Seminar on Event-Driven Distributed Systems

Event-Driven Application Architectures and Systems

Forces:

Business needs dynamic re-configuration, adaptation and scalability of applications.

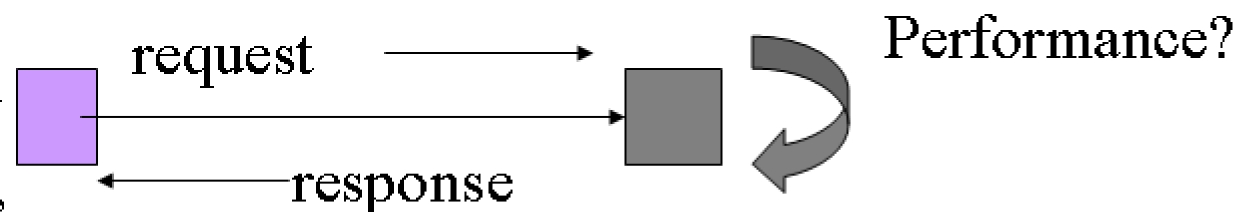
Automation of data processing and data-driven systems.

Architectural Cold-Spots in Request/Reply Systems

client

server

Knows service, must locate server, waits for response, polls service, control flow includes service call, Synchronous vs asynchr. call?



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,)

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!

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

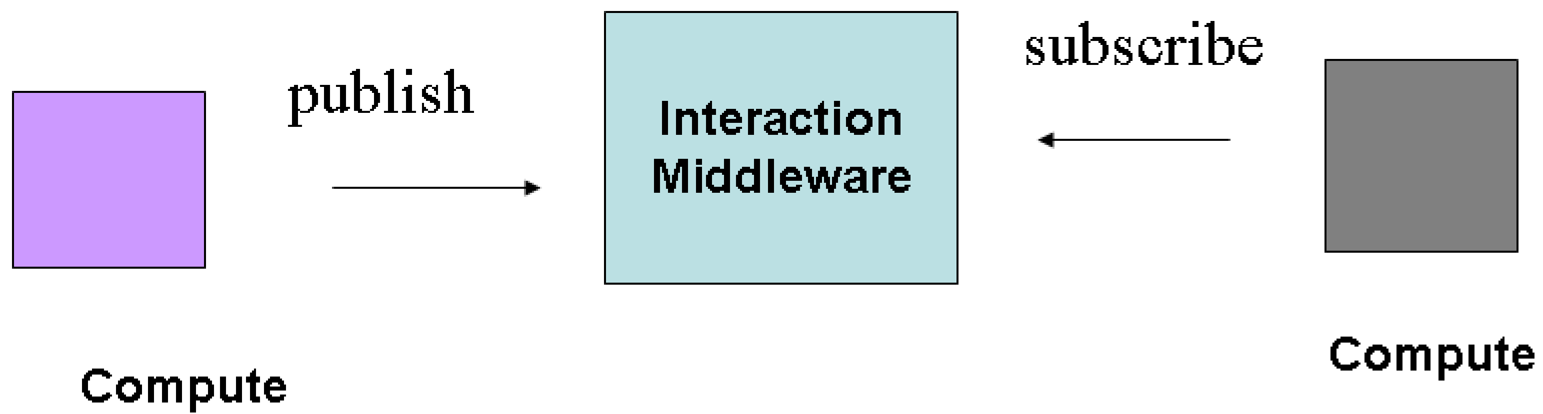
Sounds a lot like integrated circuits!

Event-Style Violations

- Encode sender/receiver information in message
- Simulate synchronous request/reply using async middleware

The use of scopes defined by administrative components provides a solution for those cases that is more in line with the separation of interaction from computation

Decoupling interaction from computation



Security Options in event-based systems

- Content-based: encryption. Problem: PKI needs to KNOW receiver as receivers public key is used to encrpyt (problem for health card e.g.) This would couple receiver and sender. Symmetric keys suffer from the distribution and trust problem. Group keys might help here, but watch out for overhead!
- Content based: Signatures. Receiver at least needs to know senders public key, sender needs to know receiver‘s security capabilities (algorithms, policies). Again requires coupling.
- Route-based. Create scopes which route information between sender and receiver without their knowledge. Can also filter and manipulate content

Manipulating content e.g. to encode sender identity was already defined as an anti-pattern for event-based systems. The same is true if used for security reasons. Only signatures could provide some useful security (along with scopes) if used properly.

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

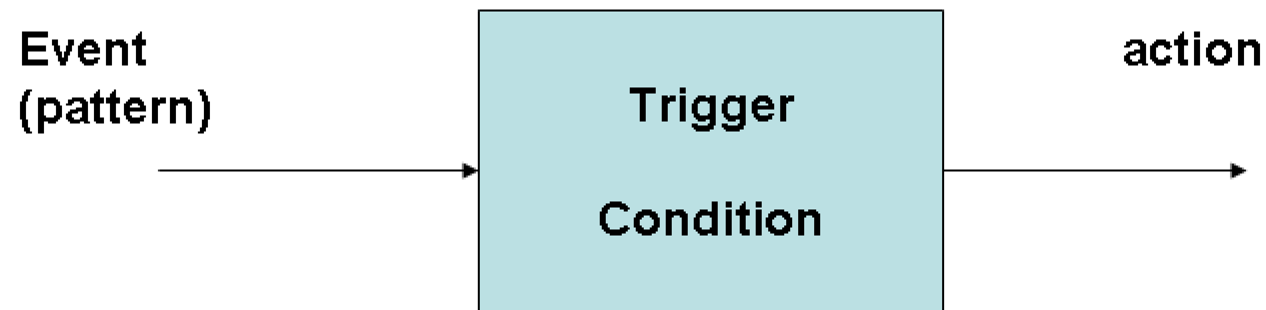
Characteristics:

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

Features of event-driven interaction

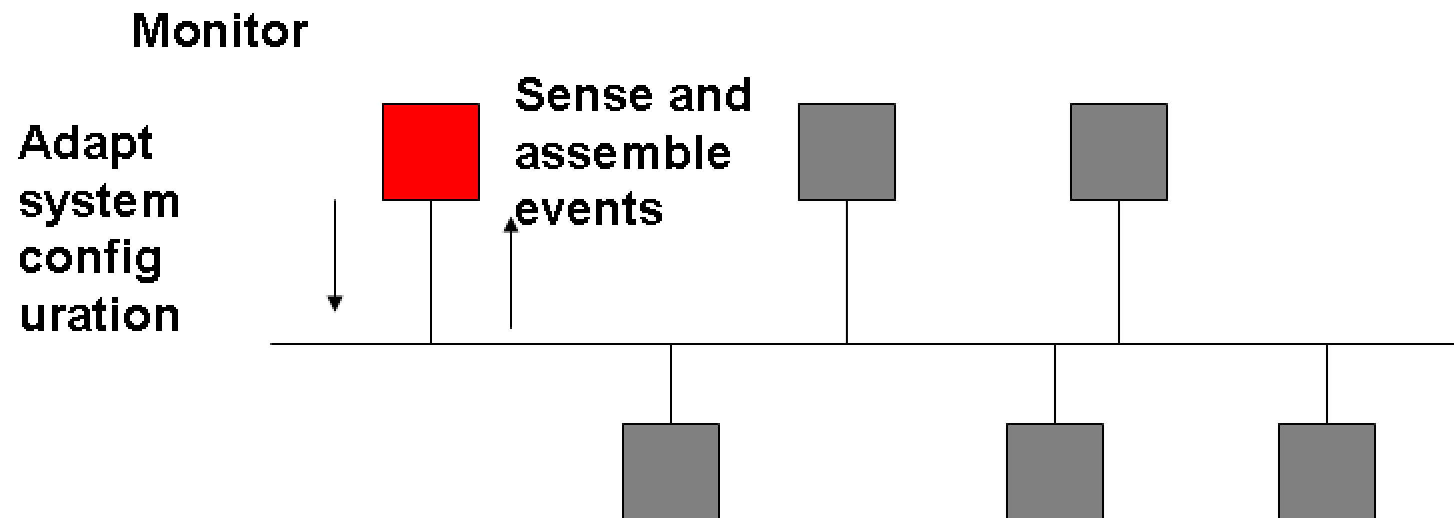
- Basic event: Everybody can send events, everybody can receive events - no restrictions, filtering etc.
- Subscription: Optimization on receiver side. Only events for which a subscription exists are forwarded to receiver. Can trigger publishing too.
- Advertisement: Optimization on sender side. Informs receivers about possible events. Avoids broadcast of every event in connection with subscriptions.
- Content-based filtering can be used for any purpose. Can happen at sender side, in the middleware or at receiver side.
- Scoping: manipulation of routes through an administrative component. Invisible assembly of communicating components through routes.

