parameter, and ' β ' (beta) the coherency-delay parameter.

This model has wide-spread applicability, including:

- * Accounts for such effects as VM thrashing, and cache-miss latencies.*
- * Can also be used to model disk arrays, SANs, and multicore processors.*
- * Can also be used to model certain types of network I/O*
- * The user-load form is the most common application of eqn.*
- * Can be used in combination with measurement tools like LoadRunner, Benchmark Factory, etc. [geekr]*

The following slides are taken from the Guerilla Capacity Planning Guide by [Gunther]

Concurrency and Coherence

Concurrency effect:

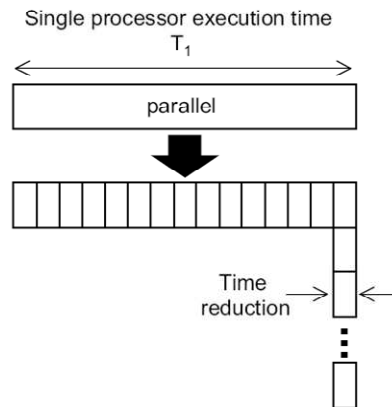


Fig. 4.4. Ideal parallelism. The uniprocessor execution time T_1 is reduced to T_1/p by equipartitioning the workload across p physical processors

Contention effect: it is really the size of the serial part of a computation that limits speedup and scalability.

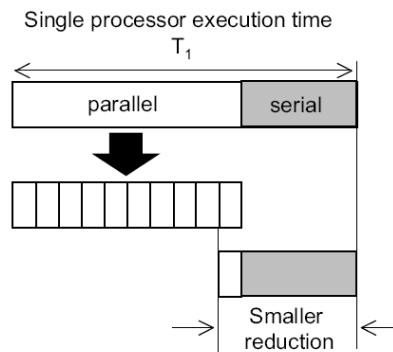


Fig. 4.5. Amdahl's law recognizes that ideal parallelism (Fig. 4.4) cannot be achieved in general because there are certain portions of the workload that can only be executed sequentially (*gray*). That aggregate portion of the total execution time is called the serial fraction

This has a profound impact on response times in a multiprocessor setup:

The impact of multiuser scaleup on response time is shown in Fig. 4.7.

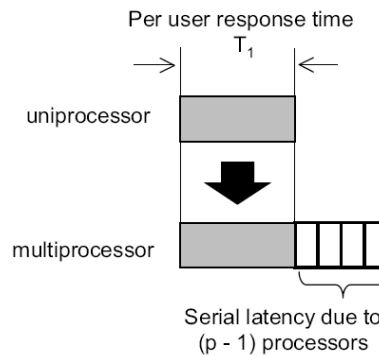


Fig. 4.7. The effect of multiuser scaleup is to stretch the response time in proportion to the number of physical processors

Actually this should be true in a single processor setup as well: adding more threads creates an increase in service time which again increases residence time (response time).

Added coherence effect (universal scalability law)

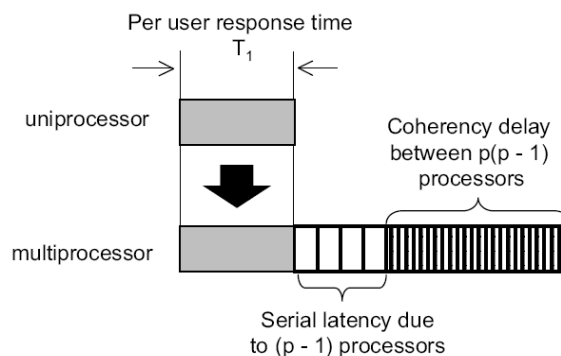


Fig. 4.8. Multiuser scaleup showing the per-user response time growing linearly with the number of processors due to serial delays (cf. Fig. 4.7), and the additional, but smaller, coherency delays increasing quadratically due to point-to-point exchanges between processors

R.Smith mentions another contributor to coherence effects and calls it the $O(N)$ Serial Bottleneck [Smith]: It describes the effect that a growing number of threads extends the time spent in serial sections of the code. This is e.g. caused by algorithms within

critical sections which operate on the number of threads in collections. The more threads the more time is spent in a critical section. Event thread-packages seem to show $O(N)$ behavior in the number of threads [vonBehren]

The resulting graph which shows a clear maximum which would not be visible with the original law by Amdahl:

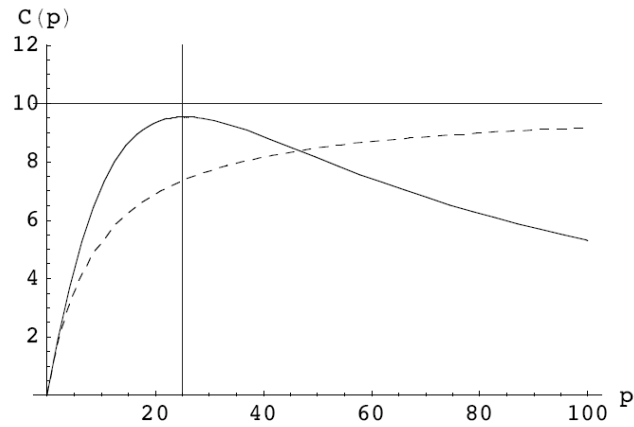


Fig. 4.9. Universal scalability characteristic (*solid*) compared with Amdahl scaling (*dashed*), which corresponds to a coherency value of $\kappa = 0$ in (4.31). A key feature of universal scaling is that a maximum can develop (here located at $p^* = 25$) depending on the values of σ and κ in (4.33). Comparison with Fig. 4.1 shows that although the system does not fail beyond p^* , its available capacity can degrade significantly

Calculation of contention and coherence parameters

The parameters of the universal scalability function control the shape of the curve and therefore contention and coherence effects.

$$C(p) = \frac{p}{1 + \sigma(p-1) + \kappa p(p-1)},$$

contention
coherence

The procedure to calculate those parameters is described by Gunther as follows:

The procedural steps for the calculation of σ and κ and $C(p)$ are as follows:

1. Measure the throughput $X(p)$ for a set of processor configurations p .
2. Preferably include an $X(1)$ measurement and at least four or five other data points.
3. Calculate the capacity ratio $C(p)$ defined in (5.2) (Sect. 5.4).
4. Calculate the efficiency C/p , and its inverse p/C (Fig. 5.3).
5. Calculate the deviation from linearity (Sect. 5.5.2).
6. Perform regression analysis on this deviation data to calculate the quadratic equation coefficients a, b, c (Sect. 5.5).
7. Use these coefficients a, b, c to calculate the scalability parameters σ, κ (Sect. 5.6.2).
8. Use the values of σ, κ to calculate the processor configuration p^* where the maximum in the predicted scalability occurs—even though it may be a theoretical configuration (Sect. 5.6.3). p^* is defined by (4.33) in Chap. 4.
9. Use σ and κ to predict the complete scalability function $C(p)$ over the full range of p processor configurations (Fig. 5.7).

The generation of test data is necessary from which the following ratios can be calculated and later be used for regression analysis: (from Gunther)

Measured CPU (p)	KRays/Sec X(p)	RelCap C=X(p)/X(1)	Efficiency C/p	Inverse p/C	Linearity p-1	Deviation (p/C)-1
1	20	1.00	1.00	1.00	0	0.00
4	78	3.90	0.98	1.03	3	0.03
8	130	6.50	0.81	1.23	7	0.23
12	170	8.50	0.71	1.41	11	0.41
16	190	9.50	0.59	1.68	15	0.68
20	200	10.00	0.50	2.00	19	1.00
24	210	10.50	0.44	2.29	23	1.29
28	230	11.50	0.41	2.43	27	1.43
32	260	13.00	0.41	2.46	31	1.46
48	280	14.00	0.29	3.43	47	2.43
64	310	15.50	0.24	4.13	63	3.13

Fig. 5.3. Example spreadsheet including the normalized capacity, efficiency, linear deviation calculations

$$C(p) = \frac{p}{1 + \sigma(p-1) + \kappa p(p-1)}$$

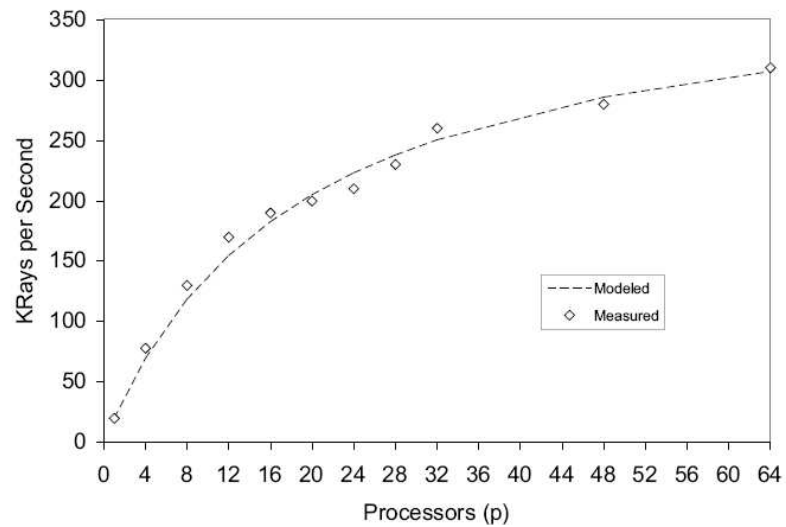
contention
coherence

$$y = ax^2 + bx + c$$

For regression analysis. Determining a and b will allow us to calculate the theoretical maximum of capacity.

The final result is a curve that can be overlayed over the test values. Gunther points out some very important properties of the universal scalability formula and its parameters:

- both parameters have a physical interpretation and can tell something about the concrete architecture
- The calculation of the theoretical maximum of a capacity curve avoids premature false peak assumptions and therefore hardware costs
- Differences between measured and calculates values can be an indicator for problems in certain configurations



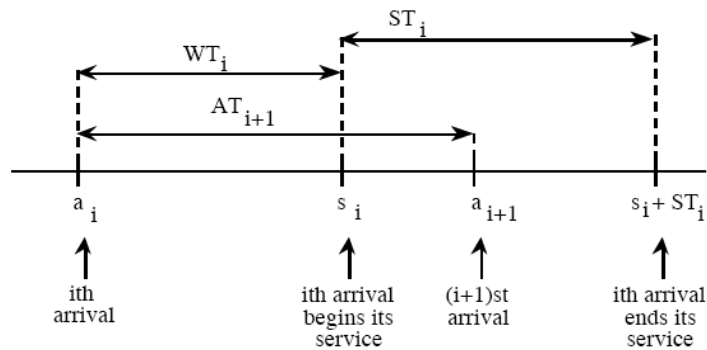
Client Distribution over Day/Week/Year

Simulation

Queuing theory quickly becomes extremely complex and no longer analytically solvable in case of multiple queues, heterogeneous hardware and non-exponential distributions. Here the only solution is to simply create empirical data, let them flow through a model and look at the results.

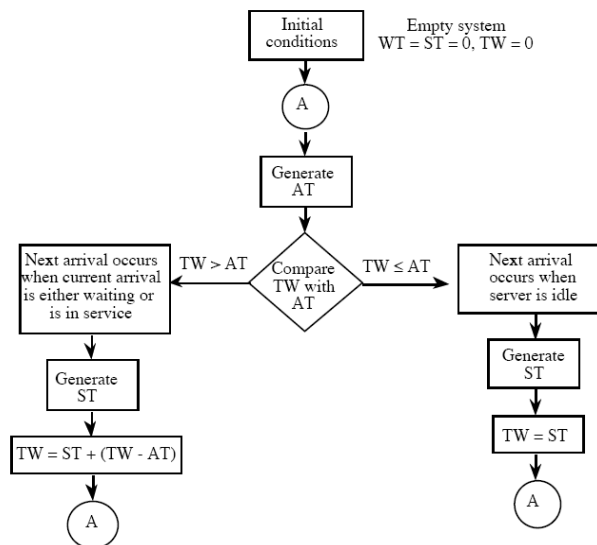
The simulation approaches I found were basically three: event-advance, unit time and activity based.

An example of an activity based simulation design can be found by [Perros]. Activity based simulation uses processes to model a system and according to Perros the method excels when simulating systems with complex interactive processing patterns.



From H.Perros, pg 94 ff. Arrival time can be during WT_i , during ST_i or when the processor is idle.

The diagram below shows how the values for the next incoming request can be calculated:



From H.Perros, pg 96 ff. Activity based simulation design of a single server queue

Event-advance simulations can simulate many days of operations within a few hours by always advancing to the next possible event. This type of discrete event simulation is described in [Pravanjan], together with a list of DES tools.

Tools for statistical analysis, queuing models and simulation

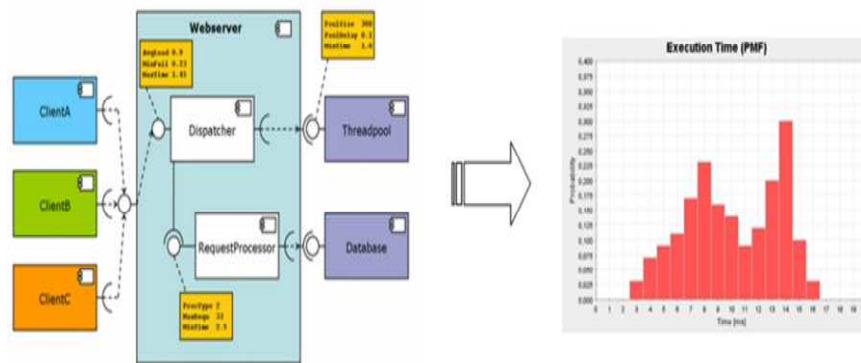


Diagram taken from the Palladio Component Model

http://sdqweb.ipd.uka.de/wiki/Palladio_Component_Model

A modelling and simulation package based on GEF/EMF especially suited for performance simulations.

PDQ *Pretty Damn Quick*. Open-source queueing modeler.
Supporting textbook with examples (Gunther 2005a)
www.perfdynamics.com/Tools/PDQ.html

R Open source statistical analysis package.
Uses the S command-processing language.
Capabilities far exceed Excel (Holtman 2004).
www.r-project.org

SimPy Open-source discrete-event simulator
Uses Python as the simulation programming language.
simpy.sourceforge.net

Example for an analysis of infrastructure based on device and architecture templates:

<http://storagemojo.com/2009/02/05/bayesian-analysis-of-it-infrastructure>

Vensim 5.9 Available

Vensim 5.9 now supports date labeling on graphs and in the Table tool. This ability makes it easier to present results to people who are not comfortable with decimal values for time. You can format the date by specifying a format string that allows dates to appear in such forms as 2009-04-09, 2009Q2, Mon Jan 1, or, for elapsed time, as 12:35:22.3 in hours, minutes and seconds. Date labeling is not available with PLE or PLE Plus. To see details on other changes and bug fixes see:

<http://www.vensim.com/new.html>

2009 System Dynamics Conference July 26-30

The 2009 System Dynamics Conference will be held in Albuquerque New Mexico USA. It should be a fun event, do consider attending. See

<http://www.systemdynamics.org/conferences/current>

Forums for Software and System Dynamics Discussion

If you have questions about Vensim or need support in using it the place to go is the Vensim forum at

<http://ventanasystems.co.uk/forum/>

Architectural Principles and Metrics

Here we are going to discuss “lessons learned” from modelling and simulation for the design and operation of large-scale systems. Architectural principles will help us to avoid bottlenecks and inefficiencies. Metrics will tell us when and what changes are needed to our system.

Architectural Principles

- avoid multi-queue service design without mechanisms to balance load further
- use small granularities for job sizes to allow balancing and uniform service times (small variance)
- track availability of service stations in multi-tier request paths closely and dynamically re-arrange request numbers of one station is out
- put limits on input request numbers
- avoid resource locking per request, use asynchronous request handling but respect the limits set
- use self-balancing mechanisms if possible instead of remote adjustments by meta-data collected
- put measurement points at input and output locations of service stations

Metrics

What are the core metrics we are interested in? (We assume averages here).

- arrival and service rates, service times
- change over time (trends) in those values
- customer timeout/cancel rate (useless computation rate)
- contention and cohesion values and trends
- service station up

Which of these metrics do we need in real-time? In other words: which of those metrics can be used for immediate, possibly automated action? How is this handled in Cloud Computing?

Changes in Perspective

<<what is essential for request construction? >>

Part IV: System Components

System Components for Distributed Media

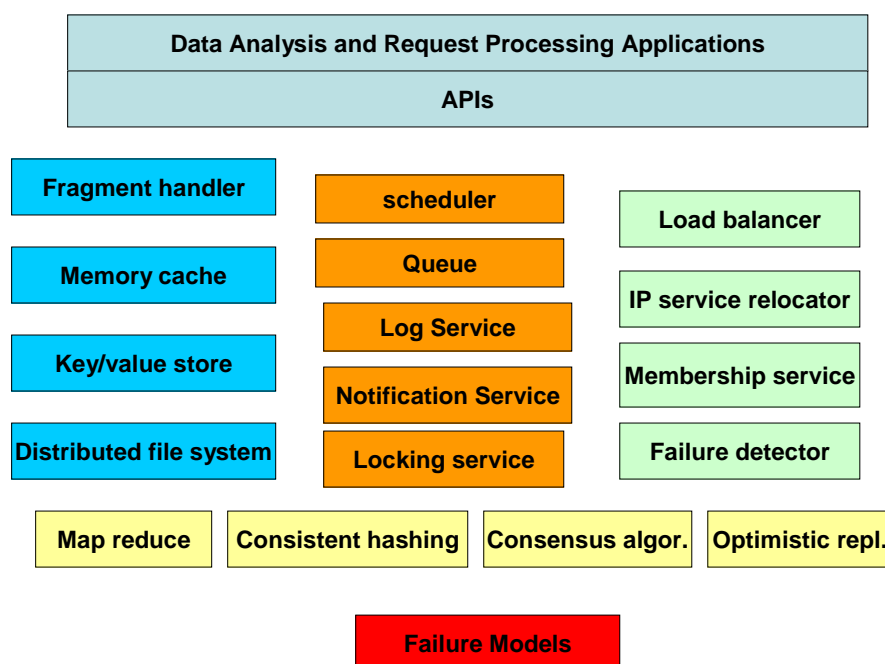
In this part of the book we are working out way down from complete sites to individual components used to scale across large numbers of requests and with a decent response time. The first chapter explains the causes of latency and how to fight them. The following chapters go into details of caching, replication and prediction as techniques for scalability. Much in these chapters is based on my own (bad) experiences from large scale portals and internet sites and I also draw heavily on wisdom collected by Todd Hoff, David Patterson, Nati Shalom and David Pritchet.

Component Interaction and Hierarchy

Latency, Responsiveness and Distribution Architecture

Low latency is not only important for shop-like applications as Todd Hoff points out in “latency is everywhere and it costs you sales - how to crush it” [Hoff] where the reader can find lots of pointers to other latency related resources. Social networks with their focus on collaboration and multi-media may be free

b



ut users still won't tolerate long waiting times for their requests. To get a grip on latency we will discuss the following topics:

- what is latency?
- How does latency develop?
- What causes latency?
- What can be done against it?

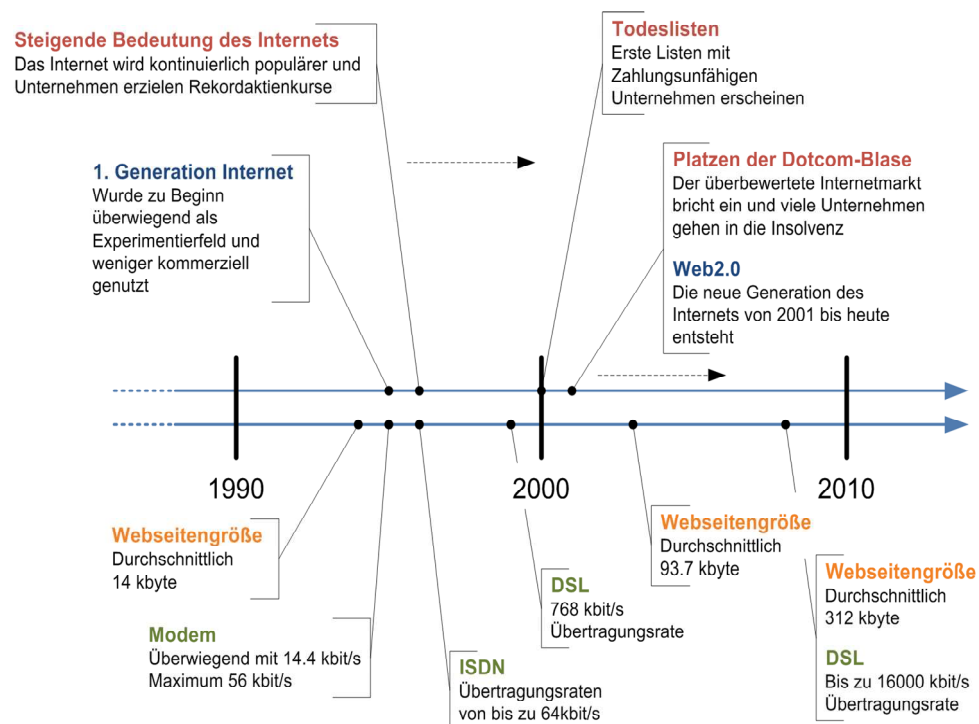
No, latency is not bandwidth (even though it has an interesting relation with it). Let's just define latency as the waiting time experienced after

sending a request until the response arrives and can be viewed or processed. Bandwidth decides how much data we will be able to send or receive in a certain time. Latency decides how fast we will get a (perhaps rather small) response. An increase in bandwidth does not improve latency but the opposite is usually true. Latency seems to be a problem that plagues especially websites since practically ever but is also extremely critical in online games, virtual worlds and realtime multimedia entertainment or collaborative and highly interactive sites. And finally and from past experience: latency is very hard to reduce once a problem is detected (and unfortunately latency problems get detected rather late in projects). Also, latency is a special and overall view of the behaviour of a system: From a latency point of view many decisions made within collaborating systems finally result in good or bad response times. How does scaling affect latency? The usual experience is that with more users/requests etc. the individual latency gets worse, sometimes event resulting in a system crash through overload.

differentiate bandwidth from latency

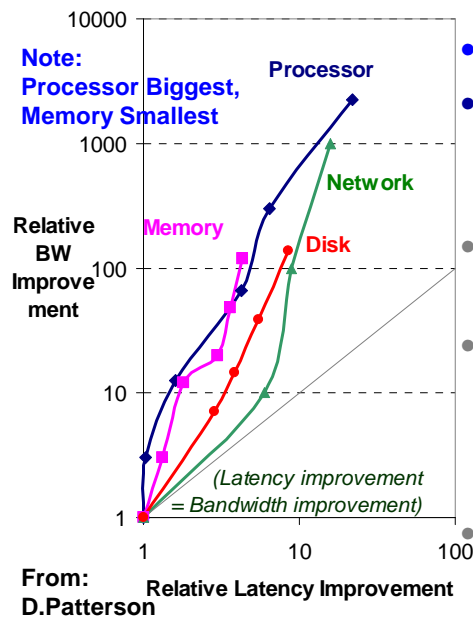
- compare with the effects of sharding

How does latency develop (compared to capacity and bandwidth)?



The slide from Till Issler [Issler] shows the growth of pages over 20 years. Users expect much more content than before and it takes a lot of punch on the server side to assemble it dynamically from different sources. David Patterson compared the development of bandwidth with the development of latency in four major areas (disk, memory, net, CPU) roughly over 20 years and came to the interesting conclusion that latency permanently lags behind bandwidth (and capacity).

Latency Lags Bandwidth (last ~20 years)



- Performance Milestones
 - Processor: '286, '386, '486, Pentium, Pentium Pro, Pentium 4 (21x,2250x)
 - Ethernet: 10Mb, 100Mb, 1000Mb, 10000 Mb/s (16x,1000x)
 - Memory Module: 16bit plain DRAM, Page Mode DRAM, 32b, 64b, SDRAM, DDR SDRAM (4x,120x)
 - Disk : 3600, 5400, 7200, 10000, 15000 RPM (8x, 143x)
- (latency = simple operation w/o contention
BW = best-case)

He also give some hints about improving latency which we will discuss shortly but the most important statement for designers and architects is that they should design for latency and not assume fundamental decreases in latency in the near future. According to Patterson the reasons for latency improvements lagging behind are that chip technology seems to help bandwidth more (pins, number of transistors), distance limitations (speed of light), better marketing of bandwidth improvements, the queuing networks of today's web sites which help bandwidth but delay requests, operating system scheduling and algorithms which favour throughput over small setup times.

What causes latency? When we follow a request from its start till a response is received and processed we notice that latency is caused by the many little delays and processing or transmission times of our request across many systems and system levels. A system call causes a context switch to kernel mode and afterwards data are copied from user to kernel buffers. Later the kernel creates a write request for a network device e.g. and the data are copied onto the wire. There the data are transmitted at a certain speed. Repeaters and other intermediates like proxies, switches etc. cause further delays and processing times. Finally at the target server our request is received and buffered. After a while an application receives a notification, changes to running mode and processes our request. The more threads or processes are busy processing other requests the more delays e.g. through context switching times our request will experience. Perhaps it needs data from backend storage systems which will cause further delays. And then the same things happen on its way back to where it originated.

Queuing theory tells us that we need to calculate the sum of all residence times in the queuing network and together with transmission and propagation times it becomes clear that the longer the service chain is the bigger the latency will get. And every component that shows especially low performance adds to it without others compensating for it.

If we look at a single processing step we notice something else: Most processing of a request shows three different phases: initialization or ramp up phase, processing phase, settle down phase. The first and the last are independent of the size of our request. They are fixed costs that apply even for a single byte. To adjust for those costs engineers tend to create wider data path or higher bandwidth connections so that more data can be transmitted or processed for the same fixed costs.

What can be done to reduce latency? From what we just said follows that many small requests are rather inefficient. We better batch requests or transmitt larger amounts of data and the same goes for disk and memory page size. Fine-grained RPC methods as have been used in classic distributed programming models like CORBA, DCOM etc. will experience a lot of latency for little data. It does not come as a big surprise that the web programming model is document centric with a larger granularity of requests.

Caching and replication have been mentionen in [Patterson] as well and from past experience I can say that they are a make or break issue for web sites or portals. It is mandatory to shorten the request path by placing data as close to the consumer as possible. Even DNS lookups should no have to travel far. Prediction is also an interesting technique to cut down on latency. Pre-fetching data is an example. Online games frequently calculate player movements ahead and disconnected from server data. Once the data from the game server cluster have reached the game client the position is corrected – which leads sometimes to jumpy movements of the character.

What else can we do on the server side or in the network? The chapter on I/O discusses strategies for efficient and fast I/O handling. A key topic here is to quickly notify applications on incoming requests. Avoidance of context switches and other concurrency techniques are discussed there as well.

Will partitioning of backend help to improve latency? This is not easy to answer correctly. At the first glance partitioning seems to improve bandwidth because it adds communication channels. One request should not get faster treatment just because of partitioning. But what if there is a queue in front of the single system and it is filled with requests? In this case distributing the request to e.g. read-only slaves will shorten latency. This is only true of course if the service times of the systems are roughly equal as we have seen in our chapter on queuing theory.

Now since we know what causes latency and what can be done to reduce it we can go ahead and optimize all request path's in our system from one end to the other. Or we can ask another question first: Where exactly is latency caused and how much of it is relevant or a call to arms? We need a good understanding of basic performance data of disks, networks, CPUs, memory etc. – this is what the chapter on hardware numbers was about. But we need something else which is a real bummer: we need to know the timings between all components involved to find out where the time is lost. And this requires a complete instrumentation of all components. As this is probably impossible to achieve we need at least to make sure that our own software is completely instrumented with timestamps and allocation/de-allocation counters (the latter just to track down irresponsible behavior by software).

Should the timestamp data directly drive scalability measures like running more virtual machines to process client requests? If responsiveness of your application is paramount to you this is probably a good idea but it comes at substantial costs as we have seen: scale-out measures are only effective if there really is an exceptionally long queue of requests lined up at the server and the latency is not in reality caused by slow backend systems. In this case having more front-end servers would be no help at all. Quite the opposite would be true: your overloaded backend systems would experience even more requests/sec. and latency would increase for all requests.

If you experience disappointing roundtrip times it will most likely mean that you will have to go with a fine comb through the complete software chain involved: Are there any bad serialization points in your processing software on the server side? You might run lots of threads but what are they really doing besides causing context switching overhead? I once had to track a performance problem in a large web site for a bank and it turned out that nine out of ten threads were always waiting at some reflection call within the Xalan subsystem needed to render the pages. Our code was multi-platform enabled and did dynamic checks for extensions available on each XSLT processor. Unfortunately the Xalan people had thought that looking up available extensions would not be a performance critical thing and put a “synchronized” statement around. We were able to move the checks to initialization time and shorten the request length considerably.

Later in this book we will cover many techniques useful to cut down on latency. Rather extreme ones like moving to “eventually consistent” algorithms which means giving up consistency for a while – or simpler ones like using a content delivery network. The sections on I/O and concurrency also provide ample opportunities to reduce the response time of requests by using more parallelism. But again: before you install an Infiniband network and scale up to x-core CPUs, twiddle with TCP and kernel settings etc. – make sure your measurements really indicate that there is a problem.

<<todd Jobson, The many flavors of system latency.. along the critical path of peak performance>>

According to Werner Vogels, CTO at amazon, the company enforces rather strict SLAs for requests: The 99.9 or 99.99 percentile of a certain request type have finish within a defined time. This leads us quickly to a core part of architecture related to latency: the creation of a distribution architecture. This can be as simple as a spreadsheet with detailed information on all our information sources and the respective protocol, responsible person, uptime range, average and slowest responses, variance in runtimes, percentiles at certain times, security issues etc.

Distribution Architecture

	Source	Protocol	Port	Avg. Resp.	Worst Resp.	Down-times	Load-bal.	Security	Contact/SLA
News	hostX	http/xml	3000	100ms	6 sec.	17.00-17.20	client	plain	Mrs.X/News-SLA
Research	hostY	RMI	80	50ms	500ms.	0.00-1.00	server	SSL	Mr.Y/res-SLA
Quotes	hostZ	Corba/IDL	8080	40ms	25 sec.	Ev.Monday 1 hour	Client	plain	Mr.Z/quotes-SLA
Personal	hostW	JDBC	7000	30ms	70ms	2 times Per week	server	Oracle JDBC dr.	Mrs.W/pers-SLA

Also add percentiles and variance for request times

Adaptations to media

Media – due to their size and timing requirements – drive even local processing systems to the limit. Media processing across different systems needs adaptations on all involved parts of those distributed systems: on the archive, producer, delivery and receiver components.

Just finding and retrieving media content requires special archive technology, organized in several processing and storage layers with frequently needed content in high-speed temporary storage and all the long term content in inexpensive tape libraries. Meta-data are used to tag any content so it can be found later again. What makes the delivery side especially critical is the fact that in many cases the requester is a human being and not a machine which means that the time to gather and deliver media is very much limited: it is the so called request time and all activities necessary to fulfill a users request have to happen during this time. Caching and replication are typical adaptations in distributed systems to deal with limited request times. We add replication especially for high-availability. Luckily in many cases with media as our main concern we are not subject to the extreme requirements for both availability as well as consistency as would be the case in an airport control tower. [Birman]

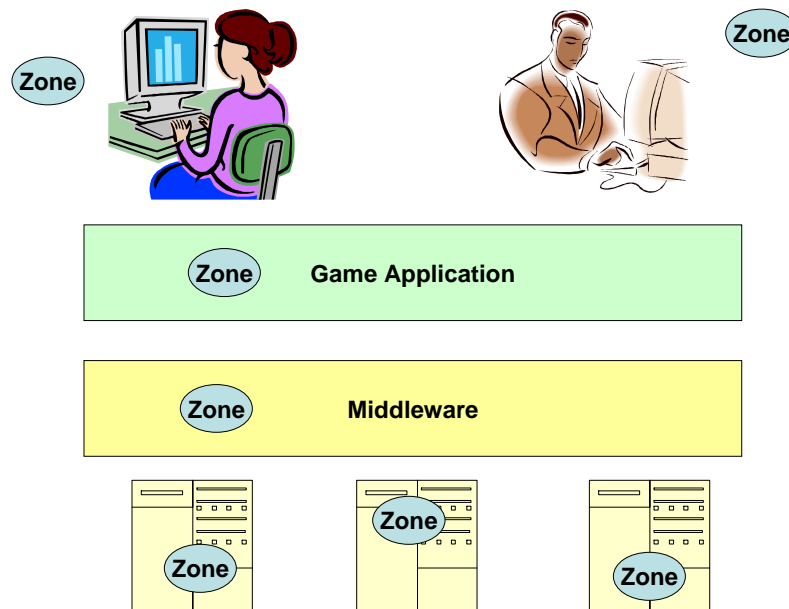
On the receiver side important adaptations for media are buffering and compensation algorithms which are able to compensate small problems in the real-time delivery process. We will see a nice example of compensation in audio streams across networks.

The adaptations necessary on the producer or processing side are in many cases what compute GRIDs or clusters (a cluster is a more homogeneous, in-house version of a GRID) can provide: ad-hoc combination of compute resources to perform e.g. rendering or image processing of media or to find and server media content in close to real-time fashion. Parallelization is one adaptation that allows e.g. parallel rendering of different frames for a computer animation.

Perhaps the most important concept of adaptation is partitioning. Partitioning simply means splitting scarce resources in a way that allows parallel access or parallel processing. Partitioning leads to independence between resources and those resources can then scale independently of each other, e.g. run on different servers. Partitioning can also mean to split complex media into fragments which can be recombined into new media containers. This way some media like complex homepages in portal architectures can be composed for every user in a different way - personalized but from a limited number of fragments. Fragments on the other hand require a proper information and distribution architecture to work and so does caching.

The downside to partitioning is that it is sometimes visible on the application level. E.g. when changing a world or zone in a MMOG is done by a transfer to a different server which is visible to the player. Or when users need to explicitly log in at specific servers to get to specific parts of a virtual world. Partitionings are just as heavily discussed as the principle of transparency in distributed systems. Some middleware for cluster computing e.g. tries to offer very fine-grained and small areas for application objects within servers (e.g. the open source game engine DarkStar, others do a coarse grained separation of action onto different servers and have no way to deal with overcrowded zones within one server.

The following diagram shows partitioning on middleware, application and user level. Ideally the middleware would be able to transparently relocate zones across machines, split zones and add more CPU power to each etc. But in many cases there are other limitations as well like the maximum number of avatars that can be displayed within a certain area so that users can still play.



Clever partitioning of the main resources in a distributed computing application saves a lot of time and money and leads to well performing applications. A distributed system with many different resources and users is currently unable to promise a complete, transparent and consistent

replication of all changes to every user. We can achieve this for special cases and limited sizes but it is not possible on the large scale.

Content Delivery Networks (CDN)

Replication is also a key concept in serving media content. Many successful online games of a large number of servers split into several data centers worldwide (see below). Online services which expect lots of requests replicate the services across a number of machines and put a load-balancing infrastructure in front of the servers.

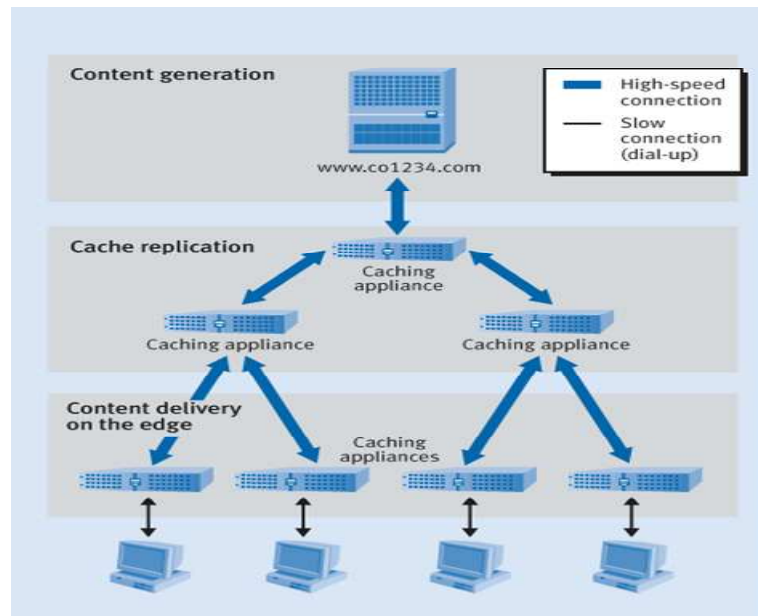
An important aspect of adaptation on the producer or sender side is the question of connections and state required to serve content. Protocols which require a permanent connection between consumer and producer will block precious server side resources for an extended period of time and do not scale well. This is one of the lesson learned from http. On the other hand those protocols must still be able to allow session state or resource sharing.

Adaptation of the delivery component knows different techniques. They range from a change in media size (compression, several levels of quality) to changes in the topology (multi-sender) to intensive use of edge caching machines. Special network protocols like multi-cast can be used where available (like for company internal TV). The use of streaming technology is also a way to control the delivery component. While e.g. an FTP server will – given enough requests – completely saturate a network channel, a streaming server will restrict network input at the configured level to avoid saturation.

Edge caching is another approach to take load from the delivery component by getting the content closer to the consumer. Companies use edge caching technology e.g. from Akamai or GroovyGecko when a larger audience is expected for webcasts etc.

The streaming of popular concerts or other events like webcasts to a large audience requires a huge amount of bandwidth at the server side as well as a high-availability infrastructure to ensure worldwide uncompromised reception of content.

But even without real multimedia content the serving of pages and images to many users stresses a companies infrastructure. For this reason edge caching networks like Akamai oder Groovy Gecko have developed. The transport content to the “edge” of the internet, i.e. closer to the final consumers. And at the same time the distribution of content ensures the scalability of events. Many companies have been caught by the so called Slashdot effect – being mentioned at Slashdot.com caused flash crowds – large numbers of users accessing the company site at the same time. The same goes for product announcements. Edge caching networks ensure enough bandwidth for an estimated number of users.



This diagram is taken from Marc Mulzer [Mulz] and describes the replication of content across several layers of caches. While this will reduce the stress on the main server machines it will significantly increase the managing efforts needed to keep those caches synchronized. A typical problem for those architectures is a sudden and important change of content which is not completely replicated to caches so that some caches will deliver outdated content even after a longer time. Distributed caches require cache invalidation protocols and in extreme cases voting protocols for global commit.

Google recently developed a new tool “WhyHigh” which was used to clarify a strange effect in CDN behaviour. According to google some clients which were closely located together and which were routed correctly to the same CDN server nevertheless suffered considerably different round-trip-times. The research paper on WhyHigh found inefficient routing (loops) and queueing delays to be responsible for those effects. It turns out that simply assuming that a CDN will create equal user experiences is wrong. [Krishnan] et.al.

Underestimating the role of caching for high-performance sites is THE major reason for unsuccessful web projects. We will come back to this topic when we discuss media fragments and personalization.

Especially interesting are currently attempts to use a more peer-to-peer like architecture for delivery of video on demand. The British BBC e.g. is trying an architecture where many different nodes can serve partial content to a consumer. The content bits and pieces are reassembled at the receiver. This avoids the typical server side bottleneck of video on demand where a server cannot deliver all content through

HA-Service Distributor

<<Whackamole, spread based>>

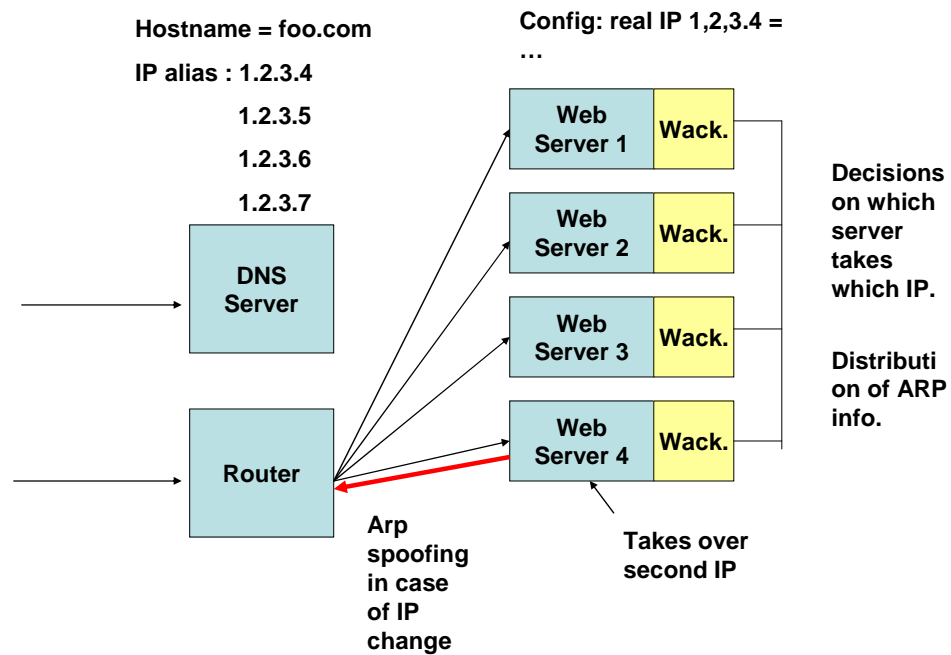
I am following Theo Schlossnagles concept of separating availability from load balancing. This separation allows us to recognize as very different services which can be implemented in a different way than the usual centrally placed load balancer/HA unit with hot standby.

What do we want to achieve?

- We want to prevent IP addresses known to clients to suddenly disappear because a server went down.
- We do NOT need transparent failover for reasons we have discussed in the clustering section above.
- We want to prevent requests being sent to “dead” hosts and hanging a long time – in other words we want immediate information on unavailable servers to prevent request stalls.
- And we want this to happen automatically without manual intervention.
- And on top of this we want this to be an inexpensive solution without having lots of specialized boxes with expensive stand-by units hanging around at every tier in our architecture.

And we can further split our component into a part that deals with IP services and how a host can service an additional IP address and a part that deals with failure detection and reaching consensus about a new, valid configuration of participating hosts. The latter sounds very generic and potentially useful for other services like replication, locking etc. That’s why we will handle this generic service later and turn it into a platform service for all kinds of other vital functions (see below also the section on component hierarchy and dependencies).

A good description of Whackamole, a peer-based high-availability component can be found in the mod_backhand description [Schlossnagle]. The mechanism differs significantly from virtual IP based load balancer/HA units which offer only one virtual IP to clients and distribute the requests internally. A peer-based HA solution looks like in the diagram below:



Two critical points are updating the ARP information in other servers when an IP address has changed to a different host. This can be done either via ARP spoofing or by distributing ARP information regularly to other hosts on the same subnet via Wackamole.

SSL Certificates in SSL connections are problematic as well as there is a binding between servers IP and the name of the service in the certificate and a whole bunch of certificates will be needed for peer based HA: Every service that works stateless or mostly stateless with some state held in a global store e.g. can be rather easily made HA with peer-based methods. Having more than just two-machine failover helps also because it allows a machine to take over more responsibilities. Wackamole supports heterogeneous hardware but if a machine takes over responsibility for another IP it needs to be able to support its services as well which puts a damper on heterogeneity of course. And can we really achieve n-1 failover with peer-based methods? N-1 in this case is only a theoretical value. We simply cannot fold n-1 loads into the one remaining server.

Distributed Load Balancers

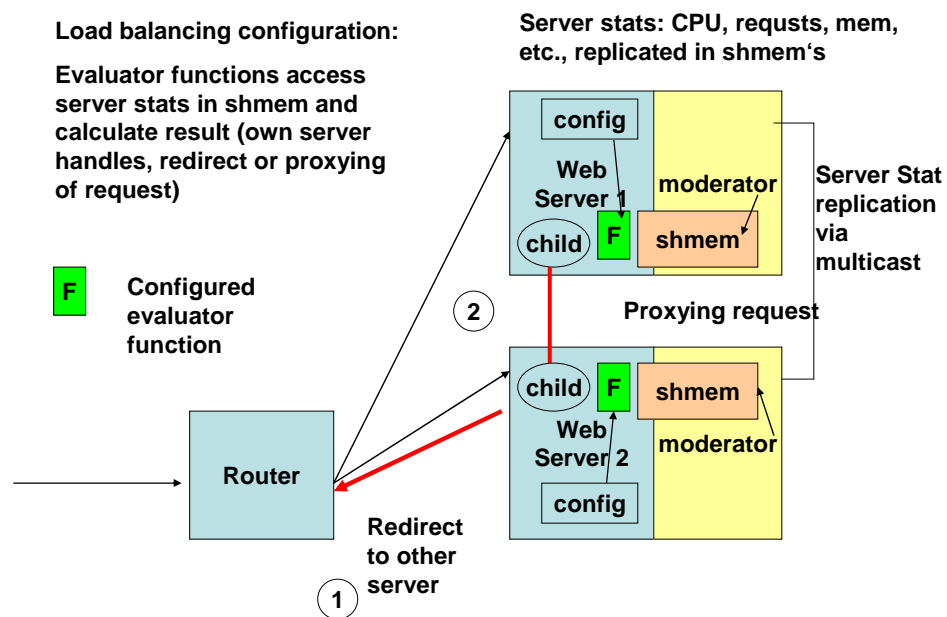
<<Mod backhand,>>

The decision to assume responsibility for an IP address is much easier than a decision to route a request to a certain server – especially if load balancing and failover are independent services so that failover does not determine who will finally handle the request. And just like in the high-availability service above we can split the service in two components: one part dealing with the replication of server statistics across machines so that every server can see them. And another part dealing with the execution of decision functions. These functions (in the diagram below designated with F) operate on the replicated server statistics and try to distribute the load evenly.

The mechanism works like that: there is a configuration of request types which tells which request should be load balanced across servers. In case of such a request a series of functions will be executed. The functions calculate a decision according to CPU load or the number of requests waiting or other parameters. Some functions implement preferences like handling a request on the receiving server. Once a server has been determined the request will either be re-directed to that server or an apache child process uses an existing connection from a connection pool to proxy the request to another server. The latter is not as effective in most cases because it forces the first server to still deal with the request by routing data back and forth. Ideally the decision is that the server who received the request originally will handle it as well.

Some things complicate load balancing enormously: many web requests are short lived (< 1 sec) and there is some overhead in replicating server statistics at a much finer granularity. A group communication protocol based on multicast though can update a small number of servers many thousand times per second – if no disk access is needed ([Birman]). We can probably use just about the same as for the failover service above.

Another problem is the weak prediction quality of parameters like CPU load. They can change so fast that they can become almost meaningless. Queue information is probably a much more useful parameter. And finally there is a chance for request thrashing when servers start re-directing requests to each other. Or requests circulating endlessly between servers. The functions can also access request parameters and detect re-directions. What can help to make load balancing easier? Strongly controlled request types and their behavior e.g. like in the SLAs of Amazon. Once we know exactly that 99.9% of a request type will finish in no longer than 2 seconds we can start calculating service times much better. Uniform hardware will also make calculations easier.



But does it have to be a push mechanism to distribute load? In the section on special web servers below we will discuss a pull based solution.

Distributed Caching – not an Optimization

There is one mechanism of adaptation that is used by almost all components, from processing to consumption of media content and this is caching. And there is a big misconception around about caching in distributed systems, based on the saying that “premature optimization is bad”. While this is frequently right in the case of caching in distributed systems it is absolutely the wrong approach: because caching in distributed systems is NOT an optimization. It is an architectural core element at several layers. And the proof for this statement lies in the fact that you can’t add caching afterwards to your distributed application without major changes. Just look at the well-know Struts architecture: without an API to ask for a cached value there is no way to re-use a previously calculated value. Instead, one has to call the specific action again to get that value.

This was fixed in the later Portle API (JSR 168).

A design of a distributed media application that does not use all available caching mechanism on the client side, network and intermediate level up to several layers of caching within backend server machines will not work at all. It will neither scale nor perform. Unfortunately the possibility to use those caching methods has to be reflected in the application design or it won’t be possible. For developers relatively knew to caching the article “Benchmark Results Show 400%-700% Increase In Server Capabilities with APC and Squid Cache” gives detailed numbers on improvements possible with caching.

[<http://www.ilovebonnie.net/2009/07/14/benchmark-results-show-400-to-700-percent-increase-in-server-capabilities-with-apc-and-squid-cache/>]

Caching and Application Architecture

Why is caching so much dependent on the application architecture?

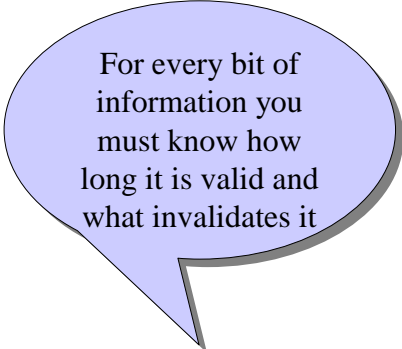
There are a number of reasons:

1. Caching requires information on the content to be cached. Can content be cached? This sounds like a stupid question but in many cases there are legal responsibilities associated with content and customers might sue if being served with outdated information.
2. How long can or must content be cached? Can is a legal aspect, must a technical aspect. Both need to form a compromise as will be shown later.
3. What about personalized content? Every piece of content served might be unique or more likely parts of it might be unique. Does this mean we cannot cache at all? Should complete content pieces (e.g. pages) with personalized content be cached?
4. What about security? Can we guarantee that the same access control rules are in place for cached content?

All these questions finally lead us to recognize that the information architecture of an application drives caching possibilities. The information architecture can simply be a spreadsheet with detailed information on each and every piece of content or content type that is used within the application.

Information Architecture – Lifecycle Aspects

Data / changed by	Time	Personalization
Country Codes	No (not often, reference data)	No
News	Yes (aging only)	No, but personal selections
Greeting	No	Yes
Message	Yes (slowly aging)	Yes
Stock quotes	Yes (close to real-time)	No, but personal selections
Homepage	Yes (message numbers, quotes) Question: how often?	Yes (greeting etc.)



For every bit of information you must know how long it is valid and what invalidates it

Frequently we will recognize also that much of the content is assembled from different bits and pieces of other content. Some of these “fragments” are personal and secret, many of them public. We can simply go ahead and on every user request start assembling the fragments and building a new piece of content that will be delivered. This is OK but as we will notice fairly quickly – it is quite expensive with respect to performance (both CPU and network). Why network performance as well? Because we now realize that those fragments are usually pulled from all kinds of backends over all kinds of protocols and with all kinds of quality-of-service associated.

Caching Strategies

There is a wide variety of caching options and strategies and just about the only one that will surely not work is to ignore caching at the start of the architecture. There is a chapter in this book on client side optimizations which includes caching as well. If you are unfamiliar with http/html level caching options take a look at the servlet book from Jason Hunter or at the book on High-Performance Web Sites.

When not to cache

Caching things makes no sense if there is no chance that the cached value is used by anybody during the lifetime of the cache value. Stream-based multimedia data are a typical example. The chunks get processed sequentially and storing them within a cache just pollutes the cache for no reason. The distribution of values across a certain type is also important: a scale free distribution (followers in twitter?) is certainly problematic to cache as only a minority of values will be used but at a high frequency [Henney]. Do you want

to cache rare thumbnails? Wikipedia seems to hold different caches for different types of content to prevent polluting a cache with information that has a low locality of reference. What about realtime information like stock quotes? They may be realtime but there is usually nothing to be said against caching them at least for a little while (20 seconds). In the worst case put a timestamp of the creation time to the values or graph so viewers can see how old values really are. This should never stop you from caching. Just about the dumbest thing I ever did with respect to caching was to not reject a business requirement for absolute realtime stock values on the homepage of a large financial portal site. Turned out that this caused huge number of XML-RPC requests against a slow backend system and it killed the homepage request performance wise.