Trigger Example: Event-Condition-Action (ECA)



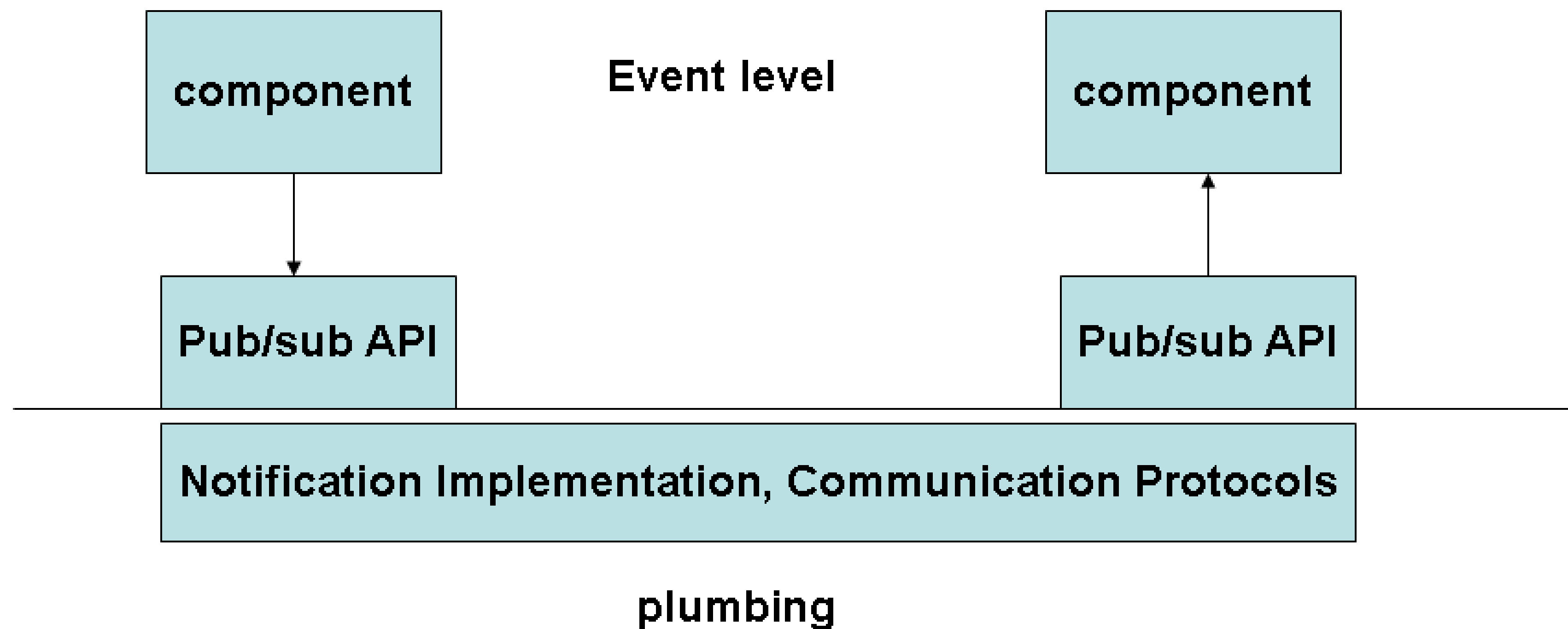
Leads automatically to a state machine like modeling of the problem domain!

Adaptive Distributed System



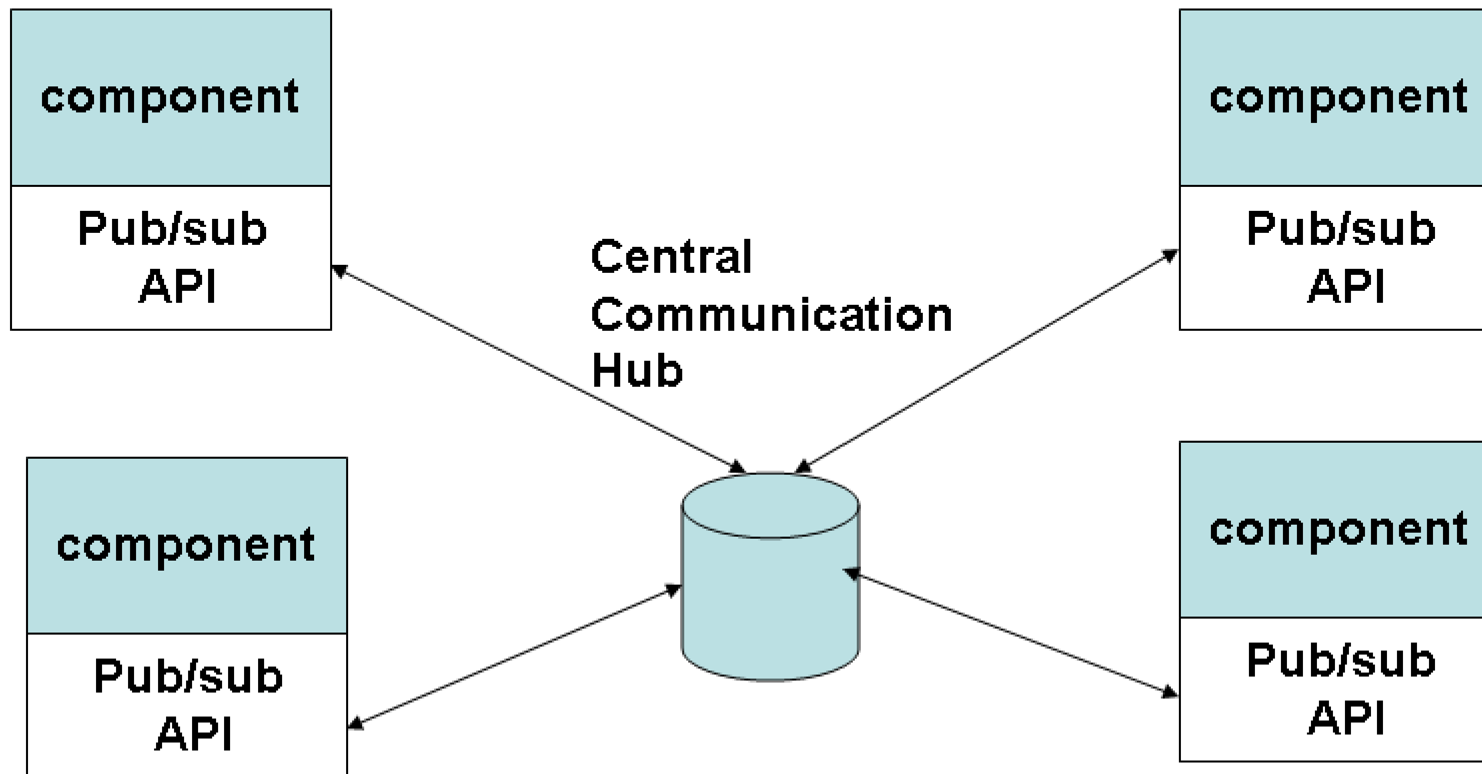
Requires the assembly of higher-level events from many lower level events (e.g. detecting an attack on a system).

Event-Architecture and Notification Implementation



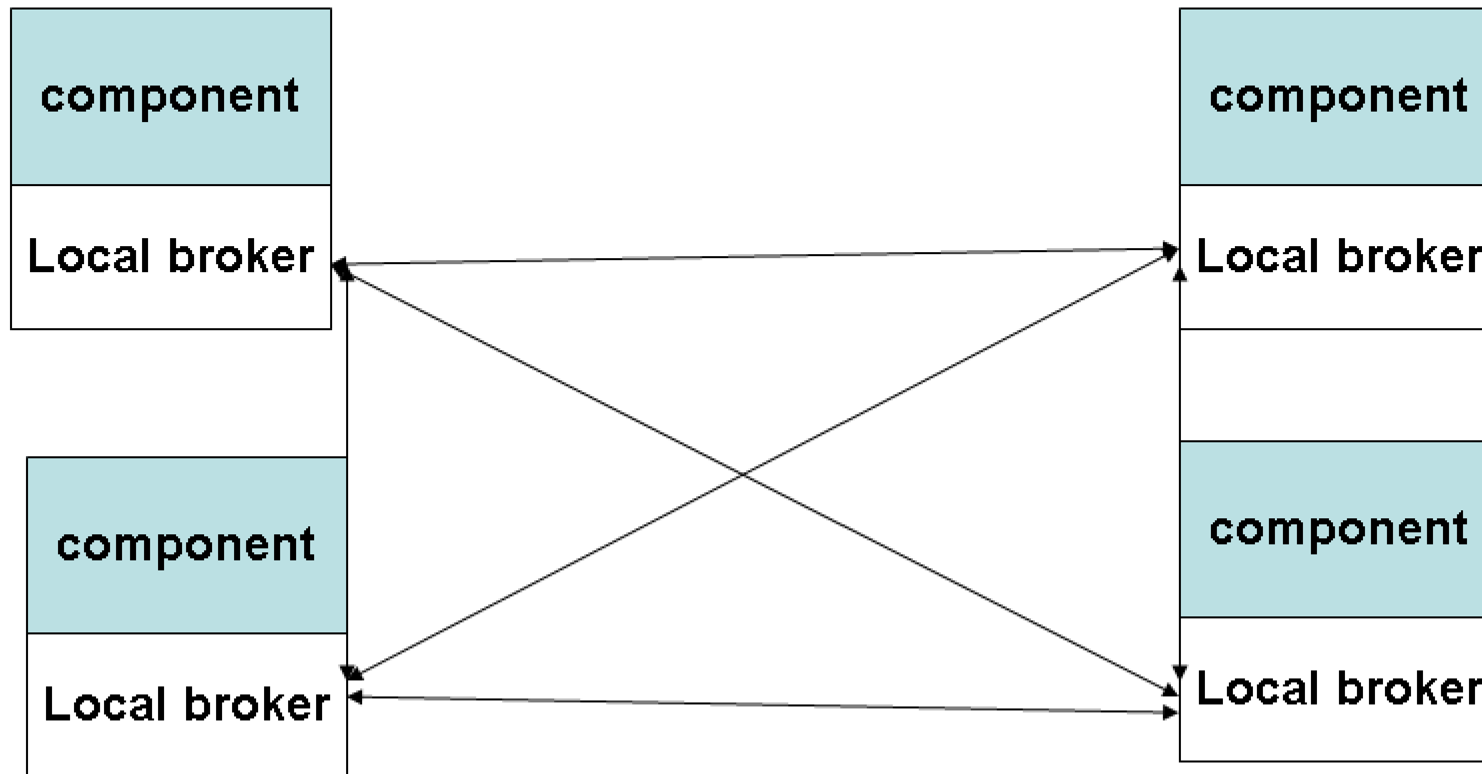
All combinations are possible: event architecture can rest on a weak, directly connected implementation (e.g. traditional observer implementation in MVC) or request-reply architecture can use true pub/sub notification mechanisms with full de-coupling)

Communication-Architecture 1: centralized



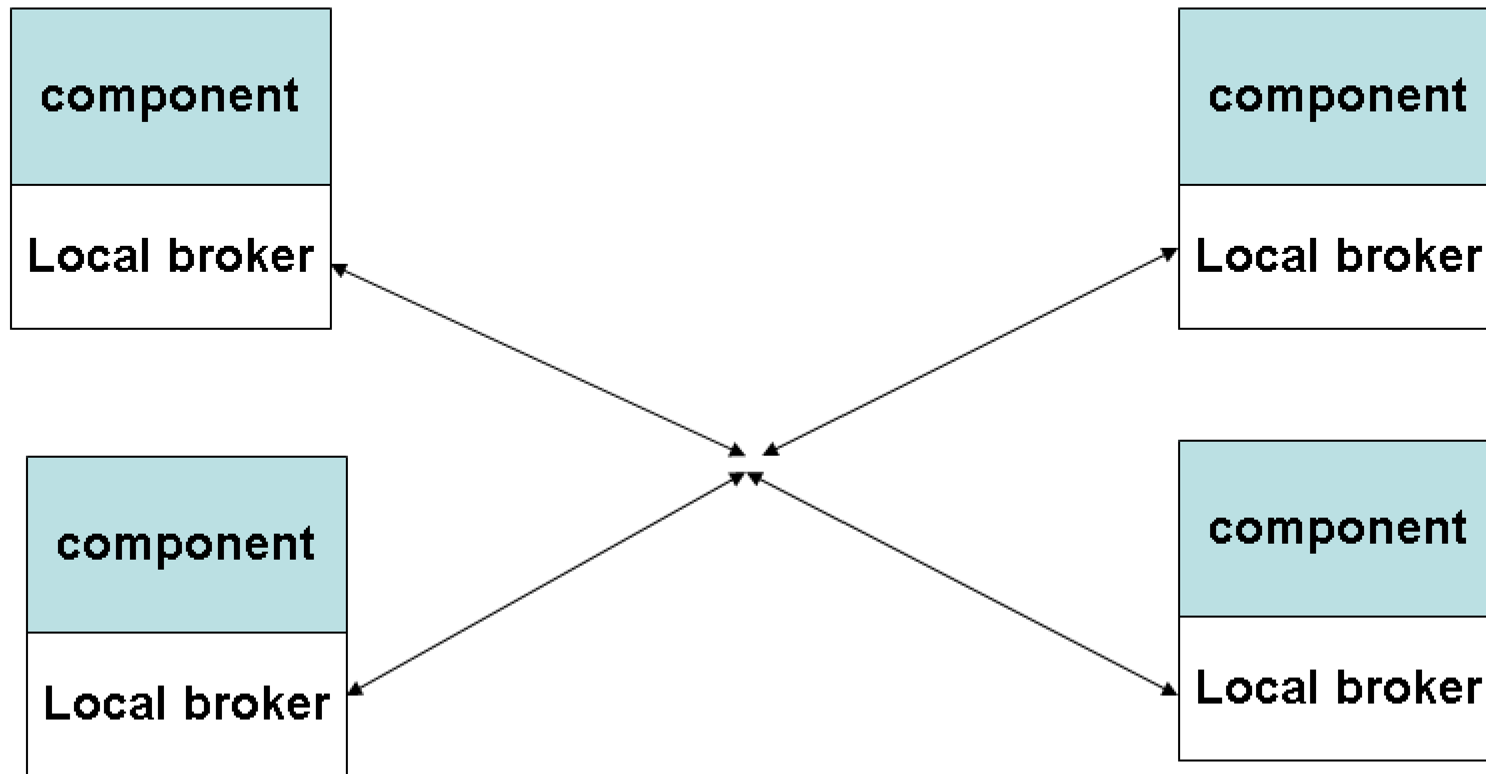
The system collects all notifications and subscriptions in one central place. Event matching and filtering are easy. Creates single-point-of-failure and scalability problems. High degree of control possible. No security/reliability problems on clients

Communication-Architecture 2: point-to-point



Local brokers need to know about each other. This puts communication and interaction knowledge into applications.
Misbehaving clients can destroy system/application semantics.
Communication overhead through point-to-point communication but good control over communication