Invalidation Events vs. Timeout

We could call it “the thundering herds of cached information” in reference to other “herds” like threads that return from waiting for a resource just to find out that the resource is busy again (see concurrency chapter) or data copies shipped around after re-partitioning of storage (see chapter on storage below). In all these cases a small change causes enormous concurrent activity to fix the situation. Here the herd is caused by some cached values becoming invalid and a whole bunch of requests is going straight for the database(s) to load the new value. Even one very busy variable can already cause this effect and lead to stalled threads at the back-end. Lucky you if your I/O is working asynchronously (see chapter on I/O) or may of your threads will simply block and wait for the result.

<<dogpile discussion>>

Invalidation of cached values is very important so make sure the invalidation mechanism is stable and able to delete larger numbers of entries at the same time.

Operational Criticality

“It is just the cache” is no longer a good argument for treating the nodes which host caches as unimportant. Our large-scale sites simple do not work without caches anymore. They would take DAYS to become operative again. This means changes to the cache infrastructure need to be incremental and the whole mechanism needs to be available 100% of the time (but not with 100% data). This is an important distinction that allows you to retire servers with only some increase in the load on backend systems.

Pre-Loading of Caches

This is highly application specific. You should really know the exact usage patterns of cached values to avoid loading the cache with unnecessary information. Content

management systems can benefit from pre-loading the caches.

Local or distributed caches

In the beginning of application servers there were only local caches available. This turned out to be one of the biggest performance problems with horizontally scaled applications. Each and every application server held its own cached values. Causing repeated access to the backends for the same data and a severe synchronization problem on top of it: if node one changed a value in the database, only its own cache got updated. The rest of the nodes would happily still serve the old data. Solving the problem with timeouts associated with the values is not really a good idea (see the discussion from above).

Distributed caches avoid those problems (I am not talking about replicating caches like the JBOSS treecache). I am going to discuss the most prominent example nowadays – memcached – below.

Partitioning Schemes

Every distributed cache needs to solve two problems: lookup must be fast from every server and a change in the cache infrastructure (e.g. more machines or a crashed machine) should not make all references invalid. The necessary algorithm is called “consistent hashing” and it is discussed in the chapter on scalable algorithms. Here we simply say that in many solutions the key of the data is hashed and compared to hashed IP addresses. The host with the smallest difference to the key hash is the one holding the value. Memcached e.g. uses a two-level hash technique to distribute values across machines and to find a value quickly within a machine.

Memory or Disk

Typically page servers for web-based content management systems use disk based caches. If they crash the cache is still available after reboot. But those caches are local ones, not distributed. It probably depends on the number of page servers, the load-balancing used to distribute load, the ability to slowly bootstrap a server etc. whether a disk cache is still a good option today.

For performance reasons the values should be held in memory as is e.g. the case with memcached.

Distribution of values

Why would you want to distribute one value across several hosts? It’s only a cached value after all. With several copies your site becomes better protected against node failures and on top of that you can distribute requests for a hot topic across several machines. But your memory requirements will double or triple. You will have to decide about the level of availability and load that your cache should provide.

Granularity

There is a simple rule regarding the granularity of cached objects: the higher the costs to re-create the value, e.g. through expensive join operations, the more important it is to cache complete, aggregated objects. At other times simply caching rows might already be enough. Twitter has some interesting papers on cache use, e.g. having one cache with only references to entries in other caches. Netlog carefully separates some values via several calls to the databases to allow incremental invalidation of only parts of objects. They trade initial construction effort against ease of use later.

Statistics

Each cache needs some administration and tweaking and this needs to be based on actual cache behaviour like hit rates. Cache instrumentation is key to performance. Unfortunately caching ruins other statistics normally: If your application can serve content from a cache the corresponding backend systems and statistic components will never see (and count) those requests. But it gets worse: Once your cache really works request numbers and behaviour in the backend systems will change dramatically, e.g. there will be much less read requests. Your architectural decisions e.g. to partition the database or to go to a No-SQL store might become questionable. This is the reason why caching is NOT a late cure for architectural mistakes which were made in the beginning (see the discussion on partitioning options below).

Size and Replacement Algorithms

<<later>>

Given that the number of followers is in all likelihood a power law distribution, tracking the mean is probably not as useful as it might first appear. For normal distributions caching with respect to the mean makes a lot more sense than for a power law distribution, which is very skewed and has potentially infinite variance. I'm not sure if the article is implying that the cache sizing is based on the mean value or whether the mean is just being offered as an interesting piece of information to make things more concrete for the reader. Kevlin Henney

<http://www.infoq.com/news/2009/06/Twitter-Architecture>

- cache coldness, cache concentration, delete after some time – problems with this approach.

Cache Hierarchies

There is not just one cache used in many web applications. It starts with the browser cache, intermediate caches (e.g. Squid), edge caches, reverse proxy caches, web server

caches, web application caches (e.g. dynacache), language caches (apc), distributed caches (e.g. memcaches), query caches of databases and so on.

Caching techniques: cache forever and explicitly expire, have a chain of responsibility. We had a generic expiration time on all objects at Digg. The problem is we have a lot of users and a lot of users that are inactive. [Chain-of-Responsibility](#) pattern creates a chain: mysql, memcache, [apc](#), PHP globals. You're first going to hit globals, if it has it you'll get it straight back, if not go to the next link in the chain, etc. Used at Facebook and Digg. If you're caching fairly static content you can get away with a file based cache, if it's something requested a bunch go with memcache, if it's something like a topic in Digg we use [apc](#).[Stump]

Memcached

I have once mistakenly thought of memcached as an in-memory database (which at that time I thought to be rather useless because most RDBMs already hold much of the data in a memory cache. Today with disks becoming tape and RAM becoming disk this might change, e.g. in the Cloud.). But memcached is no database at all, knows nothing about SQL. All it does is store key/value pairs very efficiently across a possibly very large number of servers and with the option to locate a certain value very quickly. First a client hashes a key and maps it to the responsible server for that value. Next the server hashes the key to find the locally stored value. There is no fault-tolerance nor load-balancing provided beyond a good distribution of values across machines. [Denga]

Let's start with a simple example of its use, taken from [Moon]. We need to define the server pool serving as caches.

```
$MEMCACHE_SERVERS = array(
    "10.1.1.1", //web1
    "10.1.1.2", //web2
    "10.1.1.3", //web3
);
```

Then we create an instance of a memcached client ('\$memcache') and initialize it with the server pool..

```
$memcache = new Memcache();
foreach($MEMCACHE_SERVERS as $server){
    $memcache->addServer ( $server );
}
```

Now we take a SELECT call which is either long running or of high frequency and wrap it with a call to the cache first: We first check whether the results are already in the cache, otherwise we go to the database, extract the result and put it into the cache for reuse.

```
$huge_data_for_frong_page = $memcache-
>get("huge_data_for_frong_page");
```

```

if($huge_data_for_frong_page === false){
    $huge_data_for_frong_page = array();
    $sql = "SELECT * FROM hugetable WHERE timestamp >
lastweek ORDER BY timestamp ASC LIMIT 50000";
    $res = mysql_query($sql, $mysql_connection);
    while($rec = mysql_fetch_assoc($res)){
        $huge_data_for_frong_page[] = $rec;
    }
    // cache for 5 minutes
    $memcache->set("huge_data_for_frong_page",
$huge_data_for_frong_page, 600);
} [Moon] Brian Moon, This is a story of caching,

```

This may be enough for a small to medium size website. That scale really changes many things is shown nicely by Paul Saab's discussion of adaptations made to memcached to make it perform at Facebook. [Saab]. The author mentions e.g. that connection buffer sizes became a problem eating gigabytes of RAM on memcached machines and that they had to be made shareable. Concurrent and asynchronous access by clients is another topic here. But look at the numbers given by Saab after the changes were applied:

Since we've made all these changes, we have been able to scale memcached to handle 200,000 UDP requests per second with an average latency of 173 microseconds. The total throughput achieved is 300,000 UDP requests/s, but the latency at that request rate is too high to be useful in our system. This is an amazing increase from 50,000 UDP requests/s using the stock version of Linux and memcached.[Saab]

To get an idea of what makes I/O really fast go to the chapter on Asynchronous I/O below. Extensions to memcached:

Gear6 provides a number of enhancements to standard memcached. These include:

1. *Memory utilization: Removal of the 1MB object size limit, finer grained block based memory allocation, and a cost based eviction algorithm.*
2. *Density: We use a combination of DRAM and Flash memory to lower the cost of the cache and increase the density of our solution. Currently our largest cache is 384GB per IU.*
3. *High Availability: We deploy our solution with two IU units in a cluster environment. The cluster enables two modes:*
 1. *Continuous service availability: The cluster architecture enables fail-over capabilities. This ensures that cache services are not interrupted when failures occur.*
 2. *Continuous data availability: The cluster replicates data within the cluster. This replication ensures that all cache data is always available in an alternate location, and continues to be served to users without interruption or delay. Spikes in database and application load are avoided.*

3. In addition the Gear6 Web Cache requires no client-side code modification and our cluster architecture enables disruption-free software upgrades.

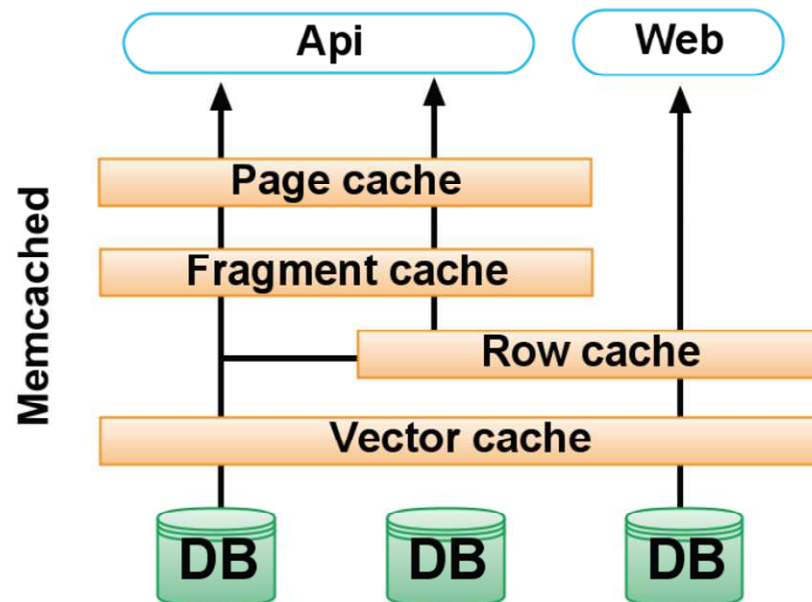
4. Reporting and Management: Gear6 Web Cache is fully instrumented and equipped with intuitive interfaces that let you see what's happening at multiple levels within your Memcached tier. We've made enhancements that automatically and continuously scan both DRAM and flash memory for failures or failure indicators. Users can drill-down on any level of their cache tier and get reports on hot keys, clients and instances.

<http://www.infoq.com/news/2009/07/webcache>

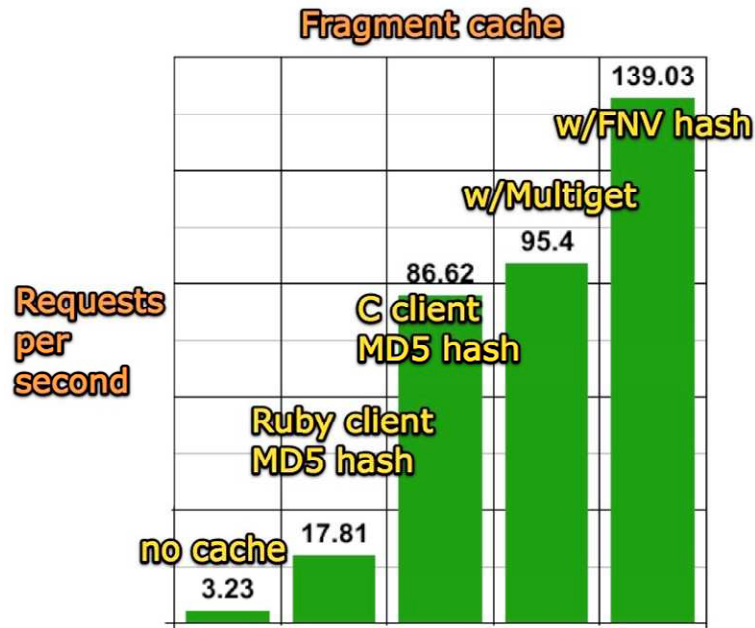
Fragment Architecture and Processor

This section could or should have gone into the chapter on caching because in all likelihood fragment handling will be done in the context of caching. When I looked at some early twitter architecture diagrams or read some papers I was surprised about the little use of caching they made. If I remember correctly the API access branch had little caching and the web part nothing at all.

This has changed obviously as the diagram below shows. It is taken from a blog entry by >> who discussed the later architectural changes to Twitter. [Weaver]



The diagram shows four levels of memcached. A fragment layer assembles re-usable bits and pieces of information across users. And look at the performance data below! The difference between uncached and fully optimized caching is almost fifty! And it shows that further optimizations like multiget or FNV <<describe>> still make a difference too.



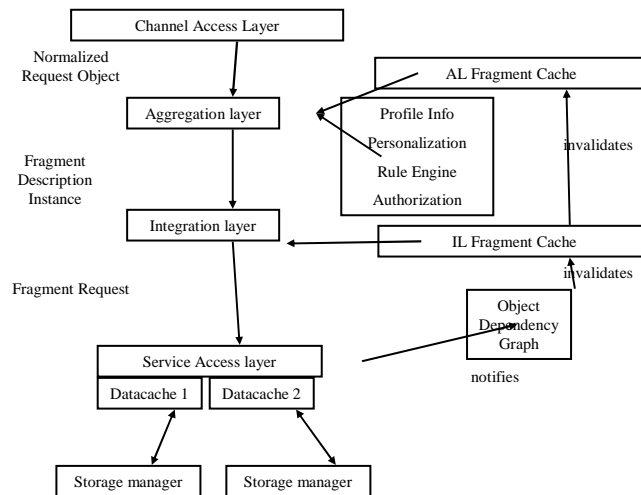
From [Weaver]

So what is a fragment architecture and how do you build one? A fragment architecture is basically a simple realization on your part. You have to realize that pages or information containers delivered to clients might be unique, customized etc. – that they still contain re-usable bits and pieces which can be used to assemble pages or containers for other users. This is sometimes not easy to realize because “personalization” in context with security make things look so very unique. But this is wrong. If you disregard the composability of pages you will learn some very hard facts about multi-tier systems: that by going to the backends for each and every bit of information will simply kill your application. This has been a core lesson learned from building large portal sites and yours truly has made that mistake once and hopefully only once.

Given your information architecture we can start to build a fragment architecture that allows us to cache fragments of different granularity, assemble them to bigger content units and at the same time guarantee that cached items will be removed if a fragment within a larger unit gets updated. For this purpose we need to capture dependencies from larger pieces on smaller fragments in the information architecture. And we have to build a runtime system that tracks the dependencies between parts, much like e.g. a relationship service from CORBA would track the deletion of embedded elements (relational integrity). Only that we would invalidate inside out: a small fragment which becomes invalid causes all its embedding “parents” to become invalid as well. Caches like Webspheres Dynacache today allow this kind of invalidation even across machines (edge cache).

This technique had been used by IBM Watson for the Olympic sites and the architecture below has been an adaptation for a large financial organization made by myself. See [Kriha] for a paper describing certain scalability problems in building large portals).

Fragment Based Information Architecture

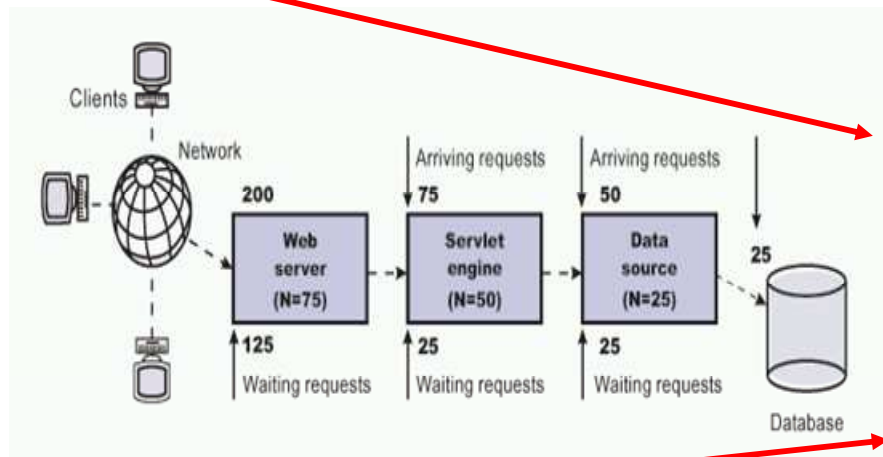


Goal: minimize backend access through fragment assembly
(extension of IBM Watson research)

Again, we don't have to do this but if we don't we will quickly learn the number one performance rule of all websites, portals, community sites etc.: In complex multi-tier applications avoid unnecessary backend requests like the plague. And the second one: Realize that all the aggregation and processing of the content needs to be done within a reasonable overall request time. Users don't wait. This limits your options for processing the content considerably and needs any kind of preprocessing of fragments etc. that is possible at all.

The diagram above mentions a SAL layer – a Service Access Layer which shields the application from unavailable or sluggishly responding backends and services. If you can't prevent the requests from your customers to reach out to unavailable services you have no control of your application. Your application will show strange behaviors depending on the availability of important services. It will stall, block and in the worst case crash due to resource exhaustion (threads, memory etc.). Controlling threads and other resources is important, there is no doubt about. But just like many other services (load balancing, IP failover service) also the fragment processor finally relies on a failure detection service which we will describe below.

Ideally your application and its different parts will form a kind of funnel that restricts incoming requests and avoids overruns.

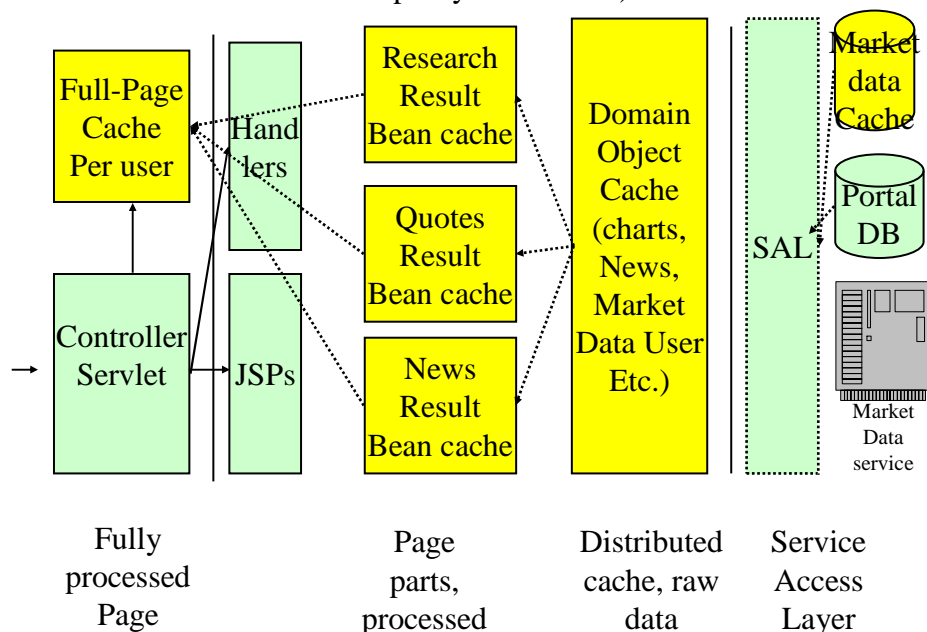


<<example of web-application funnel architecture>>

Queuing theory will help you construct the proper limits on the various entries into your application parts. But all control does not help if the threads simply block waiting for unavailable services. If you are interested in the gory details of such problems take a look at my paper on Enterprise Portal Architectures where those typical RAS issues are discussed and solutions provided [Kriha02]. Surprisingly after all these years I frequently come across brand new application designs where the same issues of reliability, scalability and availability are being completely ignored – initially...

<<portal caching architecture ,including IBM paper ref>>

Cache fragments, locations and dependencies (without client and proxy side caches)



And if I may add a third one it has to do with availability and reliability. Many content serving applications need to access different internal or external services to get fragments. With Web2.0 mash-ups have become extremely popular. Some of those are aggregated within a server application (acting like a proxy for the browser). If you look at your distribution architecture you will notice how much your application depends on the availability of all the internal and external services it uses.

Compression

The next level of adaptation is the media content during transport and storage. Compression has easy to use (e.g. via apache plug-ins) and the current development of CPU power makes the trade-off between size and time rather easy to decide. Besides browser compatibility there are no issues with compression and we will skip it for this reason here. But there is much more than compression that makes media fit for distribution: Important considerations are size (e.g. page size in www), identification (how to find the media), meta-data for better retrieval, round-the clock availability (gamers never sleep around the world), fitness for different delivery channels and formats (mobile phones vs. PC). These issues go back to the design of the media themselves and need to be solved right at the beginning. A typical example is the poor handling of images on many web sites. They are frequently of bad quality and load slowly. But there are also examples of sites with tons of pictures which are both of high quality and load blindingly fast (see www.skatemag.de).

Even more interesting is the adaptation of content for distribution.

Examples are QoS considerations for portal content (does a homepage of a financial institution need to provide absolute real-time quotes or can they be cached for a certain period of time. 10 seconds can make a world of difference as this can mean that thousands of backend requests can be avoided if a cached version can be used. The slight degradation in QoS makes the whole concept workable.

Another interesting case is to apply a divide and conquer approach on both technological as well as content level. This is e.g. used in Massively Multi-Player Online Games (MMOGs) where hundreds of thousands of users play concurrently. No central server infrastructure would be able to handle all those concurrent sessions. Therefore the game itself is divided into so called worlds which then map to different server clusters. As a player can only be in several worlds at the same time this means that the workload can be split between different clusters. The guiding principle here is that the content itself – here the game idea – supports the requirements of a large scale distributed system.
<<storage compression, wikipedia>>

Local or predictive processing

Up to now most adaptations for media in distributed systems were targeted at servers or intermediates. But the receiving client side truly becomes a center of adaptations in the case of interactive applications like multi-player online games (MMOGs) or collaborative environments like Croquet (see www.opencroquet.com). The techniques used here separate local processing time from network request time and allow for advance planning of actions on remote machines, processing of different scenarios in parallel or even distributed two phase commit. Advanced replication mechanisms

are used as well which then again include a hierarchy of storage and processing components to shorten response times.

<<diagram of croquet level architecture>>

These mechanisms will be discussed in the chapter on MMOGs and Collaborative environments. Like porting an application to a parallel processing platform like MP this approach requires a detailed analysis of potential parallelism between clients and server code. This analysis must also minimize communication load for synchronization and it is very likely that even the game design and story elements will be selected in a way to support potential independence.

Another client side adaptation has become very popular lately:

Asynchronous Javascript and XML (short AJAX) has dramatically increased client side processing. While much of this processing is done to improve usability or to create new features, it is also possible to use it for a different purpose e.g. to take load from servers. Instead of the server rendering data completely the raw data will be loaded by the client and formatted by the client processor.

<<AJAX emample of client side rendering>>

Search Engine Architecture and Integration

<<FAST example, wikipedia lucene use, separation of operation and analysis, background, cluster>>

<<anatomy of search engine, scalability in several dimensions: number of documents to index, index size, query numbers>>

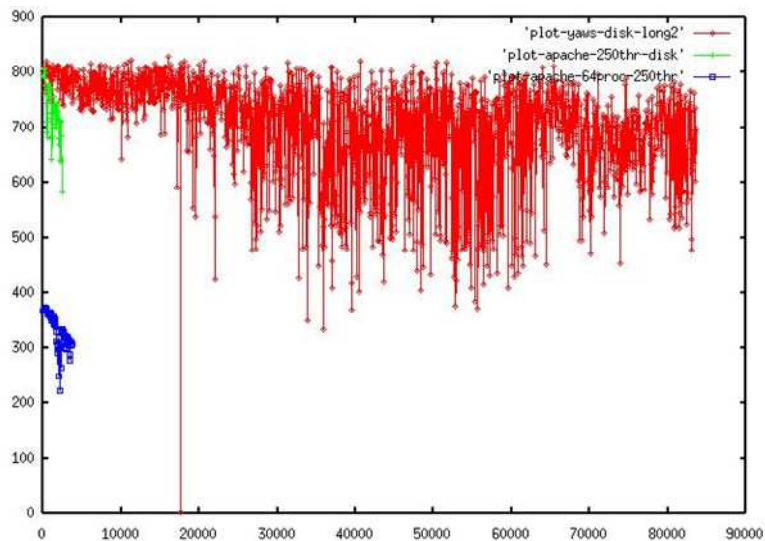
Explain separation of OLTP and OLAP systems.

[Jayme...] Jayme , Scaling/Optimizing search on netlog

Special Web Servers (light-weight)

[Laird] Cameron Laird, Lightweight Web Servers.

Youtube and others use special purpose web servers. What are the differences? Trade-offs?



Apache vs. Yaws (Erlang), from: A.Ghods, <http://www.sics.se/~joe/apachevsyaws.html>

This amazing diagram shows apache vs. yaws throughput. Apache dies at around 4000 requests/sec. The interpretation by Joe Armstrong sees the use of kernel processes and threads from Linux as the main limiting factor of the apache performance. Erlang does not use system threads. But then there must be a translation method to map those threads to different user threads within the language.

A pull based Web Server Design?

Idea: do not push requests from a load balancer to web servers, let the web servers pull the requests depending on their load. Is this feasible? Trade-offs? Would this work without LB in front (e.g. using whackamole, spread, backhand)?

What exactly is the queuing model behind the LB-WebServer combination? I think it represents parallel queuing centers, not parallel processors because each web servers has its own queue. If – due to irregularities within the processing units some service times are much longer the web server queue will be full with new requests because the LB won't be able to react quickly enough. Hajunka does not work properly in this case. Idle web servers cannot take requests from the queues of busy servers.

So either we reduce the wait queues to one within the LB and have web servers poll or we could use a group communication protocol between web servers that allows request stealing. I guess always the next request should be taken. This would require some communication to the LB as well because suddenly requests would be answered by servers which did not receive those requests from the LB.

Scheduler and parallel Processor

<<gearman etc.>>

High-availability failure detector

Whackamole (IP), group communication protocols?

and lock service

chubby <http://www.jgroups.org/>

Buffering and compensation for networked audio

Adaptation does not end at the network component level. Even at the receiver side a lot of adaptation to the distribution of media content happens. The well known browser cache is one example. Buffering of audio/video input is another. Media-Players typically don't start playing a stream right away. Instead, for a configurable amount of time or data they buffer content and start playing only when the buffer is full. Special delivery protocols try to speed up this phase (see Microsoft Media Player architecture) by starting a stream with a burst phase to fill the buffer and then fall back to the regular streaming bit rate.

Buffering unfortunately comes with the problem of buffer over- and underruns. This is finally caused by clock skew problems between sender and receiver machines. Several solutions for this problem exist, some of which still show visible or audible artifacts. One possible solution is to

timestamp every content package and try to calculate the clock deviation from those timestamps. Single content frames would then be added or removed depending on whether the receiver was about to be overrun or underrun. Other solutions try to scan the content for similar frames which could be duplicated or removed without major affect. Unfortunately these concepts still produce artifacts and sometimes require huge buffers. Stefan Werner describes in his thesis a very interesting alternative: instead of monitoring the clock skew he decided to change the playback speed in a way that kept the buffer reasonably filled. Changing between two different playback speeds provided a feedback loop that finally made the buffer level adjust to the real clock skew between the machines evolved.
<<diagram skew compensation algorithm>>

This is by far not the end of adaptation on the receiver side. It goes as far as the design of Application Programming Interfaces for video applications in a way that allows the applications to individually handle problems like lost frames or delayed content. Ideas to treat those in a generic way and to relieve the applications from the complexity of dealing with those problems have resulted in disastrous APIs which were both cumbersome to use and slow. An excellent source for lessons learned in the design of video APIs is the paper by Chris Pirazzi "Video I/O on Linux, lessons learned from SGI" [Pirazzi]. The author also maintains lurkertech.com, an excellent site for all kinds of information on video, e.g. how to deal with video fields on computers.

Data Center Architecture

[ALV] Al-Fares, Loukissas, Vahdat, A Scalable, Commodity Data Center Network Architecture (condo concept)

About routing etc. within data centers.

What are the pitfalls in multicast? For replication and caching? Performance and throughput?

Microsoft research: data center design for the cloud with geo distribution: the condo model vs. the big datacenter. Cost models etc.

Network cross-switch times (google FS paper). Multi-distribution.

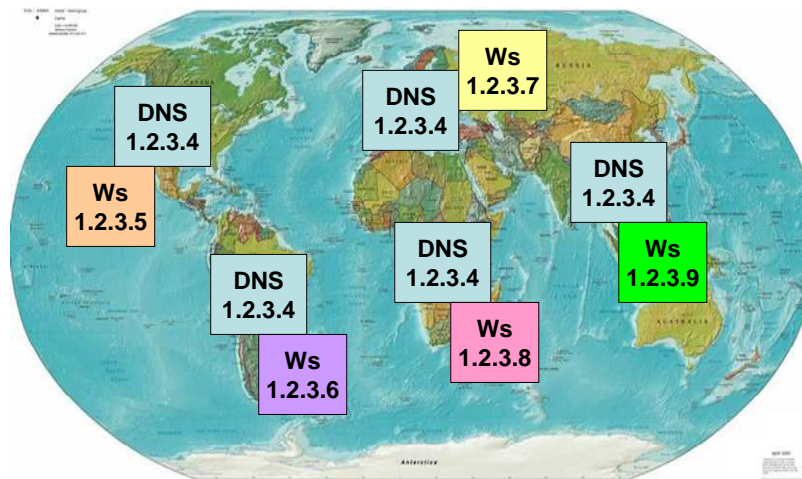
Geographically Dispersed Data Centers and Topology

- Master/Slave sites
- DNS Round-trip-time calculations for short path and fast responses (Schlossnagle)
- Anycast
- Licensing and financials
- Slow lines slowing down the application servers response (Schlossnagle)
- [CDK] R.Cocchiara, H.Davis, D.Kinnaird, Data Center Topologies for mission-critical
- Two/three site architectures, Disaster Recovery

[Cooper] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, HansArno Jacobsen, Nick Puz, Daniel Weaver and Ramana Yerneni, PNUTS: Yahoo!'s Hosted Data Serving Platform,

<http://highscalability.com/yahoo-s-pnuts-database-too-hot-too-cold-or-just-right>

Running an application in several datacenters across the world is becoming normal for large sites. The reason to distribute is not the physical proximity between clients and servers but the shortest/fastest connection between them. Distributed data centers pose a lot of problems, mostly related to replication and disaster recovery. Here I will only mention two techniques for routing clients to the best server as described in [Schlossnagle]. The first one is DNS Round-Trip-Time calculation in clients which automatically leads to the fastest responding DNS server becoming the “authoritative” one. Unfortunately changes in the Internet can make that association problematic. The other solution “anycast” is described in the diagram below. Here several local DNS servers in our distributed installations run with the same IP address and are announced to the networks. This way clients will automatically get routed to the “nearest” DNS server which is available over the shortest path. There is a chance that the next client request gets routed to a different DNS server which makes connection oriented protocols like TCP problematic because a different server did not see the initial parts of the connection and will refuse the continuation (wrong sequence number e.g.). The solution is to use UDP for DNS lookup and return the address of the local web server who will be in the same network necessarily.



Map from:
landkartenindex.blogspot.com

Scale-out vs. Scale-up

[Atwood]

June 23, 2009

[Scaling Up vs. Scaling Out: Hidden Costs](#)

In [My Scaling Hero](#), I described the amazing scaling story of plentyoffish.com. It's impressive by any measure, but also particularly relevant to us because we're on the Microsoft stack, too. I was intrigued when Markus [posted this recent update](#):
 Last monday we upgraded our core database server after a power outage knocked the site offline. I haven't touched this machine since 2005 so it was a major undertaking to do it last minute. We upgraded from a machine with 64 GB of ram and 8 CPUs to a **HP ProLiant DL785 with 512 GB of ram and 32 CPUs** ...
 The [HP ProLiant DL785 G5](#) starts at \$16,999 -- and that's barebones, with nothing inside. Fully configured, as Markus describes, it's [kind of a monster](#):

- 7U size (a typical server is 2U, and mainstream servers are often 1U)
- 8 CPU sockets
- 64 memory sockets
- 16 drive bays
- 11 expansion slots
- 6 power supplies

It's unclear if they bought it pre-configured, or added the disks, CPUs, and memory themselves. The most expensive configuration shown on the HP website is \$37,398 and that includes only 4 processors, no drives, and a paltry 32 GB memory. When topped out with ultra-expensive 8 GB memory DIMMs, 8 high end Opterons, 10,000 RPM hard drives, and everything else -- by my estimates, it probably **cost closer to \$100,000**. That might even be a lowball number, considering that [the DL785 submitted to the TPC benchmark website](#) (pdf) had a "system cost" of \$186,700. And that machine only had 256 GB of RAM. (But, to be fair, that total included another major storage array, and a bunch of software.) At any rate, let's assume \$100,000 is a reasonable ballpark for the monster server Markus purchased. It is the very definition of **scaling up** -- a seriously big iron single server.

But what if you **scaled out**, instead -- [Hadoop](#) or [MapReduce](#) style, across lots and lots of inexpensive servers? After some initial configuration bumps, I've been happy with the inexpensive Lenovo ThinkServer RS110 servers we use. They're no match for that DL785 -- but they aren't exactly chopped liver, either:

Lenovo ThinkServer RS110 barebones	\$600
8 GB RAM	\$100
2 x eBay drive brackets	\$50
2 x 500 GB SATA hard drives, mirrored	\$100
Intel Xeon X3360 2.83 GHz quad-core CPU	\$300

Grand total of **\$1,150** per server. Plus another 10 percent for tax, shipping, and so forth. I replace the bundled CPU and memory that the server ships with, and then resell the salvaged parts on eBay for about \$100 -- so let's call the total price per server \$1,200.

Now, assuming a **fixed spend of \$100,000**, we could build **83** of those 1U servers. Let's compare what we end up with for our money:

	Scaling Up	Scaling Out
CPUs	32	332

RAM	512 GB	664 GB
Disk	4 TB	40.5 TB

Now which approach makes more sense?

(These numbers are a bit skewed because that DL785 is at the absolute extreme end of the big iron spectrum. You pay a hefty premium for fully maxxing out. It is possible to build a slightly less powerful server with *far* better bang for the buck.)

But there's something else to consider: software licensing.

	Scaling Up	Scaling Out
OS	\$2,310	\$33,200*
SQL	\$8,318	\$49,800*

(If you're using all open source software, then of course these costs will be very close to zero. We're assuming a Microsoft shop here, with the necessary licenses for Windows Server 2008 and SQL Server 2008.)

Now which approach makes more sense?

What about the power costs? Electricity and rack space isn't free.

	Scaling Up	Scaling Out
Peak Watts	1,200w	16,600w
Power Cost / Year	\$1,577	\$21,815

Now which approach makes more sense?

I'm not picking favorites. This is presented as food for thought. There are at least a dozen other factors you'd want to consider depending on the particulars of your situation. Scaling up and scaling out are *both* viable solutions, depending on what problem you're trying to solve, and what resources (financial, software, and otherwise) you have at hand.

That said, I think it's fair to conclude that **scaling out is only frictionless when you use open source software**. Otherwise, you're in a bit of a conundrum: scaling up means paying less for licenses and a lot more for hardware, while scaling out means paying less for the hardware, and a *whole* lot more for licenses.

* I have *no* idea if these are the right prices for Windows Server 2008 and SQL Server 2008, because [reading about the licensing models makes my brain hurt](#). If anything, it could be substantially more.

Data Stores

Social sites inevitably need to deal with multi-media data in large proportions. This content needs to be read, written, searched, backed-up and delivered in different qualities (resolution, thumbnails) to different clients. The same goes for the archives of broadcast companies. And it is not only multi-media content that is needed. Sites need structured and semi-structured data to handle users, relations etc.

Until lately the answer to those requirements would have been either an RDBMS or a traditional file system. But with the trend to ever larger sites like amazon.com, google.com and others new forms of data stores have been invented: semi-structured column stores like Google's bigtable, key-value stores like Amazon's Dynamo and distributed filesystems like GoogleFS, ClusterFS, Frangipani, storage grids and last but not least distributed block-level stores. What is different in those architectures? Basically it is the relaxation of traditional properties of stores as we know them since many years. Posix compatibility for

file systems, transactional capabilities and strong consistency in relational databases and so on. But those large sites have discovered that they may not need all those features and the associated price in performance and throughput. They discovered that by dropping certain assumptions and store properties they could get a better performing store and they were willing to pay the price, e.g. by letting applications deal with conflicts in the store. It is the classic pattern of dropping requirements, relaxing unnecessary quality rules and pushing decisions higher up towards application semantics.

Let's start with some terminology and a collection of store criteria which define the different store types:

Requirements and Criteria

- memory store or persistent
- standard posix or SQL interfaces, REST or non-standard APIs
- unstructured data (files, key/value), semi-structured (bigtable), structured (RDBMS)
- read oriented vs. write oriented or neutral
- sequential access vs. random access
- large data sets vs. small data sets
- latency vs. bandwidth
- ACID or relaxed consistency (eventually consistent)
- Conflict resolution when (read/write) and where (store/application)
- Replicated data vs. non-replicated
- Customizable store properties vs. fixed properties
- Flat or hierarchical namespaces
- Consistency vs. availability (CAP behaviour)
- Behavior in case of extension, scaling
- Caching vs. non-caching
- Data integrity and security
- Multi-hop lookup vs. zero-hop
- Central meta-data vs. distributed meta-data
- Symmetric vs. Asymmetric design
- Failure detection and behavior
- Simple Search vs. structured search
- Many requests or few requests
- Heterogeneous hardware or standardized hardware
- Commodity hardware or special
- Load-balancing and availability guarantees
- Capacity requirements
- Programming models

<< categorization of store technologies and requirements, use dynamo paper for a start >>

The big storage categories that we know about: databases, filesystems, key/value stores and column stores, memory databases are finally made of combinations of those properties. Some of the properties can be shared, some seem to be very typical – category shaping – properties like the ability to work on highly structured data for an RDBMS.

<< terminology >>

virtualized storage:

Einig sind sich Hoff und Shackelford auch bei ihrer Kritik an den führenden Herstellern von Hypervisoren. Denn diese stellen laut Shackelford immer noch keine Dokumente bereit, die den Umgang mit virtualisierten Storage- und Netzwerkkomponenten erklären. Selbst der ebenfalls an der Diskussion beteiligte VMware-Verteter mochte nicht widersprechen und gab zu, dass es noch keine Unterlagen hierzu gebe. Dabei sind diese Themen unter Umständen ungleich komplexer als die Virtualisierung von Servern, so dass Best-Practice-Dokumente nötiger sind denn je. Insbesondere die ständig größer werdende Anzahl von virtuellen Netzwerkkomponenten sieht vor allem Hoff mit Sorge: Neben dem virtuellen Switch des Hypervisors finden sich in virtualisierten Umgebungen demnächst noch physikalische Netzwerkkarten, die selbst virtualisieren können, virtuelle Switches von Drittherstellern, Netzwerkinfrastruktur, die wie Cisco's Nexus selbst virtualisiert und nicht zuletzt der Direktzugriff der VMs auf die eigentliche Netzwerkhardware des Servers.

<http://www.heise.de/newsticker/Sichere-Virtualisierung-Viel-Laerm-um-beinahe-nichts--/meldung/136612>

External Storage Sub-Systems

Block-level, NAS, Properties of SAN, virtualized SAN etc. for scalable storage. ISILON Systems, The clustered storage revolution.

- server independent storage with multiple access paths to data
- hidden reliability mechanism by RAID levels
- transparent for client software
- scale with respect to capacity but not with concurrent access to several files [Bacher]. Why not?

<<FOB>>

[EMC] Storage Systems Fundamentals to Performance and Availability
<http://germany.emc.com/collateral/hardware/white-papers/h1049-emc-clariion-fibre-chnl-wp-ldv.pdf>

GPFS, [Schmuck]

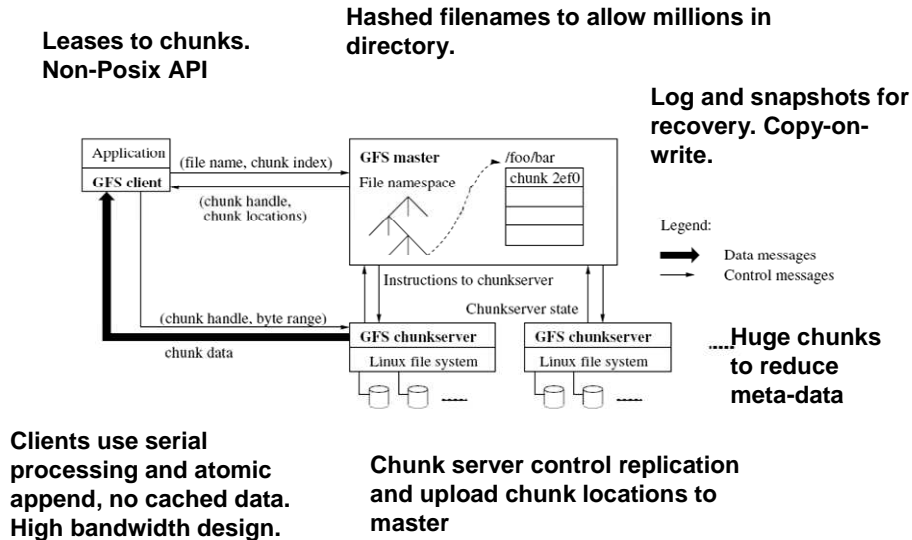
Grid-Storage/Distributed File Systems

GoogleFS is a typical representative of highly-specialized data stores for sites with huge un- or semi-structured bases of information. Key to understanding its architecture are the observations from google engineers on workload, processing etc. They discovered that:

- most files were read and written sequentially
- appending writes were frequent, random writes almost non-existent
- files sizes were huge
- only google controlles applications would use it and could be therefore co-developed. No strict Posix-compatibility needed
- 1000s of storage nodes should be supported
- Bandwidth much more important than latency
- Some inconsistencies tolerable
- No data loss allowed
- No extra caching needed

- Only commodity hardware available

This lead to a special storage-grid like architecture which is depicted in the diagram below. (taken from [Ghemawat] et.al. and extended)



Clients who want to read a file need to contact the master server first. It controls all meta-data like filenames and the mapping to chunks. Chunks are the basic unit of storage. They are huge compared to other filesystems (64MB) and they map to regular Linux files on so called chunk servers. The huge size of chunks keeps meta-data small. The separation of meta-data server and data server is a well known design pattern and is found in p2p systems like Napster as well.

To achieve reliability and availability each chunk can get replicated across several (typically three) chunk servers and in case one of those servers crashes a new one can be built up using replicas from other machines.

The master server maintains the name space for the file system. At start-up it collects chunk locations from the chunk servers and holds all mapping information in memory. Special hash functions allow millions of files within a directory. To achieve reliability and availability the master server runs a log which is replicated to a backup server. At regular intervals snapshots are taken and the log is compacted. Copy-on-write technology makes creating snapshots a fast and easy process.

Google says they have separated control and data path to achieve higher throughput and bandwidth. This means the master server has meta-data on network configurations and will make sure that chunks are distributed in a way that makes writing the three replicated chunks fast. The client writes data to those replicas and then selects a primary from the three chunk servers holding chunk replicas. The primary orders client commands and this order is then repeated at all replicas leading to a logically identical

replica at each node involved. Logically because errors during this process can lead to padding data added within chunks. This means chunks have an internal meta-data structure as well and they need not be physically identical with their replicas.

GoogleFS does not offer extra caching of chunks at the clients or servers. No cache invalidation is needed therefore. As most clients process a file sequentially anyway, caching would be futile. If the bandwidth to a file is too small, the number of replica chunks can be increased.

What kind of consistency guarantees does GoogleFS provide? A client who wants to write to chunks needs a lease from the master server. The master can control who writes to files. Most writes are appends and for those the GoogleFS provides special support: appending is an atomic operation. There is no guarantee to clients that their atomic append operation will happen at exactly the position they thought they were inside the chunk. The primary chunk server creates an order between append operations but makes sure that the individual append is atomic. Google applications are written in a way to expect changes in order and deal with them. Google applications according to [Ghemawat] also use the atomic appends as a substitute for an atomic queue for data exchange: one application writes and the other one follows reading. This allows also the implementation of many-way-merging (de-multiplexing). The principle that clients need to deal with the idiosyncrasies of GoogleFS is visible also in the handling of stale replica chunks. Clients are allowed to cache chunk locations and handles but there is no guarantee that no concurrent update process is happening and the replica chosen is stale. As most writes are atomic appends in the worst case the replica will end prematurely and applications can go back to the master to get up-to-date chunk locations.

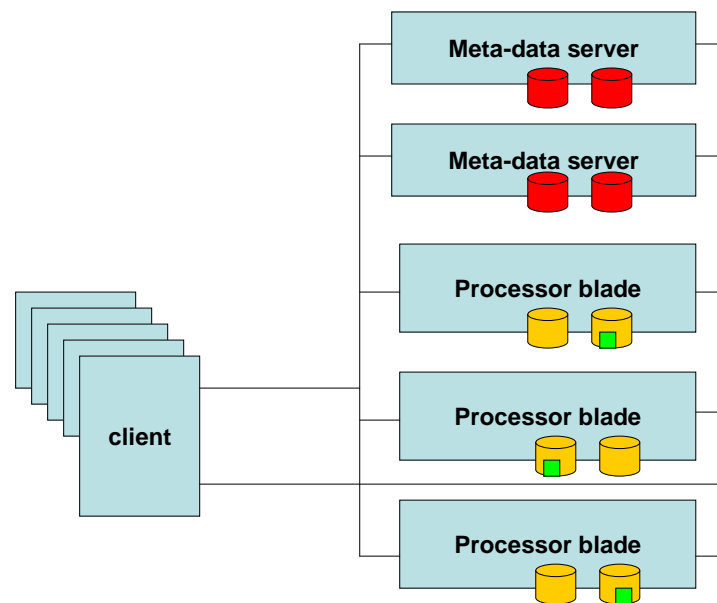
The master server can lock chunks on servers e.g. during critical operations on files. Chunk servers are responsible for data integrity and calculate checksums for each chunk. Silent data corruption happens much more often than expected and this process ensures correct replicas.

Isn't the master a natural bottleneck in this architecture? It may look like this but the data given by Google engineering says something else: the amount of meta-data held by masters (due to the huge chunk size and the small number of files) is rather small. Many hundreds of client requests seem to be no problem. The hard work is anyway done by the chunk servers.

The googleFS architecture based on commodity hardware and software is a very important building block of the google processing stack. Bigtable e.g. maps to it and many other components. The whole approach looks so appealing compared to regular drive arrays that other vendors have started to build their storage solutions also in a grid-like manner with master and slave servers running on standard blades. We will discuss one such approach for video storage, the Avid Unity Isis, below. It supports non-linear editing of HDTV video and has some different requirements

compared to googleFS, most notably the need for realtime data at high-throughput rates. Here replication is used as well but for a different purpose.

While traditional SAN or NAS storage subsystems typically present a single access point to data for clients the new grid-storage based systems use a well known pattern from peer-to-peer architectures: A split system with meta-data servers (directors) and active processors which manipulate and serve data.



The diagram shows a typical storage-grid architecture (also called “active storage”) with two redundant meta-data servers and several processor blades connected by two switched networks (switches not shown). Peer-to-peer systems are famous for their scalability and storage grid vendors claim “infinite scalability” of their architectures. Every processor blade that gets added to the grid increases bandwidth and processing capacities within the grid.

A closer look at the architecture reveals that it is not a pure p2p system due to the meta-data servers used. They are needed as a means to improve e.g. lookup performance by providing a central meta-data store – something that pure p2p systems have a problem to guarantee. Napster used a similar architecture to allow fast lookup of meta-data (where certain files are) and at the same time to delegate the raw data traffic to peers.

The storage grid excels in bandwidth, latency and redundancy as well as recovery time after a disk crash. As parts of files are distributed across the blades access to data can be parallel. A replication level of three (three copies per data unit) leads to a highly redundant system which – in case of a drive failure – starts to duplicate blocks across the whole storage cluster in parallel. This is much faster than the necessarily sequential access to a new disk in a RAID.

The downside of this architecture is exactly what causes the excellent bandwidth and latency in the first place: the loss of transparency between clients and data processors. Only during an initial phase are the meta-data servers contacted. Later clients and blades communicate directly and clients learn about data locations. If at a later time bottlenecks in the distribution of data show up special client software is needed to e.g. use alternate locations.

With respect to scalability the meta-data server presents a possible serialization point as well as the switches used to connect the components and the clients.

A special feature of storage grids is the ability to perform processing of media data within the grid. Transcoding e.g. can be performed on the processor blades. The effectiveness of those transformations probably depends on how localized processing can be done: if the processing can be done without access to further data units stored on other blades then only the costs for synchronization and control between transformation agents need to be paid. If on the other hand processing cannot only be done based on local information – as is the case in some forms of image renderings, see the example of distributed rendering with 3DSMAX, the costs of processing are comparable with the case where it is performed by the clients themselves.

<<

MogileFS <http://www.danga.com/mogilefs/>

S3: grid with focus on latency

The role of data copying and de-normalization in scalable systems: [Hoff] on using lots of disk space to de-normalize data in the context of e.g.

Google Bigtable [Chang et.al.] datastores. Tips on how to use BigTable and data duplication.>>

<<Lustre>>

Distributed Clustered Storage

Isilon Systems sells a storage system for unstructured information that looks rather similar to a distributed file system like GoogleFS except for one thing: The company claims that the system does not use central meta-data servers. Instead, all nodes within the common namespace have all meta data and all nodes can accept reads and writes for every file. And they recommend Infiniband as a high-speed network layer. According to the company papers [Isilon2006] Isilon Systems, Absolute Zuverlässigkeit durch Clustered Storage and [Isilon2006] Isilon Systems, Die Revolution des Clustered Storage the system scales linearly up to 88 nodes with an overall storage capacity of 500 Terabyte with a redundancy factor between 2 and 8.

Unfortunately the company does not say which distributed algorithm is used. The claims are interesting for a couple of reasons: First, common experience with distributed systems shows

that totally distributed systems suffer from performance and latency problems. Information can be quickly located when meta-data are everywhere but what happens when we need to write? Run a distributed lock manager as the company says? This means we need to update x machines in a consistent way using a locking algorithm.

Distributed locking can be done synchronously or asynchronously (relaxed). Synchronous locking is rather expensive and the alternative suffers from consistency problems. With respect to the distributed locking algorithm Infiniband could make a difference due to rather short latencies (which reduces the gap between necessary wait-times and actions necessary in case of node failures) and high availability of the network.

Multicast solutions will probably not scale up to 88 nodes. The company papers also claim a performance problem with separate meta-data servers due to overload. This has not been the case e.g. with GoogleFS because the meta-data machines do not server regular data. Most architectures which separate meta-data from data serving show little problems with the meta-data servers.

The Isilon architecture looks very interesting even though it runs contrary to many other distributed architectures which usually distribute a namespace across machines using either an algorithm (e.g. Distributed hashtables) or a mapping list (meta-data server). On the other hand: algorithms which involve up to 88 machines (or a majority of those) might have some serious problems with progress making in case of special failures..This needs further investigation. (The fact that Isilons “OneFS” is patented does not give me a warm feeling either – who would invest in a technology that is proprietary but presents one of the most important interfaces for a company?)

ZFS

- Logical volume manager integrated
- Silent data corruption
- Disk, raid and memory!
- Managemt for resize etc.
- Files per directory
- Fixed file size (subversion: small, video:big)
- Problem: FS nicht im kernel, dumme interfaces
- Disk scrubbing
- No overwriting of blocks, always new block and new version
- Versioning with snapshots

Database Partitioning and Sharding

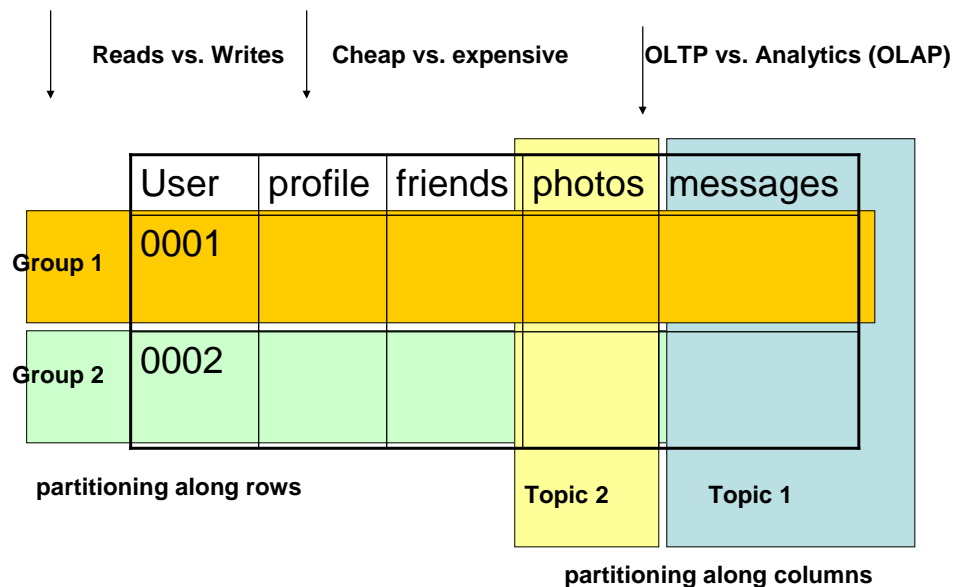
One of the best introductions to sharding and partitioning that I found is made by Jurriaan Persyn of Netlog. “Database Sharding at Netlog” is a presentation held at Fosdem 2009

<http://www.jurriaanpersyn.com/archives/2009/02/12/database-sharding-at-netlog-with-mysql-and-php/> [Persyn] and covers the basic sharding principles as well as the rationale behind breaking up your database. Interestingly the reasons also include maintenance of tables and not only performance problems.

The roadmap described by Persyn mirrors the one of Myspace to a certain degree:

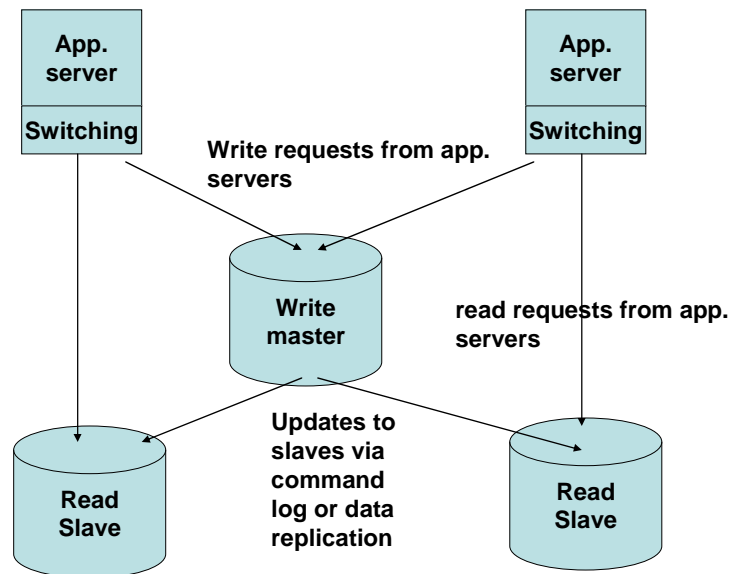
1. One server running application code and database
2. Split servers with one running the application and a separate server for the database
3. More application servers added which turns the database slowly into a bottleneck
4. Decide whether to scale-up the database server (e.g. going to a huge multicore, multicpu machine with 64bit architecture and 20 or more gigabyte of RAM. Or to disassemble the database into smaller units and stick with cheaper but more hardware.

At Netlog they were hitting the database with 3000+ requests/sec. during peak hours which caused performance and stability problems. They decided to go with cheaper but more hardware and started to disassemble the database. A database can be split along several dimensions, ranging from use criteria to categorical and growth criteria.



Perhaps the easiest and most common way to get some relief for the database is to separate read from write traffic. Assuming a rather high read/write ratio (100:1 in many cases, for social sites 10:1 seems to be a better value) we can scale out read traffic across a number of read-only replica servers ("read slaves"). Write traffic gets indirectly scaled by relieving the single write master from doing most of the reads.

<<diagram of read slaves>>



Partitioning a database according to its use (here read/write ratio) has been very common with large scale websites (wikipedia e.g. used such a set-up successfully for a while). Today this architecture has seen increasing criticism and we are going to investigate some of the reasons. First, the number of slaves is actually quite limited. Every read slave needs to be in sync with the master and with a growing number of slaves synchronization gets harder and harder. Second, we do not really scale the writes by introducing read slaves. We are actually replicating/duplicating writes across our system and thereby increasing the work that needs to be done for writes. Third, to keep the split maintainable we need a switching logic within the application servers that will transparently route reads and writes differently. Perhaps hidden in a database class which has separate instances for reads and writes. Dynamic system management should update available read slaves to achieve at least read availability. We do not improve write availability at all.

One interesting example in this context is the discussion around the read/write ratio of large sites. From looking at presentations about those sites we know that this ratio seems to be a critical factor for performance and scalability. But which r/w ratio do we actually mean? The one before introducing a caching layer in front of the DB? Or the one after? Let's assume we have a 10:1 ratio before which might be quite typical for a Web2.0 application. This led to the decision of separating read/write traffic by using a write master and several read slaves. After introduction of a caching layer this ratio might drop to 1,4:1. In the light of this change, was our DB optimization really useful? With this ratio we are no replicating almost every second request across our read-slaves! And with the overall requests reduced considerably by the cache – do we really need database partitioning at all? All these additional slave servers will cause maintenance problems due to hardware/software problems. They will lag behind in replication and cause inconsistent reads. And finally: do not

forget that these read servers will have to carry an increasing write traffic due to updates as well! We could easily end up with read slaves carrying a much higher load than the write master (who does only those reads which **MUST** be consistent – another ugly switch in our application logic) and becoming a bottleneck for the system.

Premature optimization without looking at the overall architecture (and request flow) might lead to suboptimal resource allocation.

Read/write separation is not the only way to partition a database according to its use. A very important distinction can be made between regular traffic which results from operating the system (usually called OLTP) and analytically oriented use (usually called OLAP). Of course the borders between the two are not set in stone but are design decisions. And here a very important design decision could be to absolutely keep analytics away from the operational databases. This means no complicated queries or joins whatsoever are allowed. In this architecture an asynchronous push or pull mechanism feeds data into a separate database which is then used for long running statistical analysis. Synchronization is less of an issue here. Typical use could be to calculate hits, top scores etc. in the background and post those data in regular periods. Never try to do those calculations in request time or against the operational databases.

A slightly different partitioning is along the complexity of requests. Not only queries and joins can cause a lot of load within a database. Even simple ordering commands or sorting does have a price. Some sites (e.g. lavabit) decided to minimize the load caused by sorting and put this responsibility at the application code. Yes, this has been a no-no! Do not do in application space what is the databases job. And certainly the database can do those things much more effective. But so what: the application tier scales much more easily than the database tier and scaling out via more application servers is cheap but scaling up the database server is expensive and hard.

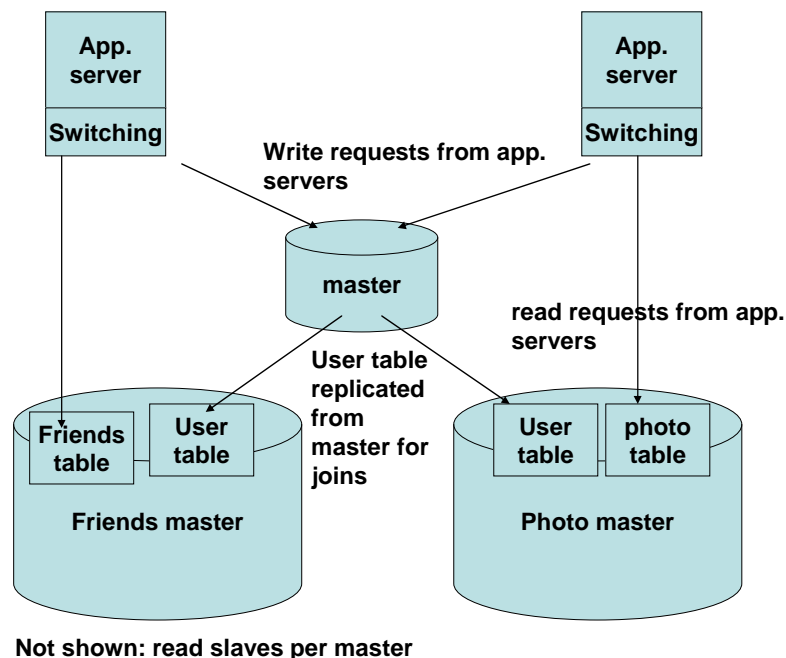
Talk about being expensive: stored procedures in masses are a sure way to cause database scalability problems. What was said about ordering or sorting is true also in case of stored procedures: try to avoid them to keep the database healthy and push the functions as much as possible into the application tier. One of the few things I would **NOT** push onto the developer is maintaining timestamps for optimistic locking purposes. And perhaps some relational integrity rules, but just some.

Finally, search engines can cause similar overhead and should be treated just like analytical programs by getting a separate replica of the database which is not tied into regular operations. Spidering or extracting data via connectors puts a lot of load on a database or application and needs a different path to data therefore. (See Jayme@netlog.eu for a presentation on scaling and optimizing search).

Up till now we have not really done any scaling on the write requests. The next partitioning scheme tries to separate write traffic according to topics.

It is called vertical partitioning and what it does is splitting the master table into several tables using the columns as a discriminator. In the example below “friends” and “photos” are now in separate databases and tables and hopefully there won’t be any joins needed involving those tables. But just in case joins become necessary there is a common pattern available that helps: replicating certain tables used for joins across databases allows complex selects and joins again. At the price of an increased synchronization effort or perhaps a sometimes inconsistent data tier.

Of course read slaves can be used to further offload read traffic in a vertically partitioned system. And it should be clear that vertical partitioning makes the switching code in our application logic even harder. Application access to several shards at once does also suffer from serialization costs. We will discuss ways to solve this problem when we present scheduling algorithms for parallel requests.



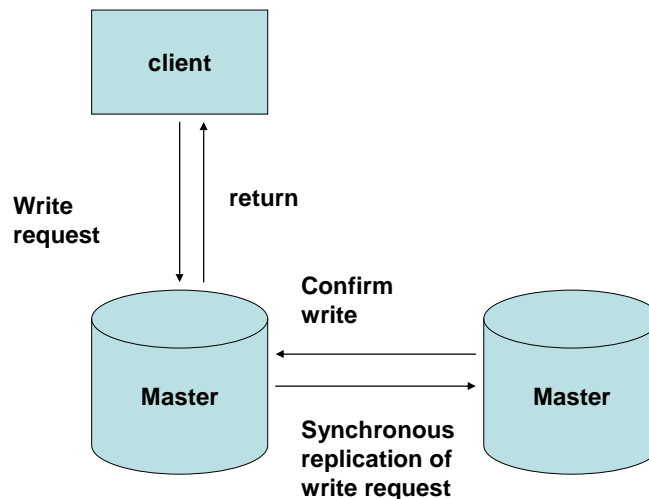
The last partitioning concept we are going to discuss is horizontal partitioning. It is needed once a tables number of rows grows extremely and causes problems along two dimensions: the sheer size of the table can cause maintenance problems when replicas need to be created or re-synchronized, during backup procedures and schema changes (alter table e.g.). And the number of connections can exceed database limits (assuming that the number of rows within a table reflects a growing use of the table as well, e.g. due to increasing numbers of users). The number of connections is quite database specific and finally depends on how those connections are implemented. Oracle connections are well known heavy-weight resources which are not only costly to created but limited in their numbers as well. An Oracle connections is mostly mapped to an operating system process which is itself a heavy-weight resource. MySql connections seem to be thread based which sounds much cheaper than an operating system process. But once we get into the hundreds of

threads we will experience serious memory allocation and context switching costs. This is discussed in depth in the chapter on I/O models. Ideally the databases would be able to separate connections from threads and dynamically assign both to each other. Such a concept of multiplexing requests across threads has been successfully used as asynchronous I/O within telecommunication equipment.

Persyn discusses other option like master-master replication or cluster set-ups [Persyn]. He points out that those architectures are geared towards better availability and single request performance, not scalability. In the case of master-master replication this is quite obvious:

As every master has to send his write requests also to the other master the number of writes per master does not get reduced.

<<master-master replication diagram>>



It is less clear in the case of cluster solutions, especially those which could work with tables across machines

<<check mysql cluster>>

The concept of horizontal partitions or shards has been used in MMOGs since many years. Everquest or World-of-Warcraft put their users into different copies of world-pieces called shards, effectively splitting the user table along the rows. This has some unfortunate consequences like friends not being able to play together when they got assigned to different shards and a new generation of game software (see Darkstar below) tries to avoid sharding at all, e.g. by further reducing the granularity of resource allocations and assignments.

So how is horizontal sharding done? First two decisions are needed according to Persyn: which column should be the key for the shards and how should the keys be grouped into shards (the shard partitioning

scheme) [Persyn]. Both decisions are dependent on your application, your data and finally require that you come up with a navigation scheme as well: how do you want to reach which data along which path? But again, also with horizontal shards duplication of other data might help to reduce navigation costs. <<check feasibility>>

A typical key is e.g. the userID given to your customers. Several algorithms can be applied to this key to create different groups (shards).

- A numerical range (users 0-100000, 100001-200000 etc.)
- A time range (1970-80, 81-90, 91-2000 etc.)
- hash and modulo calculation
- directory based mapping (arbitrary mapping from key to shard driven by meta-data table)

All methods are rather easy to apply but differ vastly with respect to their maintenance costs and effectivity. Effectivity is determined mostly by two factors: the first being how equal keys are distributed across shards to generate equal load, and second how equal the keys behave with respect to load. Numerical ranges seem to be safe with respect to distribution: we can simply define equally large ranges, or? The problem lies in changes over time: who says that after some time all keys are still alive and in use? It could be the case that ranges which were filled early in the lifecycle of a site are by now rather empty because users quit after some time. And who says that all those keys are still active? Older ones could be almost dormant and cause little load while the later ranges include many power users. The same arguments go for time ranges: distribution and activity can change dramatically leaving some shards idle and others very busy. Hashing a column value and applying the modulo operation will do a time-independent distribution of keys across shards and will probably also distribute power users equally. But what happens if a shard gets too big and needs to be split again? Using a naïve hashing/modulo algorithm will suddenly invalidate all our shard keys. Using a consistent hashing algorithm (see below the chapter on scale agnostic algorithms) will at least leave the majority of keys valid. Ideally one should know the final number of shards up-front <<check virtual shards>> which is never a good requirement.

Changes in the number or time ranges are not quite as dramatic but will require application changes in the mapping of ranges to shards. The most flexible solution with respect to growth and behavioural changes as well as maintenance problems is the directory or meta-data approach. Here a special table holds keys and their mapping to shards. We pay the price of one indirection as every application first has to lookup the shard but it allows us to change the location of keys within shards arbitrarily, e.g. by distributing power users equally across shards. This meta-data pattern is well known and used in many architectures, e.g. the media grid active storage systems for HDTV multi-media content. Persyn lists requirements for a sharding scheme and implementation:

- allow flexible addition of possibly heterogeneous hardware to balance growth
- keep application code stable even in case of sharding changes.

- Allow mapping between application view and sharding view (e.g. using shard API against a non-sharded database)
- Allow several sharding keys
- Make sharding as transparent as possible to the application developer by using an API.

At Netlog they decided to go with a directory based sharding strategy.

Now we need to discuss the consequences of a sharding strategy and how they can be made less painful. Two important techniques need to be presented in this context: support through a caching layer and how it works with shards and parallelizing requests against separate data sources.

Let's start with the consequences as mentioned by Persyn. The first problem that comes to mind is that there are no cross-shard queries anymore. This is something your architecture has to accept and compensate for by avoiding the need for those queries – which requires careful planning. Do not separate tables which need to be contacted during regular queries. One way to achieve this is to de-normalize data by keeping separate copies of tables in different shards. Persyn mentions the table of messages posted which could be stored both in the posters shard as well as in the shard of the receivers. What is the limiting factor in de-normalization? It is the need to keep the copies in sync which gets harder with the number of copies as many replication concepts had to learn the hard way.

Another – brute force approach – is to parallelize the requests for tables in different shards. While certainly possible e.g. with the use of distributed fork techniques like Gearman (by Danga.com) its limiting factor is the increase in network traffic to all servers that it causes. The beauty of shards lies in routing certain queries only to certain servers and not in creating a multicast-like scenario where all shard servers are kept busy by one request which is split and parallelized.

<<duplicated tables, parallelized queries>>

Parallel reads can lower multi-shard access costs by reducing the latency. But are they inherently bad due to increased load distribution? Why can memcached do so many requests per sec. and DBs only so few? Is this a result of the threading model used within databases?

Data consistency and referential integrity are now the applications job. Because one logical table is now split into several different instances in different databases it is not possible to use globally unique foreign keys and globally unique auto-increment mechanisms. It is now clear why many system architects of large scale sites warn against the use of auto-increment: it does not work in the context of sharded tables in different databases because it is a database local mechanism. So are regular transactions which can guarantee data consistency in the presence of concurrency. The alternative lies in using distributed transactions which are definitely a no-no given the high load of large scale sites.

Which again puts the responsibility for consistency at the application. It needs to use compensating actions in case some update to some shard went ok and others went wrong but they belonged to the same higher level unit of work.

Balancing shards is actually a second order scalability problem: You have successfully split your data into independent pieces in different databases across different servers to maximise requests per shard. And suddenly you realize that the original splitting schema does no longer give the intended results (equally distributed workload) because by some coincidence some shards concentrate power users or older shards lose users due to cancellation of membership etc. This problem is rather easily solved if you are using a directory based partitioning scheme – in other words some form of virtualization – which lets you move users (shard keys) in arbitrary ways between partitions. The meta-data based approach of directories works well also in the context of heterogeneous hardware which needs to be balanced across users.

Persyn mentions two other important side-effects of sharding: network load increases due to several independent requests to several databases and the number of database connections available might become a limiting factor. It is essential that your application does NOT keep connections open during the whole request. Otherwise the limited number of database connections will not suffice to serve the high number of requests due to the split. This may be different across databases but is generally certainly a good advice.

It should have become clear that partitioning and sharding are far from being transparent to applications. They need to understand the ways data have been split and they need to understand how to integrate data across shards without causing too much traffic or overhead. A central role in this architecture plays the distributed cache in front of the databases and its use has to be covered by special APIs.

Cache concepts with shards and partitions

- cache as a join-replacement
- cache complex objects, not only rows
- use separate queries to allow targeted invalidations of cache content

At Netlog they store cross shard data as complex objects in memcached which basically works like a distributed database integration layer in this case. This also explains the various comments from site architects that memcached should not (only) be used to cache row data (meaning single shard data) but to keep the joined data across shards in the cache because joins are rather expensive with many single database shards. This rationale may even result in a query strategy which seems to be less optimal from a database point of view, e.g. because opportunities to combine different queries are not used.

The following query example from Netlog shows the architectural dependencies between sharding and caching:

Query: Give me the blog messages from author with id 26.

1. Where is user 26?

The result of this query is almost always available in memcached.

2. On shard 5; Give me all the \$blogIDs (\$itemIDs) of user 26.

The result of this query is found in cache if it has been requested before since the last time an update to the BLOGS-table for user 26 was done.

3. On shard 5; Give me all details about the items array(10,12,30) of user 26.

The results for this query are almost always found in cache because of the big hit-ratio for this type of cache. When fetching multiple items we make sure to do a multi-get request to optimize traffic from and to Memcached.

Because of this caching strategy the two separate queries (list query + details query) which seemed a stupid idea at first, result in better performance. If we hadn't split this up into two queries and cached the list of items with all their details (message + title + ...) in Memcached, we'd store much more copies of the record's properties.

There is an interesting performance tweak we added to the "list" caches is that. Let's say we request a first page of comments (1-20), we actually query for the first 100 items, store that list of 100 in cache and then only return the requested slice of that result. A likely, following call to the second page (21-40) will then always be fetched from cache. So the window we ask from the database is different then the window requested by the app.

For features where caching race conditions might be a problem for data consistency, or for use cases where caching each record separately would be overhead (eg. because the records are only inserted and selected and used for 1 type of query), or for use cases where we do JOIN and more advance SQL-queries, we use different caching modes and/or different API-calls.

This whole API requires quite some php processing we are now doing on application level, where previously this was all handled and optimized by the MySQL server itself. Memory usage and processing time on php-level scale alot better then databases though, so this is less of an issue. [Persyn]

The mechanism of using release numbers as part of the keys is also quite nice:

** Each \$userID to \$shardID call is cached. This cache has a hit ratio of about 100% because every time this mapping changes we can update the cache with the new value and store it in the cache without a TTL (Time To Live).*

** Each record in sharded tables can be cached as an array. The key of the cache is typically tablename + \$userID + \$itemID. Everytime we update or insert an "item" we can also store the given values into the caching layer, making for a theoretical hit-ratio of again 100%.*

** The results of "list" and "count" queries in the sharding system are cached as arrays of \$itemIDs or numbers with the key of the cache being the tablename + \$userID (+ WHERE/ORDER/LIMIT-clauses) and a revision number.*

The revision numbers for the "list" and "count" caches are itself cached numbers that are unique for each tablename + \$userID combination. These numbers are then used in the keys of "list" and "count" caches, and are bumped whenever a write query for that tablename + \$userID combination is executed. The revisionnumber is in fact a timestamp that is set to "time()" when updated or when it wasn't found in cache. This way we can ensure all data fetched from cache will always be the correct results since the latest update.

Cache invalidation by new keys is a clever way to perform those invalidations without resorting to crude mechanisms like timeouts which can lead to huge traffic spikes (see caching strategies).

Next to memcaching sharded data Netlog uses parallel processing and a separate search engine to separate analytical processing from regular operations. Parallel processing means in this case splitting larger requests (e.g. to find the friends of friends for a user which has hundreds of friends (or followers)) into smaller tasks. While sounding unreasonable it seems to be true that the overhead caused by splitting a big task into many smaller ones can lead to a much faster execution of the overall request. But this must not always be true as we will discuss in the section on queuing theory where we show an example that benefits extremely from combining several requests into a larger one. <<add to algorithm section as well, can we explain the effect using queuing theory? E.g. that smaller requests of equal service time lead to better throughput?>>

Why Sharding is Bad

But there are also critical voices against sharding and partitioning of the DB. Bernhard Scheffold posed the following hypothesis (after reading the article from Zuckerberg on Facebook's architecture): Much of sharding and partitioning of the DB is simply premature optimization. Developers do not understand the set-theoretical basis of DBs and try a functional approach instead. Bad db-models create scalability problems. The DB would scale way longer than the typical developer suspects, given a decent data model.

And about the database connections: If 1024 connections to a DB are not enough it could be a problem with connection use, e.g. applications holding on to one connection for too long.

Social data examples and modeling:

most popular,
friends notification
presence indication
How scalable is the data model in opensocial.org?
<<task: evaluate scalability of opensocial schema>>

Partitioning concepts and consequences

- master/master, master/slaves, read vs. write partitioning (wikipedia)
- Scalability Strategies Primer: Database Sharding by Max Indelicato [Indelicato]
- MySQL Scale-Out by application partitioning. [Sennhauser (Various partitioning methods for data, e.g range, characteristics. Load, hash/modulo. Application aware partitioning)
- Partitioning and caching
- Database table key organization for scalability [Indelicato]
- Hscale, MySQL proxy LUA module (www.hscale.org) with some interesting numbers on DB limits discussed [Optivo]
- Vertical, horizontal, partitioned, dimensional partitioning, main lookup,

Some sites might be approaching 1 billion users in the future (skype article on PostgreSQL to scale to 1 Billion users). Netlog is using beyond 4000 shards on 80 servers. They report better maintenance of data as well due to the smaller size of shards. There is now a whole layer between application and shards which encapsulates the knowledge necessary to access the right shards from within the application. Again, sharding is way from being transparent for applications but it can be put into a special layer and therefore hidden mostly from the application. The problem lies also in the proper ways to partition your data up-front which is really hard to do.

The hardest part about implementing sharding, has been to (re)structure and (re)design the application so that for every access to your data layer, you know the relevant "shard key". If you query details about a blog message, and blog messages are sharded on the author's userid, you have to know that userid before you can access/edit the blog's title. [Persyn]

And like other site architects the Netlog people report a much better scalability of the application (server) layer than the database layer.

Data Grids and their rules of usage

Billy Newports blog:
http://www.devwebsphere.com/devwebsphere/websphere_extreme_scale/

February 06, 2009

Best practises on building data models for elastic scaling

I just read an excellent summary of the principles of doing this at this site

<http://highscalability.com/how-i-learned-stop-worrying-and-love-using-localhost-disk-space-scale>. The points especially relevant to achieving this for

WebSphere eXtreme Scale are the following.

WebSphere eXtreme Scale are the following.

Duplicate data, don't normalize it

Here, this is how common data is handled. The comments are a great

example. Duplicate the comments in to each partition and the partition

is then keyed by the main key. This allows logic for the main key to be

handled within a single partition without having to talk with other partitions which are almost 100% going to involve network IOs.

Group data for concurrent reads

Here, group related data needed for the partitioned entity underneath

this object. WXS provides a tree schema for each partitionable entity.

Placing all needed data linked to this tree keeps it all local and eliminates network hops to fetch it. This is really an amplification of

the "Duplicate data" rule.

Structure data around how it will be used

Model the data in a form compatible with the business logic to be executed on it. This makes writing the logic fast and keeps the data close the to logic. This avoids joins.

Compute attributes at write time

Add extra attributes with commonly calculated values, don't use queries

to calculate them, update the total attribute when something changes and

just query it. Assuming the queries are more frequent than the updates

then this saves a lot of time.

Create large entities with optional fields

This again is to avoid the small entities created when using a fully normalized model. Normalizing means joins and joins are

expensive so try

to avoid them if at all possible.

Define schemas in models

The framework like WXS can't manage these denormalized models automatically for you. You'll need a model which knows how to do this and does it automatically when changes are written to the model. This model can run inside the grid collocated with the data so it's going to run fast.

Place a many-to-many relation in the entity with the fewest number of elements. This basically says that rather than having a model which has a Company has a collection of employees, have a model with companies and employees with a list of companys. The list in the latter case is MUCH smaller than the other way.

Avoid unbounded queries. This is kind of dangerously obvious but if you have a tera byte of data in a grid, don't ask for a sorted list of all records and send it back to my client app. The app will die. Bound it to the top 10 or 20 items.

Avoid contention on datastore entities. This kind of goes without saying. If you use a single record in the grid all the time then it's going to bottleneck there so try to avoid or rather don't do this.

Summary

The linked article is pretty cool and summarizes much of what we already knew about the best practises on designing for DataGrids. So, here it is, enjoy.

Bernhard Scheffold:

Offenbar mißbraucht er eine relationale Datenbank als DataStore. Diese Fehlsicht scheint ja leider weit verbreitet zu sein, aber allein schon das Statment ' Structure data around how it will be used' weist ganz stark darauf hin. In einer relationalen Datenbank geht es eben darum, Daten auf alle möglichen Weisen zu verbinden und so Aussagen über das Modell zu gewinnen. Wenn er lediglich Datensätze möglichst flott "retrieven" will, dann sollte er vielleicht eher auf einen DataStore oder ein ODBMS setzen?

Ein anderes Leckerli ist 'Compute attributes at write time'. Das dahinterstehende Problem der wiederholten Berechnungen lässt sich doch weit eleganter und sicherer mit Memoization lösen.

'Duplicate data, don't normalize it': Offebar will er wirklich nur lesen und nichts aktualisieren. Das ist doch der Alptraum jeder Datenpflege!

Database based Message Queues

- Database queues for replication (Schlossnagle)

Read Replication

- Death of read replication: Brian Aker on Replication, caching and partitioning (does not like caching very much, prefers partitioning). See also Todd Hoff on using memcached and MySQL better together and the remarks of Francois Schiettecatte.

Non-SQL Stores

Toby Negrin, Notes from the NoSQL Meetup,

http://developer.yahoo.net/blog/archives/2009/06/nosql_meetup.html

Todd Lipcon, Design Patterns for Distributed Non-Relational Databases,

<http://www.slideshare.net/guestdfd1ec/design-patterns-for-distributed-nonrelational-databases?type=presentation>

Martin Fowler gave his blog entry the title “DatabaseThaw” and compared the past years with the “nuclear winter” in languages caused by Java [Fowler]. There seems to be a flood of new data storage technologies beyond (or besides) regular RDBMS. This raises the question of why this is happening? Isn't the relational model good enough?

Im remember discussions with Frank Falkenberg on the value of in-memory databases. I was not convinced of their value because the argument of keeping data in memory for faster access did not really convince me: every decent RDBMS will do effective caching. What I didn't see at that time was that the real value of different storage technology might lie in what it leaves OUT. See the following comment from the memcached homepage:

“Shouldn't the database do this?”

Regardless of what database you use (MS-SQL, Oracle, Postgres, MySQL-InnoDB, etc..), there's a lot of overhead in implementing ACID properties in a RDBMS, especially when disks are involved, which means queries are going to block. For databases that aren't ACID-compliant (like MySQL-MyISAM), that overhead doesn't exist, but reading threads block on the writing threads. memcached never blocks”[

http://www.danga.com/memcached/]

We all know that by leaving out certain features new products can be much more nimble. We also know (after reading “Innovators Dilemma”) that most products suffer from feature bloat due to competition and trying to reach the high-cost/high price quadrant.

I don’t want to overuse the term “disruptive” here because I do not believe that the new technologies are going to replace RDBMS in a general way. But it pays to ask the critical four questions when comparing a disruptive product with the newcomer:

- a) what can the newcomer NOT do (but the established technology can – this is hint about either currently or generally impossible goals, markets etc. and shows us where the new technology might have saved its strength)
- b) what can the newcomer do just about as well as the established technology? This gives us hints on general acceptability of the new technology by users of the established technology.
- c) what can the newcomer do that the established technology cannot do as well for technical or financial reasons? There won’t be many items in this bucket but they can be the deciding ones. We might find here connections to the things that were left out in a)
- d) what of the much touted features mentioned under a) are becoming less important or are outright useless in certain contexts. This is a corollary to c) and both show us the future market of the new technology.

The last point of course is especially important. If we can find some trend that requires exactly those features where the new technology excels, then we can possibly do predictions about the success of the new technology.

One of the biggest drivers of new technology trends was certainly the Web2.0 movement with cloud computing in its wake and the development of super-large-scale search engines and application platforms like google, yahoo and perhaps the success of large scale Massively Multiplayer-Online Games (MMOGs) like World of Warcraft or their non-game versions (Secondlive, OpenSimulator)

1. These platforms do a lot of multimedia processing (storage, delivers, inspection)
2. that is not necessarily or only partially transactional and
3. requires the handling of large blobs (files).
4. High-availability and huge storage areas are needed.
5. Frequently a simple key-value store and retrieval will do, the power of SQL is not needed
6. They typically use multi-datacenter approaches with centers distributed across the world.
7. They frequently need to present the “single image” or “single machine” illusion where all users meet on one platform to communicate. This requires extremely efficient replication or ultra-huge clusters.
8. Those Web2.0 applications also tend to grow extremely fast which puts a lot of strain on administration and scalability. Replication and cheap administration are not exactly those areas where RDBMS really shine.
9. Integration between different applications is not frequently done over http/web-services and not via a common database. Mash-ups work in a federated way and do not require the creation of one big data store.

Fowler calls this the move from Integration Data Stores to Application Data Stores where applications are free to store their data any way they want.

Behind the success of Flickr or Youtube a rather big storage problem is hidden: The storage of digital media at home. Digital content is growing at an alarming rate due to digital video and high-res digital pictures and to some degree audio (which is different as few people create their own audio content). Few home users know how to spell backup much less are able to implement a multi-tier backup strategy which provides safety and reasonably fast and easy access to older media. This problem is not only solved with a couple of external discs and some discipline: There are very hard and also costly problems of digital content involved: At least one of the backup discs should be at a different location for reasons of disaster recovery. That makes it less accessible and also hard to use as an archive. And then there is the question of aging formats, file systems and operating systems with unknown lifetimes.

Companies like Google will present themselves as the archive of the world. They have the distributed data centers to store content in large quantities and in a reliable way. Looking at how users currently deal with the storage problem I suspect that those services will be paid for in the near future. Of course this will raise huge concerns with respect to data security and privacy.

Another option would be to use a Peer-To-Peer protocol to turn the internet into this archive. It requires a lot of replication and defensive strategies against attacks on P2P systems like virtual subnets, re-routing requests to attackers or simply throwing away content while declaring oneself as a storage server. We will discuss those ideas a bit more in the section on advanced architectures and technologies.

Kresten Krab Thorup covered various projects or products at Qcon and I have added some more:

- Distributed DB,
- CouchDB, (Document centric database written in Erlang with a REST interface. Supports optimistic locking, crash-only consistency mode and “read operations use a Multi-Version Concurrency Control ([MVCC](#)) model where each client sees a consistent snapshot of the database from the beginning to the end of the read operation” (from the technical overview, [CouchDB]). Views operate in a map-reduce fashion taking the documents and functions as parameters. Replication is bi-directional and peer-based supporting disconnected operation and later incremental replication. Schema-free so clearly not a regular relational database or OO mapper.
- Scalaris, (distributed, transactional key-value store on P2P base with self-managing features and excellent request-scalability, programmed in Erlang. (see below)
- Drizzle, (Lightweight version of MySql see: http://drizzle.org/wiki/Drizzle_Features comes without stored procedures,

prepared statements, Views, Triggers, Query Cache and fewer field types but has a plug-in architecture for extension. Optimized for multi-core/multi-CPU architectures and lots of parallel requests.

- Rddb, Restful Ruby Document DataBase, modelled after CouchDB with the following features (from [<http://rddb.rubyforge.org/>): Documents are simply collections of name/value pairs. Views can be defined with Ruby code, mapping from a document to any other data structure, such as a String, Array or Hash. A reduce block can be defined to reduce the initial mapped data from a view. Views can be materialized to improve query performance. Datastores, Viewstores and Materialization stores are pluggable. Current implementations are RAM, file system and Amazon S3.). Clearly not a regular SQL DB.

- BigTable, HBase,
- Hypertable,
- memcached,
- Dynamo ,Amazon.com, highly available key/value store.

[DeCandia et.al.]

They are document-oriented, distributed, REST-accessible, and/or schema-free. They seem to be fallout from major large-scale Web2.0 projects (like memcaches that was written for LiveJournal.com. They certainly cannot do all the SQL magic of a full-blown RDBMS. Sometimes they go after “eventual consistency” instead of permanent consistency [Vogels].

I cannot discuss all of them but will concentrate on Scalaris because of its interesting P2P architecture and extreme scalability and on memcached due to its importance in the JEE environment for clustering.

Key/Value Stores

Semi-structured Databases

Bigtable and HBase are examples of a new type of data store. Confused by the use of “table” and “base” I found the short explanation of the store structure in [Wilson] - which made it clear that Bigtable-like stores are neither real tables nor SQL-capable databases. He cites the following definition of Bigtable: “A *sparse, distributed, persistent multidimensional sorted map*” and explains each term.

“Distributed, Persistent” means that the data are stored persistently across several machines. Typically a distributed file system like GoogleFS or Hadoop Distributed File System is used to hold the data. “The concept of a “multidimensional, sorted map” is best explained with a diagram:

.

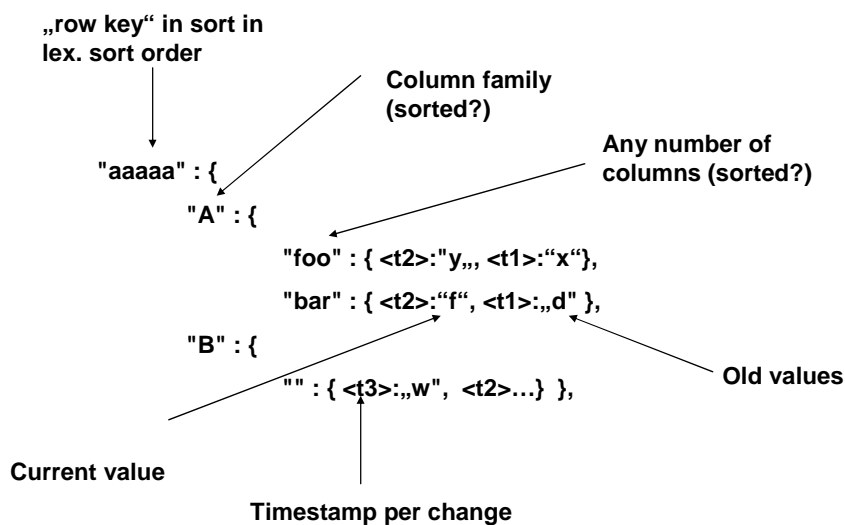
```

{ // HBase : sparse, distributed, persistent multidimensional sorted map
  "aaaaa" : {
    "A" : {
      "foo" : { <t2>:"y,,", <t1>:"x"},
      "bar" : { <t2>:"f", <t1>:,"d" },
    }
    "B" : {
      "" : { <t3>:,"w", <t2>:... } },
    }
  "aaaab" : {
    "A" : {
      "foo" : { <t3>:," world ", <t1>:... } ,
      "bar" : { <t2>:"domination,,", <t1>:"emperor" }
    }
    "B" : {
      "" : { <t1>:"ocean" } },
    }
  // ... }

```

Loosely after: [Wilson]

The diagram shows a map with keys. The first level keys are called “row keys” or simply “rows”. They are ordered lexicographically has a severe impact on the way keys need to be organized. The next level within the map is a set of keys called “column-families”. There are certain processing characteristics tied to these families like compression or indexing. Finally inside each family is an arbitrary number of column/value pairs which explains the term “sparse” used in the definition. It is possible to associate a timestamp with each value change and in this case a column/value pair is actually a list of pairs going back in time. How far is configurable. The diagram below shows the terms and the associated structure of such a map.



Loosely after: [Wilson]

How about performance and best practices for such semi-structured data stores? Wilson mentions a number of important things, the most important one probably being the organization of the row keys: `com.company.www` is a good way to write a key because in that case `com.company.mail`, `com.company.someservice` etc. all will be in close proximity to each other and an application will be able to retrieve them with just one access. To get all the available columns within such a map requires a full table scan. This situation can be improved for some columns if the data is put into column-families which provide certain indexes for faster lookup. Compression is an important topic as well and one can choose between intra-row compression and row-spanning compression. For details see [Wilson].

Forget about joins, SQL etc. and try to minimize the number of access in such an architecture. You might decide to copy and duplicate data just to minimize cross-machine data requests and a powerful memory cache holding joined data as objects is probably also a necessity. We will talk more about the use of such stores and their APIs in the chapter on cloud computing below.

Scalaris

Scalaris is a distributed key/value store (a dictionary) with transactional properties. It does not implement full SQL but does provide range queries.

There are even more reasons to take a close look at this technology: It is derived from an EU research projects called Selfman [Selfman] where a number of excellent distributed computing specialists are involved (amongst them Peter van Roy and Seif Haridi), it is written in Erlang, and it intends to solve two nasty problems: Tying scalable and flexible P2P architecture with the need for transactional safety with thousands of requests per second and creating a system with self-healing features to cut down on administration costs. And finally the Scalaris developers won the IEEE Scale Challenge 2008 with simulating wikipedia on their architecture.

The following is based on a the presentation of Thorsten Schütt and others [Schütt et.al], Scalable Wikipedia with Erlang, documentation from the company onscale.de etc.

Let's start with the wikipedia architecture and some load numbers. Today it is not uncommon for large social community sites to receive 50000 requests per second or more. (Jürgen Dunkel et.al. mention 40000/sec concurrent hits against the Flickr cache with an overall number of 35 million pictures in Squid cache and two millions in ram cache, see [Dunkel et.al], page 261) The diagram below shows the typical evolution of such a site with respect to scalability.

<<wikipedia evolution>>

In numbers , this architecture works because by far the biggest part of all requests goes to cached data. According to [Schütt et.al] this is 95% of all requests. Only around 2000 requests per second go to the database(s). This is still critical enough to cause some architectural changes especially in the storage area as we have seen. Clustering databases is sometimes not enough and the site is forced to create another partitioning at the top of the existing storage architecture as is shown in the next diagram:

<<wikipedia with partitioned DB>>

And then there is the problem of multi-site data centers which means replication of data across many machines to create the illusion of a single system.

<<mapping of tables to key values>>

DHT design: routing and transactions
Greedy routing ($O(\log n)$), qualities of service, quorum (Paxos)
Churn resistance?

A new database architecture

Is it really time for a complete re-write of database technology?
Stonebraker, Hachem and Helland argue for such a re-write. Interesting usage scenarios. According to [Stonebraker] all modern databases can be beaten easily with specialized engines with self-managing features.

Also see the discussion on Lambda the ultimate on this topic:

Part V: Algorithms for Scalability

I/O Models

Almost every system architect agrees with the statement that I/O is one of the most critical areas in system design and responsible for performance, latency and throughput. But there is surprisingly little consensus about the proper architecture for handling incoming and outgoing data. It begins with the question of how many threads should be used? This immediately leads over to the next problem: Kernel threads or green (non-preemptive) threads? The question of blocking vs. non-blocking adds to the threading problems: the costs are different between different types of threads. Asynchronous, event-driven architectures bring a new programming model with them and they handle network and disk I/O differently in many cases. Do threads have specific tasks or should they all perform the same tasks? And finally: are threads or events better?

We will discuss some of the concepts and try to answer the following architectural topics:

- Connections: how many? Lifetime? Cost for Construction and as a resource? Spoon Feeding Effects
- Threads – how many? Different functions or all the same? Kernel or green threads?
- Data handling and copying
- Memory consumption
- CPU usage and context switching/queue problems
- Resource tracking (connection free etc.)
- Threading Models for I/O
- Nio and how to model IO processing and how to properly read/write data
- Synchronization problems with select-type interfaces
- None-blocking I/O (epoll)
- Asynchronous I/O (linux interface)
- Is a staged/pipeline architecture better? (SEDA)
- Events vs. Threads
- I/O Programming Models in general and on multi-core architectures

But before we delve into the specifics of I/O processing we need to define some technical terms which will be used frequently on the next pages.

Definitions:

A context switch means a full context change from user to kernel, saving all of the previous task state and establishing the state of the newly selected task. It does not matter whether processes or kernel threads are switched. The operation is expensive and wastes cycles that could be used within a server to process requests.

Blocking means that a kernel thread or process cannot continue and needs to give up the CPU. This will be done by doing a context switch and the thread will be in

state waiting for either I/O or condition variables afterwards. A blocked thread does not compete for CPU. Blocking is like a context switch rather detrimental to server performance

Non-Blocking I/O means that a system call returns an error (E_WOULDBLOCK) if an I/O request made by the application would lead to blocking because the kernel cannot complete the request immediately. The application can then perform other tasks while waiting for the situation to improve. This is effectively a sort of polling done by the application. Because polling is costly and ineffective non-blocking I/O typically also provides a way for the application to synchronously wait for a larger number of channels to offer data (read) or accept data (write). A typical example is the “select” system call.

Synchronous processing means call and return style processing. The calling code waits for a response and continues only after the response has been received. If the called code takes longer to supply the response the caller is blocked. Synchronous is defined semantically as to require a direct response. The fact that a caller will be blocked while waiting for the response is not essential.

Asynchronous processing in a strict, semantic definition means that the calling code does not expect a direct response. The asynchronous call done by the caller may cause some action later but it would not be a response. The caller ideally will not keep state about the asynchronously requested action. In its purest form asynchronous processing leads to an event-driven architecture without a sequential call/return mechanism.

Asynchronous I/O: Not so pure forms of asynchrony send and receive events within the same layer of software as is done for I/O processing typically. This has to do with the fact that most I/O follows some request/response pattern which means a request enters the system at one point and the response leaves the system at the same point. Very popular and effective are combinations of synchronous and asynchronous processing steps as can be seen in the way Starbucks takes coffee orders synchronously while the baristas brew coffee asynchronously in the background with the customer waiting for the synchronous order/payment step in the foreground [Hohpe]

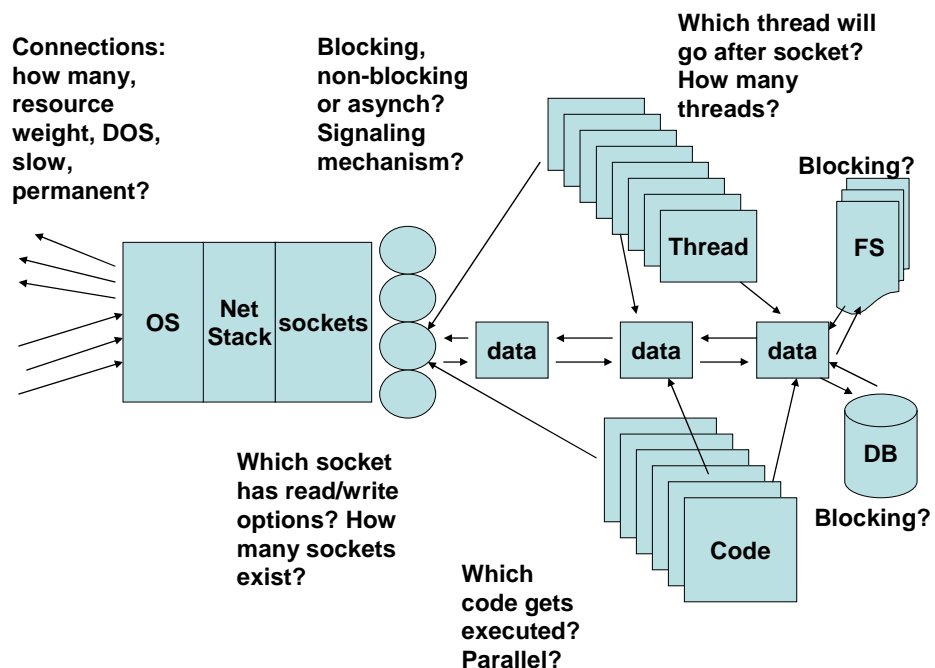
Thread denotes a virtual CPU, an associated function to be performed and a stack allocated. The stack allocation is mostly done in a static way which leads to memory under- or overallocation. Threads can be kernel threads, visible to the underlying operating system and its pre-emptive scheduler or they can be “green” or user level threads which are realized in a non-preemptive or collaborative way in the user space. Locking mechanisms are only needed in case of kernel threads as user level threads are not pre-empted. They are under control of a user level scheduler and the application. While most applications can only use a small number of kernel threads concurrently the number of user level threads easily goes into the hundreds of thousands. User level thread packages have once been popular, fell from grace when systems with multiple CPUs (and now cores) were built and are now getting popular again due to the realization that the locking primitives needed with kernel threads and shared state are exceedingly complex and on top of that a performance problem due to increased context switching and contention. It is important to distinguish the concept of concurrent kernel threads

from user level threads mapped to one kernel thread. User level threads allow the organization of processing into individual tasks without having to pay the price for concurrency control. User level thread packages need to do their own scheduling of user level threads and they cannot allow calls to block the one kernel thread. Erlang, E and other languages use this concept extensively for application code and show extreme performance by doing so. This is because the one and only kernel thread does not block unnecessarily. Instead it just continues with another available user level thread after initiating an action.

What happens if an application or server code needs to use more CPUs or cores? In other words more kernel threads? Here the answer depends on whether the code uses shared state. If it does either locking mechanism have to be used or – in case of user level threads – a second runtime needs to be started which runs in a different process. Both architectures are not perfect in every case. Currently the growing use of multi-core CPUs has raised considerable interest in solutions which make the use of multiple kernel threads possible but transparent for application code. We will discuss those approaches in the section on concurrency below, using Erlang and transaction memory as examples.

I/O Concepts and Terminology

A canonical representation of I/O might look like this:



Let's discuss the critical components from left to right.

Connections

Connections used to be a hard limit in server design. Operating Systems did not allow arbitrary numbers due to the fact that each connection needs state within the system to be held. Operating

Systems had to be changed to support larger numbers of connections.

A special problem is the question of permanent connections. Http was originally designed to close connections after every request, with http1.1 the “keepalive” option allowed a client to request several resources from a server using the same connection. From the literature it is absolutely not clear whether this is a good thing to do or not but we will give some hints below.

Slow connections force the server to respond slowly by “spoon feeding” the result to the client. This can severely impact throughput as it binds precious resources within the server and causes lots of unnecessary context switches.

The operating system typically receives data via interrupts.

Network data arrive fast and need to be stored quickly. But how will an application read those data? Reading as much as possible in one go or doing partial reads? And why is this such an important question? Given a fast network and few clients not reading all available data in one go might lead to extra stalls on the network layer which prevents the network from running at full speed.