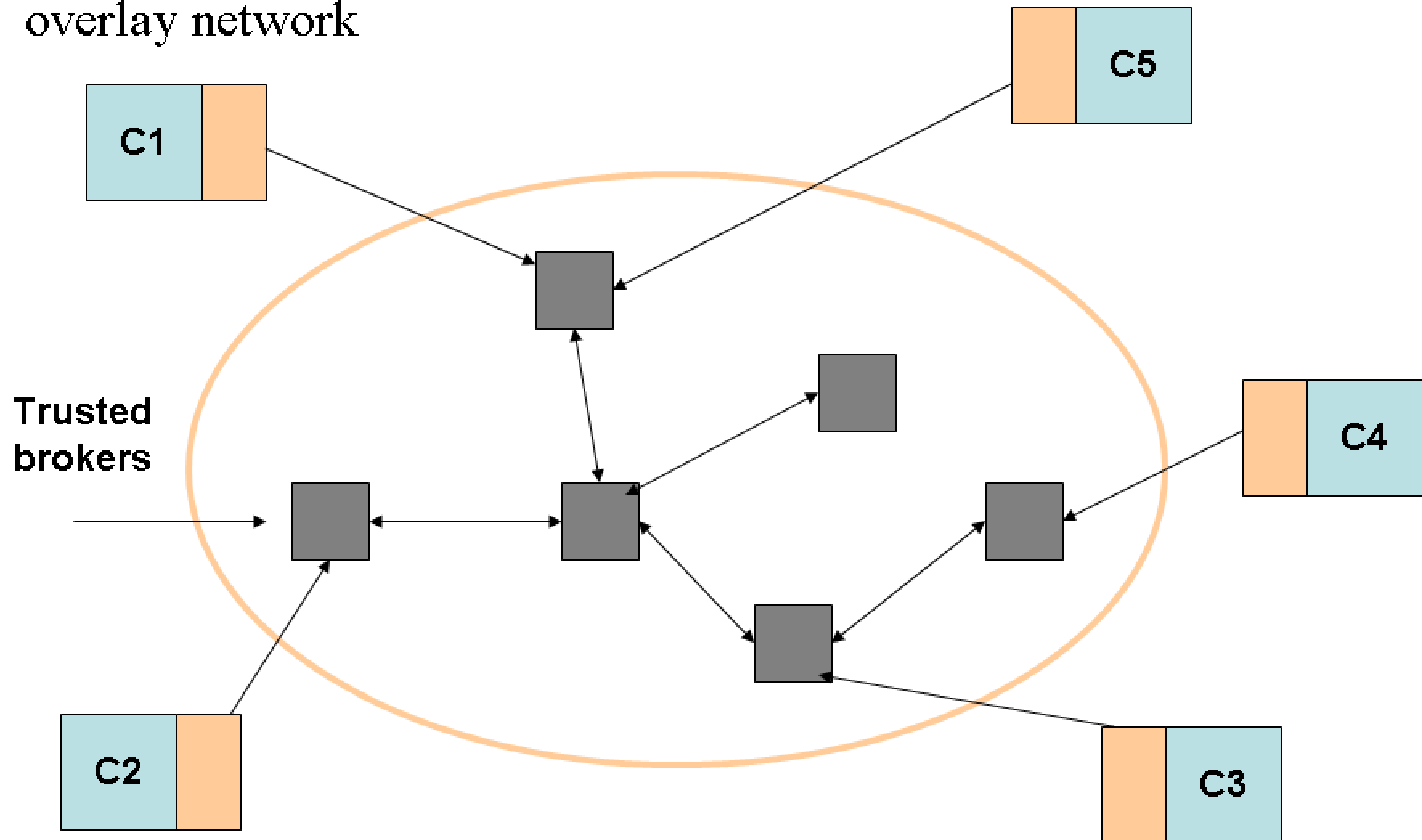
Communication-Architecture 3: multicast



Very good performance in communication layer. Bad control over system behavior. Much responsiblity in application/component layer. Security problems. No routing costs!

Communication-Architecture 4: distributed system with local brokers

Rebecca distributed notification middleware through overlay network



See: Mühl et.al. Pg. 21

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).

Distributed Notification Routing 1: flooding

Subscribe (S)

C1

Subscribe (S)

C5

S S

N

Subscribe (S)

C2

S

N

N

N

Notify (N)

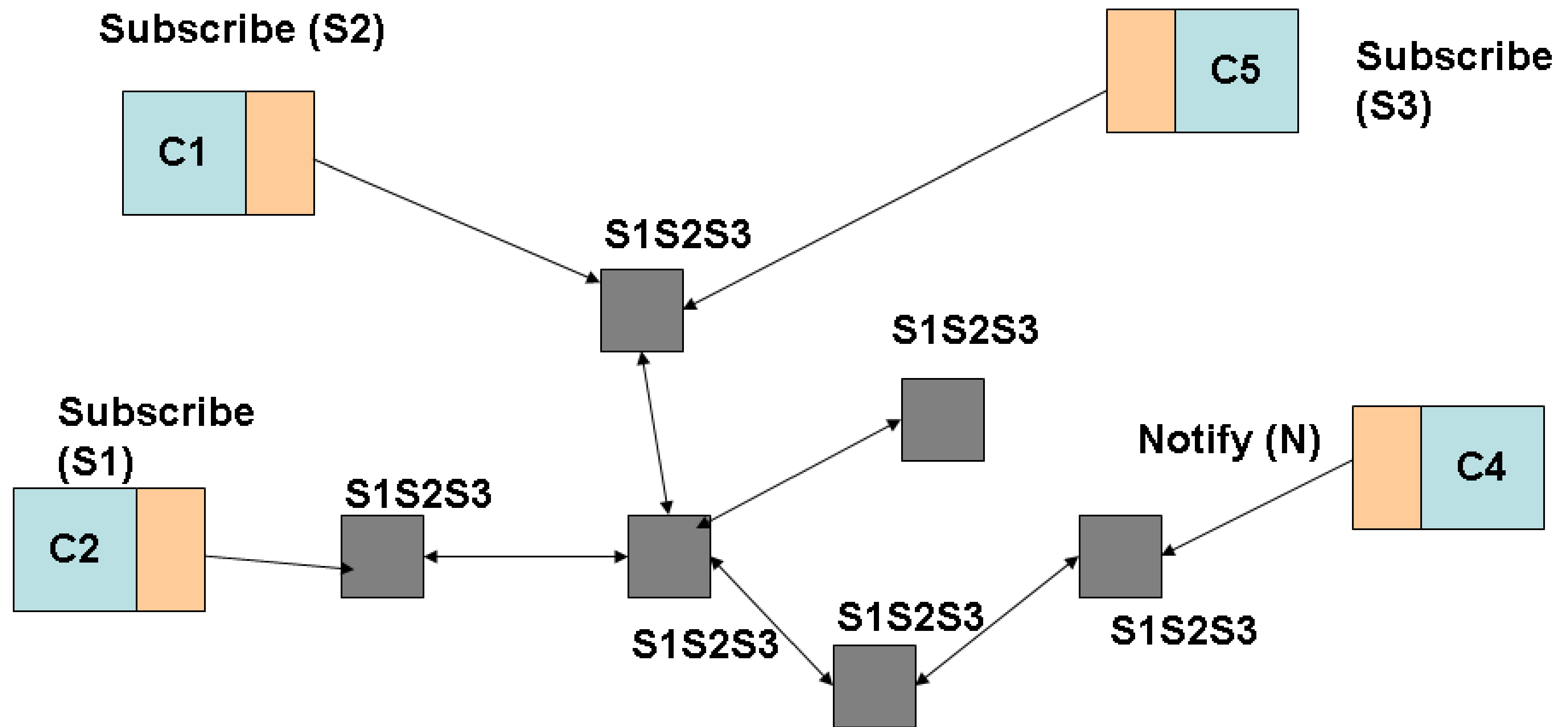
C4

N

N

With flooding notifications travel towards subscriptions which are only kept at leaf brokers. See: Mühl et.al. Pg. 22 ff. Advantages are that subscriptions become effective rather quickly and notifications are guaranteed to arrive everywhere. The price is a large number of unnecessary notifications to leaf nodes without subscribers

Distributed Notification Routing 2: simple filter-based routing



With filter-based routing subsriptions travel towards publishers. See: Mühl et.al. Pg. 23 ff. Advantages are that fewer notifications have to be transmitted. The price is large routing tables as all brokers need to know about all subscriptions/un-subscriptions and reconfiguration (subs/unsubs) causes high communication overhead.