Actually it will run WAY below speed as re-synchronization takes a lot of time (see the example the NIO section below)

The Asynchronous Web

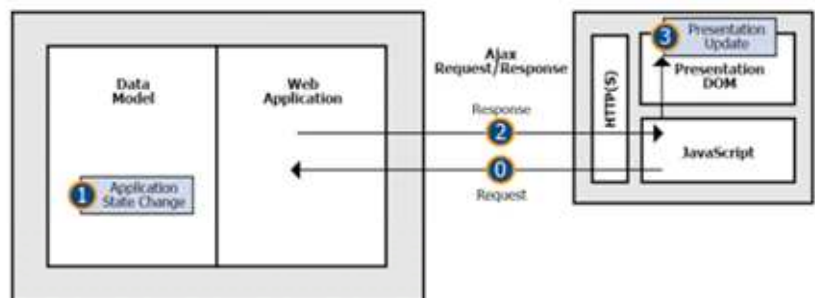
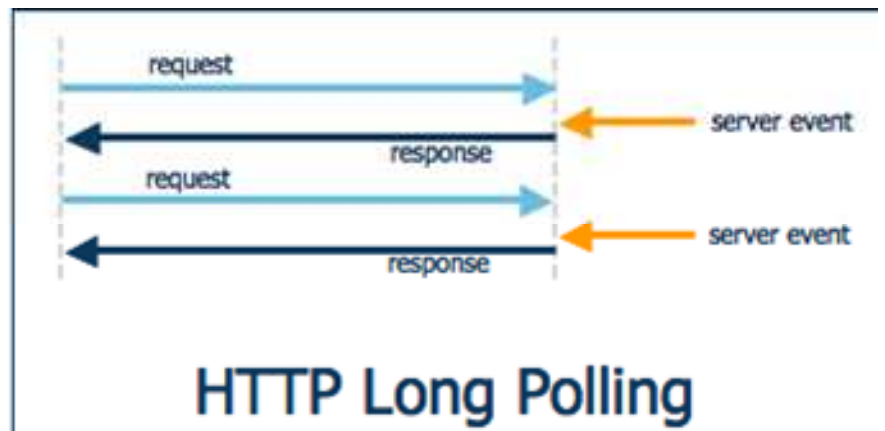
When you look at the Web today, it's pretty much based on a synchronous interaction model: The user interacts with the application, and the application responds with some updates. When you move into the asynchronous Web, you have the ability for the application to deliver state changes to the client, without the user necessarily having to initiate those updates. In effect, you can push updates to the client running inside the browser.

Asynchronous push provides information to the user instantaneously, without waiting for the user to request that information. An early, and simple, example is stock quotes that continuously change. Using asynchronous Web technologies can keep a user updated of those changes.

When you take that concept and look at it from a collaborative perspective, you can have one user of an application interact with the app, and cause changes that others will see. Instant messaging and chat applications are examples of that.

If you apply that concept to an even broader category of Web applications, you can create very sophisticated user interactions. That's especially the case in the context of social networking applications. Some social network sites now provide photo sharing that goes beyond simple posting of photos. These applications let you sit down with your friends, however far apart you may be physically, and take them through your slides, give them a slide show over the

Web. That's the kind of capability the asynchronous Web can deliver.[Maryka]



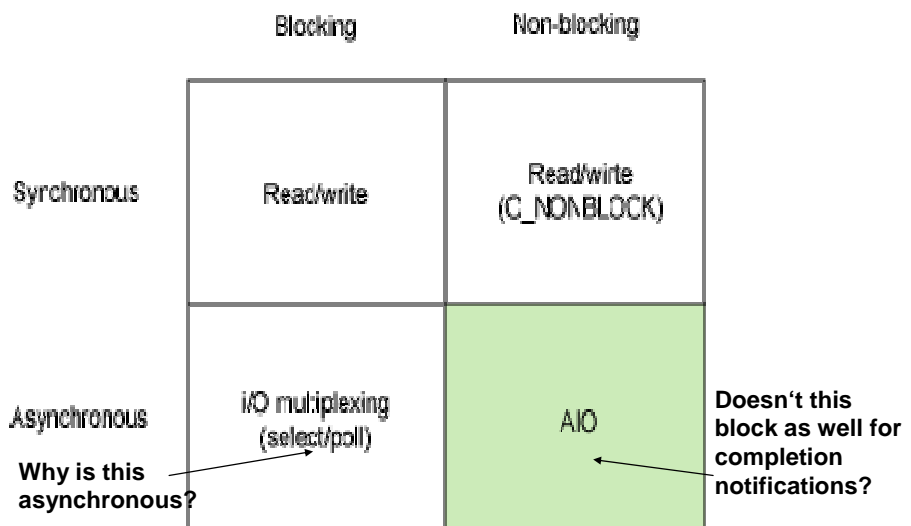
The Keep-Alive Problem

What could be wrong with keeping a – potentially expensive - connection open, waiting for more requests? Theo Schlossnagle calls it “a blessing and a curse” and points out where the problem really is. It is not only the

memory consumption of a connection which had been a problem in the past. Nor the slow algorithms dealing with event management and notification (e.g. `select()`). It is the threads being blocked and waiting for communication on this connection. The number of connections and whether they are kept open does not matter once you switch to an asynchronous handling of it: Do not block a thread waiting for more requests which might come some time later. You are starving your system for important resources. [Schlossnagle] (paper on backhand).

Actually this is a very common anti-pattern for throughput and performance. It also shows up in the handling of database connections. If a connection gets assigned to a request automatically and kept for the whole service time of the request this is very convenient for developers. But at the limited number of database connections available e.g. in Oracle doing so means severely restricting the number of requests which can be concurrently handled to the number of database connections. You need to acquire and release connections dynamically and only when and as long they are really needed.

I/O Processing Models Overview

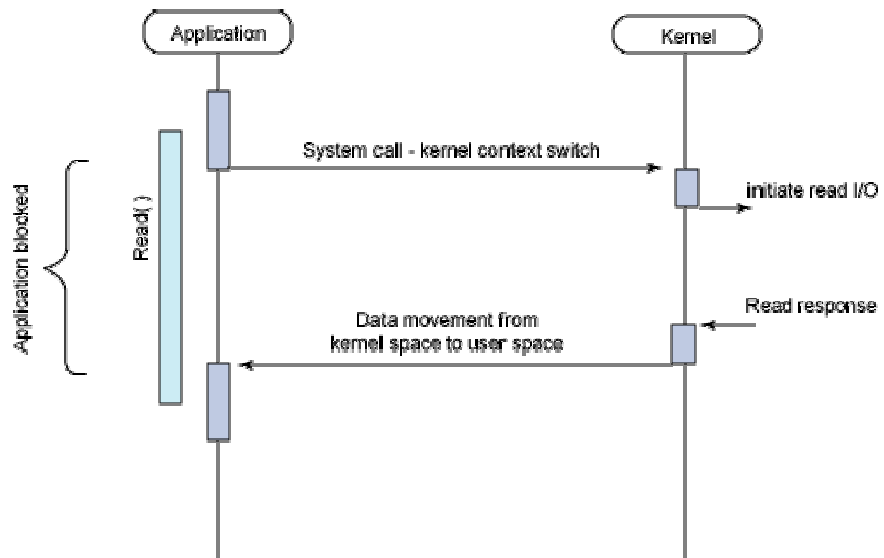


Adapted from: T.Jones Boost application performance using asynchronous I/O

Thread per Connection Model

In this model the number of sockets corresponds to the number of connections. Some of these sockets might have data to be read, some might be able to accept data to write (send). Applications have different options to find out about these sockets. The simplest

way was to just tie one thread to each socket. The thread would try to read or write and block if the socket had no data or in the write case could not accept more data. This method was used by Java up to version 1.3 and was heavily criticized.



From: [Jones]

The reasons were fourfold and had to do with exactly those threads. From a resource point of view threads show three problematic properties. First, they require a large amount of memory in the virtual machine. This memory is needed for the thread's stack and is usually fixed at startup. Many threads can easily drive a VM towards memory limits. The second problem of threads is scheduling. Scheduling is automatic in this model and not under application control. Scheduling also means context switches and those are expensive if we use kernel threads. And third as we have seen in the modelling chapter: the more threads are used, the longer the response time becomes due to contention and coherence reasons. This is especially true if those threads are mostly runnable and contend for a time-slice of the CPU. The fourth reason finally is blocking: making server code block due to data not being available again causes context switches which simply cut down on the cycles available for the application code. In the end this means that after a certain number of threads is tied to connections the system will spend most of its time with garbage collection and context switching. The "Thread-per-Connection" Model really puts us between a rock and a hard place: we want more threads to be able to service more concurrent requests. And at the same time the related overhead will diminish our ability to service those requests quickly.

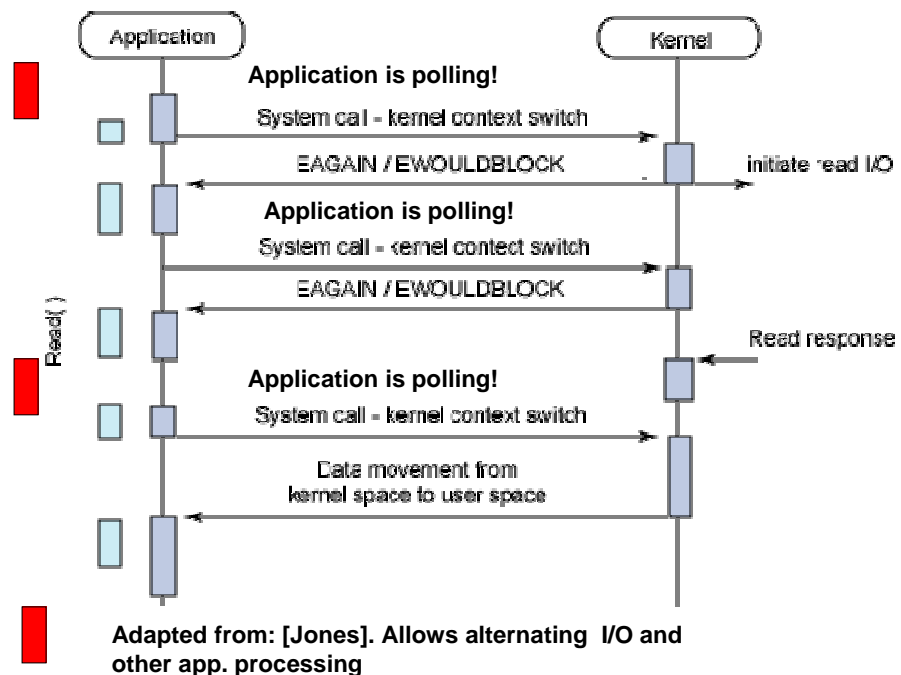
To be fair we need to acknowledge that the problems with this model come from specific implementations of it and are not necessarily intrinsic properties of the model. We will discuss an approach that tries to prove the effectivity of the thread-per-connection model by fixing some of the implementation deficits [vanBehren].

The good side of this model clearly lies in the architectural simplicity of using threads for handling I/O: the thread encapsulates all the connection related state, stores important meta-data like security information per requests and flows it across the server functions and keeps the simple, sequential programming paradigm.

What are the alternatives to threads per connection?

Non-Blocking I/O Model

Let's assume for a moment that we have got only one thread for all connections plus the application functions. Clearly this thread cannot block on one specific connection waiting for data or buffer space. A very simple form of this model would have the thread poll all connections, disks etc. in a round robin fashion and process all input and output that would not make it block.



While conceptually simple the programming turns out to be quite ugly because the application code needs to do its own scheduling of tasks (we will discuss better schemes below, e.g. the use of user level threads in Erlang that keep the programming simple). It also needs to manage the state of connections and requests explicitly

because there is not a one thread/one connection relation where the state of a request is kept on the thread stack.

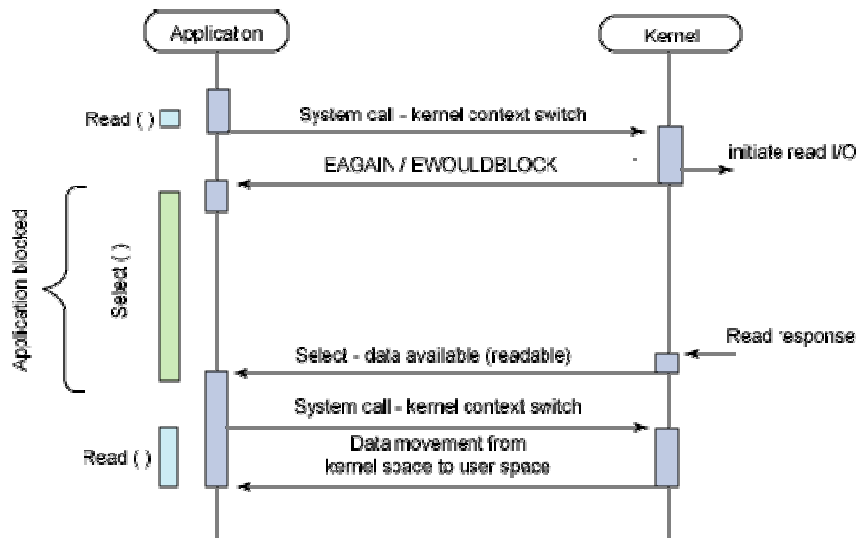
And it is not a very efficient scheme either because it can take quite a long time for the thread to react on an input source becoming available. That is why polling on non-blocking sources and sinks is usually avoided. Most systems that offer polling also offer a way to wait for a range of I/O devices within one system call (select), thereby realizing a synchronous notification model.

All non-blocking or asynchronous I/O processing shares one additional problem: partial read or write requests are possible at any time forcing the application to deal with them. This can lead to subtle errors when e.g. an application assumes that the bytes received can easily be transformed into a string of wide characters. What if the last character has only been transmitted in half? <<add code and author>>

Synchronous Notification (Multiplexing) Model

<<semantics behind interest setting and signals? Race conditions? >>

There are ways to build efficient I/O processing with only one thread. One example is to have it wait for ANY connection. This is called non-blocking I/O with synchronous notification and has been around since Unix server programming started.



From: [Jones]

It is unclear whether we should call this model an asynchronous one at all. If reads and writes happen they are performed synchronously and there is no overlapping of normal and I/O

processing within the application code. The application needs to wait synchronously for reads or writes to become possible and then needs to perform the actual reads or writes.

The system call for this features was called “select” and it allowed one thread to check concurrently on a whole array of connections represented by their file descriptors. Unfortunately for select the limit of connections was set to 32. Nowadays system calls like epoll in Linux have roughly the same functionality but deal with more connections albeit at the price of slow administration code in the kernel if the number gets really high. Other implementations even avoid this problem (see the C10k article below in “designing fast servers”)

The code for a one thread solution would look roughly like this:

```
While (true)
  Try to read non-blocking;
  Try to write non-blocking
  Do wait for specific socket event() with read or write signalling on
  On event == read X
    Turn off read signalling for socket X
    Read from specific socket X until enodata is signaled
  Done
  On event == write X
    Turn off write signalling for socket X
    Get data to write
    Write to socket X till ewouldblock is signalled
    Store the data that could not be written yet.
  Done
```

This non-blocking code is rather clear and avoids any synchronization problems. It needs to deal with different possible tasks though (reading or writing) which means explicit state management. The bad news is that the code cannot use multiple CPUs to improve throughput and that lengthy operations could starve other connections because the thread that deals with connections also has to perform other chores. And we are limited in the number of channels we can observe with select/epoll.

Just about the worst case would be if the thread has to block on either the database or the file system during the processing of a request. It turns out that not all operating systems are able to combine disk and other interfaces under the non-blocking API. If this is not possible we need to use another thread that runs in the background and accepts tasks from our thread which handles the connections, possibly via a buffered queue. Then our thread does only write into the queue and there is no danger of it blocking on some backend system. Instead, it can immediately turn back to waiting on the network. To make this work we would use two rather common patterns in non-blocking processing scenarios: The first one is to simulate non-blocking operations with blocking

background threads which simply accept tasks issued by the non-blocking foreground thread. Notifications are then delivered to the non-blocking thread either through a local channel which is part of the select range observed by the non-blocking thread. Or – assuming that the non-blocking thread comes by frequently because it waits in select with a short timeout specified – the results are placed somewhere to be picked up later. These patterns can also be used to relieve the foreground non-blocking thread from other tedious work once the channel management gets more involved.

But what happens if we want to use our multiprocessor better or there are some threads which have to block and we have no clever runtime that secretly simulates non-blocking behaviour.

There are basically three szenarios possible:

- several I/O handling threads which are responsible for different channels. They can run within the same process (which either requires explicit concurrency control by the application) or in separate processes (which would be the simplest solution)
- one I/O handling thread which delegates further processing to other worker threads.
- A combination of both with a common threadpool of worker threads.

These options are the same for asynchronous processing btw.

Two small questions arise in this context: what happens when a read or write just goes through without raising an `E_WOULDBLOCK` error? The answer in the case of non-blocking I/O mode is simple: Nothing special. The I/O gets performed synchronously and if requested, select will signal the availability of the channel at a later time.

This will be very different in the case of asynchronous I/O because here the caller does not expect the call to get through immediately. If it does though we have a problem: the completion notification will then be handled in the context of the caller and if the completion handling code just starts the next transaction we could finally blow up our stack with recursively called completion handler code. On the other hand it looks not very efficient to just forbid the caller of an asynchronous I/O function to perform the operation if it is possible without blocking. We will discuss optimizations for this case later.

The other question is about the behaviour of I/O handling threads. The answer simply is that they are not supposed to block at all.

Now we have to decide which thread is going to handle the channel(s) via select or epoll. Just one thread all the time? Any one thread at a time? All threads at the same time?

The answer pretty much depends on our operating system and its implementation of non-blocking I/O. Most of these are not able to be handled by several threads concurrently – they are not safe for

multithreading. If e.g. two threads try to change signalling behaviour concurrently, an exception will be thrown. (See the additional complexity for synchronizing channel management in [Santos]). This means we have to choose either just one permanent thread or synchronize between all threads so that just one will be the owner of the select at any time. Or assign channels statically to threads which are then solely responsible for managing them.

How does processing look in the case of all threads alternating in select management? A thread would acquire a lock for select entry and start waiting for events. Alternatively it would get suspended waiting for the lock to become available. On wakeup from select the thread will do whatever needs to be done while another thread wakes up from lock-wait and starts waiting in the select call.

With two of the three scenarios from above we have introduced the concept of worker threads. How many threads are we talking here? Not as many as we would use in a thread per connection scenario but more than two. Some authors suggest to use 2^n where n is the number of cores available. But we surely do not want to pay large context switching costs so we keep the number small.

How does the scheme with worker threads compare to the ones where the threads handling the channel will also do the processing of the requests? With the “all-in-one” threads we have a clear control of the select call semantics because it is done by one thread only and – in case we share channels - we have a suspension point for all threads waiting for access to channel management. And this means context switches! How bad are those context switches really (how frequently do they happen) and could they be replaced with a short spin-lock (busy wait)? The frequency of context switches will probably depend on the distribution and frequency of incoming and outgoing data and their associated events. With only a few events happening most threads will probably wait for socket access. With many events happening most threads will be busy serving those and there are chances that a returning thread can go directly to the next socket without wait. As the wait-time is hard to calculate a spin-lock with busy waiting is probably too dangerous but we could think about a compromise: do a short but limited spin-lock to test if socket becomes available. If not, go into wait/sleep. We could perhaps even adjust the spin-lock time depending on the frequency of events on the socket but this sounds a bit theoretical for my taste. And the read/write behaviour of threads becomes extremely critical. We would probably restrict reading and writing to a certain amount of data per event to ensure equal and calculable read/write times per thread.

Alternatively in our concept with worker threads we could propose that only one thread deals with connections at all. This thread loops between waiting in the select and either reading from a socket and writing the data into some buffered queue or getting some data

from a buffered queue and writing them into the socket that became available. All other threads would read from the queue and write into it. The thread that handles the connection needs to be fast enough to keep the other threads busy and prevent unnecessary context switches due to waits on those buffers.

Instead of contention for channel access we now have permanent hand-over costs between the I/O thread and the worker threads. Ideally the channels could be dedicated to specific threads which share a common threadpool of worker threads for delegation of requests. This would avoid contention at the channel level.

Both architectures – the one with all threads sharing the channels through a mutex or the one where only one thread does connection handling and uses worker threads for delegation – could work well. There is a tummy feeling that both could exhibit the following behaviour:

- at low load levels the processing is inefficient but tolerable
- a little bit higher the processing is really rough and stumbles along
- at even higher levels the processing runs very smoothly with almost no unnecessary context switches. The receiving worker threads would not block because there is always a request pending in the queue. The I/O thread does not block much because there is always a request pending at the channel level.
- at extremely high request levels the single I/O thread is perhaps unable to keep the workers busy. It will also be unable to administrate large numbers of channels effectively (old select and epoll problem). We would like to give our I/O thread more CPU time but this is not easy when the kernel does the scheduling. Splitting channels and adding cores will help some.

- The concept of all-in-one threads with each one being responsible for a certain number of channels might be better in the case of extremely high request levels because the threads do not need to wait often for new requests. More CPUs or cores will help but not increasing the number of threads.

Can we run a simulation to prove this gut feeling? Or should we try to program it and measure the results under load? What kind of instrumentation will we need for this? How self-controlling could the algorithms be? Wait-time and context switches are also dependent on the number of threads used. Should we try to adjust those at runtime? Should we take over scheduling in our application?

Using buffers to synchronize between threads is quite dangerous for performance: it can cause high numbers of context switches. The same is true for naive active object implementations. Scheduling of threads needs to be under the control of the server code. This means user level threads, just as required by Erlang actors.

Digression: API is UI or "Why API matters"

The cryptic title stands for an important but frequently overlooked aspect of API design: an API is a user interface for programmers. True, its design should be stable, perhaps extensible etc. But finally programmers will have to live with it and its quality - or the lack of it.

Christophe Gevaudan pointed me to an article by Michi Hennig (I know him from the former disobj mailing list) in the queue magazine of ACM on the importance of APIs. The author used a simple but striking example: the select system call API in .NET as a thin wrapper on top of the native W32 API. The way the select call was designed had already its problems but the port to windows made it worse. The author lists a couple of API defects that finally resulted in more than 100 lines of additional code in the application using it. Code that was rather complicated and error prone and that could have been avoided easily with a better interface specification of the select API.

For the non-Unix people out there: The select system call lets one thread watch over a whole group of file descriptors (read input/output/error sources). Once a file descriptor changes its state, e.g. because of data that arrived, the thread is notified by returning from the select call. The select call also allows the thread to set a timeout in case no file descriptor shows any activity.

What are the problems of the select API? The first one according to the author is that the lists of file descriptors that need to be monitored are clobbered by the select system call every time it is called. This means that the variables containing the file descriptors are used by the system call to report new activities - thereby destroying the callers settings who must again and again set the file descriptors it is interested in. The list of error file descriptors btw. seems to be rather unnecessary as most callers are only interested in errors on those sources they are really watching for input or output. To provide an error list of file descriptors to watch should not be a default.

But it gets worse: The timeout value is specified in microseconds which leads to a whopping 32 minute maximum timeout value for a server calling select. This is definitely not enough for some servers and now callers are forced to program code that catches the short timeout and transparently repeat the select call until a reasonable value for a timeout is reached. Of course - on every return from the select caused by a timeout the callers data variables are destroyed. And on top of this: the select call does not tell the caller e.g. via a return call, whether it returned due to a

timeout or a regular activity on one of the observed file descriptors. Forcing the client to go through the lists of descriptors again and again.

The author found a couple of anti-patterns in API design, one of them being the "pass the buck" pattern: The API does not want to make a decision and pushes it to the caller. Or the API does not want to carry a certain responsibility and pushed it to the client as well. A typical example in C/C++ programs is of course memory allocation. To avoid clobbering the callers variables the API could allocated memory for the notifications containing file descriptors which showed some activity. While this certainly IS ugly in those languages as it raises the question who will release that memory finally it can easily be avoided by forcing the client to allocate also those notification variables when he calls select.

But passing the buck can be more subtle: An API that does not allocate something definitely is faster. But you have to do an end-to-end calculation: somebody then HAS TO ALLOCATE memory and the performance hit will simply happen at this moment. So while the API may test faster, it does not lead to a faster solution overall.

Similiar problems show up when there is the question of what a function should return. Lets say a function returns a string. Should it return NULL or an empty string in case of no data? Does the API REALLY need to express the semantic difference between NULL and an empty string? Or is it just lazyness on the side of the API designer? How does the decision relate to the good advice to program for the "good case" and let the bad case handle by an exception?

API design is difficult as it can substantially decrease the options of clients. But avoiding decisions does not help either. The select example really is striking as it shows how much ugly code needs to be written to deal with a bad API - again and again and again...

Finally, another subtle point: The select API uses the .NET list class to keep the file descriptors. First: this class is NOT cloneable - meaning that the client can always iterate over the whole collection to copy an existing list. A mistake in a different API is causing problems here. And second: A list is NOT A SET. But select PROBABLY needs set semantics for the file descriptors - or does it make sense to have one and the same file descriptor several times in the list for input or output? This hardly makes sense but - being pragmatic - it might work. The client programmer does a

quick test with duplicate FDs and voila - it works! The only question is: for how long? The behavior of select with duplicate FDs is NOT specified anywhere and the implementors are free to change their mind at any time, lets say by throwing an exception if duplicates are found? Suddenly your code crashes without a bit of a change on your side. Using a set type in the API would have made the semantics clear. Ambiguous interfaces and one side slowly tightening the screws causes a lot of extra activities in development projects. I have seen it: A loosely defined XML RPC interface between a portal and a financial data server. And suddenly the server people decided to be more strict in their schemas...

All in all an excellent article on API design. Read it and realize that API design really is human interface design as well. It also shows you how to strike a balance between generic APIs on lower levels and specific APIs, perhaps overloaded with convenience functions, closer to applications. Method creep, parameter creep etc. are also discussed.

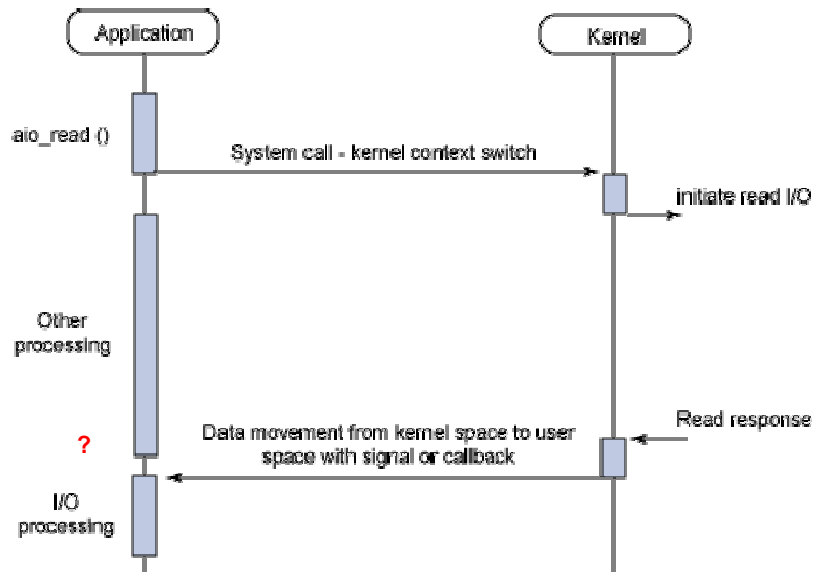
Asynchronous I/O Model

(Solaris Example vs. JDK example: kernel vs. vm). Clarify internal threads. Hand-off costs. Stack management due to immediate completion of I/O.

<<completion instead of notification, problem of synchronous calls, system thread notification, concurrency problems and race conditions, run to completion problem, programming models>>

<<interplay between app.processing and io completion: pre-emptive, parallel, polling (waiting)>>

What is the difference between non-blocking I/O with synchronous notification and true asynchronous I/O as it is depicted in the sequence diagram of Linux AIO below?



From: [Jones]. How are data moved? Is application processing interrupted? When is completion signaled? Does application wait for completion signals?

The first noticeable difference is the behaviour during initialization. Asynchronous I/O as it is frequently implemented does not assume that a call might directly go through. Typically the calling code assumes a fast return after initializing the I/O. If indeed the request could be fulfilled immediately it will create a dilemma for the calling code: should it call the completion code right away or still schedule the I/O for completion at a later time. We will discuss the problem of continuations below.

The second noticeable difference is the true overlapping of I/O processing with other applications code: while the kernel is processing the asynchronous request the application is free to process some other code.

And the third noticeable difference lies in the way notifications are handled. In the non-blocking case with synchronous notification it means that when the blocked select call returns some I/O action(s) on some channel(s) have become possible without leading to a blocked read or write call. In the true asynchronous case there is also a form of notification but it is called completion. It means that the I/O request has already been processed and the data have moved from kernel to user space or vice versa.

What is left is to inform the application about completed requests. This "completion handling" can e.g. immediately start a new asynchronous request. Or it can detect an error condition and repeat the previous request or abort it.

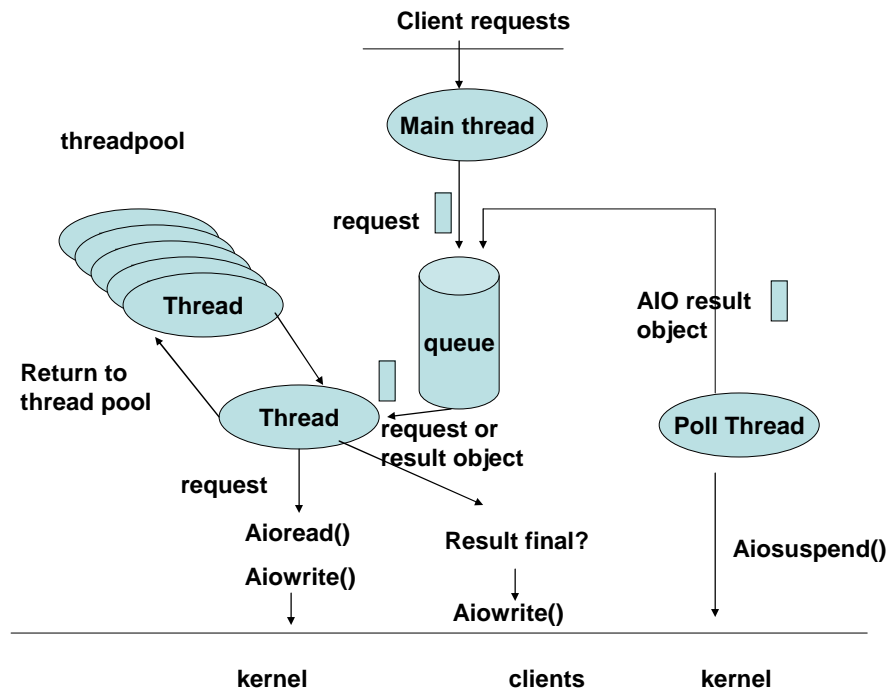
By looking at the sequence diagram we notice that there are a number of open questions regarding this handling of completed I/O requests: Who calls the completion handler? How and when does the application learn about completed requests? Does application

code run in parallel to completion handling or is it pre-empted by the completion handler?

What would be the ideal solution with respect to throughput and performance? Surely it would be necessary to do the completion notification as quick as possible to allow the next request to be started. By looking at the AIO API calls below we see that there are system calls which allow the application to learn about the status of a request either by polling or by waiting for a notification. But both seem to be rather inefficient. And waiting for a completion notification does not sound much different from waiting for a notification about a possible I/O request with non-blocking operations and synchronous notification. If on the other hand we allow the asynchronous completion handling to interrupt application code we might create race conditions e.g. if the application code was just about to prepare new data for transmission. Does this mean we have to synchronize access between completion handler and application code? This could lead to the completion handler needing to block waiting for the release of a lock. Or we settle for a solution where completion events are only sent by the kernel when the application has entered kernel state (most likely due to performing a system call). In this case we just assume that there is no chance for a race condition but we pay for it by having a non-deterministic time span between end of I/O processing and the notification of the application. This is btw. the solution used by signals.

API function	Description
<code>aio_read</code>	Request an asynchronous read operation
<code>aio_error</code>	Check the status of an asynchronous request
<code>aio_return</code>	Get the return status of a completed asynchronous request
<code>aio_write</code>	Request an asynchronous operation
<code>aio_suspend</code>	Suspend the calling process until one or more asynchronous requests have completed (or failed)
<code>aio_cancel</code>	Cancel an asynchronous I/O request
<code>lio_listio</code>	Initiate a list of I/O operations From: Tim Jones, Boosting... [Jones] The code pieces of asynchronous I/O found in the literature seem to prove those difficulties. When asynchronous I/O uses the suspend system call the difference to non-blocking I/O with synchronous notification becomes irrelevant: we have one system call for initialization, one for notification and one for checking the result. And we have also three system calls albeit in different order and function in the other case.

Let's look at an example using Sun's AIO API together with a threadpool [Sun]:



In this example the main thread receives client requests and forwards them into a queue. At the other end of the queue threads from a threadpool extract the requests and start processing. This typically involves asynchronous I/O to some other data source. The worker threads return to the threadpool and wait for new requests to arrive in the queue. A poll thread performs a blocking wait for results from asynchronous I/O and puts the result structures also into the queue. Like the original requests those structures are later extracted by worker threads which check for the status. If the request has been completed the worker thread will return the data to the clients, otherwise a new cycle of AIO read/write is started.

Raw I/O throughput in this design is also dependent on the polling thread reading the results of the AIO operations quickly and on the context switching costs of the worker threads. Sun suggests other means of notifications like signals and doors but it is unclear whether they would provide better performance.

We can compare this mechanism with the way a typical kernel handles writes to a serial device in an asynchronous way: An application writes data to a UART device. The kernel copies the data into a driver buffer, puts the application on the blocked scheduler queue and writes the first byte into the output port of the device. Once this byte is serialized and put on the wire the UART device will cause an interrupt which will extract the next character from the buffer and write it into the output port as well. Once the last byte has been consumed the interrupt code will cause a change in the state of the blocked application which becomes runnable again and returns to the user level.

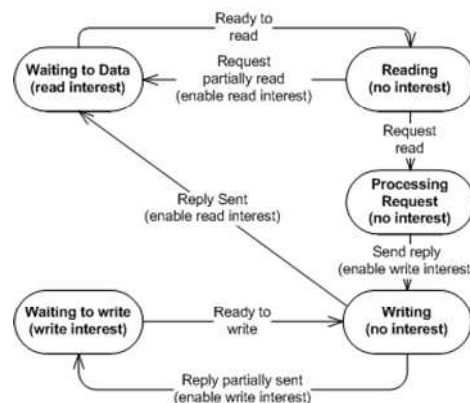
Theoretically the application could just dump the data into the kernel buffer and return immediately to continue some processing in the user level. Via some wait() or suspend() system call the application could learn about the outcome of the previous write.

Java Asynchronous NIO

[Roth] Gregor Roth, Architecture of a Highly Scalable NIO-Based Server Reactor/Proactor Patterns, framework integration,

[Santos] Nuno Santos, Building Highly Scalable Servers with Java NIO09/01/2004 <http://www.onjava.com/lpt/a/5127>

Handler State Machine



From Nuno Santos (see Resources). The state machines shows clearly the influence of non-blocking on the design of the handler which needs to maintain device or input state by itself. A regular thread would just handle one request and as long as input data are incomplete just sleep (context switch)

[Naccarato] [Giuseppe Naccarato](http://www.onjava.com/lpt/a/2672) Introducing Nonblocking Sockets 09/04/2002 <http://www.onjava.com/lpt/a/2672>

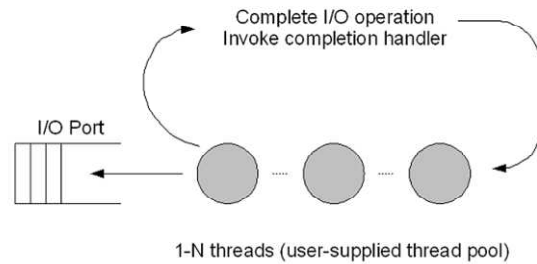
[Hitchens] Ron Hitchens, How to build a scalable multiplexed server with NIO, Javaone Conference 2006, <http://developers.sun.com/learning/javaoneonline/2006/coreplatform/TS-1315.pdf>

[OpenJDK] Notes on the Asynchronous I/O implementation, Nov. 2008

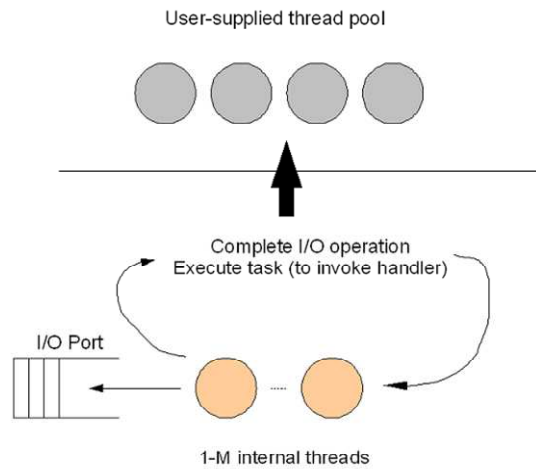
Virtual Machine Level Asynchronous I/O

The following is taken from the paper on asynchronous I/O implementation [OpenJDK] and describes various ways to supply threadpools to the async. event generator. The first concept involves a thread pool where threads extract completion events from ports of an asynchronous channel group and dispatch them to user completion handlers. When the handlers finish, the threads returns to waiting on ports.

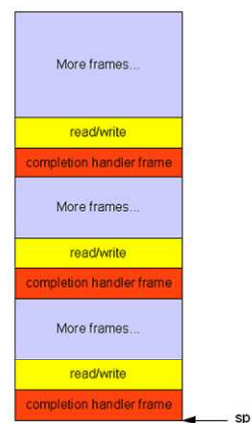
The design requires that handlers to not block indefinitely as this would finally lead to events being no longer handled. It is of course also important to set the number of threads correctly to avoid large context switching times.



The second case shows the use of two thread pools. One of them is used only internally by the event extraction logic. Those threads are not allowed to block. They will hand-off events to threads from the user supplied thread pool. Those threads in turn can block during completion handling but the threadpool itself needs to support unbounded queuing to avoid blocking the internal threads.



The paper also discusses what happens when an I/O operation can finish immediately. While this is rather nice from a performance point of view it means that the calling thread (if one from the thread pool) can start completion handling code immediately as well. And this code in turn can cause another read or write which theoretically can also finish immediately again causing the completion handler to be run and so on – until the thread stack explodes.



The implementation tries to allow several completion handler frames on the stack for performance reasons but limits its number to 16.

In “beautiful architecture” Michael Nygard describes the development of an image processing application used throughout hundreds of stores in the US where regular people can bring in their pictures and have them printed in various forms and formats [Nygard]. Here the main problem was that the main operators of the system were non-technical and in some cases even customers. For me the most interesting bit was when he described using Java NIO for image transport between store workstations and store servers. Image transport had to be highly reliable and very fast too. Nygard mentioned that this part of the project took rather long and showed the highest complexity within the project. Just matching the NIO features with high-speed networks and huge amounts of data was critical. He said that e.g. using one thread for event dispatch and manipulation of selector state is safe but can lead to performance problems. The thread used to only read a small amount of data from one channel, distribute it and go to the next channel. The high-speed network was able to deliver data so fast that this scheduling approach led to severe stalls on the network layer. They had to change the scheduling so that the receiving thread now reads data from one channel as long as there are data available. But of course this is only possible with few clients pushing files to your server. With more clients this can stall those clients considerably. Not to forget the problem of denial-of-service attacks when clients

Staged Event-Driven Architecture (SEDA)

Handover problems for individual threadpools
Call/response semantics?

We have already talked about the deficits of the request/wait cycle in multi-tier architectures. Performance or throughput problems in one tier can lead to many blocked requests upstream and finally large residence times for requests.

SEDA tries to break the request/wait cycle (which is simply a call/return pattern) by using asynchronous events between processing stages. Each so called stage runs its own thread-pool. Ideally a request enters the system at one end and leaves it at the other end without leaving any state information or allocated resources in the layers between. The diagram below shows this architecture.

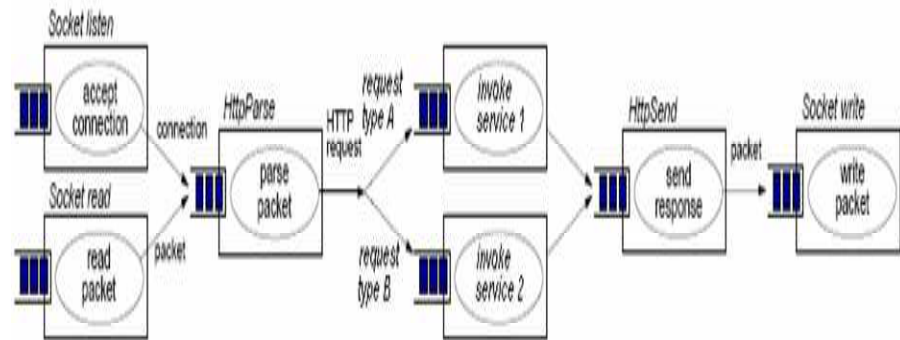


Figure 3: Our scenario mapped to the SEDA model

Realistically there needs to be some connection between the start and end of the pipeline because a request typically needs to leave the system through the same connection that has been used to deliver it in the first place. The diagram below shows how this is handled via a so called correlation ID which allows the association between a result and a connection. The only place where a synchronous I/O processing is done is right at the entrance of the system: clients wait synchronously for the response. In between stages issue requests asynchronously to the next stage downstream and do not wait for a response. Sometimes events are delivered to the same layer but in the opposite direction. This should not be confused with a simulated synchronous call semantic because the calling part in that layer does not wait for the response.

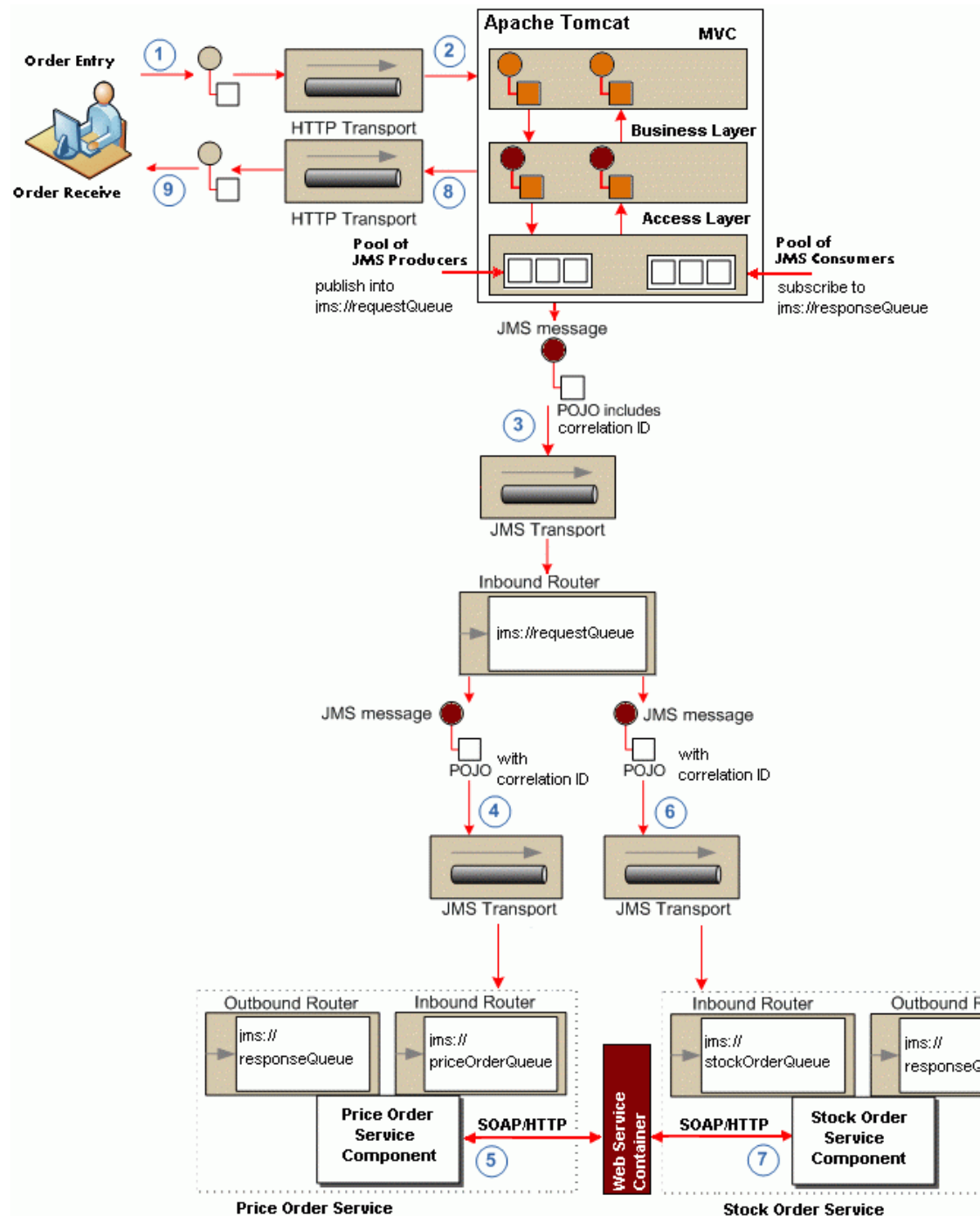


Figure 4 Overall logical flow

SEDA architectures claim much better performance than synchronous request/wait semantics. [Faler]. One critique frequently voiced concerns the way events are sent from queue to queue across different thread pools: this can lead to lots of context switches due to the necessary hand-over.

Building Maintainable and Efficient Servers

In this chapter we are going to discuss the ingredients of high-performance servers and the programming models in use to make those servers also maintainable and understandable.

Let's start with some general effects on the performance of large sites or as Jeff Darcy calls it in his "Notes on high-performance server design": "the Four Horseman of Poor Performance" [Darcy]:

1. Data copies
2. Context switches
3. Memory allocation
4. Lock contention

Zero-Copy

Probably the weakest point in this list is the first one: data copies. While even Java acknowledged the need for faster data containers by offering direct, OS-provided buffers for I/O in its later releases it is unclear how big the effect of data copies or equivalent functions like hashing really is. Using references (pointers) and length variables instead of complete buffers works well for a while but can create considerable headache later.

After all, Erlang is a functional language which keeps a separate heap per thread and forces inter-thread communication to copy data across message ports. And it does not look like Erlang suffers a performance hit by doing so. Quite contrary it is famous for the large numbers of requests it can handle. One advantage of forced copies is that both parties need not worry about concurrent access issues. I guess that many copies made in non-memory-safe languages like C and C++ are simply a result of concurrency or deallocation worries resulting from missing garbage collection and shared state multi-threading.

Avoiding kernel/user copies of large data certainly is a good idea though. Dan Kegel gives an example of `sendfile()` use to achieve a zero copy semantics. `Sendfile()` lets you send parts of files directly over the network. [Kegel]. The various data paths through an operating system are described here << zero copy techniques >>

Context-Switching Costs

We have been talking about the negative effects of context switches already. They take away processing time from functions and add overhead. But how do we avoid context switches? *The amazing thing is that, at one level, it's totally obvious what causes excessive context switching. The #1 cause of context switches is having more active threads than you have processors. As the ratio of active threads to processors increases, the number of context switches also increases - linearly if you're lucky, but often exponentially. This very simple fact explains why multi-threaded designs that have one thread per connection scale very poorly. The only realistic alternative for a scalable system is to limit the number of active threads so it's (usually) less than or equal to the number of processors. One popular variant of this approach is to use only one thread, ever; while such an approach does avoid context thrashing, and avoids the need for locking as*

well, it is also incapable of achieving more than one processor's worth of total throughput and thus remains beneath contempt unless the program will be non-CPU-bound (usually network-I/O-bound) anyway. [Darcy]

Some of these statements need further clarification. Why does the number of context switches increase with the number of threads? Given a fixed time slice per thread the number of content switches should be the same with more threads – it's just that different threads are involved. If the time slice is reduced with increasing numbers of threads we would see an increase in context switches but run into the danger of thrashing between threads without any work done. Do current systems reduce the time slice?

It is also unclear why we should see an exponential increase in context switches with more threads? Let's take a look at two other reasons for context switches besides pre-emption in the presence of more threads: blocking on I/O or condition variables. If we assume that there is a rather equal distribution of those across threads then we cannot explain the supposed exponential increase. But it sheds some light on context switch reasons in general: blocking need not lead to a context switch! It is a question of architecture (e.g. asynchronous I/O) and user level scheduling to avoid blocking for I/O or condition variables. And: blocked threads are not a problem for context switch overhead. Darcy talks about "active" threads, meaning threads in state runnable contending for the CPU. These will cause overhead.

Less active threads than processing units? This seems to be unefficient because it leaves cores idle.

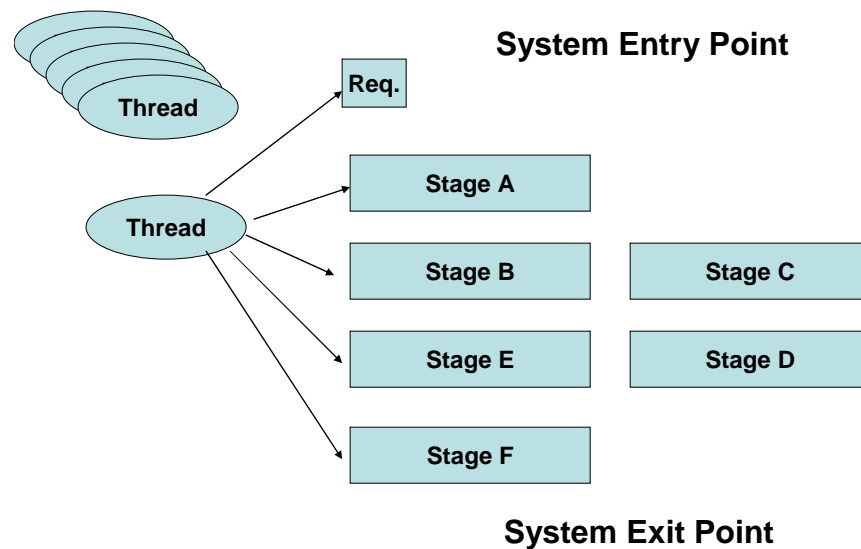
What are the lessons learned with respect to avoidance of context switching? The one thread with non-blocking I/O and user level scheduling seems to be the most effective for server applications. Instead of short time slices which allow I/O intensive processes to jump in, do their requests and block again quickly we want asynchronous I/O for interleaving of I/O requests and regular processing to reduce latency. This is very different e.g. to windows desktop OS configurations which emphasize interactivity instead of throughput. If we need or want to use more than one processor we should try to evolve the single-threaded non-blocking model by partitioning threads across either connections, processes or stages. By doing so we should avoid unnecessary context switches again. How should do this is explained by Darcy:

The simplest conceptual model of a multi-threaded event-driven server has a queue at its center; requests are read by one or more "listener" threads and put on queues, from which one or more "worker" threads will remove and process them. Conceptually, this is a good model, but all too often people actually implement their code this way. Why is this wrong?

*Because the #2 cause of context switches is transferring work from one thread to another. Some people even compound the error by requiring that the response to a request be sent by the original thread - guaranteeing not one but two context switches per request. It's very important to use a "symmetric" approach in which a given thread can go from being a listener to a worker to a listener again without ever changing context. Whether this involves partitioning connections between threads or having all threads take turns being listener for the entire set of connections seems to matter a lot less.
[Darcy]*

Here we learn what most database administrators had to learn the hard way a long time ago: a good logical model is not a good physical model in most cases. While the queue/stage architecture is conceptually very simple and nice it would cause excessive context switches if one thread can only work in one stage and needs to hand-over the results to other threads.

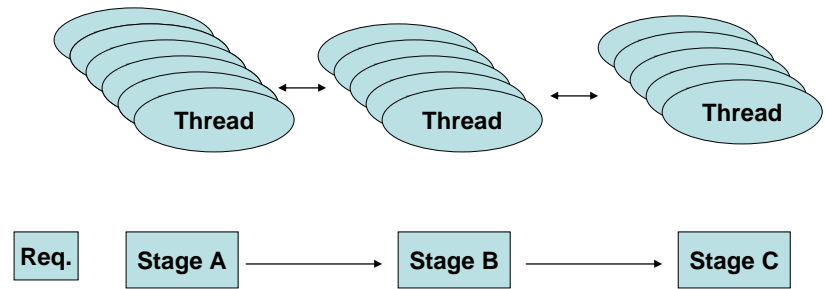
The following architecture avoids the overhead costs of frequent handover and lets one thread handle a request across all stages. Requests can be put on hold within a stage but this does not cause the thread to block and context switch. It will simply pick a new request or stage function to process.