Optimizations of filter-based routing

a) Avoid forwarding of duplicate subscriptions (identity based routing)

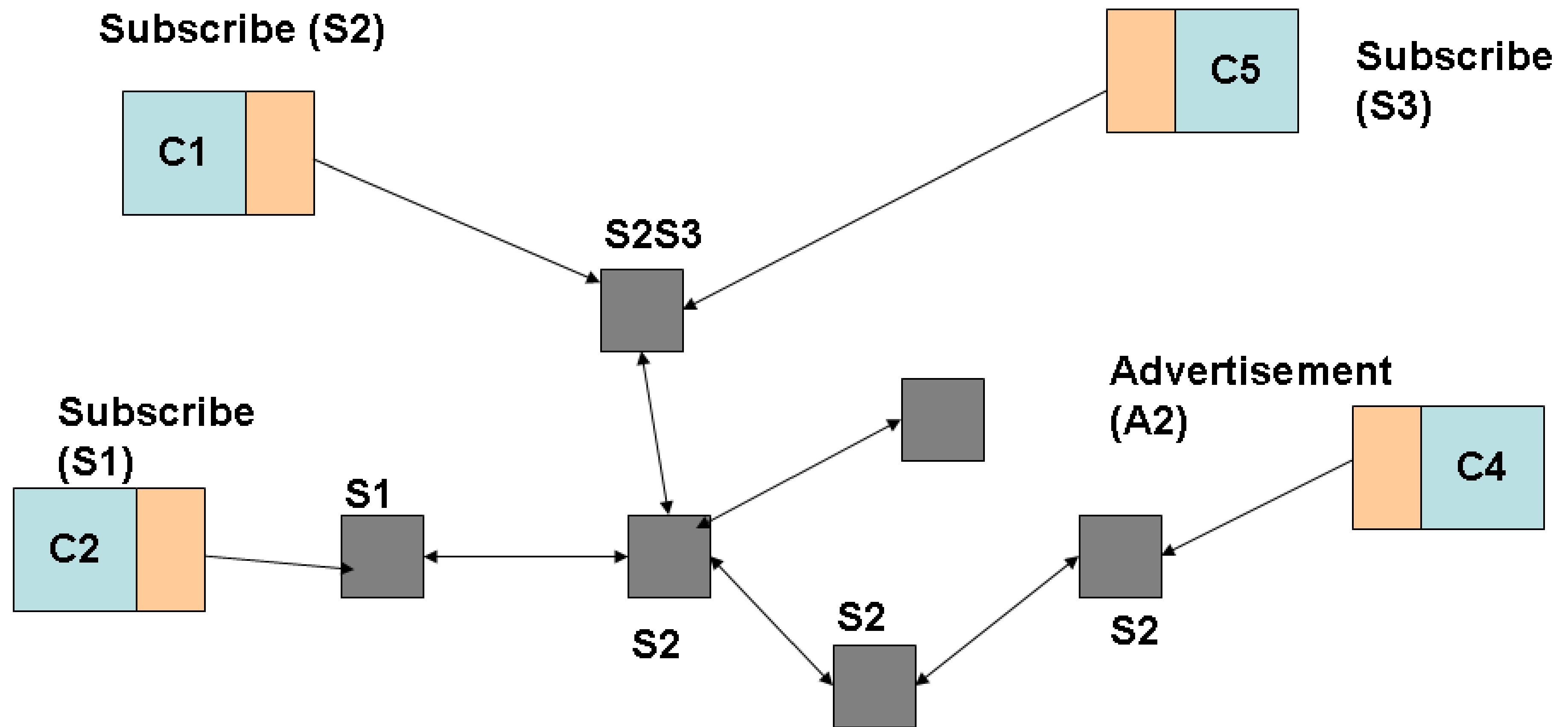
b) Avoid forwarding of partial subscriptions in case an encompassing subscription already exists (detect subscription covering)

c) Create broad, encompassing subscriptions by merging different subscriptions into a larger one, perhaps even covering some yet unsubscribed events to avoid costly reconfiguration

These optimizations try to reduce the communication overhead with forwarding subscriptions. The price lies in increased complexity (in case of unscubscriptions other subscriptions which were previously covered by this subscription now have to be forwarded instead.

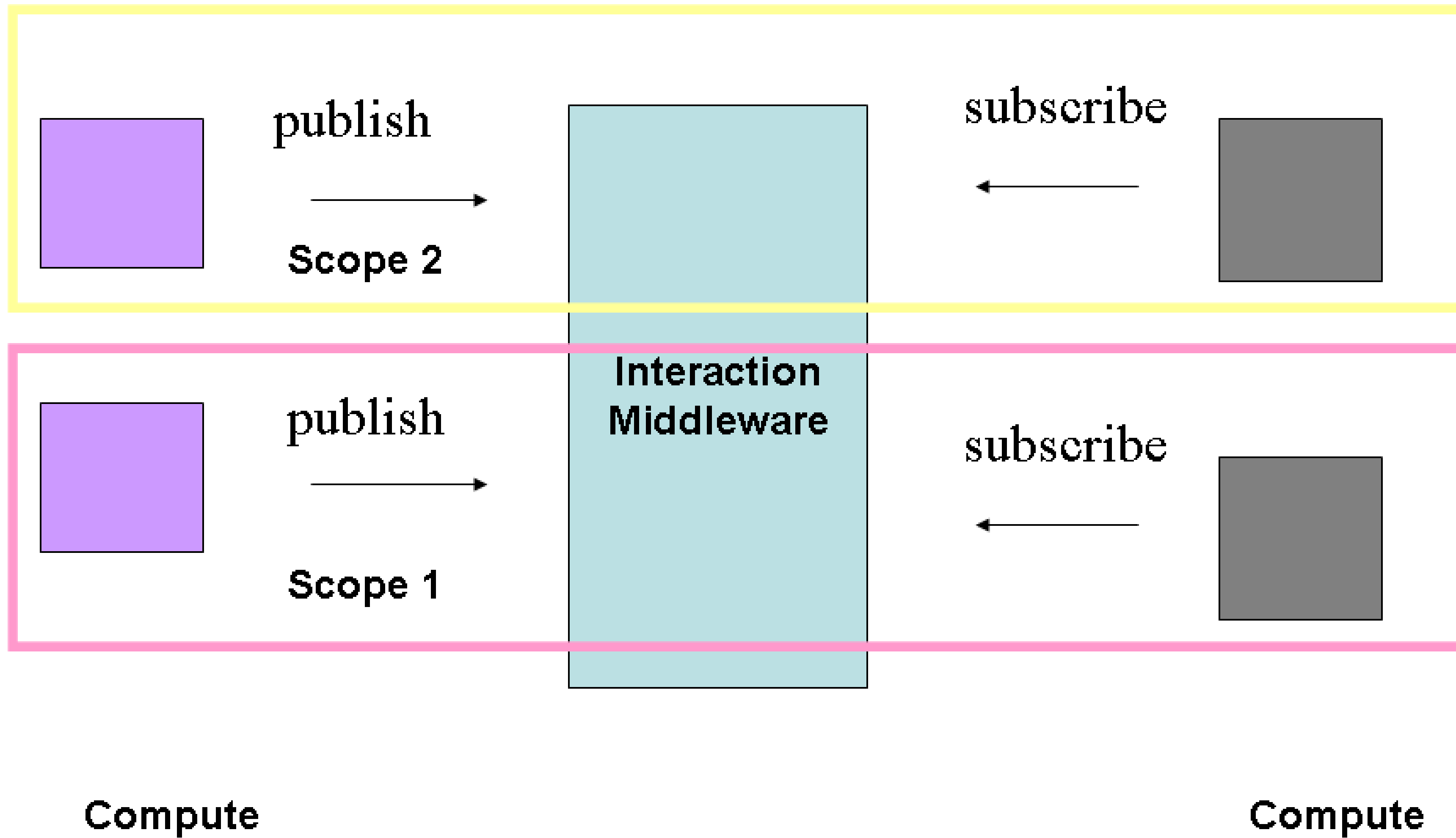
Broader subscriptions may result in notifications beeing distributed without an existing real subscription but can avoid costly re-configuration in case of new subscriptions.

Optimized filter-based routing with advertisements



Only Node C4 advertises that it will publish notifications for subscriptions of type S2. The routing system therefore only forwards S2 subscriptions towards C4. In case of a new advertisement the routing system needs to automatically forward the existing subscriptions that match the advertisement towards the node. Advertisements allow the routing system to optimize the configuration. It is also used in peer-to-peer frameworks like JXTA.