One thread processes a request across all stages with every stage controlling dispatch via return codes. Requests can be put on hold within stages but this does not block the thread.

Compare this to a naïve implementation of stages in a SEDA model where each stage has its own thread pool (even though the threads might migrate over time between stages):

System Entry Point



System Exit Point

There is considerable context switching due to hand-over between stages and associated threads. A fully symmetric thread design where each thread can run every stage (also consecutively) is much better.

Interestingly, Darcy also suggests to dynamically control the number of active threads to prevent too many threads contenting for CPU. In his example he used a counting semaphore to restrict the number of threads allowed to run. He claims that this technique works well when you don't know how when requests come in or maintenance tasks wake up. While causing additional context switches this technique again emphasizes the importance of thread reduction. We will deal more with dynamically manipulating threads in the next chapter on concurrency when we talk about the best way to deal with threads once they have acquired a lock: Pre-empt as usual or let them run longer to shorten the serialized region?

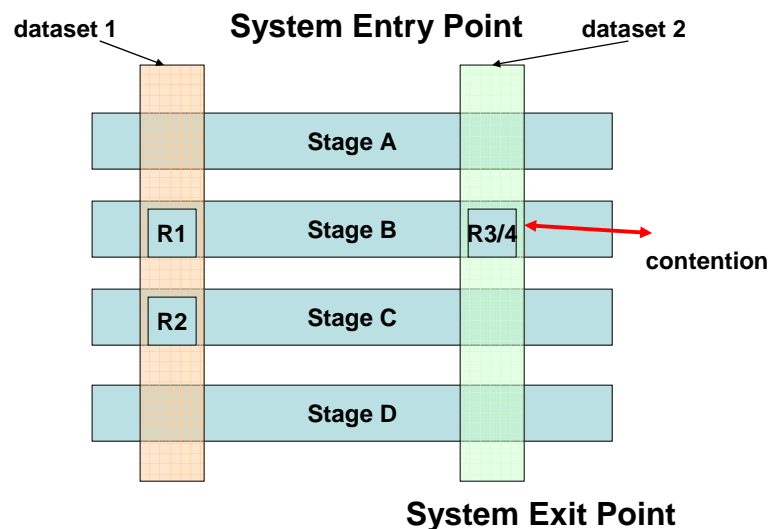
Memory Allocation/De-Allocation

Darcy also mentions a couple of memory allocation issues. Memory allocation does have an impact on server performance in two dimensions: allocation /deallocation in under contention/locking and paging costs. Pre-allocation of memory does reduce allocation costs, especially under contention but might have a negative impact on paging. What is important is to measure your allocation costs across several allocation sizes and under multithreaded conditions – you might be in for a (bad) surprise. This trick that is frequently used when data structures like collections need to be optimized for concurrent use: lock and swap complete sub-trees, branches or generations in one go and later – without holding a lock – deallocate the now isolated structure. Lookaside lists (basically pooled objects) are also useful in reducing allocation costs. If you are using a virtual machine with garbage collection make sure you understand the pro's and con's of

the different collection strategies. Generational GC e.g. can allocate memory very quickly but suffers from long lasting references. More on this topic in the chapter on concurrency. For I/O optimization it is important that your virtual machine runtime is small enough so that you can run several instances on one machine. This allows efficient partitioning of connections without an increase in contention within processes (assuming that you got enough processors for your VMs).

Locking Strategies

We will discuss locking strategies etc. in the next chapter in detail but Darcy emphasizes the effect locking does have on architecture and suggests a way to structure your code and associated locks:



The system should be designed so that contention can only exist if two requests meet within the same dataset AND the same stage. [Darcy]

Structuring your application for minimum contention is at the architecture level. But there are many smaller things that can be done to achieve high-performance servers. [Darcy] and [Kegel] mention e.g.

- use of scatter/gather
- request size measurements and optimizations
- Network optimization to batch small writes
- Page size alignments for disk and memory
- Input connection throttling when server is overloaded
- Increase default system limits (handles etc.)
- Thread memory reductions
- Putting server functions into the kernel

I/O Strategies and Programming Models

In this last chapter on high-performance I/O we will try to answer two questions:

- Are threads or events a better architecture (and which asynchronous model)?
- How much asynchronous, event-based processing should be exposed to programmers?

In “The C10K problem” Dan Kegel did a comparison of various non-blocking and asynchronous I/O system APIs (e.g. `epoll()`). [Kegel]. The results were that older API implementations sometimes have a problem dealing with large numbers of connections but the newer ones like `epoll()` and `kqueue()` are able to server tens of thousands of connections at the same time, or as Darcy says: it does not matter which of the non-blocking or asynchronous strategies one chooses – they are all largely equivalent once context switches etc are controlled.

The group of doubters with respect to asynchronous programming models starts with van Behren et.al. and their defense of threads as the superior programming model. They do not so much question the performance of event-based I/O but its ease of programming. Almost no server architecture they looked at used more than the usual control flow paradigms (call/return, parallel call, pipelines). And they show that they can achieve much the same performance and throughput with threads. Their core points are [vanBehren]:

- use user level thread packages (they recognize the context switch costs)
- be asynchronous under the hood only
- let threads allocate stack memory dynamicall (to avoid memory issue with VMs)
- change thread related algorithms to perform better than $O(N)$ in the number of threads

This list confirms what we have said in the above sections on I/O in general.

Greg Wilkins in “Asynchronous I/O is hard” [Wilkins] worries about the latest asynchronous API additions to the servlet API and gives interesting examples of the difficulties involved:

<<example of partial read error >>

```

if (event.getEventType() == CometEvent.EventType.READ) {
    InputStream is = request.getInputStream();
    byte[] buf = new byte[512];
    do {
        int n = is.read(buf); //can throw an IOException
        if (n > 0) {
            log("Read "+n+" bytes: " + new String(buf, 0, n)
                +" for session: "+request.getSession(true).getId());
        } ...
    } while (is.available() > 0);
}

```

From: [Wilkins]. The code does not take partially read characters into account and might generate an exception when converting bytes received to strings

Wilkins gives more examples e.g. writers not checking for the current I/O mode selected and concludes with the following statement:

Tomcat has good asynchronous IO buffers, dispatching and thread pooling built inside the container, yet when the experienced developers that wrote tomcat came to write a simple example of using their IO API, they include some significant bugs (or completely over-simplified the real work that needs to be done). Asynchronous IO is hard and it is harder to make efficient. It is simply not something that we want application or framework developers having to deal with, as if the container developers can't get it right, what chance do other developers not versed in the complexities have?! An extensible asynchronous IO API is a good thing to have in a container, but I think it is the wrong API to solve the use-cases of Comet, Ajax push or any other asynchronous scheduling concerns that a framework developer may need to deal with. [Wilkins]

So the right answer is to put AIO into the container? A very good demonstration of the complexities of pure AIO programming has been given by Ulf Wiger of Erlang fame in his presentation “Structured network programming - FiFo run-to-completion event-based programming considered harmful” [Wiger]. He uses a POTS (Plain Ordinary Telephony System) design to demonstrate the increase in complexity when first asynchronous programming with some blocking still allowed is used and later pure non-blocking AIO. The resulting code is absolutely non-understandable which is not a real surprise: pure, non-blocking AIO where some event loop handles all kinds of events by calling into handler routines. Those routines cannot block and therefore need to express every branch of an action as a new state. Continuations are used as well. This leads

to manual programming of complex finite state machines – something that is probably best done with the help of an explicit grammar and a compiler construction tool like ANTLR or advanced simulation tools. While it seems to be easy to build a fast, simple event-based prototype the programming model degenerates quickly when the project size increases.

It is probably a good idea to take a look at the code examples from Wiger at this point. To save some space here only the state-event matrix of the POTS is shown here.

From: [Wiger]

Global State-Event Matrix

*FIFO semantics,
asynchronous
hardware API*

	idle	getting first digit	getting number	calling B	ringing A-side	speech	ringing B-side	wait on-hook	await tone start	await tone stop	await ringing start	await ringing stop	await pid with telnr	await connect	await disconnect
offhook	0	X	X	X	X	X	0	X	X	X	D	X	X	X	X
onhook	X	0	0	0	0	0	0	0	D	D	D	D	D	D	D
digit	—	0	0	—	—	—	—	—	D	D	D	D	D	D	—
connect	—	—	—	—	0	—	—	—	D	X	X	X	X	X	X
request connection	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
reject	—	—	—	0	—	—	—	—	X	X	X	X	X	X	X
accept	—	—	—	0	—	—	—	—	X	X	X	X	X	X	X
cancel	—	—	—	—	—	—	—	—	X	D	D	D	X	D	X
start tone reply	X	X	X	X	X	X	X	X	0	X	X	X	X	X	X
stop tone reply	X	X	X	X	X	X	X	X	X	0	X	X	X	X	X
start ringing reply	X	X	X	X	X	X	X	X	X	X	0	X	X	X	X
stop ringing reply	X	X	X	X	X	X	X	X	X	X	X	0	X	X	X
pid with telnr reply	X	X	X	X	X	X	X	X	X	X	X	X	0	X	X
connect reply	X	X	X	X	X	X	X	X	X	X	X	X	X	0	X
disconnect reply	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0

Wiger emphasizes the use of blocking AIO as it is done via select() or epoll(). Here processes can block for exactly those events that fit to their current state. Ideally the processes or threads can use “inline selective receive” – a locally (logically) blocking API call which does pattern matching on events and delivers only the one that is expected, everything else is buffered for later reception by the process. In this case the process need not maintain a separate call stack as an additional bonus.

Libevent – an example event-notification library

www.libevent.org

<<what is it built with it?>>

Node.js – a new async. lib

Concurrency

<http://www.software-dev-blog.de/>

Just like in I/O processing the best way to use and deal with concurrency is a hotly debated topic. The growing number of cores within CPUs has added some additional fuel to the discussion. The trenches go along the following questions:

- what kind of and how much concurrency needs to be exposed to applications?
- what is the best way to deal with concurrency: shared state, message passing, declarative etc.
- what are the results of shared-state concurrency for the composability and stability of systems?

We will start with a short discussion on the effects of concurrency on the scalability of large-scale sites and continue with a look on various forms of concurrency in game development. Afterwards we are going into details on those concurrency forms.

Are we going to reduce latency by using concurrency techniques? The portal example from above and our discussion on latency have shown some potential here. We can take a typical request and dissect it into independent parts which can run in parallel easily. But it is not very likely that we will go very much into a fine-grained algorithmic analysis. We are looking for coarse grained units to be run concurrently. This can be requests for several backend services as in the portal example. Or it can be iterations across large document bases using one function as in the map-reduce case discussed below in the chapter on scale-agnostic algorithms.

Parallelizing I/O can theoretically reduce the overall runtime to the runtime of the longest running sub-request instead of the sum of all sub-requests. But – and this shows the various faces of concurrency: we do not have to use several cores to achieve latency reduction in I/O – non-blocking operations allow us to overlap I/O operations just as well (perhaps even better when we think about context switching costs). The parallel iteration and processing on the other hand really needs more cores to be effective. So concurrency can mean simply doing several things at the same logical time (but physically in sequence) or it can mean truly processing several things at the same physical time using more cores.

And while the examples mentioned certainly are make-or-break conditions for sites the most common use of concurrency is probably to increase the number of client requests which can be served by using more processing cores (assuming that we can somehow scale storage as well, which we will discuss later).

Tim Sweeney wrote an interesting paper on future programming languages from his perspective as a game developer (unreal engine). He discovered three areas for the use of concurrency in a game engine: shading, numeric computation and game simulation (game play logic). [Sweeney]

Three Kinds of (concurrent) Code

	Game Simulation	Numeric Computation	Shading
Languages	C++, Scripting	C++	CG, HLSL
CPU Budget	10%	90%	n/a
Lines of Code	250,000	250,000	10,000
FPU Usage	0.5 GFLOPS	5 GFLOPS	500 GFLOPS
Objects	10000's to update with 5-10 inter object touches and 60 frames/sec	Scene graph trav. Physics simulation, collision detection, path, sound	5000 visible at 30 frames/sec.
Concurrency Type	Shared state	Functional	Data parallel (embarrass.)
Concurrency Mechanisms	Software Transactional Memory	Side-effect free functions/closures, implicit thread parallelism	Data flow

Adapted from [Sweeney]

Sweeney makes a few important statements from the development of the unreal engine: shared state concurrency is a big pain. They try to avoid it or keep it as transparent to the developers as possible by running one heavyweight rendering thread, a pool of 4-6 helper threads which are dynamically allocated to simple tasks (symmetric approach to threads) and by a very careful approach to programming [Sweeney]. And it is still a huge productivity burden. The idea is to use a new form of concurrency control for game play logic with its huge number of objects with shared state and some dependencies: Software transactional memory. And for the numeric computations to use a purely functional approach with side-effect free functions which can be run by threads in parallel. Due to the data flow characteristics shader processing is anyway “embarrassingly parallel” and no longer a problem. We will discuss STM below but start with the “classic” shared state concurrency first.

Classic shared state

This approach to concurrency is called “classic” because it has been used inside of operating systems, database engines and other system software for ages. Those systems have the interesting property of concentrating concurrency control mechanism within themselves and creating a sequential, isolated processing illusion to their clients. Operating systems do this via virtual memory management and process isolation and databases use the concept of transactions to serialize data changes.

Sharing state means that two or more processes/threads/execution flows will potentially have access to the same data either at the same time or interleaved. While one can easily imagine why same-time access to data can cause havoc (especially lost updates and wrong analysis failure types) the interleaved access thing needs an explanation: What can go wrong if data is accessed by two threads? When the first thread is done, the second can do its stuff. Where is the problem? The problem is exactly in the term “done”: When the first thread is done there is really no problem giving

access to the second thread. The only problem now is to determine when the first thread is truly done. In a single core system that can't be a real problem: it is when the thread relinquishes control (yields) and gets context switched. But what if the first thread involuntarily needs to give up the core? In other words, if it gets pre-empted due to a time-slice or other scheduling policy. Then the second thread might access incomplete data. Or its updates might get lost when later on the first thread gets control again and overwrites the changes the second thread made.

We learn from this that even a single core can cause concurrency problems when shared data is used in the context of pre-emption. One core with pre-emption is as bad as two cores running in parallel. In multi-core systems we do not need the latter ingredient pre-emption: just shared data will suffice to cause all kinds of problems. The problems are mostly either:

- consistency failures
- suffering performance
- liveness defects (deadlock, livelock) and finally
- software composition and engineering problems.

And the answer to those problems in the shared state model of concurrency is always to use some kind of locking – i.e. mutual exclusion – technique to prevent concurrency for a short time. And this “cure” in turn is again responsible for some of the problems mentioned above as we will see.

For the following discussion two papers by “classic” system engineers are used extensively: They are “Real-World Concurrency” by Bryan Cantrill and Jeff Bonwick, two Sun kernel-engineers defending the shared-state multithreading approach using locks etc. [Cantrill] and “Scalability By Design – Coding For Systems With Large CPU Counts” by Richard Smith of the MySQL team at Sun, also a system engineer deeply involved in concurrency issues [Smith].

Consistency Failures

This class of errors should theoretically no longer exist in the shared state concurrency model: locks prevent concurrent use and corruption of data structures.

- lost update by overwriting previous changes
- wrong analysis by handing out intermediate, temporary (non-committed) values
- endless loops due to corrupted data structures
- race conditions and non-deterministic results due to timing differences in threads

Unfortunately performance issues force us to use locks in a rather fine-grained way (see below) which leaves ample opportunities for missing protection around some data structures. The real problem behind is actually the lack of a systematic way to prove the correctness of our locks. We are going to discuss this somewhat more below in the section on engineering issues with concurrency.

Just remember that those problems were the ones we went out to fix via locks originally.

Performance Failures

There is a rather simple relation between locks and performance: the more coarse grained locks are used the safer the concurrent system becomes due to even longer serialized, non-concurrent sections and the slower it will be. And the more fine-grained locks are used we will see better performance and throughput at the price of more deadlocks, livelocks and consistency problems. Remember that our original goal in using concurrency was not so much to speed up the individual function but to increase the number of requests being processed. The following problems have a negative effect exactly on our ability to run more requests by forcing the requests to wait for one request within a locked, serialized section.

- coarse grain locking of top-level functions or data structures
- pre-emption with locks held
- broadcast vs. signal handling: thundering herds
- false sharing

coarse grain locking of top-level functions or data structures

But what does “coarse-grained” mean in this context? It simply means locking the entry to a frequently used, entry-level function which lets only one process or thread get into the system and do useful work while all others have to wait at the entry. And the same effect can be achieved with locking large data structures like e.g. a complete table in a database. With a table lock no other thread can work with rows in that table even if the threads would use completely different and independent rows each. “Lock breakup” (this does NOT mean to take away a granted exclusive access by force!) discussed below is a strategy to break coarse grained locks up into much finer sections of serialized processing.

The following table (taken from [Goetz]) gives a good idea how coarse grained locking affects performance. It compares the throughput of the classic Java Hashtable (top-level synchronized) with ConcurrentHashMap and its fine grained locking.

Threads	ConcurrentHashMap	Hashtable
1	1.00	1.03
2	2.59	32.40
4	5.58	78.23
8	13.21	163.48
16	27.58	341.21
32	57.27	778.41

From: [Goetz], Java theory and practice: Concurrent collections classes - ConcurrentHashMap and CopyOnWriteArrayList offer thread safety and improved scalability

In large-scale systems such numbers are hard to ignore and warrant the effort to break up coarse grained locks.

pre-emption with locks held

In the chapter on I/O processing we have already learned that a high volume of context switches is a sure performance killer, mostly caused by too many threads. And from our queuing theory section we know that more threads means longer individual request service times as well. Now we can top those negative effects by allowing threads which hold locks to be pre-empted. This is about as bad as it can get for throughput. More interesting are the concepts needed to work around that problem, e.g. by letting the kernel know about the locks (see below).

thundering herd problems

“Scheduler thrashing. This can happen under Unix when you have a number of processes that are waiting on a single event. When that event (a connection to the web server, say) happens, every process which could possibly handle the event is awakened. In the end, only one of those processes will actually be able to do the work, but, in the meantime, all the others wake up and contend for CPU time before being put back to sleep. Thus the system thrashes briefly while a herd of processes thunders through. If this starts to happen many times per second, the performance impact can be significant.” [JargonFile]

Generations of Java developers have learned to use the broadcast mechanism of “notifyAll()” instead of the signal mechanism “notify()” on grounds of improved software stability. As notifyAll() wakes up all threads waiting on a

mutex it does not matter if some of those threads actually wait for something else to happen: All are woken up, all will have to check their special condition before accessing the resource (“guarded wait”) and all except one will fall back to waiting for the resource to become available again. Slight mistakes in the notification algorithms are inconsequential in this case.

I think that even the original argument based on robustness of the code is wrong: it actually hides a software bug in the notification algorithm used which should be fixed instead of covered up. And just think about the consequences for system performance: a possibly large number of threads wakes up (context switch) to do a short check on the condition variable and go back to sleep (context switch). This is far from effective and should be avoided like hell. If you are not sure about your locks and who is going to wait for what you need to build a model or lock-graph and perhaps track your locking solution with a model checker (see the SPIN/Promela section in our modelling chapter).

False Sharing

[Cantrell] mentions a rather tricky complication of local cache synchronization in multi-CPU systems which depends on the synchronization granularity, i.e. the width of a cached line. If elements of an array are used as locks there might be a situation where two of those data elements show up in one cache line. It can happen now that two CPUs are in contention for different elements of the same cache line. The solution proposed by Cantrell et.al. is to pad array elements so that two cannot show up in the same cache line.

Liveness Failures

The next list of failures all deals with the application no longer making progress. The best known and most feared reason for this is the so called deadlock. A situation where two threads each hold a resource which the other thread tries to lock next. This is not possible of course as the resource is already held. A deadly cross-over of requests for resources who’s importance seems to be largely determined by the specialty of the respective persons: theoreticians tend to emphasize the non-deterministic character of such deadlocks which turns them into a permanent threat without a safe model or theoretical concept for prevention. Practicioners also do not like deadlocks but do not hate them so much as the reasons for a deadlock are easily determined from the stacks of the involved threads and can therefore be fixed easily.

- reader/writer lock problems
- deadlock
- livelock

Reader/Writer problems are a bit more subtle to see They are caused by the rule that once a writing request is made no more read

requests are accepted. This means that a currently active read request which tries to do a recursive read request (i.e. to acquire the read lock again will be blocked – still holding the same lock already and thus preventing the write request from ever getting the lock and without being able to make progress itself. But again the situation is easily fixed post mortem. Livelocks usually happen more on the higher levels of architecture or in the context of lock-free synchronization (see below).

Software Composition/Engineering Failures

This section deals with general concurrency problems and their impact on software. The first topic, visible locks, seems not so important but has a major impact on debugging and scheduling abilities. The next question is about composable systems using locks and the answer as in many cases depends on ones perspective, just like with deadlocks. A short discussion on the performance impact of lock-free techniques follows and the section ends with some remarks on provable correctness.

Visible lock owners: mutex vs. semaphore use

The question is: who knows that thread A has acquired a lock on some resource? If a mutex is used then the kernel usually knows the lock owners identity and if it is not freed in time it is rather easy to find the culprit. If – like with semaphores or some condition variables – nobody knows about lock owners explicitly, lock failures and problems are very hard to track and the reasons for poor performance are hard to find. The lock graph and overview of a system are still very important problem solving utilities.

And there is another reason for making lock owner visible: If the scheduler knows that a certain thread has acquired a lock it can try to prevent this thread from being pre-empted. This is similar to the situation in real-time systems with priority scheduling and a low priority process holding a lock to a resource. If a high priority resource is blocked waiting for the lock to become available it makes sense to give the lock holding low priority thread the higher priority as long as it hold the lock. This shortens the time until the high priority thread has to wait for the lock to become available. Remember: we do NOT break locks by taking them away from their owners by force!
Both reasons are discussed in detail in [Cantrell].

composable systems in spite of shared state concurrency with locks?

Does the use of locks and shared state concurrency automatically lead to non-composable systems? It probably depends on your idea of composability. Let's try an analog problem first: Does the lack of garbage collection in C/C++ lead to non-composable systems? Theoretically the answer

is yes because in many cases the responsibility for heap memory allocated via malloc/free or new/delete is unclear and requires the assembler of a system to take a look at the source code of those components to figure out the responsibilities for freeing the memory. Practically those systems are assembled every day and the composability problems are not seen as major. The same is true for locks and shared state concurrency as has been shown e.g in [Miller]. And of course Cantrell et.al. are right in saying that despite those problems components using locks are successfully assembled every day. You just don't know whether some problem might show up at runtime.

Performance impact of test-and swap with lock-free synchronization

We will discuss lock-free synchronization below but just a short statement on performance costs. These are estimated to run from 2-4 times the costs of traditional locking techniques [Jäger], [Smith]. But practitioners responsible for large scale system design find this to be a good trade-off against cumbersome locking problems at runtime and complicated code to maintain [Sweeney].

Provable correctness?

<<CSP, SPIN, Promela>>

Classic Techniques and Methods to deal with shared state concurrency

This section discusses some well-known techniques and idioms to prevent the negative impact of locking on performance. Spin-locks are a way to avoid context switching overhead. There is no doubt that the secret to better performance lies in fine-granular locking. The concept of lock-breaking can either be used to reduce lock granularity either in a temporal or a spatial way. Very interesting techniques involve different generations of data which are either swapped under a short lock or simply retired with the option of still being available in case someone needs them. Finally, the most important technique is probably a clever architecture which separates hot paths from cold paths and uses lock breaking only where it is really needed.

Fighting Context-Switch Overhead: Spin-locks

I started using spin-locks (also called busy-wait) when Unix started running on multi-processor machines. Suddenly it was no longer enough just to block interrupts from intervening with critical work in the kernel. It was now necessary to prevent other CPUs from doing the same. In the I/O section we have discussed the costs of context switching caused by too many runnable threads. Here we are talking about context switches caused by too many CPUs fighting for a resource or condition. Using a regular sleep/wakeup idiom which puts the losing thread on a wait

queue is just too expensive due to the context switch involved.

The golden rule here is to let a thread busy wait for a resource of condition becoming available. This works of course only when the algorithms involved guarantee that the lock will never be held for a long time. This excludes e.g. I/O from being done by the lock holder.

<<how is this used within Java VMs in connection with synchronized??>>

lock breaking in time: generation lock and swap, memory retiring

The time spent under a lock does always have a serious impact on throughput. Keeping that time short is of paramount importance. To achieve this we can try to move all non-critical parts of an algorithm outside of the lock (e.g. not synchronizing a complete Java method but using synchronized blocks within the method). Or we need to make sure that we do not set a lock too high within a data structure if the modifications will only affect small parts within (table lock vs. row lock).

But what if the algorithm has to work on a large data-structure under lock? Here the generational-swap idiom can help a lot: We allocate a new data-structure, lock the existing one and swap references from the old one to the new one and release the lock. Now we can clean up the old data structure taking our time to do so because we are not holding a lock. An idea related to this idiom has been described by [Cantrell] et.al. using the lock breakup of hashtables as an example. Hash-tables need to be re-organized frequently to scale access time in case of growing data. This would imply reorganizations of large amounts of data, copying them over to new containers. Instead of copy-and-destroy we are using memory retiring in this case: We keep the old data containers and when a request comes we check whether it is for the old or new containers. This check of course is done under a short time lock.

lock breaking in space: per CPU locking

Besides keeping the lock time short we can try to decrease the number of cores getting hit by the lock. If we manage to partition a resource across the number of CPUs available we can set partial locks which will only affect one CPU instead of all. This partitioning of course is highly application dependent but it can pay off a lot to assign events, data-structures etc. to certain CPUs.

lock breaking by calling frequency: hot path/cold path

But before we do major code restructuring to achieve lock breaking and a smaller lock granularity it is very important to find out where we should put our efforts to get the most bang for our bucks. A calling frequency analysis with the help of a profiler will quickly show the hot and cold paths in our software. We do not want to use fine-grained locking in code that is only run once – as is typical for initialization or shutdown code. Here we can safely use coarse grained high-level locks which will protect large parts of our code. In frequently called code though we need to use the different kinds of lock breaking techniques explained above.

Making code perform better by making locks more granular is a tedious activity which has a major impact on the overall architecture. Sometimes it cannot be avoided but if you are in the lucky position to start a new project you might want to go back to the I/O section and take a new look at [Darcy] and his proposed structuring of a parallel architecture. Or you might want to take a closer look at the next sections on alternative architectures which avoid locks as much as they can.

threading problem detection with postmortem debug

Two tools will help you find threading/locking problems in your code. In case of a deadlock a system dump will allow you – with the help of your debugger – to recreate thread states and discover the deadly crossover of resource allocation that caused the deadlock.

But frequently your major concern will be poor performance which is probably caused by threads not really running parallel but contending for some resources most of the time. A so called thread analyzer traces all threads and the functions called and shows the blocking graph of your software. If you detect frequent cases of a group of threads waiting for one condition or lock you know where to start re-organizing your code.

Transactional Memory and Lock-free techniques

Still within the shared state concurrency paradigm but trying to overcome the performance of liveness problems are a couple of technologies which try to reduce or eradicate one of the core problems: locking on a level that is visible to programmers. The basic approach behind those techniques is well established: Transactions separate different processing flows without bothering the programmer with locking tables or rows. They do this essentially by creating a “shadow world” for each processing flow and no write in this world becomes visible till the transaction (and iff) completes. While mostly transparent to programmers (which only have to mark the beginning and end of a unit of work) transactional protection tends to lock resources and thereby prevents more concurrent use and in the worst case can lead to a deadlock.

A special version of transactions uses so called “optimistic” locking. In this case locks are held only at two points in time: when variables are read and when the transactions commits and variables are written back. During write-back the transaction system checks whether one of the variables that have been read has changed (by some other process flow). We say the read set has changed and presumably the results – the write set – are now invalid because they depend on what was read before. There are many possible ways to detect the change: Sometimes a timestamp is used which is taken when a variable is read and compared to the timestamp value at the time of write-back. One could also store copies of the variables read or create a versioning system for all changes (see MVCC below). If there was a conflicting change in the read set we need to abort/rollback the transaction and start from fresh.

It is easy to see the appeal behind optimist locking in transactions: we just go ahead at full speed and do all the reads and modifications necessary and at the end we check for possible conflicts. The trade-off is clear: we have much shorter serialized sections in our processing because the resources mostly stay in an unlocked state. This means we are using concurrency much better. But this gain in concurrency can easily be lost by larger numbers of transactions with conflicts and rollbacks. As the number of conflicts will probably increase over time and the number of processes involved (the more processes work on the same data and the longer they do it the more likely we will end with conflicting read set changes) systems using optimistic locking strongly advise against longer running transactions.

“Shadow world” (transactional) approaches do have some drawbacks as well: They require all participating resources to be able to “roll back” in case of a conflict. This means they are not allowed to create external side-effects which are not revokeable through roll back. The other drawback is that in case of a conflict and roll back user provided input needs to be re-acquired because the new situation might necessitate a different input.

Looking back at the beginning of the chapter on concurrency we realize that we can now add transaction monitors to operating systems and databases. They all relieve programmers from the need to explicitly deal with concurrency and locks. As the need to use concurrency and locking explicitly is probably tied to high-throughput processing it comes as no surprise that transactional technologies have been a focus of mid-range and mainframe systems till now.

But this is about to change drastically and the change is driven by hardware: Because the CPUs have pretty much reached the end in cycle speed (already a signal cannot reach all places within a die before the next signal is issued) hardware vendors are taking a different route to increase performance and throughput even further. Instead of bumping up the CPU clock frequency the number of cores present within a CPU is increased. We will soon be talking 80 core CPUs. This won't be a big issue for server side developers used to build application servers, runtime containers and

virtual machines. But it will have a major impact on everybody else. Your desktop application will need to use those additional cores just to keep up its current speed because a single core will no longer run at such extreme clock frequencies. This means application developers will have to look for places in their code to use concurrency.

“Finding parallelism” by Kai Jäger describes the forces behind this development as well as the technologies proposed [Jäger]. The first one is so called “Software Transactional Memory” and it is basically an implementation of optimistic transactions in memory.

How would programmers use this? Below pseudo-code is given which shows that programmers only have to put brackets around the code pieces which should be handled atomically by STM. Here the keyword “atomic” is used.

```
procedure Transfer(from, to, amount)
  atomic
    result := call Withdraw(from, amount)
    if result = true then
      call Deposit(to, amount)
    end if
  end atomic
```

Pseudo-Code for software transactional memory (STM), from Kai Jäger, Finding Parallelism [Jäger]

What is happening under the hood? Here STM needs to compare read-sets and write-sets of transactions efficiently to figure out whether one has to fail:

Put differently, a transaction fails when there is an intersection between its read set and the write set of another transaction that commits before it. STM implementations can use this fact to validate a transaction's read set very quickly and without having to actually compare memory contents. This however means that transactions are only atomic with respect to other transactions. When shared-data is accessed outside of transaction, race conditions may still occur.[Jäger] pg. 20.

We do not compare memory contents, all it takes is to compare change states. And it is clear that this comparison needs to be atomic and uninterrupted as well or we will have inconsistent data.

So what are our core requirements for committing STM transactions?

- efficient
- atomic

- lock-free
- wait-free
-

Jäger mentions STM implementations that use regular locking techniques for commit but this just makes deadlocks possible again and might have performance problems. Something like this is done in conventional transactions systems with optimistic locking.

When we say efficient atomic comparisons we mean some form of “compare and swap” technology (CAS) with hardware support. In CAS we can compare a word or double word of memory with an expected value and atomically update the memory word with a new value if our expected value is still the current value of the memory cell as it is shown in the code below:

```

procedure CompareAndSwap(value by reference, expected, new)
  atomic
    current := reference
    if current = expected then
      reference := new
    end if
    return current
  end atomic

```

Use of CAS. Value represents memory location, expected represents the originally read value and new the new value to be set in case expected==value from Kai Jäger, Finding Parallelism [Jäger]

Using this approach we get two additional benefits: it is lock-free and wait-free. Wait-free simply means we will not be context-switched and put on some wait-queue till a lock becomes available. Lock-free is much more interesting. It means we do not exclude any other thread from working (except for the atomic CAS instruction itself which would prevent other threads from accessing the same memory cell in that instant of time).

Lock breaking revisited:

But more importantly, we do not hold a lock and continue modifying some shared, critical data structure until we release the lock. This has a big advantage with respect to system stability and consistency: We have touched on this above in the discussion on “lock breaking”. Lock breaking NEVER means killing a thread and releasing the associated lock by force. This would simply lead to unclear and potentially inconsistent data because nobody knows what the thread had been doing when it was killed

and the lock released. Lock breaking always only means to reduce the granularity of locks. It gets even worse: if we are not allowed to kill a thread holding a lock, what about crashed threads? Using this model we are not allowed to have a thread terminate AT ALL within a critical section. How would we guarantee this? (see the talk by Joe Armstrong on Erlang concurrency, [Armstrong]).

Here we do not lock at all and therefore have no deadlocks or inconsistencies to expect. And this is more important event than the gain in concurrency.

What if our goal is not to finish our transaction (make our shadow copy the valid one) but to wait for a certain condition to become true? Condition variables are used for this purpose and if a thread finds a condition to be false it needs to be blocked and go on a wait queue if it cannot expect the condition to become available within a very short period of time (busy waiting). Just using the STM mechanism for waiting on condition variables would simply mean we are always busy waiting for the variable to become available. The automatic roll-back mechanism of transactions would force us back to start every time. Is there a way the STM mechanism can figure out that the thread should be blocked instead? And how should STM know when to wake it up again? Again the answer is in watching the read-set of the thread waiting for a condition variable to get a certain value: Once the check was done and the value was wrong for the thread it makes no sense to let the thread run until a write set of another set shows a change in the read set of the blocked value. The pseudo code below shows a thread blocking on a condition variable and STM used to control access.

```
procedure Withdraw(account, amount)
  atomic
    balance := call GetBalance(account)
    if balance < amount then
      retry
    end if
    newBalance := balance - amount
    call SetBalance(account, newBalance)
  end atomic
```

A thread waiting for a condition variable, implemented lock-free and with automatic change detection from Kai Jäger, Finding Parallelism [Jäger]

How can the system detect changes in the read set? From looking at the code above we see that changes need to be tracked even through function calls (GetBalance()). This looks harder than it really is: A Java VM e.g.

can use the load and store instructions in combination with a flag for atomic sections to define and track read/write sets quickly (perhaps even replacing the regular load/store interpretation with one for atomic sections to avoid tracking the flag).

The use of STM and lock-free synchronization primitives is not undisputed.

Use wait- and lock-free structures only if you absolutely must. Over our careers, we have each implemented wait- and lock-free data structures in production code, but we did this only in contexts in which locks could not be acquired for reasons of correctness. Examples include the implementation of the locking system itself, the subsystems that span interrupt levels, and dynamic instrumentation facilities. These constrained contexts are the exception, not the rule; in normal contexts, wait- and lock-free data structures are to be avoided as their failure modes are brutal (livelock is much nastier to debug than deadlock), their effect on complexity and the maintenance burden is significant, and their benefit in terms of performance is usually nil.[Cantrill] pg. 24ff.

I would like to add the scalability problems with any kind of optimistic synchronization method: The more objects or resources are involved within one atomic operation and the longer the operation will take the higher the likelihood of a conflict and a forced rollback. In this case the effects on throughput and performance will clearly be negative. The fact that STM is up to four times slower than traditional locking on the other hand may not really be a problem for most applications if they gain in consistency and ease-of-programming by using STM.

Generational Techniques

We have seen the use of locks to prevent clients from seeing inconsistent data. The other solution for the shared state concurrency problem was to compare the read-set of an operation against other write sets and detect changes. This prevents inconsistencies at the moment of an attempted synchronization (i.e. when a client tries to make her read-set the valid one).

To get this to work an idea of history of processes is already required: We need to remember what the client had seen (read) originally to be able to compare it to the current situation. We can extend this notion of history and discover that versioning is an alternative to shared state concurrency: we get rid of the shared state by never updating any value. Instead we always create a new version after a change. Now we only need to keep track of who has been working with which version. This seems to be the idea behind Multi-version concurrency control (MVCC). The following uses the explanation of MVCC by Roman Rokytssyy in [Rokytssyy].

The goal of MVCC is to allow most transactions to go through without locking, assuming that real conflicts will be rare – in other words it is an optimistic concept as well. Every transactions gets assigned a unique, increasing ID when it starts. On every write to a record that ID is written into the latest version of this record which becomes the current one. The previous version is stored as a diff to the current one (as is done in source

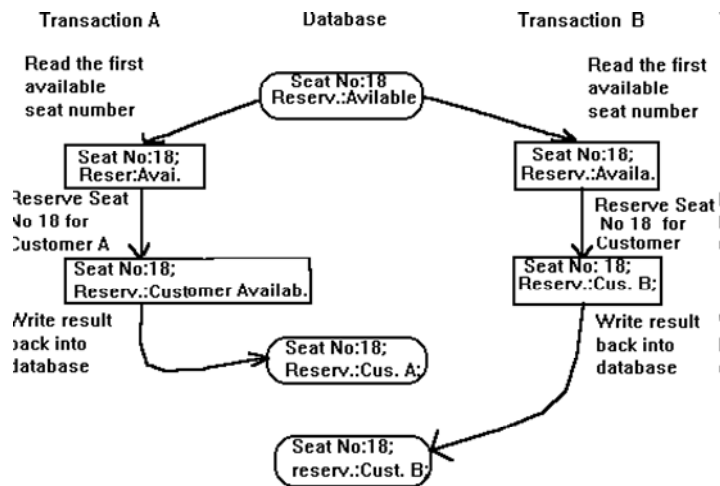
code control systems as well). Doing so allows the system to reconstruct older values if necessary (even though the hope is that this won't be necessary in most cases).

On a read to a record the IDs of the transaction and the currently saved ID are compared and the system checks whether the ID stored with the record belongs to a transaction that was completed before the reading transaction started. In this case there is no conflict at all and the current value is the valid one.

If the transaction ID stored with the value is younger than the reading transaction there are several choices: If the stored transaction ID is still running we cannot assume that the stored value is valid: the transaction might abort and we get a dirty read failure if we use the value nevertheless. If the writing transaction committed during the lifetime of our transaction it depends on the serialization level we want to achieve: In strict mode we cannot use the current value and need to reconstruct the value at the time our transaction was started. This guarantees that whatever our transaction sees comes from one consistent moment in time. But it does not mean the value read is really the most current one if the writing transaction committed in the mean time.

We might be able to accept a lower level of isolation though by accepting something like "read committed": rows added later to a table e.g. might not affect our business logic. Re-reading a value might give a different albeit committed result.

The following diagram shows exactly this problem using the example of oversold airline seats:



From [Rokytsky]

For an even better explanation on MVCC see [Harrison].

Finally a short note on the consequences to applications written against systems with different locking rules: As large-scale sites use major refactorings and changes in technology quite frequently this might be helpful. Rokytssky cites an IBM/Oracle dispute on the possibility of deadlocks when porting applications written for Oracle to DB2: *“As a result of different concurrency controls in Oracle and DB2 UDB, an application ported directly from Oracle to DB2 UDB may experience deadlocks that it did not have previously. As DB2 UDB acquires a share lock for readers, updaters may be blocked where that was not the case using Oracle. A deadlock occurs when two or more applications are waiting for each other but neither can proceed because each has locks that are required by others. The only way to resolve a deadlock is to roll back one of the applications.”* [Rokytssky]

If have chosen this quote because it nicely contradicts the claim in [Cantrill] that lock-based systems like OSs and DBs can be made composable. Here at least source code changes are actually necessary to make it work.

There are also more traditional uses of generations or versions, e.g. as generation counters during re-acquisition of locks. *“When reacquiring locks, consider using generation counts to detect state change. When lock ordering becomes complicated, at times one will need to drop one lock, acquire another, and then reacquire the first. This can be tricky, as state protected by the first lock may have changed during the time that the lock was dropped—and reverifying this state may be exhausting, inefficient, or even impossible. In these cases, consider associating a generation count with the data structure; when a change is made to the data structure, a generation count is bumped. The logic that drops and reacquires the lock must cache the generation before dropping the lock, and then check the generation upon reacquisition: if the counts are the same, the data structure is as it was when the lock was dropped and the logic may proceed; if the count is different, the state has changed and the logic may react accordingly (for example, by reattempting the larger operation)”*. [Cantrill] et.al.

Task vs. Data Parallelism

<<about decomposition techniques and the real parallel distributed monsters, infiniband and 10Gb influence on architecture?>>

Introduction to parallelism

Introduce the problem

Traditionally, Computer system consists of Processor, Memory system, and the other subsystem. The processor takes the instructions sequential one after one. Also the traditionally software has been written for serial computation.

Of course, we still get fast computer, and from time to time the processor frequency and the transistors count into the microprocessor get doubled. But suddenly the scientists discovered that we are near to reach processor frequency limitation. Then they try to discover new methods to improve the

processor performance, such as: put many low frequency and power consuming cores together, specialized cores, 3D transistor, etc. Now Multi-Core processors become the fashion of our industry. Now you can find in the market Dual-Core, Quad-Core, Octal-Core, and more will come. Actually the scientist discover that adding more cores into the processor will provide more performance without suffering from increasing the processor frequency. By nurturing the Multi-Core processors have the ability to process the tasks in parallel and provide more performance. But there are two problems:

** Most of current software did not designed to support parallelism i.e. to scale with the count of the processors.*

** The Parallelism is not easy for most developers.*

Introduce the parallelism

parallelism is form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel").

Simply, the Parallel is all about decomposing one task to enable concurrent execution.

Parallelism vs. Multi-Threading, you can have multithreading on a single core machine, but you can only have parallelism on a multi processor machine. Create threads will not change your application architecture and make it a parallel enabled application. The good mainstream developers are comfortable with multi-threading and they use it in three scenarios:

- 1- Background work for better UI response.*
- 2- Asynchronization I/O operation.*
- 3- Event-Based asynchronization pattern.*

Parallelism vs. Concurrent, you can refer to multiple running threads by concurrency but parallelism no. concurrent often used in servers that operate multiple threads to processing the requests. But parallelism like I said it's about decomposing one task to enable concurrent execution.

Parallelism vs. Distributed, Distributed is form of parallel computing but in distributed computing a program is split up into parts that run simultaneously on multiple computers communicating over a network, by nurturing the distributed programs must deal with heterogeneous environments, network links of varying latencies, and unpredictable failures in the network and the computers.

Parallelism in depth:

Parallelism Types:

There are two primary types of the parallelism:

** Task Parallelism*

** Data Parallelism (A.K.A. Loop-Level parallelism)*

Task Parallelism:

A group of tasks that can be executed simultaneously by multiple processors. Task parallelism is achieved when each processor executes different threads on the same or different data. The threads may execute the same or different code. For example: Imagine that you have four tasks you don't care which one will finish first. These tasks could be: Open an image file and process it and then save it.

Data Parallelism:

Data Parallelism usually manipulates a shared data that can be accessed by multi-threads in safety way. Data parallelism also known as loop-level parallelism and it's seems like SIMD, MIMD. For example: Imagine that you have a huge array of data (such as: bitmap Image, text file) and you want process this array/huge data in parallel.

There are two kind of data parallel, first: Explicitly Data Parallel, and Implicitly Data Parallel. In explicitly data parallel you just write a loop that executed in parallel as I mentioned. But In implicitly data parallel you just call some method that manipulates the data and the infrastructure is the responsible to parallelize this work. .NET platform provide LINQ (Language Integrated Query) that allow you to use the extension methods, and lambda expressions to manipulate the data like the dynamic languages. See the next figure to know the implicitly data manipulation and the parallelism in the implicitly data manipulation (implicitly data parallelism):

Task Parallelism vs. Data Parallelism

Bingo, if you really understand the previous sections, so you may ask who is better task parallelism or data parallelism? But unfortunately there is no standard answer for this question, usually the answer depend on the situation.

For example: if you want process many large files (i.e. folder content many large files). This question is: Do I should process the file contents in parallel (Data Parallel) or the independent files in parallels (Task Parallel)? Even this question don't have standard answer, to get the right answer for this question you should ask yourself the following questions:

** Is the files content large data, or just a few megabytes.*

** Is the file data processing will be forward only (such as: Fixing, Counting), or the data processing depends on themselves (such as: Sorting).*

** If you will choice data parallel (process the file contents in parallel), so how do you will manage the reading and the processing operation, and the required synchronization.*

Before I answer this question, I would like to say: The parallel programming is hard, because in parallel world the code behavior tend to be nondeterministic.

In our situation, I think processing the file contents in parallel will be better, because the HDD usually is slow and don't provide better support to concurrent reading or writing, so make many concurrent reading request to such slow device will be help. But if we make the loaded data processing in parallel this will be better, See the next table. Beside this we can perform per-fetch in our data parallel algorithm to load the next chunk of data while the loaded data process to keep the HDD busy and this will improve the performance.

Parallelism in real-world

Before I start speak about the parallelism in mainstream, I should speak about the mainly current parallelism applications. Servers have long been the main commercially successful type of parallel and concurrent system. Their main workload consists of mostly independent requests that require little or no coordination and share little data. As such, it is relatively easy to build a parallel Web server application, since the programming model treats each request as a sequential computation. Building a Web site that scales well is an art; scale comes from replicating machines, which breaks the sequential abstraction, exposes parallelism, and requires coordinating and communicating across machine boundaries. High-Performance Computing followed a different path that used parallel hardware, and optimized parallel software because there was no alternative with comparable performance, not because scientific and technical computations are especially well suited to parallel solution. Parallel hardware is a tool for solving problems.

Today the popular programming models-MPI and OpenMP—are performance focused, error-prone abstractions that most of developers find it difficult to use. In game programming, the developers emerged as another realm of high-performance computing, with the same attributes of talented, highly motivated programmers spending great effort and time to squeeze the last bit of performance from complex hardware. So what about the mainstream applications and developers? In spite of the fact that said the parallel computing is hard, today there are big trend to make the parallel computing more deterministic, and easy. Today you can found many easy parallelism frameworks, and debugging tools, such as:

** Intel Parallel Studio*

** Microsoft CCR and DSS*

** MS PPL - Microsoft Parallel Pattern Library (will released in 2009 Q4)*

** MS .NET 4 - Microsoft .NET Framework 4 (will released in 2009 Q4)*

** Java 7 (will release in 2009)*

** PRL - Parallel Runtime Library (Beta released in June 2009)*

In next table you can see a comparison between the above frameworks.

The previous offered features could changes in the produce final release. Your choice for the parallel framework should depend on your platform, and your application. For example Parallel Runtime Library designed to work will with high performance computing in first place. But Microsoft .NET Framework 4.0 parallelism API designed to support extensibility, and wide applications.

<http://www.hfadeel.com/Blog/?p=136>

#

*If you want to be serious about parallelism, perhaps you should also talk about the kind of parallelism used by the largest systems in the world - Blue Gene, Roadrunner, Jaguar, et al. - and not just the parallelism within a single relatively weak system. These big machines are distributed systems in the sense that they do not have shared memory, but they are **not** generally characterized by heterogeneity, long/unpredictable latency etc. as you claim distributed systems are by nature. The most common programming model/library is MPI, though shmem and UPC also have their fans and new alternatives appear all the time.*

These “esoteric” systems and approaches are becoming more common, as just about anyone can afford a rack full of PCs and an Infiniband or 10GbE switch. They also bear some significant resemblance to the large systems put together by folks like Google, Amazon, or (increasingly) MS.

By Jeff Darcy on Jun 1, 2009

Java Concurrency

<http://www.infoq.com/presentations/brian-goetz-concurrent-parallel>

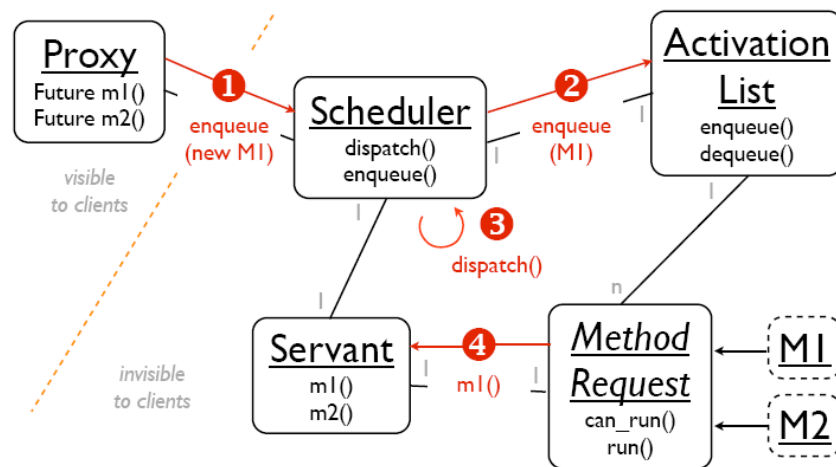
Active Objects

The following is based on a talk by Andy Weiß on the active object pattern [Weiß]. This pattern is e.g. used by Symbian servers to provide fast services without the need for explicit locking and interprocess or intertask synchronization methods. While Symbian servers typically use only one real thread other runtimes can use multi-core architectures efficiently as well.

The secret of active objects is the guarantee that at any time there will be only one thread active inside an active object and that calling a method of an active object is done asynchronously. If you consider calling methods as sending messages (like Smalltalk does) there is very little difference between active objects and the actor concept of Erlang which is explained in the next chapter. The biggest difference exists in the use of “future”

objects in active objects which allows the calling party to synchronously wait for a return value if it wants to do so. In purely asynchronous processing there won't be a return value of the call at all: answers will also be delivered via an asynchronous callback to the caller. To programmers active objects present the typical "method call" paradigm of OO languages with a pseudo synchronous return option..

The diagram below shows a Java like implementation of an active object system. Different languages which e.g. control or manipulate method dispatch are of course able to implement it in a much more elegant way (they don't need the proxy classes etc.).

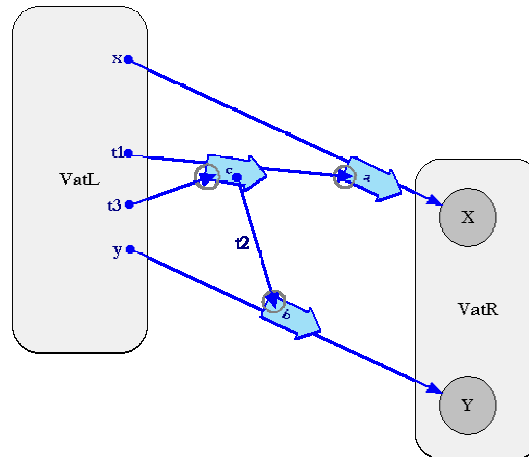


Here a client wants to call method `m1()` of a certain object. Instead of calling `m1()` directly the method call is intercepted by a proxy of the object and delegated to the internal – application level - scheduler. The method call is put into a so called activation list which acts much like a regular run-queue only with methods or functions instead of threads. After registering its wish to call `m1()` and the registration in the activation list the client who tried to call `m1()` returns immediately and continues processing. The method call to `m1()` is therefore an asynchronous call. As we have seen in the I/O section all asynchronous calls need some way to get back to the caller later. This can be via some callback function or as in this case via the use of a correlation ID or object. Here "futures" play the role of correlation objects (in the language "E" they are called "promises". A future or promise is a handle to the results of an asynchronous call. The handle can be used in several ways. One way is to use it as a parameter in other calls (remember: the processing which is represented by the handle is for from being done yet). The handles can be stores in collections, handed over etc. But if a caller tries to get to the result of the asynchronous processing two things can happen: The result is not there yet and the caller will be suspended (here: returns to the scheduler so that a different function or method can be processed). Or the

result is already there and will be returned to the caller immediately which continues processing.