Decoupling organization from computation: scopes



Interfaces of Event-Based Systems

-Subsribe: a component places a subscription

-Unsubscribe: a component withdraws a subscription

-Publish: a component publishes an event

-Notify: a component receives an event

-Advertise (optional): a component declares to publish events in the future

There are logical implications behind those interfaces which can be expressed in temporal logic. Things to express are e.g. the idempotency of operations (what happens if a component subscribes twice for the same event?), dependencies etc. (see safety and liveness later)

Modeling of event-based Systems with traces

S0 (A0) -> S1 (A1) -> S2 (A2) -> S3 (A3) -> S4 (A4) …

Or with collapsed states and actions:

S0, S1, S2, S3, S4, …..

A trace is a sequence of states (see Mühl et.al. 25 ff). Properties of traces can be described with temporal logic. Even concurrent distributed processes can be described that way.

Formal Specification of event-based Systems with temporal logic

Operators:

-Always: a predicate or expression holds for all sub-traces or places within a trace

-Eventually: a predicate of expression will hold for at least one place in a subtrace or one subtrace

-Next: a predicate or expression holds for the second place in a subtrace or for the second subtrace

-Logical operators and quantifiers

-Atomic Predicates P

A specification is a set of traces. A system in compliance with a specification will only show behavior (traces) which are part of this set.

(see Mühl et.al. 25 ff).

Examples of specifications of event-based Systems with temporal logic

Operators:

-Always: a predicate or expression holds for all sub-traces or places within a trace

-Eventually: a predicate of expression will hold for at least one place in a subtrace or one subtrace

-Next: a predicate or expression holds for the second place in a subtrace or for the second subtrace

-Logical operators and quantifiers

-Atomic Predicates P

A specification is a set of traces. A system in compliance with a specification will only show behavior (traces) which are part of this set.

(see Mühl et.al. 25 ff).

Basic rules of event-based Systems expressed informally

Safety:

- receive notifications only if subscribed to them
- received notifications must have been published before
- receive a notification only at most once

Liveness:

- start receiving notifications some time after a subscription was made

Note that at most once notification may not be enough as it implies that a system crash might lose notifications (better: persistent notifications with exactly once semantics) and also: there is no guarantee about the time lag between subscriptions and the beginning of notifcations. No ordering rule is given for a simple system.

(see Mühl et.al. 25 29).

Basic rules of event-based Systems expressed with temporal logic

Safety:

Always[notify(Y,n) => [n element of SubscriptionSet(Y)] AND
[n element of PublishingSetOf(SomeComponent)] AND
[eventually always NOT notify(Y,n)]]

Liveness:

Always[Always(Filter F ElementOf SubscriptionSet(Y) =>
[Eventually always(pub(X,n) AND n element of
MatchedEvents(F) => Eventually notify(Y,n))]]

Note that this is a specification for the logical behavior of a system. Real (physical) behavior of a system might differ and the problem lies in either preventing the difference or making it disappear over time (self-repairing, self-stabilizing system)

(see Mühl et.al. 25 29 and the discussion on self-stabilizing and fault-tolerant systems pg. 264ff).

Ordering in Distributed Event-Systems

- -FIFO Ordering: a component needs to receive notifications in the order they were published by the publisher
- -Causal Ordering
- -Total Ordering: events n1 and n2 are published in this order. Once a component receives n2 not other component in this system is allowed to receive n1.

Total Ordering is orthogonal to the other two ordering relations.

Concepts: time, ordering and failures in distributed systems

-Time: no global system time exists. Useful: interval timers

-Ordering: partial order means that some elements of a set are ordered, but others are not affected by the ordering relation. Total order means all elements are ordered

-Failures: either fail-stop like (all of a system stops and we have a clean failure state) or byzantine (several parts of a system can fail in unpredictable and changing ways which may prevent the system from achieving a new, consistent state.

- In asynchronous systems certain failures can prevent progressing to a safe state or e.g. the correct functioning of a detection mechanism (try to detect A and B as a complex event. What happens if B does not arrive due to network failures?)

Engineering Event-Driven Systems

-Subscriber issues (quenching, initialization, learning about events)

-Publishing issues (meta-data encoding, no subscriber behavior, when to create event)

- Cross-cutting concerns (transactions, activities, locality of reference, Quality-of-Service, Sessions)

- Administration concerns: how to bundle notifications, separation of events, creation of hierarchies of events, security through routing configuration)

- Causality and Complex Events

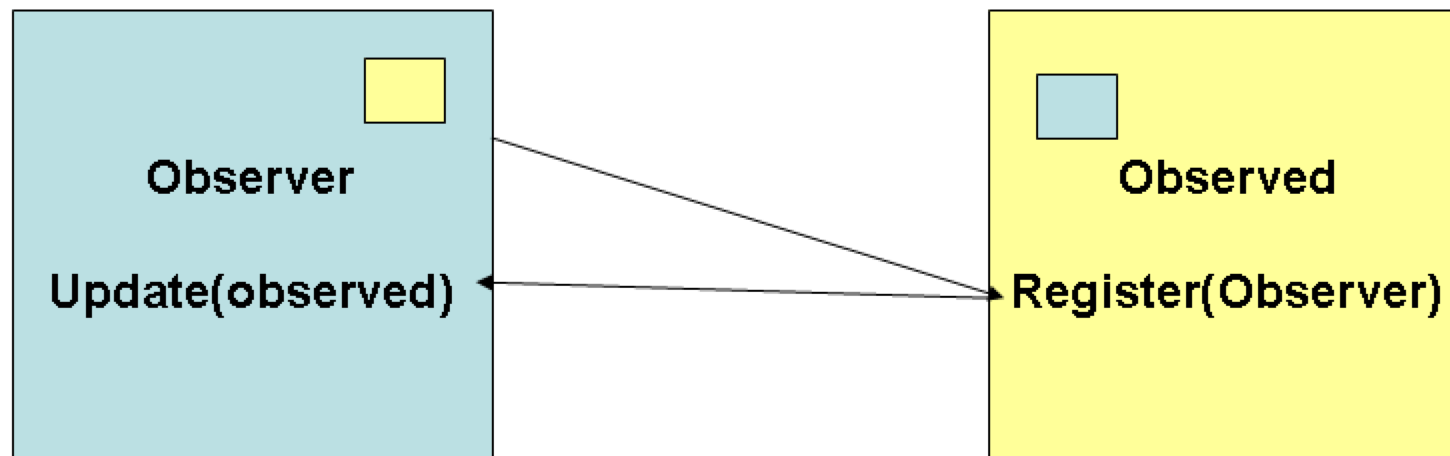
- Scoping

- Filtering: Non-deterministic Automata, filter combining,

Observer-Pattern: engineering problems

Init: tight coupling between components

Coarse granular events. Unclear semantics: object, delta, old + new etc. No QoS through filtering



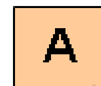
Unreliable communication, no once and only once delivery guarantees. No message interception points for aspect injection. No transparent scope administration for configuration

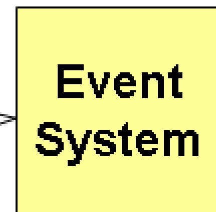
Unreliable callback with danger of livelock and inconsistencies (concurrency). Timing not decideable.

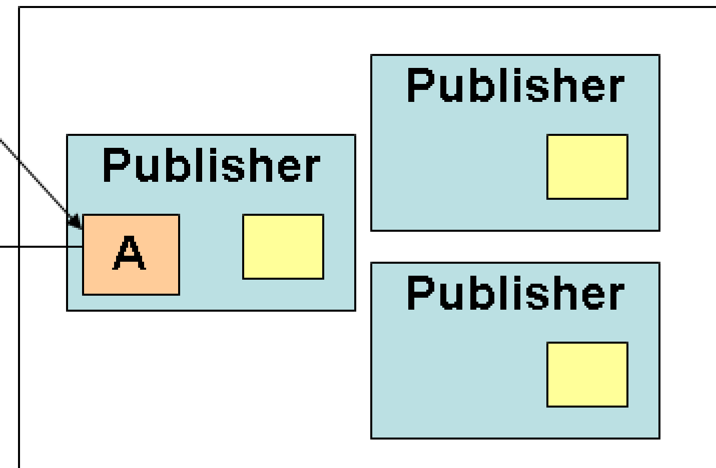
Event-Driven: Engineering Solutions

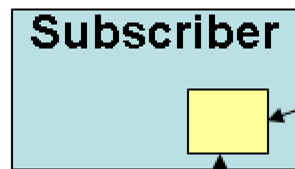
Init: System access through dependency injection

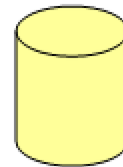
Event source configuration through aspect injection: event adapter











Quenching/Reliability QoS

QoS control: no publishing without subscriber

Local transactions, bundling of components for locality of ref.

reliable communication, no once and only once delivery guarantees.

Efficient Matching

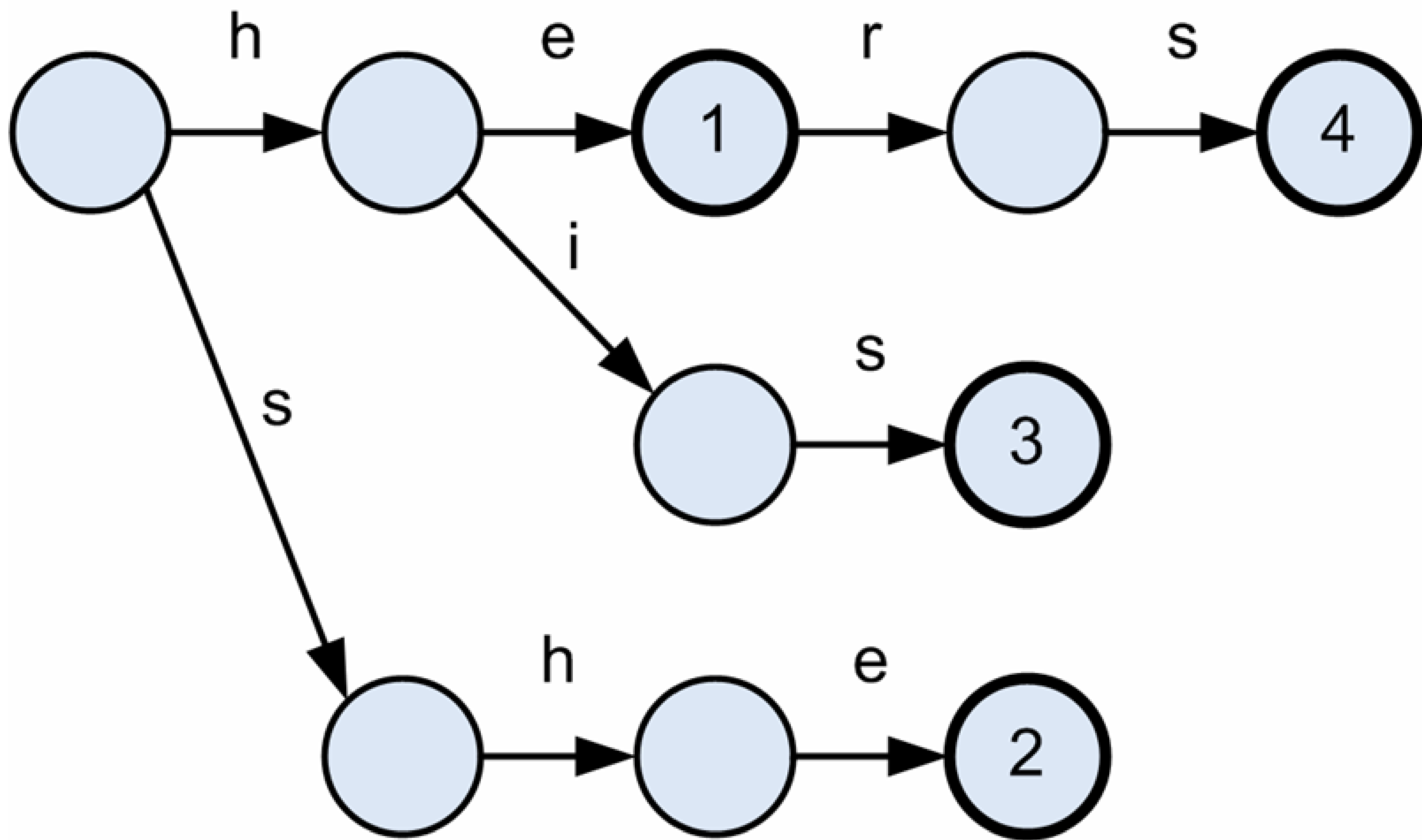
-Sequential filtering is too expensive

-Filter combining needed

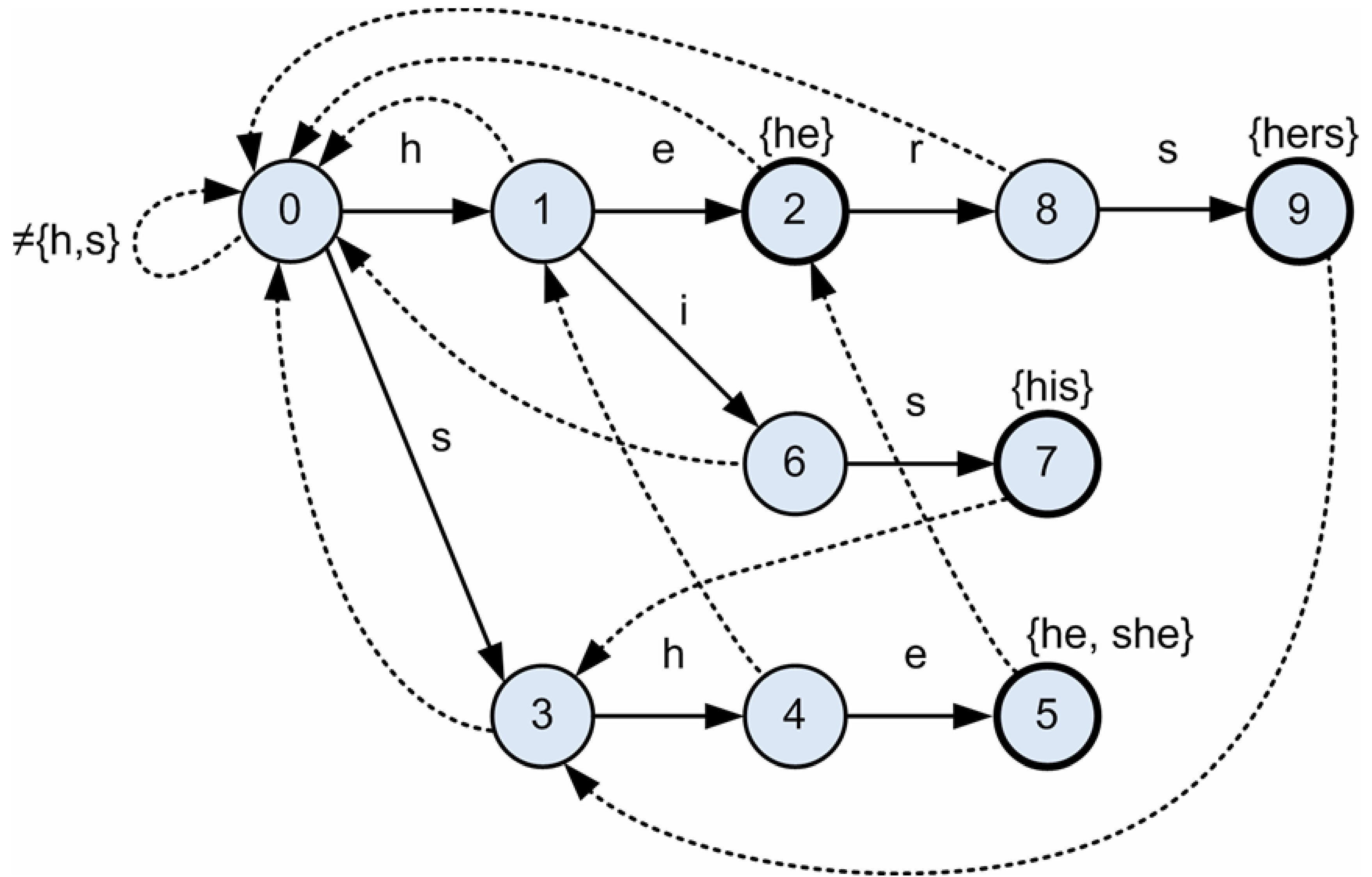
-Non-deterministic automata

-Deterministic automata

Do not recalculate predicates for every filter. A common search space needs to be created. Automata need to be generated for every filter definition.