The use of futures or promises is an extremely elegant way to avoid dealing with threads and asynchronous processing explicitly. Complexity of concurrent processing is reduced significantly. Even remote objects can be treated as futures as is done e.g. in “E” which allows some really surprising optimizations by avoiding unnecessary roundtrips.

Remote Pipes with Promises

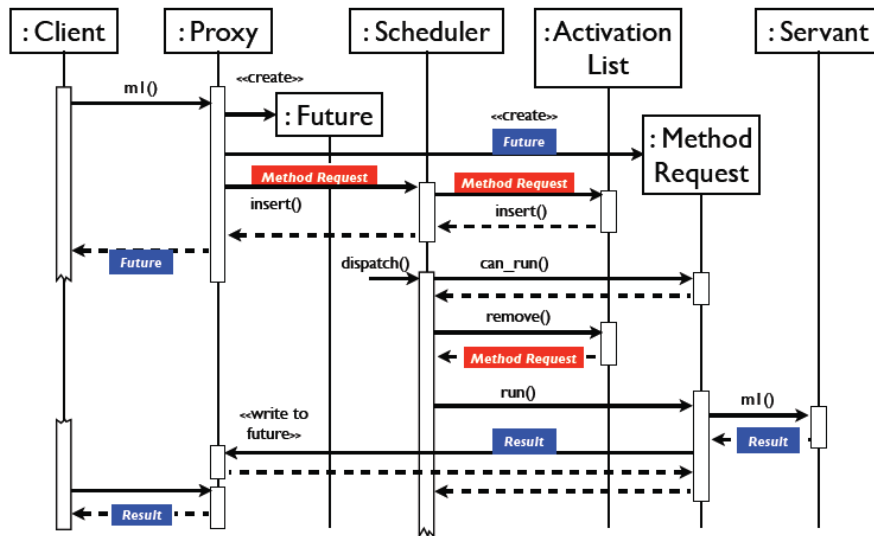


B represents a promise to a future result and is already forwarded to X – this saves a roundtrip from VatL to get the result from Y and forward it to X

To make this work in the remote case a promise does not contain a value once it is calculated. It functions as a function proxy only. <<check>>

What is happening behind the scene in an active object system? At one time the scheduler will pick the registered call to m1() in the activation list, wrap it into a method call object and execute the call. The call itself will end in the so called servant which contains the real m1() call. Here the call will be executed, the the result filled into the future object. If the client now tries to get the value from its future it will succeed in doing so. After the asynchronous execution the call will return to the scheduler and the next method will be selected for execution. The scheduler will make sure that there are never two concurrent calls to methods within one object executed.

The diagram below shows the sequence of calls for one asynchronous execution of a method m1():



The active object pattern allows many specializations, e.g. ways to determine when it makes sense to execute an asynchronous call (it might depend on some condition) and whether only one thread is used or several. The use of only one thread is highly efficient because it never blocks as long as executable method calls are registered and the context switching is really only exchanging user level functions. If more threads are used, e.g. on a multi-core platform, we have to make sure that we avoid thread numbers higher than the number of cores. And also fewer registered methods than available threads which would then only be put to sleep by the kernel (context switches). How is pre-emption handled in the active object pattern? From a user level all method calls are non-preemptive: At the end of a method or function call is the return to the user level scheduler. Everything else is non-blocking and asynchronous. From the kernel point of view the thread executing a method (or the threads) is pre-empted. But due to the fact that the user level scheduler will not allow another thread to enter an object while one method of this object is executed there are no shared data and therefore no consistency problems and no locks needed. An active object implementation which uses only one thread – as in the Symbian servers e.g. – will have some impact on application architecture: method calls need to be rather short to avoid an unresponsive system. The clients can use yield() methods to return back to the scheduler and allow other methods to be executed.

<<how would we implement something like transactions in a multi-core environment?>>

The Erlang Way

Lately Erlang has become a very popular language, albeit in special areas as it seems. Distributed key-value stores (Scalaris), messaging systems (ejabberd, RabbitMQ), databases (CloudDB) and even Raytracers have been built <<Ref. to Bader/Stiegler>>. In his talk on Functions +

Messages + Concurrency = Erlang Joe Armstrong mentions e.g scalability and error recovery as well as reliability as core properties of the language. Interestingly he insists that availability, stability, concurrency and recovery are all intertwined and mutually dependent things.

What makes Erlang well suited for large scale, extremely reliable systems with up to 9 nines of availability? And why e.g. is thread switching in Erlang so much faster? (Stack small due to continuations and short tasks?) Certainly a core feature of Erlang is its actor model of processing and concurrency.

Miller lists the key principles of the actor model in Erlang: [Miller]

- no shared state
- lightweight processes , not tied directly to kernel threads, not OS processes but fast to create, cheap and in large numbers available. Scheduled in user space controls pause and resume.

- Asynchronous message passing (with delayed receive?)
- Mailboxes to buffer incoming messages
- Message retrieval with pattern matching

And the list from Ulf Wiger's blog looks quite similar. His key properties for Erlang style concurrency are:

- Fast process creation/destruction
- Ability to support >> 10 000 concurrent processes with largely unchanged characteristics.
- Fast asynchronous message passing.
- Copying message-passing semantics (share-nothing concurrency).
- Process monitoring.
- Selective message reception.

If there is any single defining characteristic of Erlang-style Concurrency, it is that you should be able to model your application after the natural concurrency patterns present in your problem.(Wiger Blog entry, 6 Feb. 2008)

Wiger claims that Erlang can theoretically support 120 million processes and that he saw consistent performance up to 20 Mio. Processes with creation times around 4 micro seconds. Those numbers make me think again about the three concurrency models used by Sweeney for the unreal engine. He wants to use Software Transactional Memory to update 10000+ objects containing the game logic. Could he use Erlang processes instead?

Unlike Wiger to me Erlang is ideally suited for concurrency because functions are stateless and side-effect free (also called "referentially transparent") and variables can be assigned a value only once a read does not need protection from concurrent access. Messages are copies of data and immutable as well.

Wiger claims that the asynchronous message passing style of Erlang does not fit well to massive data parallelism but many current application architectures seem to be covered quite well by it.

Lets take a look at some code:


```

temperatureConverter() ->
    receive {From, {toF, C}} ->
        From ! {self(), 32+C*9/5}, temperatureConverter(); ...etc

```

```

start() -> spawn(fun() -> temperatureConverter() end).

```

```

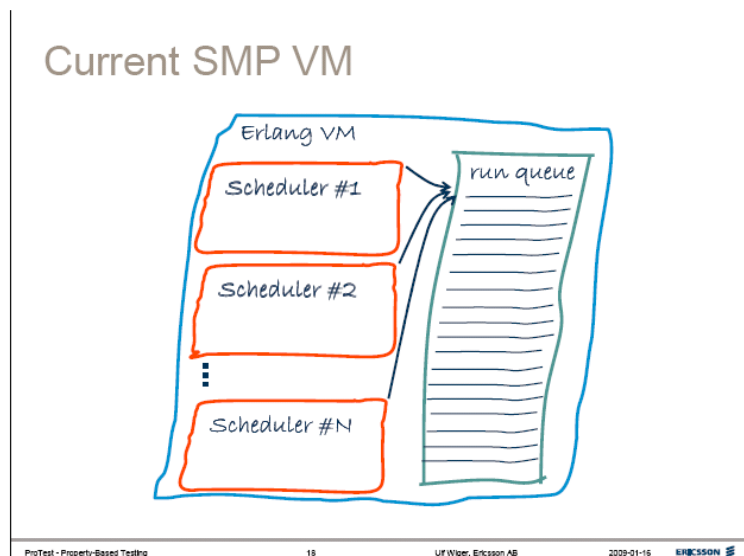
convert(Pid, Request) ->
    Pid ! {self(), Request},
    receive {Pid, Response} -> Response end.

```

The start() function spawns the converter process and returns its process identifier. The convert function uses the process identifier to call the converter process, sending the current identifier as the source, then blocks in a receive waiting for a response on the current's process mailbox, which is subsequently returned.

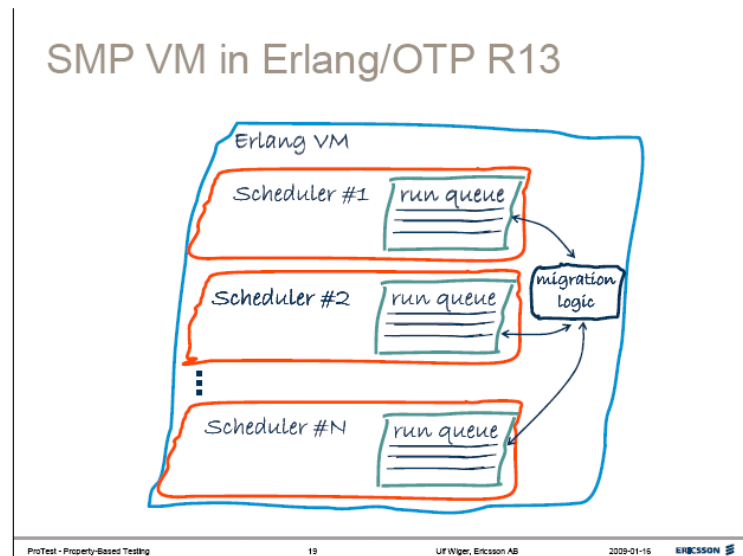
**Actor/Process use in Erlang,
From [Miller]**

And how does scheduling work in Erlang? According to Wiger in multi-core Erlang scheduling of threads is preemptive from the user perspective. The following slides from Wiger show implementations of multi-core support and a very interesting benchmark. The first slide shows several scheduler instances working on one run queue. How can this be? What about share-nothing? Here schedulers compete for access to the queue and locking/ mutex mechanisms (no matter how “soft”) will have to be used to prevent corruption. The slides show nicely that – while for applications the shared nothing approach is certainly kept up – the Erlang runtime system internally needs to deal with shared state in various forms.



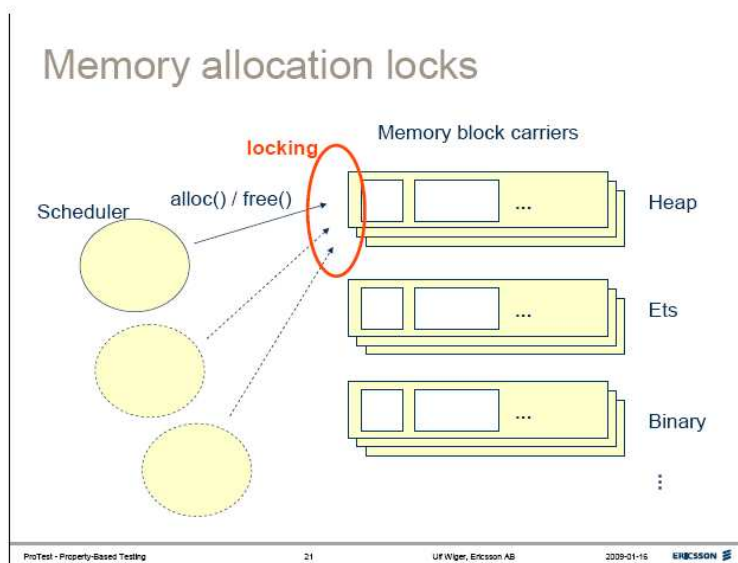
From: U. Wiger

And of course the typical shared state problems with concurrency also show up: bad performance e.g. due to locking or exclusive access. The next slide shows a better approach where each scheduler controls a subset of tasks. Access to the individual run queues is now no longer shared and needs no synchronization. On top of that the lessons from queuing theory are applied as well: multiple wait-queues are problematic because if one is empty a busy one cannot offload easily. Here task migration is used to avoid the problem.



From: U. Wiger

Internal shared state problems do not only show up with run-queue handling. Memory allocation is another typical problem zone. While on the application level all Erlang processes have their own, separate heap, this is not the case within the runtime memory allocation management as the next slide explains:



And this is not an Erlang problem only: If you are using Java with concurrency, make sure you measure the allocation times necessary for large pieces of heap memory. You might be in for a surprise!

The following benchmark where Erlang performs badly shows a very interesting aspect of dealing with threads: We have already seen that too many threads in a runnable state lead to much context switching and long response time. Here we see the opposite effect: too few threads (in other words: things to do within the Erlang application) lead to threads/cores being permanently put to sleep and woken up again – unnecessary context switches causing bad performance.

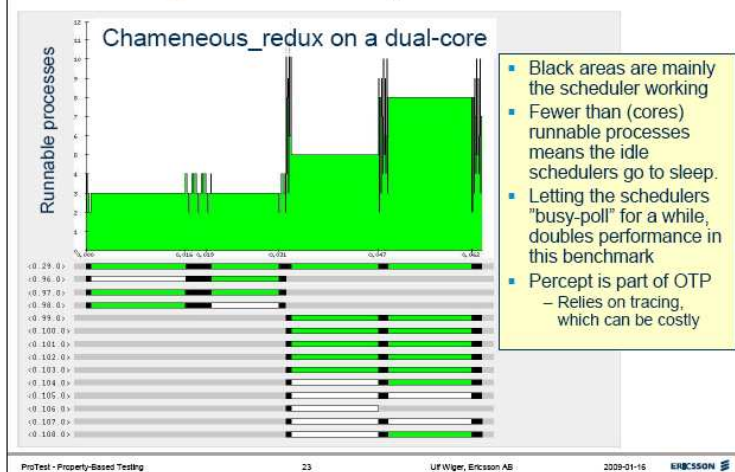
There are other benchmarks that are less flattering to Erlang. One of the worst known to-date is the "chameneos_redux" in the Computer Language

Shootout. It is basically centered around rendezvous, and a very poor match for message-passing concurrency (esp of the granularity that Erlang supports). One may note that the Scala entry, using much the same approach as Erlang, timed out...

We note that the best entries use some form of shared-memory mutex (spinlocks, MVars, etc.) The difference in performance is staggering. To add insult to injury, the OTP team has observed that this benchmark runs slower the more cores you throw at it.

On the next slide, we will try to see what is going on. [Wiger] Multi-Core Erlang pg. 22ff.

Profiling with Percept



According to Wiger the differences when other communication mechanism like shared memory spinlocks were used are “staggering”. But this is only a sign that one concurrency paradigm does not fit all bills. With tightly coupled, number crunching applications the message passing and process creation overhead in Erlang might be a problem. But again, the typical social network site does not have such requirements.

In the chapter on autonomous, selfmanaged systems we will see how agents in Erlang can create hierarchical feedback loops. For a more systematic look at concurrency concepts like declarative concurrency, message passing and shared state I recommend [vanRoy].

<<also: Actors on the JVM, Kilim, Scalaris>>

Multicore and large-scale sites

What does the trend towards multicore really mean for us developers of large scale sites? At first glance it looks like more cores simply means more requests per second are possible without changes to our software. This is what Joe Armstrong meant when he said that multi-core is good for legacy software: more things can run in parallel if they are independent. That’s a big IFF, but still... But is there nothing we need to worry about? More cores means slower cores! And this means that our requests suddenly may run longer than before. If we cannot afford to do so we need to either make our requests shorter (doing less work) or start splitting them up into parallel parts. And this means several threads working on one requests. While this is certainly possible we need to make sure that these threads really are available at the same time. Otherwise our request processing might take much longer. And we have to do so with a minimum of context switches per thread too! This is not easy to do!

<<architecture??>> It is a sync/async pattern with several async threads
<<slides half-sync/half async pattern>

<<sharing is good, but only on the social level with copies!!>>

Scale agnostic algorithms and data structures

Principles:

Decentralize, denormalize, don't share, be eventually consistent, parallelize, be asynchronous, specialize, cache

- Long-tail optimization (watch parallel processing for extreme delays)
- beyond transactions, large scale media processing
- combine requests into one – split large tasks into many smaller ones. Both can reduce execution time, but when and how?
- partitioned iteration (map/reduce)
- hadoop, hbase, big-table paper, google application engine, gfs,
- mostly consistent/correct approaches: win be losing some things?

Performance through imperfection? Code for the “good/fast” case and live with the failures? (relaxing of constraints etc.)

- eventually consistent (epidemic) protocols
- algorithms dealing with heterogeneous hardware environments (faster/slower server combinations e.g.) as expressed by Werner Vogels in “a word on scalability”
- consistent hashing.
- Central meta-data/decentral data combinations like media grids or Napster (but watch for downsides like loss of indirection and virtualization)
- MVCC [Rokytssky]
- Sharding logic (vertical sharding avoids downtime by just adding new columns and tables)
- Snapshots and Synchronization points

Graph data structures and processing:

<http://comlounge.tv/databases/cltv45>

<http://highscalability.com/blog/2010/3/30/running-large-graph-algorithms-evaluation-of-current-state-o.html>

[DeCandia et.al.] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall and Werner Vogels, “Dynamo: Amazon's Highly Available Key-Value Store”, in the *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007. <http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>

[Vogels] Werner Vogels, Eventually Consistent – Revisited,

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

[Vogels] Werner Vogels, Eventually Consistent, Building reliable distributed systems at a worldwide scale demands trade-offs—between consistency and availability. ACM queue,

http://portal.acm.org/ft_gateway.cfm?id=1466448&type=pdf

-

Scalability is a core requirement for today's media processing systems. What if your system hosts millions of media content of different kind and you would like to sift through those data asking specific questions?

A good architecture splits its code into two parts: a scale and distribution agnostic level and a scale and distribution aware lower level. This looks much like the way P2P networks use distributed hash tables. [Holl] pg 133,

For such an architecture the selection of the proper abstractions for the higher levels is of paramount importance. Holland describes systems that are so large that the set of records of a certain type and sets of related records cannot be kept on one system under the control of one resource manager. But instead of asking for distributed transactions to assure consistency according to Holland truly large scale applications use the abstraction of an entity on the upper, scale agnostic layer. And these entities are explicitly defined as not spanning machines and unable to support distributed transactions. So some of the scalability problems are reified and represented on the higher levels as abstractions and others can successfully be hidden in lower levels. The split between explicit representation and transparent function is one of the most critical decisions in distributed systems and it gets more critical with the size of a distributed application (where size means either users, content, timing requirements or all of this).

Partitioned Iteration: Map/Reduce

One of those splits has gotten very famous: the map/reduce algorithm used by Google to sift through its huge document base represents a clever split of a processing algorithm into two different parts. Joel On Software describes the steps toward this clever separation of code in an excellent article on functional programming ideas. [Spolsky]. The core idea really comes from functional programming and its concept of higher order functions. If we look at a typical iteration over some data we might notice something peculiar:

```
For (i=0;i<AllDocuments;i++)  
    Document=nextDocument();  
    Result=Process(Document)  
    Write(Result)
```

This code mixes the iteration and the processing steps and also forces the whole processing into a sequential mode: one document after the other is processed. Using the functional concept of higher order functions we can split the iteration from the processing:

```
Map(Documents, ProcessingFunction)  
    For (i=0;i<AllDocuments;i++)  
        New Thread(Document=nextDocument();  
        ProcessingFunction)
```

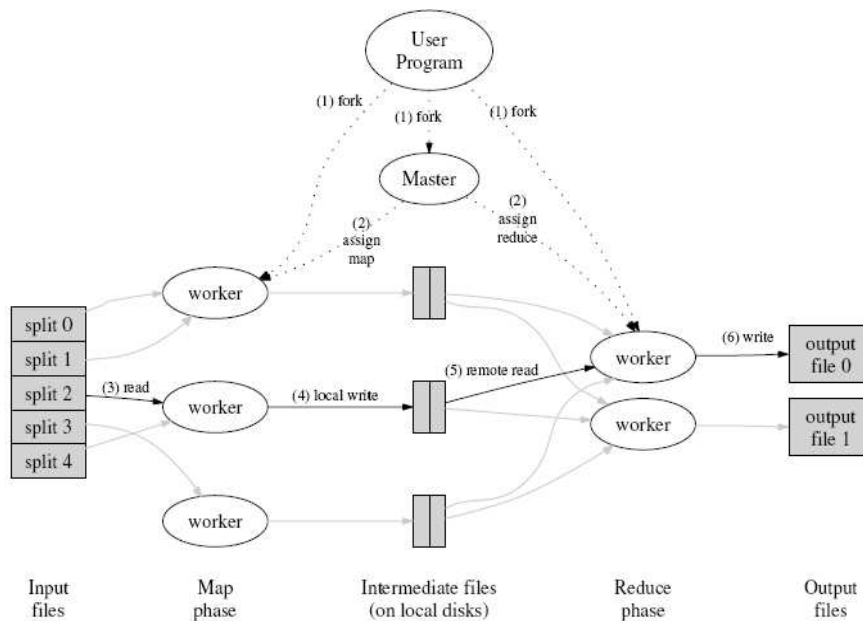
This new "map" function can accept any processing function we give it. The processing function can be created by application programmers while the map function can do very fancy distribution and parallelization of the documents and the processing function, e.g. send partitions of the

document base together with the processing function to different servers and handle all the distributed system logic and failure handling transparent to the application programmer!

Google engineers have invented the map function and they combined it with a second step, the so called “reduce” where the results can be aggregated according to some user defined reduce function that is also a higher order function like the process function handed over to map.

This architecture leads not only to usability improvements but also allows google to sift through its complete database hundreds of times a day and with many different hypothesis embedded in processing functions.

The diagram below shows some of the architectural elements used in this system. Of course one constraint must be fulfilled: It must be possible to apply the processing function to individual documents without side-effects. In other words the processing of one document does not influence the processing of other documents.



MapReduce: Simplified Data Processing on Large Clusters
 Jeffrey Dean and Sanjay Ghemawat of Google Inc.

A short example can show how much optimization the split into a distribution agnostic and a distribution aware part of the application allows: When millions of documents are sent to servers for processing strange effects can show up. When thousands of servers are used, some of those servers will fail. But they won't fail immediately in most cases. Instead, they start getting slower, e.g. because the disk develops more and more bad blocks which must be re-allocated etc. This leads to a very long tail for a map/reduce run: Almost all servers are ready except for a few which are still calculating. A clever map algorithm will account for those servers, monitor the processing and add duplicate processing requests when servers show malfunction. This reduces overall processing time by

as much as 30% - but you sure don't want your application programmers having to deal with reliability and distribution problems.

(Hadoop)

We basically see the same effect as with the Chubby/Paxos implementation at Google: there is a huge gap between the theoretical algorithm and the realities of its distributed and reliable implementation.

Incremental algorithms

Some algorithms require all elements of a computation to be re-processed in case a new element is added. A good example is the calculation of a mean. If this is done by again adding all the known elements and dividing them by their number the algorithm will not really scale with larger numbers. An alternative strategy is to calculate the new result from the last result and the newly added element. It is easily seen that this will require far fewer memory accesses and scale much better.

The pattern can be generalized to all kinds of incremental calculations.

Fragment algorithms

We have just seen that sometimes the addition of a new element requires the re-processing of many old elements. But we need to take a close look: is it really the case that the WHOLE algorithm applied to each element needs to be repeated? Or is it possible that some intermediate result of the algorithm still holds? In that case we have a fragment of the algorithm's result that we can cache and re-use in the calculation of the new result. This might increase throughput by orders of magnitude as I have just seen in an image comparison web application.

Long-tail optimization

When a large number of processing units is used the chance for some of them developing problems during the execution of e.g. a map-reduce job is rather high. An effective work distribution algorithm checks for slow machines and reschedules the respective tasks.

consistent hashing

(memcached etc.)

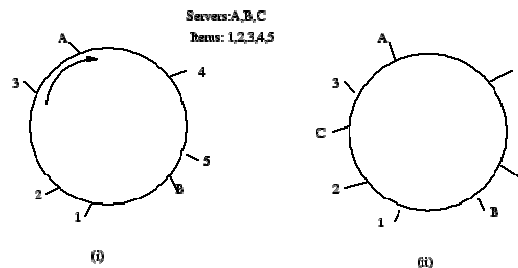
[Kleinpeter] Tom Kleinpeter, Understanding Consistent Hashing, <http://www.spiteful.com/2008/03/17/programmers-toolbox-part-3-consistent-hashing/>

[White] Tom White, Consistent Hashing, http://weblogs.java.net/blog/tomwhite/archive/2007/11/consistent_hash.html

[Karger] David Karger, Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web <http://citeseer.ist.psu.edu/karger97consistent.html>

[Karger et.al.] David Karger, Alex Sherman, Web Caching with Consistent Hashing

(i) Both URLs and caches are mapped to points on a circle using a standard hash function. A URL is assigned to the closest cache going clockwise around the circle. Items 1, 2, and 3 are mapped to cache A. Items 4, and 5 are mapped to cache B. (ii) When a new cache is added the only URLs that are reassigned are those closest to the new cache going clockwise around the circle. In this case when we add the new cache only items 1 and 2 move to the new cache C. Items do not move between previously existing caches. [Karger et.al.]



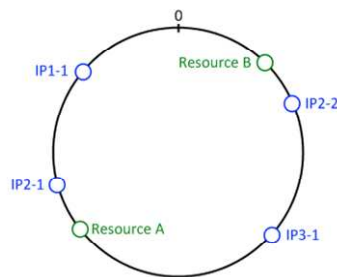
The term “consistent hashing” stands for a family of algorithms which intend to stop the “thundering herds of data” as Tom Kleinpeter calls the phenomenon of wild data re-arrangements caused by changes in the configuration of storage locations. A consistent hash function is a function that changes minimally as the range of the function changes [Alldrin]. Functions that associate a certain data item with a certain storage location in an automatic way are used in many areas. Distributed Hash Tables [DHT] rely on this technique as well as horizontal data partitioning schemes where e.g. certain user types are distributed across replica machines. No matter whether a real hash function is used to map data to locations or whether certain data qualities are used to map to a range of machines: the number of machines or locations is a parameter of the mapping function and if this number changes the mappings change as well.

It is a classical second order scalability problem: first order scalability partitions data across storage locations and makes both access and storage scalable. Second order scalability – here represented by consistent hashing - needs to make the partitioning scalable in the face of machine changes and additions. Some important criteria for what we want to achieve: *“First, there is a “smoothness” property. When a machine is added to or removed from the set of caches, the expected fraction of objects that must be moved to a new cache is the minimum needed to maintain a balanced load across the caches. Second, over all the client views, the total number of different caches to which a object is assigned is small. We call this property “spread”. Similarly, over all the client views, the number of distinct objects assigned to a particular cache is small. We call this property “load”. [Karger]*

To give a simple example from [karger] et.al: A simple hash function like
 $X \rightarrow ax + b \pmod{P}$

With P being the number of machines available would have a “thundering herd” characteristics if e.g. used to partition data across a distributed cache and the number of machines in this cache changes. The change could be caused by crashes or increasing load. Suddenly almost every cached item is on the wrong machine and therefore unreachable. The caches would have to be flushed anyway because invalidation events would also no longer reach the right machines.

The following diagram shows one way to achieve consistent hashing in a DHT ring. The example is taken from [Kleinpeter].



Here the whole hash range forms a ring with the first and the last hash value being next to each other. Two resources have been mapped into the ring via their hash values. And three nodes have also been mapped into the ring at random positions using hashes of their IP addresses. Node #2 is bigger and has two IP addresses (or locations on the ring) therefore. The following rule applies: A node is responsible for all resources which are mapped between his own position and the position of its predecessor (when walking the ring clockwise).

There are some obvious advantages to this scheme: you can deal with heterogeneous hardware easily by handing out more IP numbers. You can slowly bootstrap new servers by adding IPs in a piecemeal fashion and in case of a server crash the load should be distributed rather equally to the other machines. [Kleinpeter]

Let's take a look at what happens when a server crashes. If #2 crashes the resources A and B are re-assigned to new nodes. So far so good but in practice a number of problems will have to be dealt with:

- How do we know that #2 is down? We don't want to hang in network stacks for a long time.
- What happens to the stored resources? The new nodes do not have replicas. We can either design a read-through cache which makes the newly responsible nodes turn around and fetch the data from some store (difficult because the simple key/value interface does not transport parameters needed to re-create the data e.g. from some backend service). Or we let the cache-read request fail at the client and the client goes to the storage to get the original value.
- And what happens if unfortunately Michael Jackson's newly found very last video clip shows up and is mapped to node #1? Then we learn that our load partitioning using random hashes cannot deal with a very uneven distribution of requests for specific data. The "load" property mentioned by [Karger] only assures that a small number of objects is mapped to a node. It does not take the number of requests into account.
- The random distribution of resources and nodes may lead to uneven load distribution.
- There are no provisions yet for availability of data. This may not be necessary for a cache but is certainly needed for other applications. Also: more and more caches are of vital importance for large sites which are no longer able to re-generate all content needed from scratch and in a reasonable time.
- The last point has also consequences for the new nodes: They cannot just copy the data from another node because there is none with the same data.
- IP numbers are not the ideal type to use for nodes. Some form of virtual tokens would be better.
- Membership information about nodes and tokens need to be kept and maintained (e.g. via gossip protocols) by each node.

The Amazon Dynamo implementation as described in [DeCandia] also uses consistent hashing in a similar way but shows some improvements with respect to the deficiencies just mentioned:

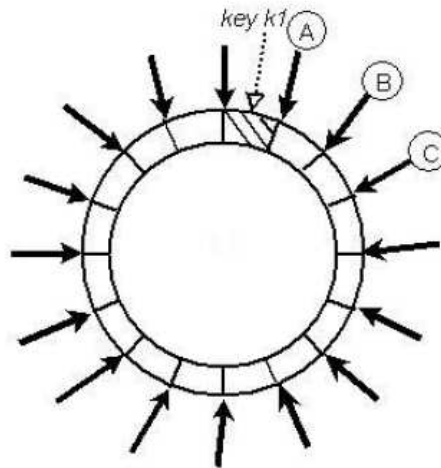
"The fundamental issue with this strategy is that the schemes for data partitioning and data placement are intertwined. For instance, in some cases, it is preferred to add more nodes to the system in order to handle an increase in request load. However, in this scenario, it is not possible to add nodes without affecting data partitioning. Ideally, it is desirable to use independent schemes for partitioning and placement" [DeCandia]

The Dynamo architecture finally ended up dividing the ring into equally sized partitions which were assigned to virtual tokens and nodes and replicating the data across several nodes. This brought several advantages and disadvantages like

- having partitions in one place/node which made archiving and snapshots easier
- needing a coordination process to decide on partition/node associations
- gossiping of compact membership information between nodes

- being able to move replicas to a new node incrementally
- avoiding costly data scans at local nodes in case of configuration changes

The diagram below shows the chosen solution. For a detailed description of the Dynamo store – especially its eventually consistent features see [DeCandia].



From: DeCandia et.al, Dynamo, Amazons highly available Key/Value Store

The necessary helper services like coordination, eventual consistency and membership/failure detection are discussed below. Amazon puts a lot of emphasis on SLAs which determine the runtime of services rather strictly, e.g. at the 99.99 percentile. Consistent lookup and atomic merges are further requirements on DHTs and we will take a closer look at the Scalaris DHT which is implementing those requirements in the section on leading edge architectures below.

For a connection with replication see: Honicky, Miller, [HM], Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution, UCSC.

<<scalis: consistent lookup, atomic merge>>

beyond transactions, large scale media processing

when to give up the idea of (distributed) transactions and how to cope with the fallout.

mostly consistent/correct approaches:

win be losing some things? Performance through imperfection? Code for the “good/fast” case and live with the failures? (relaxing of constraints etc.)

Failure Detection

Ping based network approach

Membership protocols

algorithms dealing with heterogeneous hardware environments

(faster/slower server combinations e.g.) as expressed by Werner Vogels in “a word on scalability”

Striping works best when disks have equal size and performance. A non-uniform disk configuration requires a trade-off between throughput and space utilization: maximizing space utilization means placing more data on larger disks, but this reduces total throughput, because larger disks will then receive a proportionally larger fraction of I/O requests, leaving the smaller disks under-utilized. GPFS allows the administrator to make this trade-off by specifying whether to balance data placement for throughput or space utilization. [Schmuck] pg. 4.

Shortlived Information

- group communication based service for social information (presence, same page etc.)
(Schlossnagle)

Sharding Logic

Scheduling and Messaging

(Gearman), ejabberd,

Task and processing Granularity with same block size, task time etc.

Collaborative Filtering and Classification

* * Taste Collaborative Filtering - Based on the Taste project which was incorporated into Mahout, including examples and demo applications

* Naive Bayes Implementations - Implementations of both traditional Bayesian and Complementary Bayesian classification are included

* Distributed Watchmaker Implementation - A distributed fitness function implementation using the Watchmaker library, along with usage examples

<http://www.infoq.com/news/2009/04/mahout>

Und hier die eigentlich Projekt Homepage:

<http://lucene.apache.org/mahout/>

Clustering Algorithms

* Distributed Clustering Implementations - Several clustering algorithms such as k-Means, Fuzzy k-Means, Dirchlet, Mean-Shift and Canopy are provided, along with examples of how to use each

<http://www.infoq.com/news/2009/04/mahout>

Und hier die eigentlich Projekt Homepage:
<http://lucene.apache.org/mahout/>

Number Crunching

* Basic Matrix and Vector Tools - Sparse and dense implementations of both matrices and vectors are provided

*Hier ist der Link zur Übersicht auf InfoQ:
<http://www.infoq.com/news/2009/04/mahout>

Und hier die eigentlich Projekt Homepage:

<http://lucene.apache.org/mahout/>

Consensus: Group Communication for Availability and Consistency

- spread, virtual synchrony [Schlossnagle], Spread toolkit [Amir et.al.]
- Fault-tolerant PAXOS implementation as an example of synchronous (quorum) group communication. [Google]
- Bryan Turner, The Paxos Family of Consensus Protocols, [Turner]. Good explanation of the Paxos protocol.
- CAP theorem/eventually consistent paper by Werner Vogels
- Backhand, wackamole (Schlossnagle)

Most distributed systems have a need for some form of agreement or consensus with respect to certain values or states. Locking is a typical example, replication of critical values or commands another. Just to solve basic questions like who is currently responsible for what a reliable mechanism is needed. Reliable meaning that it should work even in the presence of machine or network failures and despite the famous impossibility theorems of FLP and CAP.

We are going to look at two different consensus algorithms with different performance and reliability guarantees. The first is PAXOS, a well know and frequently used distributed consensus algorithm from Lamport. The other one is based on group membership and virtual synchrony. The implementation section in between discusses some lessons learned by Google engineers when they implemented Paxos for the Chubby lock service.

Paxos: Quorum based totally ordered agreement

In the presence of failures consensus is reached when a majority within a static group of nodes agrees on a certain value. This is called a quorum. Some rules apply to guarantee consistent decisions:

With N being the number of nodes, a client must at least write to WQ nodes and read from RQ nodes with $WQ + RQ > N$. In our example we have 5 machines, $WQ = 3$ and $RQ = 3$. In that case every write or read will have a majority. Individual writes will have a timestamp or counter associated which lets a client detect the latest version of a value.

The following is based on [Turner]. The basic Paxos protocol knows several roles in addition to the client. The node receiving a client request is called proposer. It needs to become a leader to process the request. Acceptors are basically the voters in the protocol and learners store and retrieve values. In practice these roles are rolled together at each node, sometimes even the client. But for the purpose of demonstration we will keep them mostly apart.

Let's again assume we have five nodes who together perform the Paxos algorithm. Such groups are a frequent pattern in distributed systems to e.g. provide locking, reliable storage of few but important values, coordination of tasks etc. (see also [Resin])

A client sends a request to one of the participating nodes. If the node is up it will

- a) propose himself as a leader for this request to the others
- b) Collect acceptance messages from the others
- c) once accepted as a leader send the request to every other node
- d) wait for confirmation from acceptors and values from learners

<<basic paxos diag>>

[Message Flow : Basic Paxos (one instance, one successful round)]

Client Proposer A1 A2 A3 L1 L2

-----> Request

----->Prepare(N)

----->Prepare(N)

----->Prepare(N)

<----- Promise(N,{Va,Vb,Vc})

<----- Promise(N,{Va,Vb,Vc})

<----- Promise(N,{Va,Vb,Vc})

Client (now) Leader A1 A2 A3 L1 L2

-----> Accept!(N,Vn)

-----> Accept!(N,Vn)

-----> Accept!(N,Vn)

<----- Accepted(N,Vn)

<----- Accepted(N,Vn)

<----- Accepted(N,Vn)

<----- Response

<----- Response

(modified after [Turner], A= Acceptor, L=Learner, N=Instance, V=Value)

This process flow naturally splits into two phases: an initiator phase where leadership is decided and a data processing phase where values are read or written. The leadership principle ensures a total order of values and the protocol makes progress as long as there is a quorum of nodes available. Please note that the Promise response from acceptors can contain a value from a previous instance run by a different leader which crashed during the accept

phase. In this case just one of the acceptors might have seen the accept command with this value and it is now essential for the new leader to take this value as the value for his first round so all other acceptors learn the previously committed value.

“As long as quorum is available” is a critical point in the architecture of Paxos. The minimum number of active nodes to achieve a consensus is $2F + 1$ assuming F concurrently failed machines. With only $2F$ nodes we could experience a network partition problem and we would not know which half of the nodes is correct. Why is this important? Because with the number of assumed concurrent machine failures the write quorum (WQ) and the read quorum (RQ) grows as well.

This means we have to do more writes and reads which slows request handling down. But it gets worse. The Paxos protocol is based on synchronous acknowledgements – nodes always have to reply for the protocol to make progress. And there are at least two rounds of this req/ack pattern needed per request (we will talk about optimizations shortly). [Birman] concludes therefore that Paxos is a very reliable albeit slow protocol to achieve consensus. He talks about tens of requests per second, Google reports fivehundred but we need to realize that a tightly coupled high-speed distributed processing would probably use a different algorithm for replication. This is not a problem for many cases where Paxos is used today: locking of system resources, some critical replication of small values etc. are no problem at all.

There are many optimizations possible in Paxos. The protocol obviously benefits from a stable leader who could process more requests within one instance without having to go through the leadership agreement first. A sub-instance number added to the Accept command will take care of that extension which is called Multi-Paxos.

Accept!(N, I, V_n)

Accepted(N, I, V_n)

Accept!(N, I+1, V_n)

Accepted(N, I+1, V_n)

....

Another optimization (Generalized Paxos) concerns the values and their mutual dependencies. If a leader can detect that certain concurrent requests from clients are commutative, it can bundle those requests within a single round and further reduce the number of rounds needed. Don't forget: in a quorum system even reads need to go to several nodes before a value returned can be considered consistent!

Proposed Series of operations by two clients received at a node (global order). A state machine protocol maintains two values A and B:

1:Read(A)
2:Read(B)
3:Write(B)
4:Read(B)
5:Read(A)
6:Write(A)
7:Read(A)

1, 2 and 5 are commutative operations. So are 3 and 6 and finally 4 and 7. The node batches the operations into three rounds:

1. Read(A), Read(B), Read(A)
2. Write(B), Write(A)
3. Read(B), Read(A)

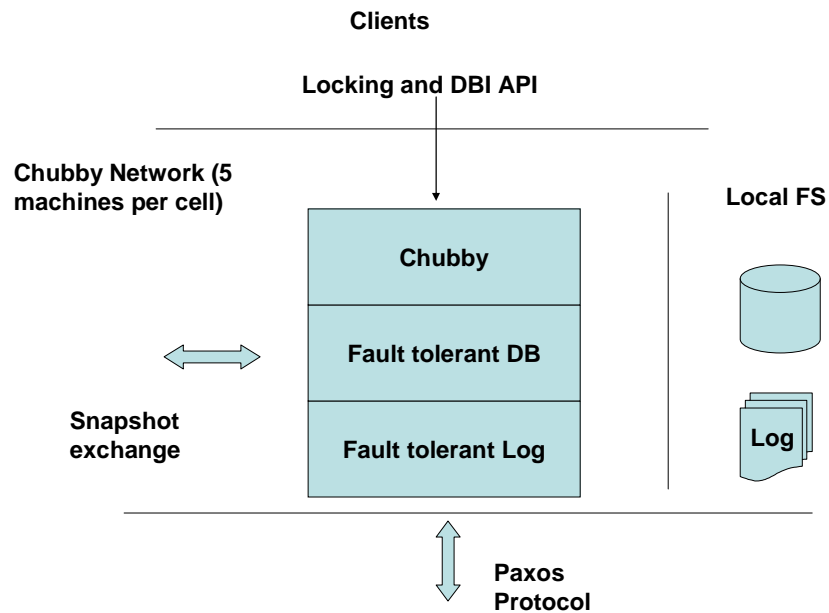
(after [Turner])

The final optimizations turn the leader-based Paxos protocol into something that resembles more membership protocols based on multi-cast virtual synchrony and is called Fast Paxos. Here Clients send messages directly to acceptor/learner nodes. The nodes send accepted messages to each other and the leader and only in case of conflict the leader sends out the canonical (his) accept message to resolve conflict. This is very similar to letting nodes communicate freely via multicast with one node sending out the defined order of those messages every once in a while. This protocol can be further improved with respect to message delays when there is a mechanism in place which lets acceptor/leader nodes not only detect conflicts (this is ensured by messages being sent to all participants) but also to resolve conflicting requests automatically.

Paxos Implementation Aspects

One of the best papers on distributed systems engineering available is “Paxos Made Live – An Engineering Perspective” by Chandra, Griesemer and Redstone of Google [Chandra] et.al. It describes the considerable engineering effort needed to create a fault tolerant log running on a cell of five machines. On top of this log other functions like a fault-tolerant store and locking mechanism have been built which were described already in the section on components needed in ultra large systems.

<<chubby arch>>



After: [Chandra] et.al

In my eyes the paper is also a clear calling for well-tested open source implementations for all kinds of group communication needs (membership, consensus, failure detection). Such a component is clearly needed in large systems but the effort to turn an algorithm into a robust service implementation is huge.

The paper is divided into sections on Paxos, Algorithmic challenges, Software engineering and finally unexpected failures. In the Paxos part [Chandra] et.al describe a rather regular use of Multi Paxos with propose phases prevented by sticking to one leader called master. The whole API for the log already routes client requests to arbitrary replicas to the one master node. This is essential for good performance with Paxos.

The algorithmic challenges consisted of significant performance improvements using leases for master and replicas and better robustness in case of disk errors. Even an extension to the protocol had to be made due to unexpected failures at nodes.

A look at the Paxos protocol in the context of a necessary quorum for guarantee consistency makes it clear that even a simple read against the log would involve a full Paxos round of requests against a read quorum. But in case of a fixed master, shouldn't it be able to return a read value from its own store? The problem lies in the fact that other replicas can at any time decide to start a new round of leader election, perhaps without notifying the master. This could have led to a new read value and the old master would then return stale data from its store.

To prevent unnecessary churn of masters a master is granted a lease. As long as the lease is valid the master knows that it will be

the only one to answer requests and can therefore take read values right from its own store and return them to clients.

Leases are certainly going to improve the performance of Paxos due to lesser rounds needed. But they are nevertheless dangerous: What happens in case of a master having problems? Or being disconnected? To make progress a new master must be elected and then the question arises: what happens to the lease at the old master? What if it only experienced temporary performance problems and wants to continue now? How could it now about a potential network partition without doing a Paxos round and asking for a quorum? The Google engineers do not tell exactly how they distinguish those cases and what happens to the lease. And there are more problems with the implementation of the protocol:

In the presence of temporary network outages or disconnects the Paxos protocol might cause two master nodes to fight for control with each increasing their instance number every time they come back. The problem was solved with forcing the master to regularly run full Paxos rounds and to increase their instance numbers at a certain frequency.

Something else might lead to a fast churn rate: what if the nodes participating in consensus run some other processes as well? If the load caused by those processes becomes too high it might affect the ability of a master to respond quickly enough to requests from his peers – who might conclude that the master is dead and start a new election. This means there must be a scheduler service available which can guarantee a certain response time to some processes. << add this to scheduling >>

According to the google engineers the higher level lock-service protocol requires a request to be aborted when the master changes during the request – even if it becomes master again during the request. This forced the designers to implement a so called epoch number for a master. It is a global counter of master churn at a specific master. Losing mastership and later acquiring it again will lead to a new epoch number. As all requests are conditional of their epoch number it is now easy to decide when a request has to be aborted.

Disc corruption was another interesting challenge within the implementation of Paxos. As every Paxos node makes promises during rounds, it cannot be allowed that the results of those rounds are changed behind the back of the protocol. File corruption is prevented using checksums and a way to distinguish an empty disk (new) from a disk with inaccessible files (error) was needed. To this purpose a node writes another marker in the Google File System and when it reboots it checks for the marker. If it finds one it knows that the disk is corrupt and starts rebuilding it by contacting the other replicas and getting the latest snapshot of the system state.

Snapshots are needed to condense an ever growing log of actions and commands into a static state. Snapshots have a number of requirements that are hard to fulfil:

- sometimes a snapshot spans several resources which are independently updated
- in most cases it is impossible to stop the system to take a snapshot
- snapshots must be taken quickly to keep inconsistencies small
- a catch-up algorithm is needed to get the changes after a snapshot has been taken.

I will leave it to the reader to learn about other optimizations like database transactions using complex Paxos values and concentrate on a few but critical experiences in software engineering.

[Chandra] pg. 9

The Google engineers used four essential techniques to achieve fault-tolerance and reliability:

1. An explicit model of the Paxos algorithm.
2. Runtime consistency checking
3. Testing
4. Concurrency restrictions

A consensus protocol like Paxos is used to implement the state machine approach of distributed processing and lends itself to an implementation using a state machine specification language. From own experience I can say that having such a grammar which can be turned into code via a compiler construction tool is an incredible advantage over having complex events and states directly implemented in software. Protocol problems are much easier to find this way.

Implementors of large scale systems fear one thing especially: runtime corruption of data structures. This is a well known problem in storage technologies (ZFS was once thought to destroy disks only because it contained test and validation code which detected silent data corruption). The same goes for memory corruption in unsafe languages like C or C++ and so on. The google engineers reported that they used extra databases to hold checksums of other database information.

Testing needs to be repeatable to have any value. Code needs extensive instrumentation to generate test input and output. A rather unnerving fact is the tendency of fault tolerant systems to hide errors. A node that is wrongly configured will try forever to join some group just to be rejected again and again. A casual observer will only notice that this node has probably crashed and is now catching up without realizing the systematic error behind. In our chapter on modelling ultra large systems we have shown how hardware engineers use markov chains to put a probability on

certain state changes which could be used to detect systematic errors.

Within the Chubby/Paxos implementation a deliberate effort was made to avoid multi-threading. While partially successful the engineers had to admit that many components had to be made concurrent later on for performance reasons [Chandra] pg. 12

Agreement based on virtual synchrony

<http://www.jgroups.org/>

spread

Optimistic Replication

“Thou shalt not copy” is usually a good advice in IT. Every copy automatically raises the question of up-to-dateness. The more copies the more trouble to keep them synchronized. But in many cases either performance/throughput arguments or availability of resources force us to create copies. And sometimes scalability forces us to even give up on a central consistency requirement: that all copies have to have the same state as the master before a client gets access to one of the copies. Long distance and poor latency exclude a pessimistic replication strategy as well. Using synchronous requests over several rounds to achieve consensus is just too expensive.

Lately the concept of “eventual consistency” has become popular, e.g. with Amazon’s key/value store called dynamo. Werner Vogels has written extensively about their use of eventual consistent techniques like handoff-hints etc. [Vogels] and [DeCandia].

Let us go back to the problem of multiple copies and see what it takes to bring them into eventual consistence and what this means for clients. To do so we need to answer the following questions:

1. **who does the update? Single Master or multiple masters**
2. **What is updated? State transfer or operation transfer?**
3. **How are updates ordered?**
4. **How are conflicts handled/detected?**
5. **How are updates propagated to replica nodes?**
6. **What does the system guarantee with respect to divergence?**

Roughly after [Saito]

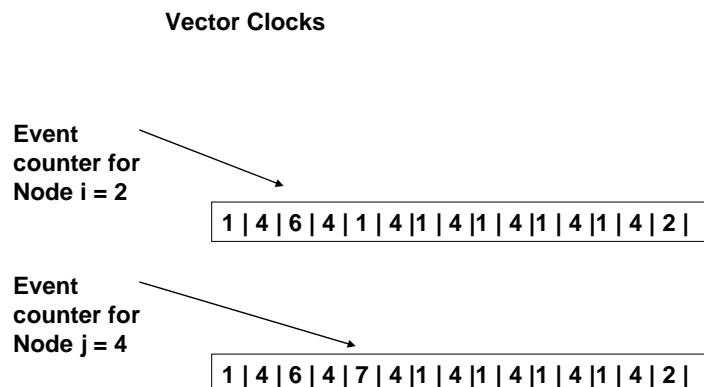
The following is a discussion of selected topics from [Saito et.al.]. They describe asynchronous replication algorithms and its problems in great detail. Before we start the discussion let's mention some systems and applications which use optimistic replication and accept eventual consistency. DNS and usenet news are very popular examples and their excellent scalability has been proven many times. They flood updates through their network successfully. CVS is another optimistic replication schema. It accepts concurrent updates by allowing offline operation but flags potential conflicts. P2P file sharing comes to mind as well as PDA – PC replication of personal user data. I do mention those successful applications of optimistic replication to overcome the uneasy feeling in the tummy once transactional guarantees are no longer available. But fact is: many applications can live perfectly and some only with optimistic replication.

The question of single-master vs. multiple master is rather critical for replication systems. A single master excludes scheduling and conflict detection problems and – surprisingly – may scale much better than a multi-master replication system. The reason might be the increase in conflicts and conflict resolution overhead once multiple masters accept concurrent updates. While at the same time a single master can serialize access easily and with little – especially no networking – costs. [Saito] et.al. pg. 10.

Do we transfer the state of complete objects or do we transfer individual operations that – executed at the target site – will produce perfect copies according to the distributed state-machine principle? This depends very much on the application. State transfer seem ideal for small objects, expensive calculation costs and low latency connections. Operation transfer allows semantically rich treatment at the receiver side, saves

potentially network bandwidth and transmit times. Both use different techniques to detect and handle conflicts (e.g. using chunks for incremental updates).

There are numerous ways to detect conflict. From gossiping about object state between machines and comparing their timestamps to comparing causal histories of updates with vector clocks and so on. Vector clocks are the swiss army knife of creating order in distributed systems.



Vector clocks are transmitted with messages and compared at the receiving end. If for all positions in two vector clocks A and B the values in A are larger than or the same as the values from B we say that Vector Clock A dominates B. This can be interpreted as potential causality to detect conflicts, as missed messages to order propagation etc.

To solve conflict we can use Thomas' write rule which leads to older objects slowly to disappear from the replicas. CVS pushes the question of conflict handling in certain cases to the user of the application. Fully automatic ways to deal with conflicts will frequently have a price to be paid in consistency.

How updates are propagated depends on the topology of the network and how users of our replication system will interact with it. Here session behaviour is a very important point because most applications need to guarantee at least consistent sessions.

Session Guaranties with optimistic replication

“**Read your writes**” (RYW) guarantees that the contents read from a replica incorporate previous writes by the same user.

“**Monotonic reads**” (MR) guarantees that successive reads by the same user return increasingly up-to-date contents.

“**Writes follow reads**” (WFR) guarantees that a write operation is accepted only after writes observed by previous reads by the same user are incorporated in the same replica. No jumping back in tome with a replica that missed some writes.

“**Monotonic writes**” (MW) guarantees that a write operation is accepted only after all write operations made by the same user are incorporated in the same replica. (read set of client received from replica will show those write events)

After [Saito]. Remember that it is transparent to the client which replica answers a request

Werner Vogels of Amazon correctly points out that applications which violate the first two conditions are very hard to use and understand.

Finally the question of divergence of replicas needs to be answered. And here the solutions are rather limited. Epsilon consistency with its famous example of an international bank account comes to mind: A bank which wants to restrict the damage fom overdraft in five regions will set a limit of $x/5$ per region.. [Birman] gives some interesting numbers on the behaviour of epidemic distribution protocols which seem to show a high degree of reliability.

- session consistency
- epidemic propagation
- vector clocks

Failure Models

Time in virtually hosted distributed systems

[Williamson]

```
[root@domU-12-31-xx-xx-xx-xx mf]# ping 10.222.111.11
PING 10.222.111.11 (10.222.111.11) 56(84) bytes of data.
64 bytes from 10.215.222.16: icmp_seq=2 ttl=61 time=473 ms
64 bytes from 10.222.111.11: icmp_seq=4 ttl=61 time=334 ms
64 bytes from 10.222.111.11: icmp_seq=5 ttl=61 time=0.488 ms
64 bytes from 10.222.111.11: icmp_seq=6 ttl=61 time=285 ms
64 bytes from 10.222.111.11: icmp_seq=7 ttl=61 time=0.577 ms
64 bytes from 10.222.111.11: icmp_seq=8 ttl=61 time=0.616 ms
64 bytes from 10.222.111.11: icmp_seq=9 ttl=61 time=0.794 ms
64 bytes from 10.222.111.11: icmp_seq=10 ttl=61 time=794 ms
```


64 bytes from 10.222.111.11: icmp_seq=11 ttl=61 time=0.762 ms
64 bytes from 10.222.111.11: icmp_seq=14 ttl=61 time=20.2 ms
64 bytes from 10.222.111.11: icmp_seq=16 ttl=61 time=0.563 ms
64 bytes from 10.222.111.11: icmp_seq=17 ttl=61 time=0.508 ms
64 bytes from 10.222.111.11: icmp_seq=19 ttl=61 time=706 ms
64 bytes from 10.222.111.11: icmp_seq=20 ttl=61 time=481 ms
64 bytes from 10.222.111.11: icmp_seq=22 ttl=61 time=0.868 ms
64 bytes from 10.222.111.11: icmp_seq=24 ttl=61 time=1350 ms
64 bytes from 10.222.111.11: icmp_seq=25 ttl=61 time=4183 ms
64 bytes from 10.222.111.11: icmp_seq=27 ttl=61 time=2203 ms
64 bytes from 10.222.111.11: icmp_seq=31 ttl=61 time=0.554 ms
64 bytes from 10.222.111.11: icmp_seq=32 ttl=61 time=678 ms
64 bytes from 10.222.111.11: icmp_seq=34 ttl=61 time=0.543 ms
64 bytes from 10.222.111.11: icmp_seq=35 ttl=61 time=25.6 ms
64 bytes from 10.222.111.11: icmp_seq=36 ttl=61 time=1955 ms
64 bytes from 10.222.111.11: icmp_seq=41 ttl=61 time=809 ms
64 bytes from 10.222.111.11: icmp_seq=43 ttl=61 time=2564 ms
64 bytes from 10.222.111.11: icmp_seq=44 ttl=61 time=7241 ms

As you can appreciate, this has some considerable knock-on effects to the rest of our system. Everything grinds to a halt. Now I do not believe for a moment, this is the real network delay, but more likely the virtual operating system under extreme load and not able to process the network queue. This is evident from the fact that many of the pings never came back at all.

[VMWare] Time in VMWare ...

The problem is that distributed algorithms for consensus, locking or failure detection all rely on rather short and predictable latencies to predict a failure reliably (meaning long timeouts) and at the same time achieve a high throughput (meaning short timeouts). VMs will try to catch up by delivering timer interrupts faster but this mechanism can clash with higher level time setting protocols badly when run at the same time. Overcompensation is one possible result.

Problem: how to monitor cloud app performance externally (Gomez?)

Williamson:

Following on, I noticed that cloudkick, the cloud performance monitoring people, published their own findings on the network latency, and digging into their graphs, we find a complete correlation with our own data.

Part VI: New Architectures

- media grid
- Peer-to-Peer Distribution of Content (bbc)
- Virtual Worlds
- Cloud Computing??
- Web app APIs from Google and Yahoo
- Scalaris with transactions
- Selfman self-management and feedback loops with agents

Cassandra and Co.

(Todd Hoff, MySQL and Memcached, the end of an era?,

<http://highscalability.com/blog/2010/2/26/mysql-and-memcached-end-of-an-era.html?printerFriendly=true>)

Design Patterns for Distributed Non-Relational Databases (Cloudera, Todd Lipcon). Very good schematics on row/column and mixed storage and log structured merge trees. (perhaps better in algorithms and bigtable discussion). The points are: automatic scalability. Huge growth. Non intelligent reads dominate. Mostly no transactions. Cassandra, MonoDB, Voldemort, Scalaris...

With a little perspective, it's clear the MySQL+memcached era is passing. It will stick around for a while. Old technologies seldom fade away completely. Some still ride horses. Some still use CDs. And the Internet will not completely replace that archaic electro-magnetic broadcast technology called TV, but the majority will move on into a new era.

LinkedIn has moved on with their Project Voldemort. Amazon went there a while ago.

Digg declared their entrance into a new era in a post on their blog titled Looking to the future with Cassandra, saying:

The fundamental problem is endemic to the relational database mindset, which places the burden of computation on reads rather than writes. This is completely wrong for large-scale web applications, where response time is critical. It's made much worse by the serial nature of most applications. Each component of the page blocks on reads from the data store, as well as the completion of the operations that come before it. Non-relational data stores reverse this model completely, because they don't have the complex read operations of SQL.

Twitter has also declared their move in the article Cassandra @ Twitter: An Interview with Ryan King. Their reason for changing is:

We have a lot of data, the growth factor in that data is huge and the rate of growth is accelerating. We have a system in place based on shared mysql + memcache but its quickly becoming prohibitively costly (in terms of manpower) to operate. We need a system that can grow in a more automated fashion and be highly available.

It's clear that many of the ideas behind MySQL+memcached were on the mark, we see them preserved in the new systems, it's just that the implementation was a bit clunky. Developers have moved in, filled the gaps, sanded the corners, and made a new sturdy platform which will itself form the basis for a new ecosystem and a new era.

[Building Large AJAX Applications with GWT 1.4 and Google Gears](#)

In this presentation from QCon San Francisco 2007, Rajeev Dayal discusses building applications with GWT and Google Gears. Topics discussed include an overview of GWT, integrating GWT with other frameworks, GWT 1.4 features, developing large GWT applications, integrating GWT and Google Gears, the architecture of a Google Gears application, Google Gears features and the Google Gears API.

Adaptive, Self-Managed ULS Platforms

www.selfman.org: European Research on self-managed systems

[Andrzejak] Artur Andrzejak, Alexander Reinefeld, Florian Schintke, Thorsten Schütt, On Adaptability in Grid Systems, Zuse Intitute Berlin

[vanRoy] Peter van Roy, Self Management and the Future of Software Design, <http://www.ist-selfman.org/wiki/images/0/01/Bcs08vanroy.pdf>

[vanRoy] Peter van Roy, The Challenges and Opportunities of Multiple Processors: Why Multi-Core Processors are Easy and Internet is Hard (short piece on conflicting goals in p2p and emergent behaviour like the intelligence of google search)

[vanRoy] Peter van Roy, Overcoming Software Fragility with Interacting Feedback Loops and Reversible Phase Transitions. (again the concept of feedback loops for control)

[Northrop] Linda Northrop, Scale changes everything,

[Gabriel] Richard Gabriel, Design beyond human abilities

[SEI]

[UK]

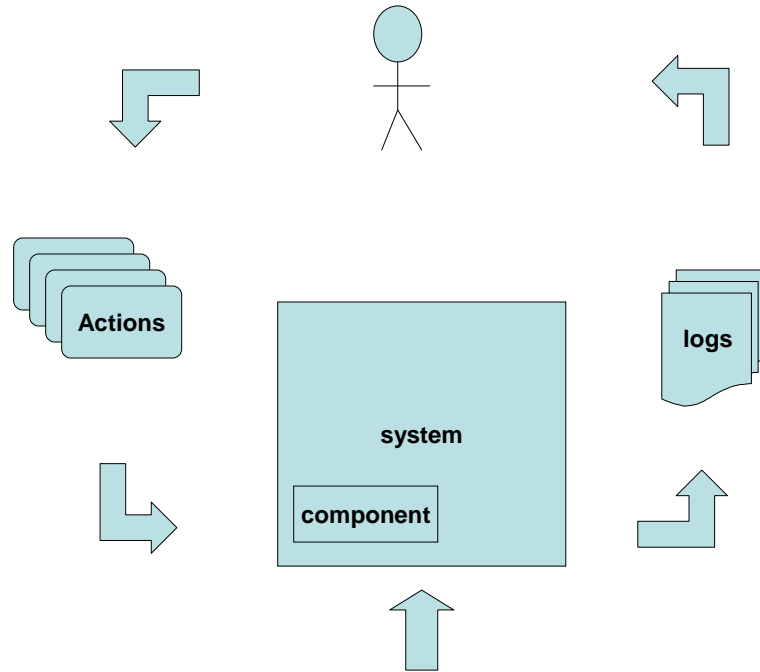
“Human-in-the-loop”

The approach in this book has been a rather practical one: take a look at real ULS sites and investigate the architectures and methods used to build them. The assumption behind is that while the practices and technologies used certainly are different in ULS, it is still conventional engineering that is used to build them. Even though it is a more complex kind of engineering that is needed and which includes the social environment explicitly. And even though it is a kind of engineering that stumbles from roadblock to roadblock only to re-engineer what was built before to make it adapt to new challenges.

But this approach is not undisputed. There are at least two groups of researchers who go way beyond and challenge the engineering approach to ULS in general:

ULS design will have to move beyond computer science and electrical and electronics engineering-based methodologies to include building blocks from seven major research areas: human interaction; computational emergence; design computational engineering; adaptive system infrastructure; adaptable and predictable system quality; and policy, acquisition, and management.
[Goth]

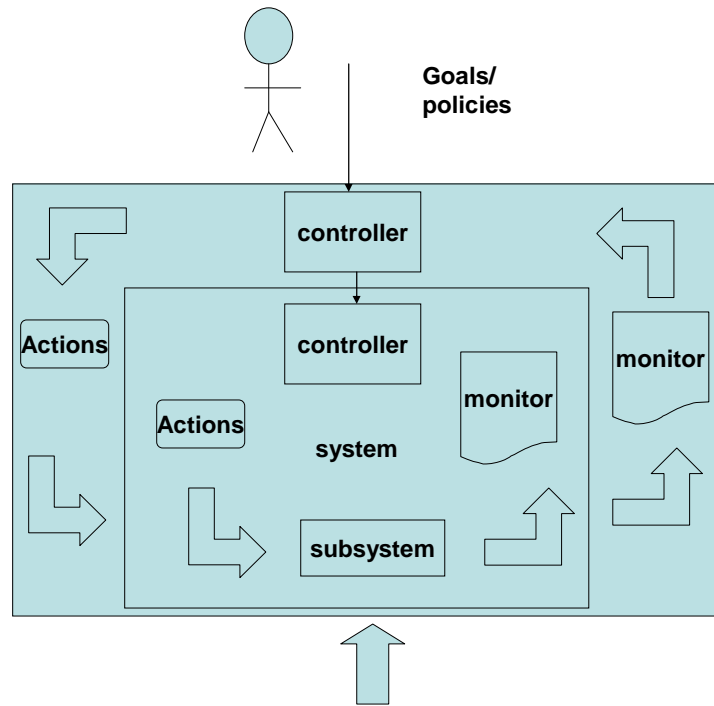
Both groups of researchers share the above statement more or less but differ in the engineering approach and especially in the scope of their vision. But take a look first at the approach that criticized by both groups an which I call “human in the loop”:



Currently there is always a human involved in the basic feedback loops which keep systems in a stable state while enduring external inputs and forces. And of course humans were needed to build the whole system in the first place. Adaptation, the change of a system in processes, structures etc. is done manually. <<def adaptation, static, dynamic, evolution>>

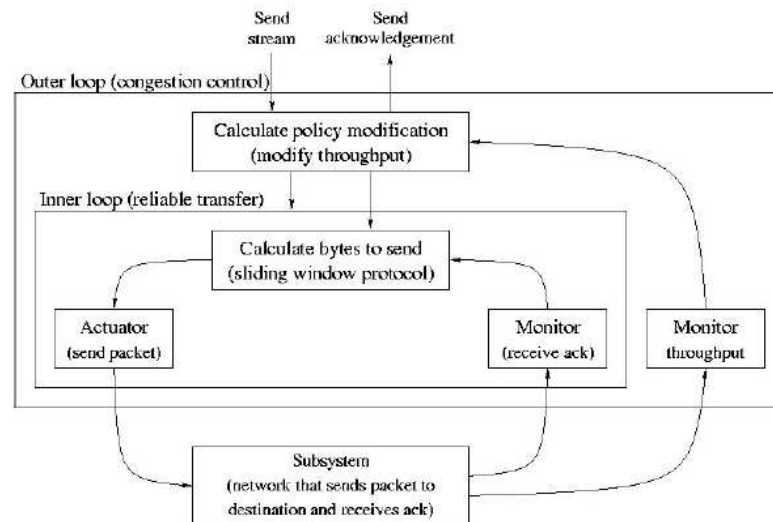
Self-management with interacting, hierarchical feedback loops

The selfman.org project, headed by Peter van Roy tries to replace manual management with the concept of self-regulation by hierarchically organized feedback loops.



<<feedback, stygmergy, management, open close, math>>.

The following diagram shows a real example of interacting feedback loops in the TCP protocol:



TCP feedback loops, after [vanRoy]

The research group uses structured overlay networks as an example of self-regulating/healing architecture and built self-management algorithms on top of the SON platforms. Consistent lookup times, reliable merging of partitioned rings with eventual consistency, range queries and finally even distributed transactions on top of SONs have been developed.

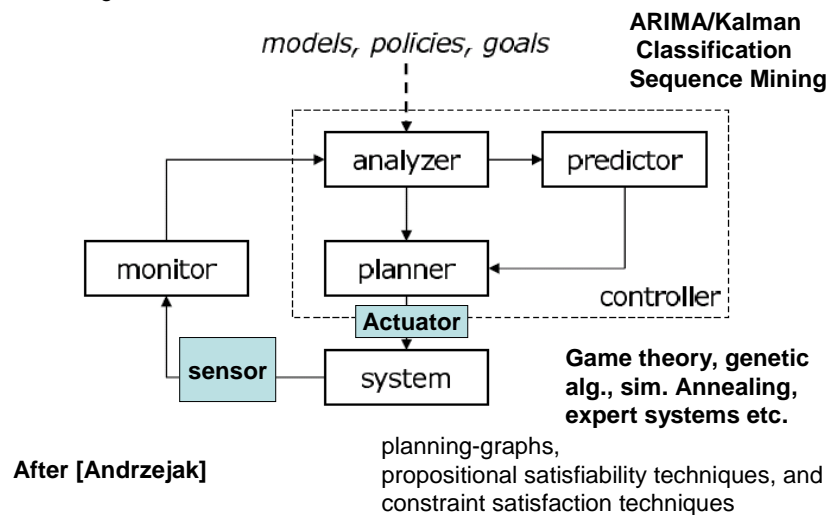
<<description of selfman.org sub-projects>>.

The engineering view behind is based on good software architecture principles: separation of interfaces from implementation and making architectural elements explicit. <<[Haridi] on Kompics.>>

The concept of complex systems as “systems of sub-systems connected via hierarchical feedback loops” is already a rather demanding view on ULS architectures given that there is no general systems theory yet and the complexity of intertwined feedback loops soon gets challenging and .

<<diagram of feedback components: planner, analytics, decisions, policies etc.>>

architecture of the managed system, its state, the allowed management actions, desired target system states and the optimization goals. Event triggered condition-action rules for management of networks and distributed systems



The programming problem certainly generalizes the autonomic computing problem, since in all by few exceptions the means to attain the self-managing functionality is software. Does it mean that the effort of formalization for self-management is similarly high as in the programming problem? This is not necessarily the case, since in the domain of self-management the required solutions are simpler (and more similar to each other) than in the field of programming, and so the benefits of domain-specific solutions can be exploited. A further step to reduce the effort of formalisation would be the usage of machine learning to automatically extract common rules and action chains from such descriptions [3]. Other tools are also possible, including graphical development environments (e.g. for workflow development), declarative specification of management actions used in conjunction with automatic planning, or domain-specific languages, which speed-up the solution programming.

Complete fault-tolerance is neither possible nor beneficial. One goal of autonomic computing is to hide faults from the user and to first try to handle such situations inside the system. Some faults cannot be detected, like whether an acknowledgement or calculation just takes a very long time, or was lost during data transmission. This is also known as halting problem [30] which states that no program can decide whether another program contains an endless loop or not. [Zuse..]

The paper raises some very interesting theoretical questions like the observation of one program through another (halting problem) and how it is applied at runtimes instead at code. But the methods mentioned for decision making, planning and even analytics and prediction are far from being engineering technologies. They are pure science and it will take a while until we will be able to use some in real systems.

Emergent Systems Engineering

But the group that met for OOPSLA06 to discuss ULS seems even more radical. Linda Northrop gave a presentation with the title “scale changes everything” and she and her group of researchers including Richard Gabriel and Doug Schmidt went out to investigate even larger systems. They rejected a core assumption made about the engineering of ULS: that they could be built consisting of billions of lines of reliably working code with incremental improvements to today's software technology: “Scale changes everything”.

Some core observations from this group:

<<list of features of ULS>>

What Is an Ultra-Large-Scale (ULS) System?



A ULS System has unprecedented scale in some of these dimensions:

- Lines of code
- Amount of data stored, accessed, manipulated, and refined
- Number of connections and interdependencies
- Number of hardware elements
- Number of computational elements
- Number of system purposes and user perception of these purposes
- Number of routine processes, interactions, and "emergent behaviors"
- Number of (overlapping) policy domains and enforceable mechanisms
- Number of people involved in some way

ULS systems will be interdependent webs of software-intensive systems, people, policies, cultures, and economics.

ULS systems are systems of systems at internet scale.



<<why scale changes everything>>

Scale Changes Everything



Characteristics of ULS systems arise because of their scale.

- Decentralization
- Inherently conflicting, unknowable, and diverse requirements
- Continuous evolution and deployment
- Heterogeneous, inconsistent, and changing elements
- Erosion of the people/system boundary
- Normal failures
- New paradigms for acquisition and policy



These characteristics may appear in today's systems and systems of systems, but in ULS systems they dominate.

These characteristics undermine the assumptions that underlie today's software engineering approaches.



The group also states a paradigm shift in the approach to build those systems. According to them neither classic engineering

- largely top-down and plan-driven development
- requirements/design/build cycle with standard well-defined processes
- centrally controlled implementation and deployment
- inherent validation and verification

nor the agile approach

- fast cycle/frequent delivery/test driven
- simple designs embracing future change and refactoring
- small teams and retrospective to enable team learning
- tacit knowledge

will work on the scale of ULS. [Northrop]

A quote from Greg Goth shows the scope of this research approach clearly:

Where a traditionally engineered software system might be like the detailed blueprints for a building, with everything laid out in advance, the creation of a ULS architecture is more like the evolution of the city itself: The form of a city is not defined in advance by specifying requirements; rather, a city emerges and changes over time through the loosely coordinated and regulated actions of many individuals. The factors that enable cities to be successful, then, include both extensive infrastructures not present in individual buildings as well as mechanisms that regulate local actions to maintain coherence without central control. (from page 5 of the ULS report) [Goth]

In the context of this thinking fundamental questions are raised:

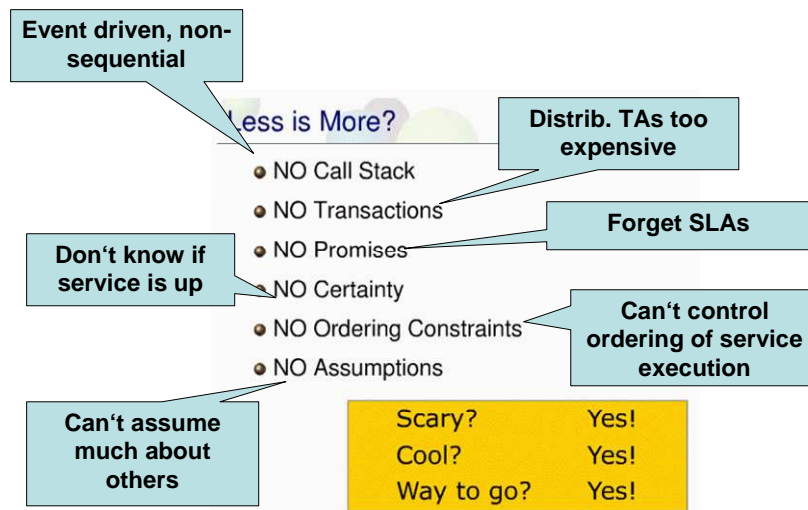
- are requirements really useful to build systems that span 25 and more years?
- Can we even use traditional “design” thinking to build things of such complexity and size?
- How do you bootstrap such systems (Kelly’s question on how to build a biotope)
- Do these systems emerge or are they built according to an engineering plan?
- Are the control loops hierarchical or network-like?
- How do we tie heterogeneous components into one system? Is there ONE system?
- Collusion is normal in those systems
- Traditional science thinking is towards small and elegant algorithms. Those systems are big and sometimes ugly conglomerates of smaller pieces.
- Second order cybernetics: the builder are part of the system

Both research approaches are certainly fascinating but I seriously doubt that they are in any way representative of the type of ULS we have been discussing in this book. Sites like Facebook or Flickr, Youtube or Google do go to great length to avoid some of the characteristics mentioned in the ULS of Northrop. The desing rules are actually trying to put the problem space into a shape that allows the application of engineering techniques to achieve reliable systems: create requests of same, standardized runtime behaviour. Control requests tightly with respect to frequency and side-effects. Partition data as much as possible. Avoid services which create unduly disruptions to your infrastructure and so on. And yes, despite a carefule use of monitoring and logging there are humans in the feedback loop that makes the existing systems scalable and reliable.

Scalability by Assumption Management

Perhaps this is anyway the right way to approach the problem: if it does not fit to our engineering abilities – bring it into a shape that will fit. Gregor Hohpe of Google, author of the famous book on application integration patterns, collected a number of design guidelines for highly scalable systems. The following is taken from his talk at Qcon London with the title “Hooking stuff together - programming the cloud”. [Hohpe]

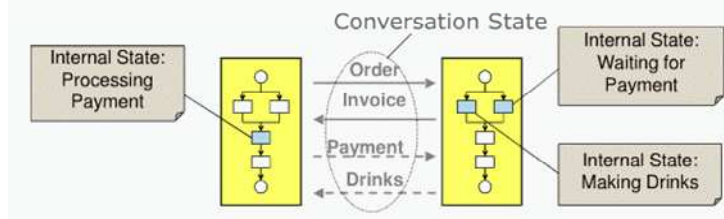
<<less is more>>



After Gregor Hohpe, Qcon Talk

Conversations

- Series of related messages between parties
- Not handled at lower layer
- Endpoints keep some conversation state
- Protocol design



Hohpe uses the example of Starbucks to demonstrate throughput optimizations: accept some loss to achieve maximum throughput. This sounds a bit like “eventual consistency” and we could call it “eventual profitability” perhaps. In ULS design it clearly emphasizes the need to re-think request types and functions in the overall system context. Who cares about a tossed coffee every once in a while if they can save on a very expensive transactional protocol? Overlapping processes are necessary to achieve high throughput.


Starbucks Does not Use 2-Phase Commit Either

- Start making coffee before customer pays
- Reduces latency
- What happens if...

Customer rejects drink	➔	Remake drink Retry
Coffee maker breaks	➔	Refund money Compensation
Customer cannot pay	➔	Discard beverage Write-off

<<what now>>

Now What?



- Live with uncertainty
- Simplicity is King
- Interaction
- Asynchrony
- New programming models
- Behold the Run-time
- Patterns Renaissance

To be able to live with very few assumptions we need to re-design our services and functions to e.g. make them order independent. Like in our discussion of Paxos we see again that commutativity of requests allows extreme optimizations, ideally full parallelization.

Living With Uncertainty

ACID (before)	ACID (today)
<ul style="list-style-type: none"> ● Atomic ● Consistent ● Isolated ● Durable 	<ul style="list-style-type: none"> ● Associative ● Commutative ● Idempotent ● Distributed
Predictive Accurate	Flexible Redundant

Order of execution does not matter!

Service is either a natural or our protocol needs to achieve it!

Almost every architect of a ULS mentions simplicity as a core design feature. We should probably attempt to define it a bit better: what do they really mean with simplicity? Here the statement about a clear failure mode being better than some complex failsafe architecture is interesting.

Simplicity is King

- Even simple things become complicated in a distributed environment
- If it looks complicated on paper it's likely to be impossible in practice
- If you can't understand it, other developers likely won't either
- A well understood failure scenario can be better than an incomprehensible and unproven "failsafe" system

Interaction has always been the magic behind distributed systems [Wegner]. Interaction is what makes those systems so very different from sequential algorithms. I believe that we need to favour living systems over code analysis in the future: a service is only a service if it is available. Code is very different to a running instance which we can interact with! (halting problem?)

Focus on Interaction

- In the OO world interaction is essentially free
- Powerful structural mechanisms: inheritance, composition, aggregation
- In the cloud, more focus shifts to interaction. Structural composition mechanisms are limited.

Hohpes emphasize on asynchronous interaction does not come as a surprise anymore: we have already seen that synchronous wait times are just too expensive to achieve high throughput.

Asynchrony

- Exchange through messages, not RPC
- Waiting for the results of an HTTP request is not a smart use of a 3 GHz processor
- Request and response message typically handled by different parts of your program, even if the same TCP connection
- Reduced assumptions about timing and state

New programming models like map-reduce are needed to process data in ULS. Hohpe's final point here is to emphasize the difference between some logical model and its execution within a distributed and parallel environment. This requires extensive monitoring and tracking.

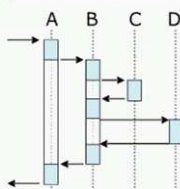
<<map reduce>>
<<runtime>>

Behold the Run-time

● Call Stack

```
void a() {
  b();
}

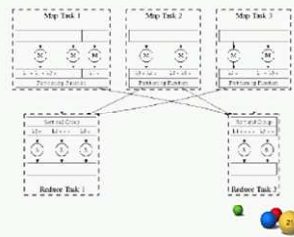
void b() {
  c();
  d();
}
```



● MapReduce

```
map(in_key, data)
  → list(key, value)

reduce(key, list(values))
  → list(out_data)
```



According to Hohpe Cloud-Computing dodges the bullets of the research groups mentioned earlier by restricting features and cutting down on assumptions and guarantees provided to clients. And Hohpe explicitly says that some application scenarios are probably unfit for running in the cloud. Giving up on transactions e.g. is certainly a hard thing to do for many applications. Here the work of the selfman.org group might come in handy by providing a transactional DHT and standard components which realize broadcast and other functions within an active component (actor) concept. Let's take a closer look at Cloud Computing concepts now.

Cloud Computing: The Web as a platform and API

How do we use those new platform APIs with their special storage technology?
Pricing and API use?

[zülch] paper

[Williamson] Alan Williamson, has the EC2 cloud become over subscribed?
http://alan.blog-city.com/has_amazon_ec2_become_over_subscribed.htm#
(cloud computing is not the most cost effective way of running an enterprise if the majority of them are running all the time). According to our monitoring, the newly spun up machines in the server farm, were under performing compared to the original ones. At first we thought these freaks-of-nature, just happened to beside a "noisy neighbor". A quick termination and a new spin up would usually, through the laws of randomness, have us in a quiet neighborhood where we could do what we needed. (noisy neighbours)
Amazon is forcing us to go to a higher priced instance just because they can't seem to cope with the volume of Small instances.
we discovered a new problem that has crept into Amazon's world: Internal Network Latency.
ping between two internal nodes within Amazon is around the 0.3ms level,
App architecture: shut off instance and hope that the new one will be better.

On Polling being bad in clouds: Polling is bad because AppEngine applications have a fixed free daily quota for consumed resources, when the number of feeds the service processed increased - the daily quota was exhausted before the end of the day because FF polls the service for each feed every 45 minutes. [Zuzak] Ivan Zuzak Realtime filtering and feed processing
<http://izuzak.wordpress.com/2010/01/11/real-time-feed-processing-and-filtering/>

[google] Entity Groups and Transactions
<http://code.google.com/appengine/docs/python/datastore/transactions.html>

[Hohpe]

Amazon S3 architecture:
<http://blogs.zdnet.com/storage/?p=416>
<<check pricing at 15 cent/gig/month>>

the Guide to Cloud Computing from Sun.

http://www.sun.com/offers/docs/cloud_computing_primer.pdf

(mentions capital expenditure advantages as well, defines saas, paas, iaas, open storage concepts in new sun fire 4500,

[http://www.ibm.com/developerworks/web/library/wa-cloudflavor/index.html?](http://www.ibm.com/developerworks/web/library/wa-cloudflavor/index.html?ca=dgr-jw22CC-Labyrinth&S_TACT=105AGX59&S_CMP=grsitejw22)

[ca=dgr-jw22CC-Labyrinth&S_TACT=105AGX59&S_CMP=grsitejw22](http://www.ibm.com/developerworks/web/library/wa-cloudflavor/index.html?ca=dgr-jw22CC-Labyrinth&S_TACT=105AGX59&S_CMP=grsitejw22)

provisioning, deployment, architecture

http://www.theserverside.com/news/thread.tss?thread_id=54238

http://www.devwebsphere.com/devwebsphere/websphere_extreme_scale/

```
*      Storage made easy with S3
<http://www.ibm.com/vrm/newsletter\_10731\_5146\_110766\_email\_DYN\_2IN/wqxcg
83948394> (Java technology)
*      Cloud computing on AIX and System p
<http://www.ibm.com/vrm/newsletter\_10731\_5146\_110766\_email\_DYN\_3IN/wqxcg
83948394> (AIX and UNIX)
*      Is there value in cloud computing?
<http://www.ibm.com/vrm/newsletter\_10731\_5146\_110766\_email\_DYN\_4IN/wqxcg
83948394> (Architecture)
*      Cultured Perl: Perl and the Amazon cloud, Part 2
<http://www.ibm.com/vrm/newsletter\_10731\_5146\_110766\_email\_DYN\_5IN/wqxcg
83948394> (Linux)
*      Realities of open source cloud computing: Not all clouds are
equal
<http://www.ibm.com/vrm/newsletter\_10731\_5146\_110766\_email\_DYN\_6IN/wqxcg
83948394> (Open source)
*      The role of Software as a Service in cloud computing
<http://www.ibm.com/vrm/newsletter\_10731\_5146\_110766\_email\_DYN\_7IN/wqxcg
83948394> (Web development)
```

Mark Andreesen, Internet Platforms on his blog.

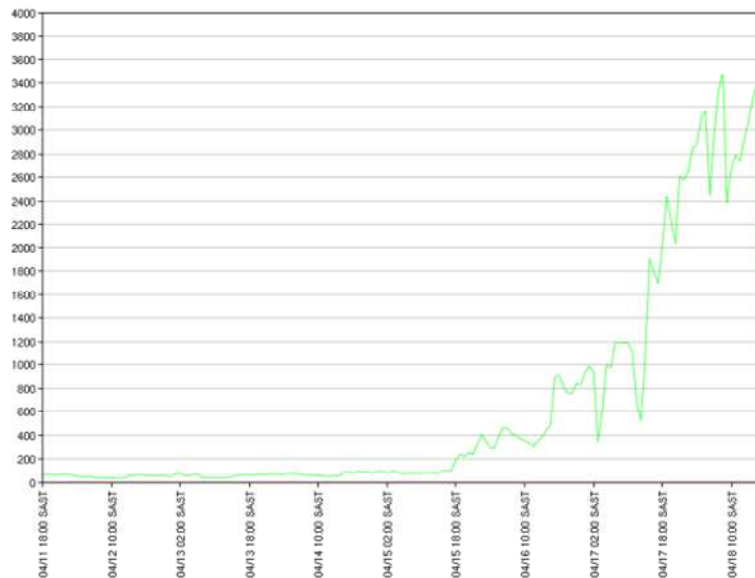
[Shalom] Nati Shalom, Latency is everywhere.

http://natishalom.typepad.com/nati_shaloms_blog/2009/03/its-time-for-auto-scaling-avoid-peak-load-provisioning.html

- middleware virtualization
- cloud APIs and datastores
- best practices for cloud apps

Dr. Strüker also from the University of Freiburg talked about communicating things, calculating clouds and virtual companies. He also used the famous Animoto example.

Animoto scalability on EC2, from Brandon Watsons blog



Animoto faced extreme scalability problems and solved them by using EC2. Brian Watson questioned the rationale behind adding 3000 machines practically over night:

Amazon loves to hold out Animoto as an example of the greatness of their platform. They love to show the chart on the left here. In a couple of days, usage of the Animoto service exploded. There's an accounting of the event in a [blog post by the AWS team](#). If you do the quick math, they were supporting approximately 74 users per machine instance, and their user/machine image density was on the decline with increased user accounts. The story they like to tell from this chart is "wow, we were able to spin up 3000 machines over night. It's amazing!" What I see is more along the lines of "holy crap, what is your code doing that you need that many instances for that many users?" I don't mean to impugn Animoto here, but I don't want the point to be lost: the profitability of your project could disappear overnight on account of code behaving badly.

[Watson]

I found especially interesting what Strüker said about cloud computing. He gave some interesting numbers on the size and numbers of datacenters built by Google, Amazon and now also Microsoft. According to him Microsoft is adding 35000 machines per month. Google uses 2 Mio. machines in 36 datacenters worldwide. But the way this compute power is used surprised me even more. The first example was the conversion of 11 Mio. New Your Times articles to pdf. Instead of building up an internal infrastructure of hundreds of machines somebody decided to rent compute power from the Amazon Elastic Compute Cloud EC2 and ended up with the documents converted in less than a day for only 240 dollar. Then he mentioned the case of animato, a company creating movies from pictures. Interesting about this case is that animato used the EC2 cloud to prepare for incredible growht. I don't remember the exact numbers but the growth of requests was so big that without an existing, scalable infrastructure, the users of animato would have experienced major breakdowns. There would have been no way to increase compute power quickly enough to comply with this growth rate. But the last cases were even more astonishing. They were about businesses using the cloud to do all kinds of processing. This includes highly confidential stuff like

customer relationship handling which touches the absolute core of businesses. I was surprised that companies would really do this. In large corporations this type of processing is done internally on IBM Mainframes. The whole development could spell trouble for the traditional IBM Mainframe strategy as a new presentation at infoq.com already spells out: Abel Avram asks: [Are IBM's Cloud Computing Consulting Services Generating a Conflict of Interests?](#)

Qcon: Host: **Gregor Hohpe**

The Web has become the application delivery platform of choice. After an initial focus on the presentation layer, business services and middleware components are moving to the web as well. Supported by core services like Amazon's EC2 compute cloud and S3 storage services, and using application services like Google's GData APIs these applications don't just run over the web, they run on the web.

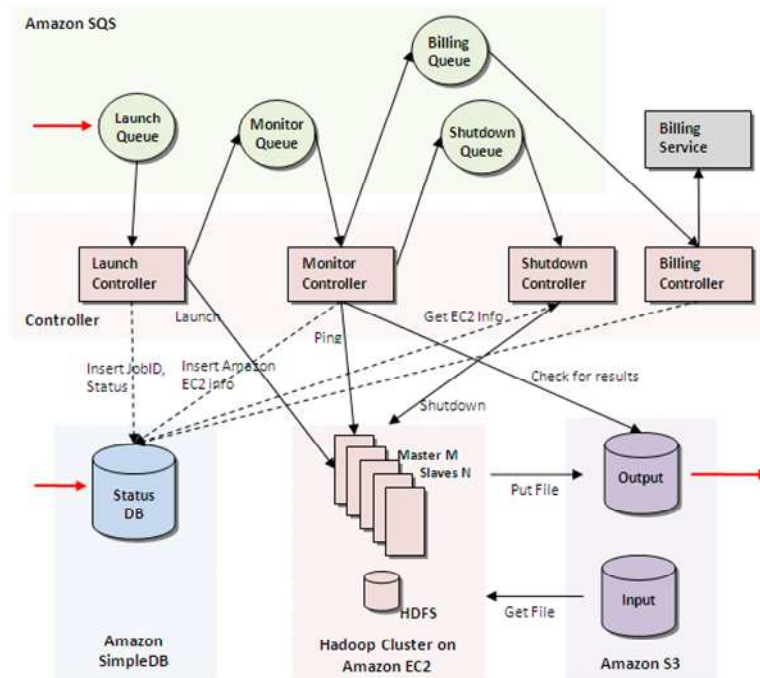
What does this mean for application developers? How do you deploy an application to the Web? Will applications be composed by dragging web-based components together? Do we still have to fiddle around with JavaScript and brittle APIs? This track invites experts who have been living the cloud to share their experiences and give hand-on advice.

Moving to the Grid will affect your application architecture considerably, according to Joseph Ottinger. He explains core J2EE architectural features like the assumption of request/response patterns and what is needed to move toward a dynamic grid infrastructure. [Ottinger]

Canonical Cloud Architecture

The canonical cloud architecture that has evolved revolves around dynamically scalable CPUs consuming asynchronous, persistently queued events. We talked about this idea already in [Flickr - Do the Essential Work Up-front and Queue the Rest](#). The cloud is just another way of implementing the same idea. [Hoff], Canonical Cloud Architecture

What is this about asynchronous, persistently queued events and scalability via CPUs? Sounds similar to Darkstar architecture for MMOGs.



(from [Hoff], canonical cloud arc.)

Cloud-based Storage

[Glover]

-REST based API to S3, 15 cent/gig/month plus transfer costs, flexible access token generation (e.g. time-limited access to storage parts), global name space for spaces. Twitter stores user images on S3.

<<REST API example for store and update >>

Latest from Architecture

<http://www.infoq.com/architecture/>:

Presentation: Google Data API (G-Data)

Frank Mantek discusses the Google Data API (GData) including decisions to use REST rather than SOAP technology, how the API is used, numerous examples of how GData has been used by clients, and future plans for evolving the API. A discussion of how GData facilitates Cloud Computing concludes the presentation.

(Presentations)

Cloud-based Memory (In-Memory-Data-Grid)

We are on the edge of two potent technological changes: Clouds and Memory Based Architectures. This evolution will rip open a chasm where new players can enter and prosper. Google is the master of disk. You can't beat them at a game they perfected. Disk based databases like SimpleDB and [BigTable](#) are complicated beasts, typical last gasp products of any aging technology before a change. The next era is the age of Memory and Cloud which will allow for new players to succeed. The tipping point is soon. [Hoff], Cloud-based Memory

Will ram become disk and disk become tape? Does this really scale? What is the role of MVCC?

Time in Virtualized Environments

[DynaTrace] SLA monitoring
[VMWare] Time Keeping in VMWare Virtual Machines
[Harzog]
[Dynatrace] Cloud Service Monitoring for Gigaspaces

The Media Grid

Make abstract:

“The Media Grid is a [digital media](#) network infrastructure and software-development platform based on new and emerging distributed computational grid technology. The Media Grid (<http://www.MediaGrid.org/>) is designed as an on-demand public computing utility that software programs and web sites can access for digital content delivery (graphics, video, animations, movies, music, games, and so forth), storage, and media processing services (such as data visualization and simulation, medical image sharpening and enhancement, motion picture scene rendering, special effects, media transformations and compositing, and other digital media manipulation capabilities). As an open platform that provides digital media delivery, storage, and processing services, the Media Grid's foundation rests on Internet, web, and grid standards. By combining relevant standards from these fields with new and unique capabilities, the Media Grid provides a novel software-development platform designed specifically for networked applications that produce and consume large quantities of digital media. As an open and extensible platform, the Media Grid enables a wide range of applications not possible with the traditional Internet alone, including: on-demand [digital](#) cinema and interactive movies; distributed film and movie rendering; truly immersive multiplayer games and virtual reality; real-time visualization of complex data (weather, medical, engineering, and so forth); telepresence and telemedicine (remote surgery, medical imaging, drug design, and the like); telecommunications (such as video conferencing, voice calls, video phones, and shared collaborative environments); vehicle and aircraft design and simulation; computational science applications (computational biology, chemistry, physics, astronomy, mathematics, and so forth); biometric security such as real-time face, voice, and body recognition; and similar high-performance media applications”
Dr. Dobb's Journal, November 2005

The Media Grid

A public utility for digital media

By Aaron E. Walsh

- interaction
 - ad-hoc
 - mobile
 - swarming
 - combination of p2p and GRID technology
- <<swarming effect diagram>>

Peer-to-Peer Distribution of Content (bbc)

Video on Demand use case, problems with bandwidth.

Solution: p2p streaming

www.selfman.org !!!

Meanwhile, a portion of the BBC's vast archive of audio and video material may also be accessed via MyBBCPlayer. The software may also let viewers to buy items via the BBC Web site, which would be a big leap from the current public service features of the BBC's online sites.

The announcement was made in August at the U.K.'s broadcasting headliner event, the Edinburgh Television Festival, by the recently appointed head of the BBC, Mark Thompson ("director-general" in BBC-speak). "We believe that on-demand changes the terms of the debate, indeed that it will change what we mean by the word 'broadcasting'," he said. "Every creative leader in the BBC is wrestling with the question of what the new technologies and audience behaviors mean for them and their service," he went on. "[MyBBCPlayer] should make it easier for users to find the content they want whenever and wherever they want it."

It seems straightforward enough: a major content provider has made a smart move with technology anticipating the growing surge of interest in on-demand TV. But that interpretation misses some of both the political nuances of the BBC's intentions and its possibly explosive impact on the programming market in not just the U.K., but globally as well.

The trial he's referring to is some 5,000 carefully selected consumers who will be offered a version of IMP (Interactive Media Player), a prelude to MyBBCPlayer delivered to the PC that is set to evolve into the full commercial release if plans come to fruition.

The underlying technology platform on which MyBBCPlayer and IMP are built is provided by U.S.-based firm Kontiki. The company's peer-to-peer solution is increasingly being used as weapon of choice for delivering large media files over IP, according to Kontiki CEO Todd Johnson. "We are unique in using a legal way to use peer-to-peer—buttressed by rights protection—to make mass consumption of these kinds of properties a reality," he claims.

The advantage of P2P for this application is that it avoids the need to pump out huge files centrally; instead, a network of collaborating computers team up to share the workload with the content neatly splitting into many component pieces, all reassembled at the user's PC after locating the nearest and easiest nodes from which to retrieve the next needed element. This way quality of service isn't constrained at any point during the delivery chain. "At peak periods this means successful delivery even with relatively modest amounts of backup infrastructure," Johnson says—acknowledging that this is exactly how "pirate" services like Gnutella and Grokster have been moving content for quite some time.

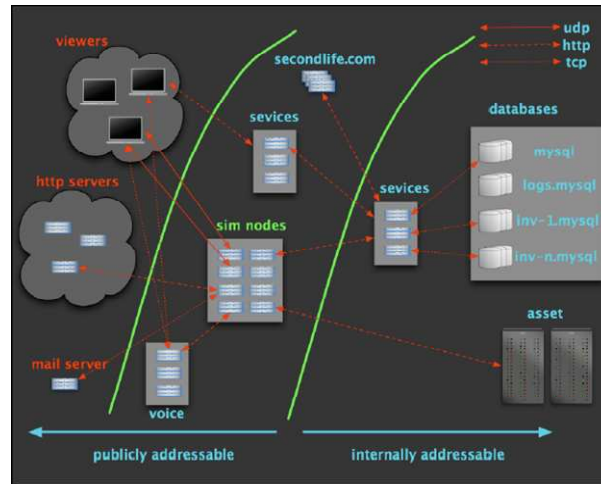
„is bittorrent deployed by a huge broadcaster“

<<diagram with myBBCPlayer, swarming infrastructure and BBC archive plus website for billing>>

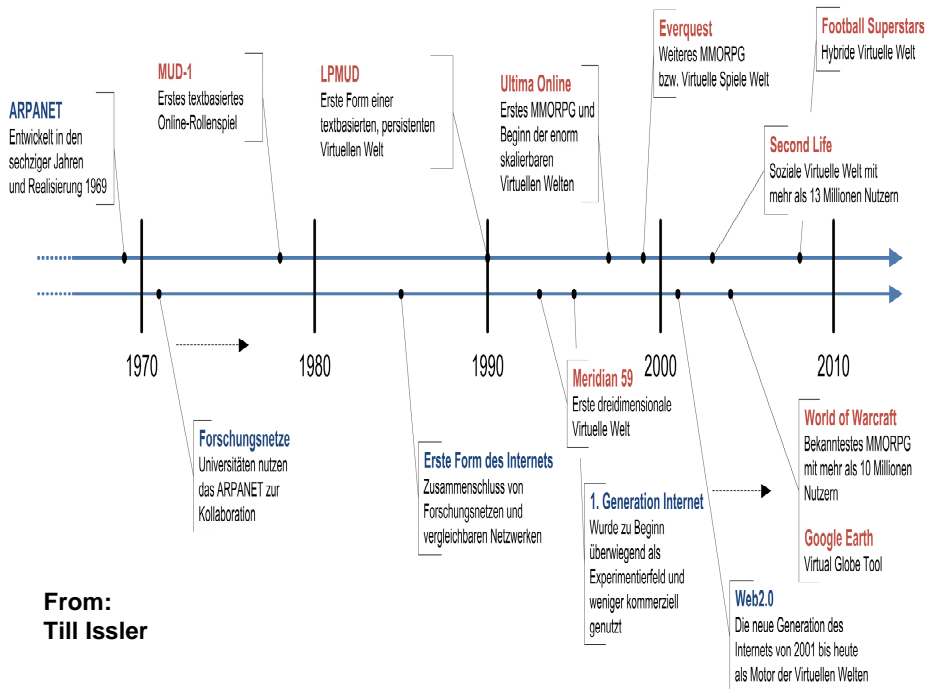
<<diagram from kontiki architecture>>

Virtual Worlds (Secondlife, DarkstartWonderland) – Architecture for Scalability

(wikipedia article on new architecture of secondlife)



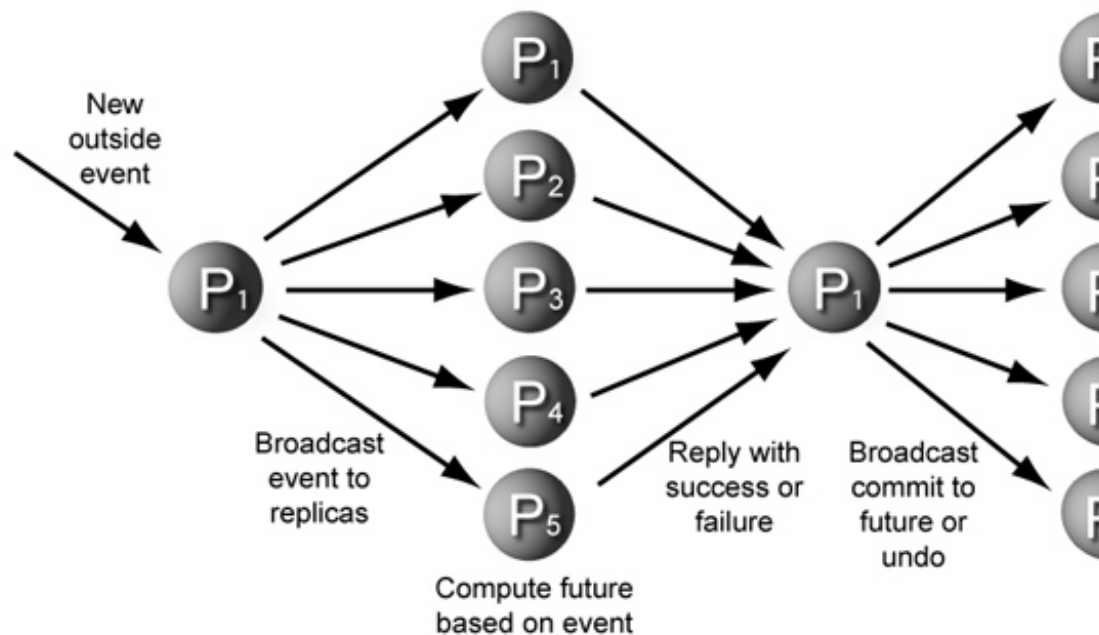
Jim Waldo of Sun - famous for his critique of transparency in distributed systems wrote a paper on the new game platform Darkstart. But in this paper he turns around and claims that for his new project it was necessary to build transparent distributed features because of the special environment of 3D games. He claims that game programmers are unable to deal e.g. with concurrency explicitly. Darkstar splits requests into short and limited tasks which can be transparently distributed to different cores or machines. To achieve consistency all data store access is transacted with an attached event system. We will see how this scales in the long term.



Immersive multi-media based collaboration (croquet)

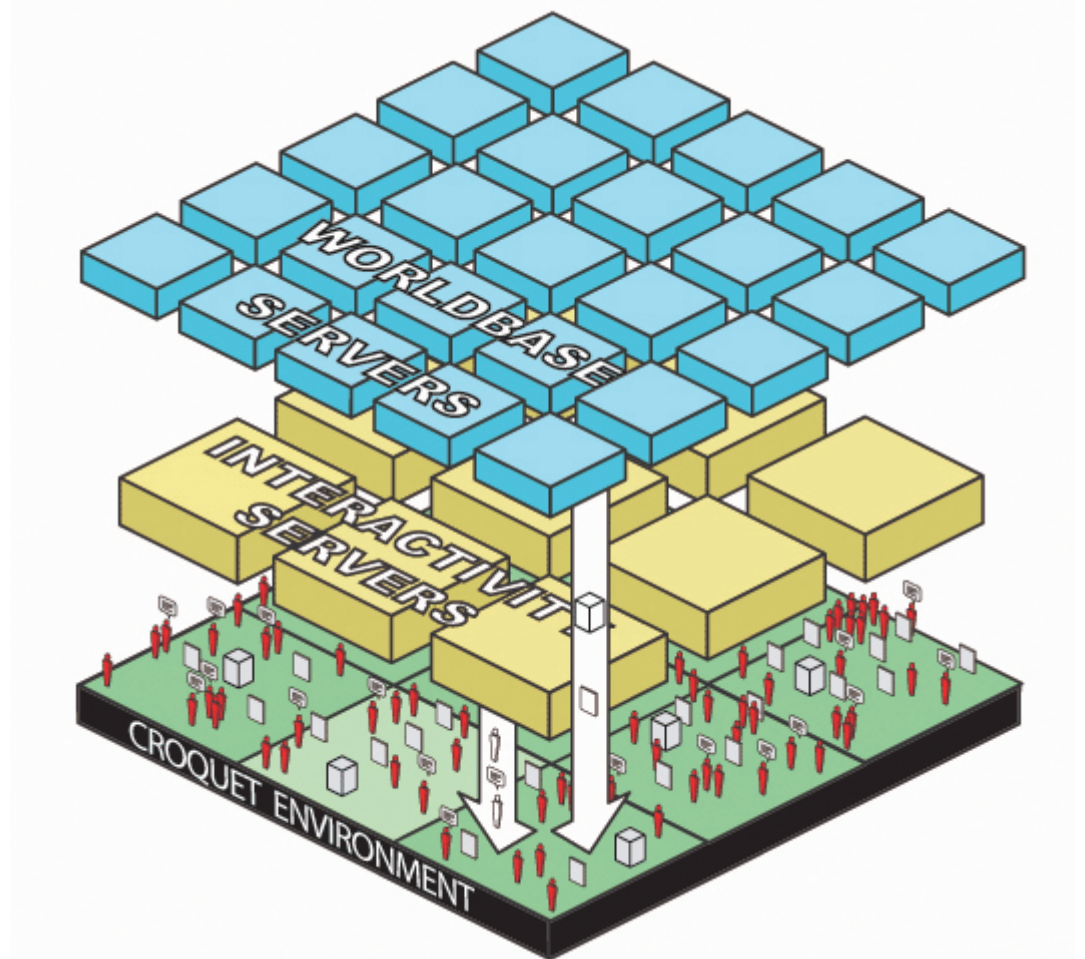
- The effects of interaction
- replication instead of proxies
- separating requests from local processing time
- specialization through hierarchies of servers

Replicated, independent objects:



hierarchies of servers

<<vat concept with router diagram>>



Part VII: Practice

A scalable bootstrap kernel

<<build a small kernel for a scalable site that allows growth. Put the scalability mechanisms in place early on. See how this works financially. How many collocated servers? Compare with cloud computing costs, open source cloud? Core services needed?>>

Exercises and Ideas

Data Storage

- take a look at a social graph model and speculate about its scalability
- build some storage grid components based on open standards:

Ideas with Grid Storage for HDTV

- Build micro-grid with Lustre (standard FS)
- Calculate capacity curve (Gunther)
- Build Grid-Gateway to support posix apps and measure throughput
- Investigate existing Video apps for interfaces to other storage types
- Build scheduler (based on hadoop) for transcoding and indexing
- Build administration tools for soft backup and restore, disaster recovery etc.
- Use of ZFS for NAS/SAN combo.

Modeling and Simulation

- program a simulation of one-queue servers with one or two service stations. The Palladio simulation environment from KIT Karlsruhe seems to be a good candidate for this.

Performance Measurements and Profiling

Distributed Algorithms

- use a group communication software to synchronize one variable across servers. Grow the number of servers and watch for performance problems. How far does multicast go? What is the effect of REAL high speed networks on reliability and liveness?

Measurements

- use of a mediawiki installation for
 - load-tests
 - performance tests

- profiling (cache, DB, PHP)
- monitoring and alarming

Compare the results with those from “modelling and simulation”. This is currently done in my course on “system engineering and management”.

According to GOMEZ we will get a restricted test account for their global test environment which would let us test the application externally.

Going Social

- Take a web-application and extend it with social features. How should a social data model look like? (Open social, hierarchical etc.)
- Use Semsix as a testbed (currently a thesis which I am mentoring)

Failure Statistics

Collect real-world failure statistics on e.g. network partitionings, disk failures. Consider dependencies between distributed algorithms and specific hardware architectures (time in V

Part VIII: Resources

Literature:

[Narayanan] Arvind Narayanan und Vitaly Shmatikov, "Robust De-anonymization of Large Sparse Datasets",

http://www.cs.utexas.edu/~shmat/shmat_oak08netflix.pdf

- <http://blog.stackoverflow.com/category/podcasts/> (bzw: <http://itc.conversationsnetwork.org/series/stackoverflow.html>)

Der Stack Overflow Podcast ist eine wöchentliche Serie in der Joel Spolsky und Jeff Atwood über Software-Architektur und Themen rund um Software-Technologie reden.

Interessant im Zusammenhang mit der Ultra-Large-Scale Sites Veranstaltung sind insbesondere die Berichte über die Architektur der stackoverflow.com Community.

- Web Services Architecture book
- Ed Felten..
- Globus.org
- Tecmath AG
- Stefan werner thesis
- Bbc article
- Bernard Traversat et.al., Project JXTA 2.0 Super-Peer Virtual network.

Describes the changes to JXTA 2.0 which introduced “super-peers” for performance reasons – though they are dynamic and every peer can become one. Good overview on JXTA.

- Ken Birman et.al, Kelips: Building an Efficient and Stable P2P DHT Through increased Memory and Background Overhead. I read it simply because of Birman. Shows the cost if one wants to make p2p predictable.
- Petar Maymounkov et.al. Kademia: A peer-to-peer Information System based on the XOR metric. <http://kademlia.scs.cs.nyu.edu/> An improvement on DHT technology through better organization of the node space. Interestingly, edonkey nets want to use it in the future.
- Atul Adya et.al (Micr.Res.), Farsite: Federated, Available and Reliable Storage for an Incompletely Trusted Environment. very good article with security etc. in a distributed p2p storage system. How to enable caching of encrypted content etc.
- Emit Sit, Robert Morris, Security Considerations for Peer-to-Peer Distributed Hash Tables. A must read. Goes through all possible attack scenarios against p2p systems. Good classification of attacks (routing, storage, general). Suggests using verifiable system invariants to ensure security.
- M.Frans Kaashoek, Distributed Hash Tables: simplifying building robust Internet-scale applications (<http://www.project-iris.net>) . Very good slide-set on DHT design. You need to understand DHT if you want to understand p2p.
- A Modest Proposal: Gnutella and the Tragedy of the Commons, Ian Kaplan. Good article on several p2p topics, including the problem of the common goods (abuse) http://www.bearcave.com/misl/misl_tech/gnutella.html
- Clay Shirky, File-sharing goes social. Bad news for the RIAA because Shirky shows that prosecution will only result in cryptographically secured darknets. There are many more people than songs which makes sure that you will mostly get the songs you want in your darknet. Also: do your friends share your

music taste? quite likely. http://www.shirky.com/writings/file-sharing_social.html
Don't forget to subscribe to his newsletter – you won't find better stuff on networks, social things and the latest in p2p.

- Project JXTA: Java Programmer's Guide. First 20 pages are also a good technical overview on p2p issues.
- www.cachelogic.com. Note the rising „serious“ use of bittorrent by software and media companies.
- Olaf Zimmermann et.al., Elements of Service-oriented Analysis and Design, 6/2004, www.ibm.com/developerworks
- Ali Arsanjani, Service-oriented modeling and architecture, 11/2004 www.ibm.com/developerworks
- Guido Lares et.al., SOA auf dem Prüfstand, ObjectSpektrum 01/2005. Covers the new benchmark by The Middleware Company for SOA implementations
- <http://www.akamai.com/en/html/services/edgesuite.html> for a description of the edge caching architecture and service
- Gamestar Magazine 08/2005
- Dr. Dobb's Journal, November 2005. The Media Grid. A public utility for digital media By Aaron E. Walsh
- BBC turns to P2P for VOD, <http://www.streamingmedia.com/article.asp?id=9205>
-

- Peer-to-Peer, Harnessing the Power of Disruptive Technologies, Edited by Andy Oram, 2001, O'Reilly. Contains good articles on different p2p applications (freenet, Mixmaster Remailers, Gnutella, Publius, Free Haven etc). And also from Clay Shirkey: Listening to Napster. Recommended.
- Peer-to-Peer, Building Secure, Scalable and Manageable Networks, Dana Moore and John Hebler. Definitely lighter stuff than Andy Oram's collection. Missing depth. Covers a lot of p2p applications but few base technology.
- www.openp2p.org, the portal to p2p technology. You can find excellent articles e.g. by Nelson Minar on Distributed Systems Topologies there.
- Project JXTA: Java Programmer's Guide. First 20 pages are also a good technical overview on p2p issues.
- Upcoming: 2001 P2P Networking Overview, The emergent p2p platform of presence, identity and edge resources. Clay Shirkey et.al. I've only read the preview chapter but Shirkey is definitely worth reading.
- It's not what you know, it's who you know: work in the information age, B.A.Nardi et.al., http://www.firstmonday.org/issues/issue5_5/nardi/index.html
- Freeriding on gnutella, E.Adar et.al., http://www.firstmonday.org/issues/issue5_10/adar/index.html, claims that over 70% of all gnutella users do not share at all and that most shared resources come from only 1% of peers.
- Why gnutella can't possibly scale, no really, by Jordan Ritter. <http://www.monkey.org/~dugsong/mirror/gnutella.html>. An empirical study on scalability in gnutella.

- A Modest Proposal: Gnutella and the Tragedy of the Commons, Ian Kaplan. Good article on several p2p topics, including the problem of the common goods (abuse) http://www.bearcave.com/misl/misl_tech/gnutella.html

- Clay Shirky, File-sharing goes social. Bad news for the RIAA because Shirky shows that prosecution will only result in cryptographically secured darknets. There are many more people than songs which makes sure that you will mostly get the songs you want in your darknet. Also: do your friends share your music taste? quite likely. http://www.shirky.com/writings/file-sharing_social.html Don't forget to subscribe to his newsletter – you won't find better stuff on networks, social things and the latest in p2p.
- Bram Cohen, Incentives Build Robustness in Bit Torrent. Explains why the bit torrent protocol is what it is. Bit torrent tries to achieve „pareto efficiency“ between partners. Again a beautiful example how social and economic ideas mix with technical possibilities in p2p protocol design: why is it good to download the rarest fragments first? etc.
- Bob Loblaw et.al, Building Content-Based Publish/Subscribe Systems with Distributed Hash Tables. Nice paper on DHT design with a content based focus (not topic based as usually done). Experimental, good resource section.
- M.Frans Kaashoek, Distributed Hash Tables: simplifying building robust Internet-scale applications (<http://www.project-iris.net>) . Very good slide-set on DHT design. You need to understand DHT if you want to understand p2p.
- Ion Stoica (CD 268), Peer-to-Peer Networks and Distributed Hash Tables. Another very detailed and good slide set on DHT designs. (CAN/Chord/freenet/gnutella etc.). Very good.
- Emit Sit, Robert Morris, Security Considerations for Peer-to-Peer Distributed Hash Tables. A must read. Goes through all possible attack scenarios against p2p systems. Good classification of attacks (routing, storage, general). Suggests using verifiable system invariants to ensure security.
- Moni Naor, Udi Wieder, A simple fault tolerant Distributed Hash Table. Several models of faulty node behavior are investigated.
- Distributed Hash Tables: Architecture and Implementation. A usenix paper which discusses transactional capabilities of a DHT based DDS.
- www.emule-project.net/faq/ports.htm shows the ports in use by emule-related protocols. Shows that several emule-users behind a NAT/router/firewall need individual redirects established at the firewall to allow incoming connections to be redirected to a specific client.
- OCB Maurice, Some thoughts about the edonkey network. the author explains how lookup is done in edonkey nets and what hurts the network. Interesting details on message formats and sizes.
- John R. Douceur et.al (Microsoft Research), A secure Directory Service based on Exclusive Encryption. One of many articles from Microsoft research which try to use P2p technologies as a substitute for the typical server infrastructure in companies.
- John Douceur, The Sybil Attack, Can you detect that somebody is using multiple identities in a p2p network. John claims you can't without a logical central authority.
- Atul Adya et.al (Micr.Res.), Farsite: Federated, Available and Reliable Storage for an Incompletely Trusted Environment. very good article with security etc. in a distributed p2p storage system. How to enable caching of encrypted content etc.

- W.J. Bolosky et.al, Feasibility of a Serverless Distributed Filesystem deployed on an Existing Set of PCs. Belongs to the topics above. Interesting crypto tech (convergent encryption) which allows detection of identical but encrypted files.
 - Ashwin R. Bharambe et.al, Mercury: A scalable Publish-Subscribe System for Internet Games. Very interesting approach but does not scale yet. Good resource list at end.
 - Matthew Harren et.al, Complex Queries in DHT-based Peer-to-Peer Networks. How do you create a complex query if hashing means “exact match”? E.g. by splitting the meta-data in many separate hash values. Interesting ideas for search in p2p.
 - Josh Cates, Robust and Efficient Data management fo a Distributed hash table, MIT master thesis.
 - Peter Druschel at.al, PAST: a large-scale, persistent peer-to-peer storage utility. Excellent discussion of system design issues in p2p.
 - Bernard Traversat et.al, Project JXTA: A loosely-consistent DHT Rendezvous walker. Read this to get the idea of DHT in an unreliable environment. Very good.
 - John Noll, Walt Scacchi, Repository Support for the Virtual Software Enterprise. Use of DHT for software engineering support in distributed teams/projects.
-
- Petar Maymounkov et.al. Kademlia: A peer-to-peer Information System based on the XOR metric. <http://kademlia.scs.cs.nyu.edu/> An improvement on DHT technology through better organization of the node space. Interestingly, edonkey nets want to use it in the future.
 - Zhiyong Xu et.al. HIERAS: A DHT based hierarchical P2P routing algorithm. Shows that one can win through a layered routing approach which e.g. allows optimization through proximity.
 - Todd Sundsted, The practice of peer-to-peer computing. A series of entry level articles from www.ibm.com/developerworks (e.g. trust and security in p2p)
 - <http://konspire.sourceforge.net> A comparison with bittorrent technology. Interesting. What limits the download in a p2p filesharing app? Also get the overview paper on konspire from that site.
 - NS2 – the network simulator. A discrete event simulator targeted at network research. Use it to simulate your p2p networks. (from <http://www.isi.edu/nsnam>)
 - Zhiyong Xu et.al, Reducing Maintenance Overhead in DHT based peer-to-peer algorithms.
 - Bernard Traversat et.al., Project JXTA 2.0 Super-Peer Virtual network. Describes the changes to JXTA 2.0 which introduced “super-peers” for performance reasons – though they are dynamic and every peer can become one. Good overview on JXTA.
 - Ken Birman et.al, Kelips: Building an Efficient and Stable P2P DHT Through increased Memory and Background Overhead. I read it simply because of Birman. Shows the cost if one wants to make p2p predictable.
 - Krishna Gummadi et.al, The impact of DHT Routing Geometry on Resilience and Proximity. Compares several DHT designs. Quite good. Findings are that neighbour flexibility is more important than route selection flexibility. Proximity selection techniques perform well.

- Mark Spencer, Distributed Universal Number Discovery (DUNDi) and the General Peering Agreement, www.dundi.com/dundi.pdf
- http://www.theregister.com/2004/12/18/bittorrent_measurements_analysis/print.html An analysis of the bittorrent sharing system.
- Ian G.Gosling, eDonkey/ed2k: Study of a young file sharing protocol. Covers security aspects.
- Heckmann, Schmitt, Steinmetz, Peer-to-Peer Tauschbörsen, eine Protokollübersicht. www.kom.e-technik.tu-darmstadt.de
- A Distributed Architecture for Massively Multiplayer Online Games, Chris Gauthier Dickey Daniel Zappala Virginia Lo

Larry Lessig, How creativity is strangled by the law,
http://www.ted.com/index.php/talks/larry_lessig_says_the_law_is_strangling_creativity.html

[Issl] T.Issler, Potentiale und Einsatz von Virtuellen Welten entlang der Wertschöpfungskette der Automobilindustrie, Diplomarbeit 2008, HDM/IBM

[Rodr]
 Alex Rodriguez , RESTful Web services: The basics
 IBM , 06 Nov 2008 http://www.ibm.com/developerworks/webservices/library/ws-restful/index.html?S_TACT=105AGX54&S_CMP=B1113&ca=dnw-945

[Holl] P.Holland, Life beyond Distributed Transactions: an Apostates Opinion

[Rodr] A. Rodriguez, RESTful Web Services: The Basics,
<http://www.ibm.com/developerworks/webservices/library/ws-restful/index.html>

[Seeg] M.Seeger, Anonymity in P2P Networks, thesis HDM 2008,

[Pink] D.H.Pink, A Whole New Mind,

[Mühl], Gero Mühl et.al., Distributed Event-Based Systems

[Luck] D.Luckham, Complex Event Processing

[Dean] J. Dean and S. Ghemawat of Google Inc,
 MapReduce: Simplified Data Processing on Large Clusters
<http://labs.google.com/papers/mapreduce.html>

[Ghemawat] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, The Google File System, Google <http://labs.google.com/papers/gfs-sosp2003.pdf>

"Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters"
 — Paper von Hung-Chih Yang, Ali Dasdan, Ruey-Lung Hsiao und D. Stott Parker, [Yahoo](http://www.yahoo.com) und [UCLA](http://www.ucla.edu), veröffentlicht in Proc. of ACM SIGMOD, pp. 1029--1040, 2007. (Dieses Paper zeigt, wie man MapReduce auf relationale Datenverarbeitung ausweitet)

[Saito] Yasushi Saito, Marc Shapiro, Optimistic Replication,
<http://www.ysaito.com/survey.pdf>

[Chandra] Tushar Chandra, Robert Griesemer, Joshua Redstone, Paxos Made Live - An Engineering Perspective <http://www.chandrakin.com/paper2.pdf>

[Tomp] C.Tompson, Build it. Share it. Profit. Can Open Source Hardware Work? Wired Magazine, 16.11

[Bung] S.Bungart, IBM. Talk at HDM on the future of IT.

[Edge] Edge Architecture Specification, <http://www.w3.org/TR/edge-arch>

[Mulz] M.Mulzer, Increasing Web Site Performance: Advanced Infrastructure and Caching Concepts
http://www.dell.com/content/topics/global.aspx/power/en/ps1q02_mulzer?c=us&cs=555&l=en&s=biz

[Heise119014] Heise news, Zürcher Forscher erstellen Modell für Erfolg von Internet-Videos

[Crane] Riley Crane and Didier Sornette, Robust dynamic classes revealed by studying the response function of a social system, Proceedings of the National Academy of Sciences, Vol. 105, No. 41. (October 2008), pp. 15649-15653.

[Game] You have gained a level, Geschichte der MMOGs, Gamestar Sonderheft 08/2005

[enisa] European Network and Information Security Agency, Virtual Worlds, Real Money – Security and Privacy in Massively-Multiplayer Online Games and Social and Corporate Virtual Worlds
http://www.enisa.europa.eu/doc/pdf/deliverables/enisa_pp_security_privacy_virtualworlds.pdf

[Kriha02] Enterprise Portal Architecture, Scalability and Performance Analysis of a large scale portal project <<url>>

[Borthwick] John Borthwick, the CEO of Fotolog
<http://www.borthwick.com/weblog/2008/01/09/fotolog-lessons-learnt/>

[Little] M. Little, The Generic SOA Failure Letter
<http://www.infoq.com/news/2008/11/soa-failure>

[HeiseNews119307] Blackberry Storm Käuferansturm legt website lahm,
- <http://www.heise.de/newsticker/Blackberry-Storm-Kaeuferansturm-legt-Website-lahm--/meldung/119307>

[Kopparapu] Chandra Kopparapu, Load Balancing Servers, Firewalls, and Caches

[Haberl] Karl Haberl, Seth Proctor, Tim Blackman,
Jon Kaplan, Jennifer Kotzen, PROJECT DARKSTAR

Sun Microsystems Laboratories

[Pirazzi] Chris Pirazzi, Video I/O on Linux: Lessons Learned from SGI,
<http://lurkertech.com/linuxvideoio/>

[Fowler] Martin Fowler, distributed document-oriented databases,
<http://martinfowler.com/bliki/DatabaseThaw.html>

[InfoQ] distributed document-oriented databases
<http://www.infoq.com/news/2008/11/Database-Martin-Fowler>

distributed document-oriented databases
http://qconsf.com/sf2008/tracks/show_track.jsp?trackOID=170

[Purdy] Cameron Purdy, The Top 10 Ways to Botch Enterprise Java Technology-
Based Application Scalability and Reliability
<http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-4249.pdf>

[P2PNEXT] <http://www.p2p-next.org/>

[Gabriel] Richard P. Gabriel, Design beyond human abilities ,
<http://dreamsongs.com/Files/DesignBeyondHumanAbilitiesSimp.pdf>

[Scalaris] <http://www.zib.de/CSR/Projects/scalaris/>
http://www.ist-selfman.org/wiki/images/1/17/Scalaris_Paper.pdf
<http://www.ist-selfman.org/wiki/images/9/95/ScalarisLowRes.pdf>

<http://www.ist-selfman.org/wiki/images/d/d5/PeerTVLowRes.pdf>
http://www.ist-selfman.org/wiki/index.php/SELFMAN_Project

<http://p2pcomputing.blogspot.com/> links to p2p dist-sys.

[vanRoy] Peter van Roy, Self Management and the Future of Software Design,
<http://www.ist-selfman.org/wiki/images/0/01/Bcs08vanroy.pdf>

[vanRoy] Peter van Roy, The Challenges and Opportunities of Multiple
Processors: Why Multi-Core Processors are Easy and Internet is Hard (short piece
on conflicting goals in p2p and emergent behaviour like the intelligence of google
search)

[vanRoy] Peter van Roy, Overcoming Software Fragility with Interacting
Feedback Loops and Reversible Phase Transitions. (again the concept of feedback
loops for control)

[Bray] Tim Bray, Presentation: "Application Design in the context of the shifting
storage spectrum", Qcon 2008-12-01

[Fowler] Martin Fowler, DatabaseThaw,

<http://www.theregister.co.uk/2008/11/22/braykeynote/> on Bray Keynote, notes that memcached was a result of a large web2.0 site (LiveJournal.com)

[Hoff] Todd Hoff, google video, Youtube Architecture,
<http://highscalability.com/youtube-architecture>

[Hoff] Todd Hoff , A Bunch of Great Strategies for Using Memcached and MySQL Better Together <http://highscalability.com/bunch-great-strategies-using-memcached-and-mysql-better-together>

[Hoff] Todd Hoff, Facebook Tweaks – how to handle 6 times as many memcached requests,
http://highscalability.com/links/goto/545/396/links_weblink

[Hoff] Todd Hoff, Myspace Architecture, <http://highscalability.com/myspace-architecture>

[Hoff] Todd Hoff, Scaling Twitter: making Twitter 10000 Percent Faster,
<http://highscalability.com/scaling-twitter-making-twitter-10000-percent-faster>

[Blaine] Blaine, Big Bird, Scaling Twitter slides,
<http://www.slideshare.net/Blaine/scaling-twitter>

[Mituzas] Domas Mituzas, Wikipedia: Site internals, configuration, code examples and management issues, MySQL Users Conference 2007,
<http://dammit.lt/uc/workbook2007.pdf>

[Bergsma] Mark Bergsma, Wikimedia Architecture
<http://www.nedworks.org/~mark/presentations/san/Wikimedia%20architecture.pdf>

<http://highscalability.com>

[AboutDrizzle] about Drizzle, http://drizzle.org/wiki/About_Drizzle

[Dunkel et.al.] Jürgen Dunkel, Andreas Reinhart, Stefan Fischer, Carsten Kleiner, Arne Koschel, System-Architekturen Für Verteilte Anwendungen, Hanser 2008

[ProgrammableWeb] Overview of mashups, <http://www.programmableweb.org>

[CouchDB] CouchDB Technical Overview,
<http://incubator.apache.org/couchdb/docs/overview.html>

[Chang et.al.] Chang, Dean, Gemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, Gruber, Bigtable: A Distributed Storage System for Structured Data
<http://labs.google.com/papers/bigtable.html>

[DeCandia et.al.] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Voshall and Werner Vogels, “Dynamo: Amazon's

Highly Available Key-Value Store”, in the *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.
<http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>

[Vogels] Werner Vogels, Eventually Consistent – Revisited,
http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

[Vogels] Werner Vogels, Eventually Consistent, Building reliable distributed systems at a worldwide scale demands trade-offs—between consistency and availability. ACM queue,
http://portal.acm.org/ft_gateway.cfm?id=1466448&type=pdf

[Henderson] Cal Henderson, [Building Scalable Web Sites](#)

[Henderson] Cal Henderson, Scalable Web Architectures – Common Patterns and Approaches, presentation, <http://www.slideshare.net/techdude/scalable-web-architectures-common-patterns-and-approaches/138>

[SocialText] Story of Caching,
http://www.socialtext.net/memcached/index.cgi?this_is_a_story_of_caching

[Turner] Bryan Turner, The Paxos Family of Consensus Protocols
<http://brturn.googlepages.com/PaxosFamily.pdf>

[Turner08] Bryan Turner, The state machine approach
<http://brturn.googlepages.com/StateMachines08.pdf>

[IBM] IBM System Journal on Continuously Available Systems
<http://www.research.ibm.com/journal/sj47-4.html>

[Golle] Philippe Golle, N. Ducheneaut, Keeping Bots out of Online Games. In proc. of *2005 Advances in Computer Entertainment Technology*.
<http://crypto.stanford.edu/~pgolle/publications.html>

[Kleinpeter] Tom Kleinpeter, Understanding Consistent Hashing,
<http://www.spiteful.com/2008/03/17/programmers-toolbox-part-3-consistent-hashing/>

[White] Tom White, Consistent Hashing,
http://weblogs.java.net/blog/tomwhite/archive/2007/11/consistent_hash.html

[Karger] David Karger, Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web
<http://citeseer.ist.psu.edu/karger97consistent.html>

[MySQL] MySQL, Memcached hash types, MySQL HA/Scalability Guide
<http://dev.mysql.com/doc/mysql-ha-scalability/en/ha-memcached-using-hashtypes.html>

[MemCachedFAQ] Memcached faq,
<http://www.socialtext.net/memcached/index.cgi?faq>

[ViennaOnline] Offizielle Erklärung zum Bildausfall im Euro Halbfinale 2008
<http://www.vienna.at/magazin/sport/specials/euro2008/artikel/offizielle-erklaerung-zum-bild-ausfall-im-euro-halbfinale/cn/news-20080626-02235595/?origin=rssfeed>

[Telegraph] Euro 2008: Power cut leaves football fans in the dark 27 June 2008
<http://www.telegraph.co.uk/news/uknews/2195423/Euro-2008-Power-cut-leaves-television-fans-in-the-dark.html>

[Kelly] Kevin Kelly, Predicting the next 5000 Days of the Web
http://www.ted.com/index.php/talks/kevin_kelly_on_the_next_5_000_days_of_the_web.html

[Hoff] Todd Hoff, Scribe- Facebooks scalable logging system
<http://highscalability.com/product-scribe-facebooks-scalable-logging-system>

[Hoff] Todd Hoff, How I learned to Stop Worrying and Love Using a Lot of Disk Space to Scale, <http://highscalability.com/how-i-learned-stop-worrying-and-love-using-lot-disk-space-scale>

[Henderson] Cal Henderson, Scalable Web Architectures, Common Patterns and Approaches, <http://www.slideshare.net/techdude/scalable-web-architectures-common-patterns-and-approaches/138>

[Jordan] Kris Jordan Tips on REST for PHP <http://www.krisjordan.com/>

[Duxbury] Bryan Duxbury, Rent or Own: Amazon EC2 vs. Colocation Comparison for Hadoop Clusters <http://blog.rapleaf.com/dev/>

[Schlossnagel] Theo Schlossnagel, Scalable Internet Architectures

IO related:

[Santos] Nuno Santos, Building Highly Scalable Servers with Java NIO 09/01/2004 <http://www.onjava.com/lpt/a/5127>

[Naccarato] [Giuseppe Naccarato](#) Introducing Nonblocking Sockets 09/04/2002 <http://www.onjava.com/lpt/a/2672>

[Hitchens] Ron Hitchens, How to build a scalable multiplexed server with NIO, Javaone Conference 2006,
<http://developers.sun.com/learning/javaoneonline/2006/coreplatform/TS-1315.pdf>

[Roth] Gregor Roth, Architecture of a Highly Scalable NIO-based Server, 02/13/2007,
Dan Kegel's "[The C10K problem](#)" <http://www.kegel.com/c10k.html>

[Darcy] Jeff Darcy, Notes on High Performance Server Design
<http://pl.atyp.us/content/tech/servers.html>

[Liboop] Event library for asynchronous event notification,
<http://liboop.ofb.net/why>

[JargonFile] Thundering Herd Problem,
<http://catb.org/~esr/jargon/html/T/thundering-herd-problem.html>

[Simard] Dan Simard, AJAX, Javascript and threads: the final truth
<http://www.javascriporkata.com/2007/06/12/ajax-javascript-and-threads-the-final-truth/>

[vonBehren] Rob von Behren, Jeremy Condit, Eric Brewer, UCB, Why Events Are A Bad Idea (for high-concurrency servers)
<http://citeseer.ist.psu.edu/681845.html>

[Welsh] Welsh, Culler, et al. – 2001, SEDA: an architecture for well-conditioned, scalable Internet services –
<http://citeseerx.ist.psu.edu/showciting;jsessionid=B3029AE01E363095959876C62C88CC85?cid=7065>

[DeShong] Brian DeShong, Designing for Scalability, April 5, 2007
<http://media.atlantaphp.org/slides/2007-04-bdeshong.pdf> (excellent examples on de-normalization etc.)

[Graf] Markus Graf, Workflow und Produktionsaspekte einer CG-Animation im studentischen Umfeld, Bachelor-Thesis, HDM 2009-02-13

[NY Web Expo 2.0-Panel Discussion] Building in the Clouds: Scaling Web2.0 Writeup by Kris Jordan, <http://www.krisjordan.com/2008/09/18/panel-discussion-building-in-the-clouds-scaling-web-20/>, about metrics, CDNs, user behavior, quotas and the danger developers create for service clouds.

[Boer] Benjamin Boer, The Obama Campaign – A programmers perspective, ACM Queue, Jan. 2009 <http://queue.acm.org/detail.cfm?id=1508221>

[Meyer] B. Meyer, Software Architecture: Object Oriented Versus Functional, in: Domidis Spinellis, Georgios Gousios, (Ed.), Beautiful Architecture – Leading Thinkers Reveal the Hidden Beauty in Software Design

[Sletten] Brian Sletten, Resource-Oriented Architectures: being “in the Web”, in: Domidis Spinellis, Georgios Gousios, (Ed.), Beautiful Architecture – Leading Thinkers Reveal the Hidden Beauty in Software Design

[Turatti] Maurizio Turatti, camelcase, The CAP Theorem,
<http://camelcase.blogspot.com/2007/08/cap-theorem.html>

[Kyne] Frank Kyne, Alan Murphy, Kristoffer Stav

Clustering Solutions Overview: Parallel Sysplex and Other Platforms, Clustering concepts, Understanding Parallel Sysplex, clustering, comparing the terminology, IBM Redbook 2007

<http://www.redbooks.ibm.com/redpapers/pdfs/redp4072.pdf>

[Morrill] H. Morrill, M. Beard, D. Clitherow, Achieving continuous availability of IBM systems infrastructures, IBM SYSTEMS JOURNAL, VOL 47, NO 4, 2008 MORRILL, BEARD, AND CLITHEROW pg. 493

<http://www.research.ibm.com/journal/sj/474/morrill.pdf>

[Clarke] W.J. Clarke et.al., IBM System Z10 Design for RAS

<http://www.research.ibm.com/journal/rd/531/clarke.pdf>

[ITIL3] OGC Common Glossary, ITIL Version 3 (May 2007),

<http://www.best-management-practice.com/officialsite.asp?FO=1230366&action=confirmation&tdi=575004>

[LSHMLBP], Th. Lumpp, J. Schneider, J. Holtz, M. Mueller, N. Lenz, A. Biazetti, D. Petersen, From high availability and disaster recovery to business continuity solutions, HA approaches, in: IBM SYSTEMS JOURNAL, VOL 47, NO 4, 2008 <http://www.research.ibm.com/journal/sj/474/lumpp.pdf> (good explanation of HA concepts, clustering etc.)

[STTA] W.E. Smith, K.S. Trivedi, L.A. Tomek, J. Ackaret, Availability analysis of blade servers systems, in: IBM SYSTEMS JOURNAL, VOL 47, NO 4, 2008 <http://www.research.ibm.com/journal/sj/474/smith.pdf> (shows state-space models like Markov Models, Semi Markov Processes etc. for availability calculation. Nice failure tree of blade system architecture)

[CDK] R. Cocchiara, H. Davis, D. Kinnaird, Data Center Topologies for mission-critical business systems, in: IBM SYSTEMS JOURNAL, VOL 47, NO 4, 2008 <http://www.research.ibm.com/journal/sj/474/cocchiara.pdf> Disaster recovery concepts, two and three site architectures

Jboss Tree Cache – clustered, replicated, transactional, http://www.jboss.org/file-access/default/members/jbosscache/freezone/docs/1.4.0/TreeCache/en/html_single/index.html#d0e2066

[Miller] Alex Miller, Understanding Actor Concurrency, Part 1: Actors in Erlang http://www.javaworld.com/javaworld/jw-02-2009/jw-02-actor-concurrency1.html?nhtje=rn_031009&nladname=031009javaworld%27senterpris_ejavaal

Statistics, Modeling etc.

PDQ *Pretty Damn Quick*. Open-source queueing modeler. Supporting textbook with examples (Gunther 2005a) www.perfdynamics.com/Tools/PDQ.html

R Open source statistical analysis package. Uses the S command-processing language. Capabilities far exceed Excel (Holtman 2004).

www.r-project.org

SimPy Open-source discrete-event simulator
Uses Python as the simulation programming language.
simpy.sourceforge.net

[Burrows] Mike Burrows, The chubby lock service for loosely-coupled distributed systems, Google paper,

[Pattishall] Dathan Vance Pattishall, Federation at Flickr – doing Billions of Queries per Day,

[Indelicato] Max Indelicato, Scalability Strategies Primer: Database Sharding, <http://blog.maxindelicato.com/2008/12/scalability-strategies-primer-database-sharding.html>

[Hoff] Todd Hoff, “latency is everywhere and it costs you sales - how to crush it”

[Pritchett] Dan Pritchett, Lessons for lowering latency

[ALV] Al-Fares, Loukissas, Vahdat, A Scalable, Commodity Data Center Network Architecture

[Google] Google's Paxos Made Live – An Engineering Perspective

[Laird] Cameron Laird, Lightweight Web Servers – Special purpose HTTP applications complement Apache and other market leaders. (Evaluation criteria and lists of special purpose web servers)

[Sennhauser] Oli Sennhauser, MySQL Scale-Out by application partitioning. (Various partitioning methods for data, e.g range, characteristics. Load, hash/modulo. Application aware partitioning)

[Ottinger] Joseph Ottinger, What is an App Server? (Good comparison of J2EE architecture properties like request/response with a dynamic Grid environment).

[Lucian] Mihai Lucian, Building a Scalable Enterprise Applications using Asynchronous IO and SEDA Model, 2008 (with performance numbers)

[Jones] Tim Jones, Boost application performance using asynchronous I/O, posix AIO API. (on Linux IO models)

[Pyarali et.al] Pyarali, Harrison, Schmidt, Jordan, Proactor – An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events

[Lavender et.al.] Lavender, Schmidt, Active Object – An Object Behavioral Pattern for Concurrent Programming

[Gilbert et.al.] Seht Gilbert, Nancy Lynch, Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services (on the CAP Theorem), see also Vogels

[Indelicato] Max Indelicato, Distributed Systems and Web Scalability Resources, (excellent list from his blog)

[Smith] Richard Smith, Scalability by Design – Coding for Systems with Large CPU Counts, SUN.

[Trencseni] Morton Trencseni, Readings in Distributed Systems,
<http://bytepawn.com> (excellent resource for papers)

[HAProxy] Reliable, High Performance TCP/HTTP Load Balancer,
www.haproxy.lwt.eu

[Hoff] Todd Hoff, canonical cloud architecture, (emphasizes queuing).

[Karger et.al.] David Karger, Alex Sherman, Web Caching with Consistent Hashing <http://www8.org/w8-papers/2a-webserver/caching/paper2.html#chash2>

[Watson] Brandon Watson, Business Model Influencing Software Architecture (questions Animoto scaling lessons on EC2)
<http://www.manyniches.com/cloudcomputing/business-model-influencing-software-architecture/>

[geekr] Guerilla Capacity Planning and the Law of Universal Scalability,
<http://highscalability.com/guerilla-capacity-planning-and-law-universal-scalability>

[Optivo] Hscale, MySQL proxy LUA module (www.hscale.org) with some interesting numbers on DB limits discussed

[Amir et.al.] Amir, Danilov, Miskin-Amir, Schultz, Stanton, The Spread toolkit, Architecture and Performance

[Allamaraju] Subbu Allamaraju, Describing RESTful Applications, Nice article talking about locating resources when servers control their namespace. Bank API example.

[Schulzrinne] Henning Schulzrinne, Engineering peer-to-peer systems, Presentation. 2008. Excellent overview of p2p technology.

[Loeser et.al.] Loeser, Altenbernd, Ditze, Mueller, Distributed Video on Demand Services on Peer to Peer Basis.

[Scadden et.al.] Scadden, Bogdany, Clifftor, Pearthree, Locke, Resilient hosting in a continuously available virtualized environment, in: IBM Systems Journal Vol. 47 Nr. 4 2008 (on serial and parallel availability)

[Miller] Alex Miller, Understanding Actor Concurrency, Part 1: Actors in Erlang
www.javaworld.com 02/24/09
<http://www.javaworld.com/javaworld/jw-02-2009/jw-02-actor-concurrency1.html?nhtje=rn_030509&nladname=030509>

[ThinkVitamin.com] ThinkVitamin.com, Serving Javascript fast

[Yu] Wang Yu, Uncover the hood of J2EE clustering,
<http://www.theserverside.com/tt/articles/article.tss?l=J2EEClustering>

[Yu] Wang Yu, Scvaling your Java EE Applications, Part 1 and 2
<http://www.theserverside.com/tt/articles/article.tss?l=ScalingYourJavaEEApplications>
<http://www.theserverside.com/tt/articles/article.tss?l=ScalingYourJavaEEApplicationsPart2>

Terracotta scalability: <http://www.infoq.com/infoq/url.action?i=595&t=p>
<http://www.infoq.com/infoq/url.action?i=614&t=p>
<http://www.infoq.com/infoq/url.action?i=749&t=p>
<http://www.infoq.com/infoq/url.action?i=602&t=p>
<http://www.infoq.com/infoq/url.action?i=440&t=p>

<http://www.infoq.com/bycategory/contentbycategory.action?idx=2&ct=5&alias=performance-scalability>

what is an appserver?
<http://www.theserverside.com/tt/articles/article.tss?l=WhatIsAnAppServer>
Java EE APIs list too, cloud computing changes app server

<http://www.theserverside.com/tt/articles/article.tss?l=AreJavaWebApplicationsSecure>

[Resin] Scaling Web Applications in a Cloud Environment using Resin 4.0,
Technical White Paper, coucho 2009

[Perros] Harry Perros, Computer Simulation Techniques, The Definitive
Introduction, <http://www.csc.ncsu.edu/faculty/perros//simulation.pdf>

[Saab] Paul Saab, Scaling memcached at Facebook,
http://www.facebook.com/note.php?note_id=39391378919&id=9445547199&index=0

[Viklund] Andreas Viklund, Empyrean, Performance – When do I start worrying?
<http://pravanjan.wordpress.com/2009/03/24/performance-when-do-i-start-worrying/>

[Bray] Tim Bray, Sun Cloud API Restful API
<http://kenai.com/projects/suncloudapis/pages/Home>

Websphere eXtreme Scale <http://www-01.ibm.com/software/webservers/appserv/extremescale/>

[Levison] Ladar Levison, Lavabit-Architecture – Creating a Scalable Email
Service <http://highscalability.com/LavabitArchitecture.html>

[Xue] Jack Chongjie Xue, Building a Scalable High-Availability E-Mail System
with Active Directory and More, <http://www.linuxjournal.com/article/9804>

[Pravanjan] Pravanjan, Performance – when do I start worrying?

[Hoff] Are Cloud Based Memory Architectures the Next Big Thing? 03/17/2009, www.highscalability.com

[Persyn] Jurriaan Persyn, Database Sharding at Netlog, Presentation held at Fosdem 2009 <http://www.jurriaanpersyn.com/archives/2009/02/12/database-sharding-at-netlog-with-mysql-and-php/>

MVCC:

[Rokytsky] Roman Rokytsky, A not-so-very technical discussion of Multi Version Concurrency Control, http://www.firebirdsql.org/doc/whitepapers/tb_vs_ibm_vs_oracle.htm

[Webster] John Webster, DataDirect S2A: RAID for a Petabyte World, Aug. 2008, <http://www.illuminata.com>

[Coughlin] Tom Coughlin, The Need for (Reliable) Speed, Coughlin Associates, Aug. 2008

[Stedman] Geoff Stedman, Aktive Speichertechnik, Grid Speicher, in FKT 3/2008

[Bacher] Bacher Systems EDV GmbH, Storage einmal anders betrachtet, Newsletter 2/2008, <http://www.bacher.at>

[Venners] Bill Venners, Twitter on Scala – a conversation with Steve Johnson, Alex Payne and Robey Pointer, April 2009 http://www.artima.com/scalazine/articles/twitter_on_scala.html
(talks about Ruby problems with stability, building type systems in dynamic languages, just like the developers with static languages build dynamic features over time. Scala advantages and disadvantages.

[Glover] Andrew Glover, Storage made easy with S3, <http://www.ibm.com/developerworks>

[Maged et.al.] Maged Michael, José E. Moreira, Doron Shiloach, Robert W. Wisniewski
IBM Thomas J. Watson Research Center
Scale-up x Scale-out: A Case Study using Nutch/Lucene
<http://www.cecs.uci.edu/~papers/ipdps07/pdfs/SMTPS-201-paper-1.pdf>

[Chu et.al.] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, Kunle Olukotun, Map-Reduce for Machine Learning on Multicore, <http://www.cs.stanford.edu/people/ang/papers/nips06-mapreducemulticore.pdf>

[Bartel] Jan Bartel, Proposed Asynchronous Servlet API, http://www.theserverside.com/news/thread.tss?thread_id=40560

[Schroeder] B.Schroeder, M.Harchol-Balter, Web Servers under overload: How scheduling can help, in Charzinski, Lehnert, ITC 18, Elsevier Science

[Wilkins] Greg Wilkins, Asynchronous I/O is hard,
http://blogs.webtide.com/gregw/entry/asynchronous_io_is_hard
(on partial reads/writes and other problems)

[Sun] Thread Pools Using Solaris 8 Asynchronous I/O,
http://developers.sun.com/solaris/articles/thread_pools.html

[Palaniappan] Sathish K. Palaniappan, Pramod B. Nagaraja, Efficient data transfer through zero copy: Zero Copy – Zero Overhead,
<http://www.ibm.com/developerworks/linux/library/j-zero-copy/>

David Patterson, Why Latency Lags Bandwidth,
and What it Means to Computing
http://www.ll.mit.edu/HPEC/agendas/proc04/powerpoints/Banquet%20and%20Keynote/patterson_keynote.ppt

[Maryka] Steve Maryka, What is the Asynchronous Web and How is it Revolutionary? http://www.theserverside.com/tt/articles/article.tss?track=NL-461&ad=700978&l=WhatistheAsynchronousWeb&asrc=EM_NLN_6729006&uid=5812009

[Shalom] Nati Shalom, Auto-Scaling your existing Web Applications,
http://library.theserverside.com/detail/RES/1242406940_306.html?asrc=vcatssc_s_itepost_05_15_09_c&li=191208

[Sweeney] Tim Sweeney, The Next mainstream Programming Language: a Game Developers Perspective, <http://www.cs.princeton.edu/~dpw/popl/06/Tim-POPL.ppt>

[Jäger] Kai Jäger, Finding parallelism - How to survive in a multi-core world
Bachelor thesis at HDM Stuttgart 2008

[Schneier] Bruce Schneier, Interview on cloud-computing
<http://www.vnunet.com/vnunet/video/2240924/bruce-schneier-cloud-security>

[Fountain] Stefan Fountain, What happens when David Hesselhof meets the cloud, experiences with AWS, <http://www.infoq.com/presentations/stefan-fountain-hasselhoff-cloud>

[Elman] Josh Elman, glueing together the web via the facebook platform,
<http://www.infoq.com/presentations/josh-elman-glue-facebook-web>

[Armstrong] Joe Armstrong, Functions + Messages + Concurrency = Erlang

<http://www.infoq.com/presentations/joe-armstrong-erlang-qcon08>

[Wardley] Simon Wardley, cloud, commoditisation etc.

<http://www.slideshare.net/cpurrington/cloudcamp-london-3-canonical-simon-wardley>

[Oracle] Oracle® Database Concepts, 10g Release 2 (10.2) Part Number B14220-02 , Chapter 13 Data Concurrency and Consistency (on statement or transaction read level consistency, MVCC use and isolation levels possible. Essential reading for the web site architect).

[Harrison] Ann W. Harrison, Firebird for the Database Expert: Episode 4 - OAT, OIT, & Sweep,

http://www.ibphoenix.com/main.nfs?a=ibphoenix&page=ibp_expert4

[Wilson] Jim R. Wilson, Understanding Hbase and BigTable,

http://jimbojw.com/wiki/index.php?title=Understanding_Hbase_and_BigTable

[Wilson] Jim R. Wilson, Understanding Hbase Column-family performance options http://jimbojw.com/wiki/index.php?title=Understanding_HBase_column-family_performance_options

[Goetz] Brian Goetz, Java theory and practice: Concurrent collections classes - ConcurrentHashMap and CopyOnWriteArrayList offer thread safety and improved scalability

<http://www.ibm.com/developerworks/java/library/j-jtp07233.html#author1>

[Ellis] Jonathan Ellis Why you won't be building your killer app on a distributed hash table

<http://spyced.blogspot.com/2009/05/why-you-wont-be-building-your-killer.html#>

[Bain] Tony Bain, The Problems with the Relational Database (Part 1) –The Deployment Model <http://weny.ws/1Xx>

[EMC] Storage Systems Fundamentals to Performance and Availability

<http://germany.emc.com/collateral/hardware/white-papers/h1049-emc-clariion-fibre-chnl-wp-ldv.pdf>

[Schmuck] Frank Schmuck, Roger Haskin, GPFS: A Shared-Disk File System for Large Computing Clusters, Proceedings of the FAST 2002 Conference on File and Storage Technologies Monterey, California, USA January 28-30, 2002

http://db.usenix.org/events/fast02/full_papers/schmuck/schmuck.pdf

[Avid] Avid Unity Isis,

http://www.avid.com/resources/whitepapers/Avid_Unity_ISIS_WP.pdf

[Northrop] Linda Northrop, Scale changes everything, OOPSLA06

<http://www.sei.cmu.edu/uls/files/OOPSLA06.pdf>

Goth, Greg. [Ultralarge Systems: Redefining Software Engineering?](#) *IEEE Software*, 2008

Gabriel, Richard P. [Design Beyond Human Abilities](#)

[Heer] Jeffrey Heer, Large-Scale Online Social Network Visualization,
<http://www.cs.berkeley.edu/~jheer/socialnet/>

[Hohpe] Gregor Hohpe, Hooking Stuff Together – Programming the Cloud
<http://www.infoq.com/presentations/programming-cloud-gregor-hohpe>

[Goth] Greg Goth, Ultralarge Systems: Redefining Software Engineering, IEEE Software March/April 2008

[Jacobs] Adam Jacobs, The pathologies of Big Data, ACM Queue
<http://queue.acm.org/detail.cfm?id=1563874>

[Henney] Kevlin Henney, Comment on Twitter Architecture
<http://www.infoq.com/news/2009/06/Twitter-Architecture>

[Saab] Paul Saab (notes) facebook developer blog, Friday, December 12, 2008 at 12:43pm
<http://www.facebook.com/people/Paul-Saab/500025857>

[Weaver] Evan Weaver, Architectural changes to Twitter,
<http://blog.evanweaver.com/about/>

[google] Entity Groups and Transactions
<http://code.google.com/appengine/docs/python/datastore/transactions.html>

[Scheurer] Isolde Scheurer, Single-Shard MMOG EVE online, HDM 2009,
<http://www.kriha.de/krihaorg/dload/uni/..<< >>>

[Stiegler] Andreas Stiegler, MMO Server Structures, Why the damn thing always lags! HDM 2009 <http://www.hdm-stuttgart.de/~as147/mmo.pdf> slides:
<http://www.hdm-stuttgart.de/~as147/mmo.pptx>

[Seeger] Marc Seeger, Key-Value Stores – a short overview... << >>

[Spolsky] Joel Spolsky, Can your Programming Language do that? Article on functional programming and map reduce in <http://www.joelonsoftware.com>

[Adzic] Gojko Adzic, Space Based Programming,
<http://gojko.net/2009/09/07/space-based-programming-in-net-video/>

[Krishnan et.al.] Rupa Krishnan Harsha V. Madhyastha Sridhar Srinivasan
Sushant Jain§
Arvind Krishnamurthy Thomas Anderson£ Jie Gao, Moving Beyond End-to-End Path Information to Optimize CDN Performance

[DynaTrace] The problem with SLA monitoring in virtualized environments
<http://blog.dynatrace.com/2009/09/23/the-problem-with-sla-monitoring-in-virtualized-environments/>

[VMWare] Time Keeping in VMWare Virtual Machines
http://www.vmware.com/pdf/vmware_timekeeping.pdf

[Harzog] Bernd Harzog Managing Virtualized Systems – Pinpointing performance problems in the virtual infrastructure April 2008
<http://www.vmworld.com/servlet/JiveServlet/previewBody/3420-102-1-4432/Managing%20Virtualized%20Systems%20-%20APM%20experts%20Apr08.pdf>

[Dynatrace] Cloud Service Monitoring for Gigaspaces
<http://blog.dynatrace.com/2009/05/07/proof-of-concept-dynatrace-provides-cloud-service-monitoring-and-root-cause-analysis-for-gigaspaces/>

[Chiew] Chiew, Thiam Kian (2009) Web page performance analysis. PhD thesis, University of Glasgow. <http://theses.gla.ac.uk/658/01/2009chiewphd.pdf>

[Schroeder] Bianca Schroeder, Eduardo Pinheiro, Wolf-Dietrich Weber, DRAM Errors in the Wild: A Large-Scale Field Study
<http://www.cs.toronto.edu/~bianca/papers/sigmetrics09.pdf>

[Cooper] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, HansArno Jacobsen, Nick Puz, Daniel Weaver and Ramana Yerneni, PNUTS: Yahoo!'s Hosted Data Serving Platform,
<http://highscalability.com/yahoo-s-pnuts-database-too-hot-too-cold-or-just-right>

[Cantrill] Brian Cantrill, Dtrace Review, Google Video
<http://video.google.com/videoplay?docid=-8002801113289007228#>

[Shoup] Randy Shoup, eBay's Challenges and Lessons from Growing an eCommerce Platform to Planet Scale HPTS 2009 October 27, 2009

[Click] Cliff Click, Brian Goetz, A crash-course in modern hardware, Video, JavaOne 2009 <http://www.infoq.com/presentations/click-crash-course-modern-hardware>

[Ristenpart] Thomas Ristenpart □ Eran Tromer † Hovav Shacham □ Stefan Savage □, Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds
<http://people.csail.mit.edu/tromer/papers/cloudsec.pdf>

[Heiliger] Jonathan Heiliger Real-World Web Application Benchmarking
<http://www.facebook.com/notes/facebook-engineering/real-world-web-application-benchmarking/203367363919> Discusses the effects of memory access times in a highly optimized infrastructure

Index

—**M**—

Media 5

—**P**—

People 5

—**S**—

Social Media 5