Aho-Corasick Algorithm without error-links. From: Matthias Schmidt, Intrusion Detection Systems



Aho-Corasick Algorithm with error-links. From: Matthias Schmidt, Intrusion Detection Systems

Trie-Filter Automaton (Aho-Corasick Alg.)

Eine in diesem Kontext sehr interessante Eigenschaft dieses Algorithmus ist, dass er die Signaturen nicht einzeln, sondern alle in einem Schritt vergleicht. Formal ausgedrückt, sollen alle Vorkommnisse des Musters P (also die Signaturen) in einem Text T (also dem überwachten Netzwerkverkehr) gefunden werden, wobei die Elemente von P und T aus einem endlichen Alphabet Σ stammen. Muster und Text sind in so genannten Feldern gespeichert, mit T[1…n] und P[1…m], also ist n:=|T| und m:=|P|. Die Komplexität während der Suchphase ist O(n+z), also linear. z ist hierbei die Anzahl der Vorkommnisse des gesuchten Musters im Text und muss berücksichtigt werden, da bei jedem Treffer eine Funktion aufgerufen werden muss, die die Treffer speichert oder ausgibt. Diese lineare Laufzeitkomplexität wird allerdings teilweise dadurch erkauft, dass der Algorithmus vor dem eigentlichen Suchvorgang eine Initialisierungsphase benötigt, in der der so genannte Keyword-Tree, oft auch Trie genannt, erzeugt werden muss. Dieser Vorgang hat zusätzlich noch eine Komplexität von O(m). Es ergibt sich also eine gesamte (immer noch lineare) Zeitkomplexität des Algorithmus von O(m+n+z).

In der Initialisierungsphase wird aus den Signaturen ein Baum erzeugt (dieser repräsentiert also die Menge der Muster P), der folgende Eigenschaften hat:

- Jede Kante ist mit einem Zeichen aus P versehen

- Es dürfen von einem Knoten niemals 2 oder mehr Kanten mit gleichen Zeichen ausgehen.

- Ein Weg startet immer von der Wurzel aus und endet bei einem Knoten. Wege repräsentieren Teilworte aus P.

- Die Wurzel repräsentiert das leere Wort. Dieser Baum hat die gleichen Eigenschaften wie ein endlicher Automat. Die Knoten, die eine Zahl enthalten, repräsentieren hierbei die Endzustände des Automaten, also die Elemente aus P.

Trie-Filter Automaton (Aho-Corasick Alg.) cont.

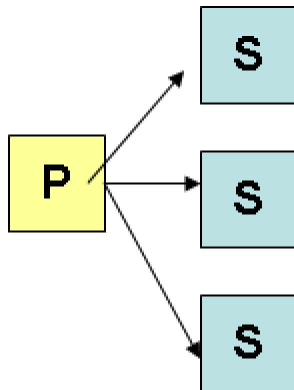
Ein solcher Trie kann folgendermassen erzeugt werden: - Man beginnt immer bei der Wurzel und folgt dem Pfad, der durch die Zeichen der Wörter von Pi vorgegeben wird. - Endet der Pfad, bevor alle Zeichen eines Wortes aus P betrachtet wurden, so werden dem Baum neue Kanten und Knoten für die noch fehlenden Zeichen hinzugefügt. - Bei einem Endknoten (also einem vollständigen Wort aus P) wird der Bezeichner i von Pi für diesen Knoten gespeichert. Da nun ein Teilwort (Suffix) im Baum auch das Präfix eines längeren Wortes darstellen kann, müssen noch so genannte Fehlerlinks hinzugefügt werden. Diese sind im unten dargestellten Bild gestrichelt eingezeichnet. Normale Zustandsübergänge, die durch Verwendung eines Symbols aus P erreicht werden können, sind als ungestrichelte Pfeile eingezeichnet. Die Zustände 2, 9, 7 und 5 sind Endzustände, also vom endlichen Automat akzeptierte Eingaben. > Ein Fehlerlink wird durchlaufen, wenn für das folgende Zeichen des betrachteten Strings aus P kein normaler (ungestrichelter) Übergang vorhanden ist. Zur Demonstration der Funktion des Algorithmus soll das Eingabewort "ushers" gewählt werden, im Wesentlichen orientiere ich mich hierzu an den Ausführungen von [2]. Für das erste Zeichen "u" wird im Startzustand verweilt, da hierfür kein Übergang in einen anderen Koten existiert.. Nun wird mit dem zweiten Zeichen, "s", fortgefahren. Der Automat geht dadurch in Zustand 3 über. Durch Eingabe des dritten Zeichens, "h", geht der Automat erfolgreich in Zustand 4 über, danach durch Eingabe von "e" in Zustand 5. Da der Knoten 5 mit "he" und "she" markiert ist, weiss er nun, dass diese beiden Strings im Text gefunden werden und kann diese Information speichern oder ausgeben. Allerdings ist der Eingabestring noch nicht zu Ende. Somit wird mit dem fünften Buchstaben "r" fortgefahren. Da der Automat hierfür keine gültige Transition besitzt, wird die Fehlerfunktion ausgeführt, also dem Fehlerlink gefolgt. Er wechselt dadurch in Zustand 2, da "he" das längste gültige Suffix von "she" ist. In diesem Zustand 2 wird nun eine Transition für "r" gefunden und der Automat kann in Zustand 8 wechseln. Durch Eingabe des letzten Zeichens "s" kann der Automat noch in den Zustand 9 überführt werden und findet hiermit den String "hers" im Text. Da keine weiteren Eingabezeichen vorhanden sind bricht der Algorithmus an dieser Stelle ab. Ergebnis sind drei gefundene Strings: "she", "he" und "hers".

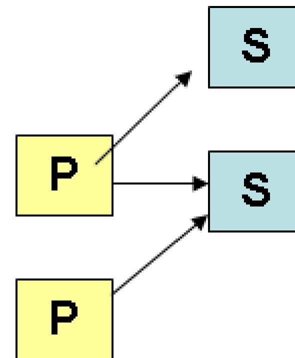
Trie-Filter Automaton (Aho-Corasick Alg.) cont.

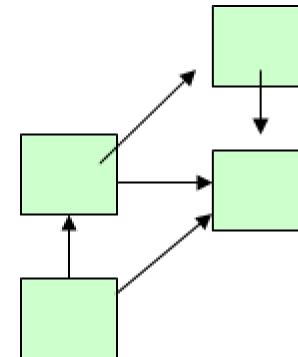
Dieses Verfahren ist extrem schnell, aufgrund der linearen Komplexität auch für eine grosse Anzahl an Signaturen. Dies ist in einem NIDS von grosser Wichtigkeit, da selbst ein übliches Fast Ethernet Netzwerk (Übertragungsrate 100MBit/s) ca. 23234 durchschnittlich grosse TCP/IP-Pakete mit jeweils 538 Byte (also 4304 Bit, Annahmen: Kollisionsfreies Netz, 26 Byte Ethernet-Frame + durchschnittlich 500 Bytes Nutzdaten (z.B. TCP/IP) + 12 Byte Inter-Frame Gap) pro Sekunde überträgt. Dies bedeutet, dass jedes Paket in durchschnittlich 43 Mikrosekunden komplett analysiert werden muss. Für heutige NIDS sind 100MBit-Netze zwar kein Problem mehr, Gigabit-Netze stellen aber sehr wohl für viele Systeme noch ein Problem dar. Dass eine naive Suche bei einer solch grossen Datenmenge keinen Sinn hat, wird schnell deutlich wenn man die worst-case Laufzeitkomplexität von O(m * n), also quadratisch, betrachtet.

From: M.Schmidt, Intrusion Detection Systems

Data Flow Topologies







Simple dissemination: Stock quotes, news, content distr. networks

Multi-Source: markets, suppliers, channels

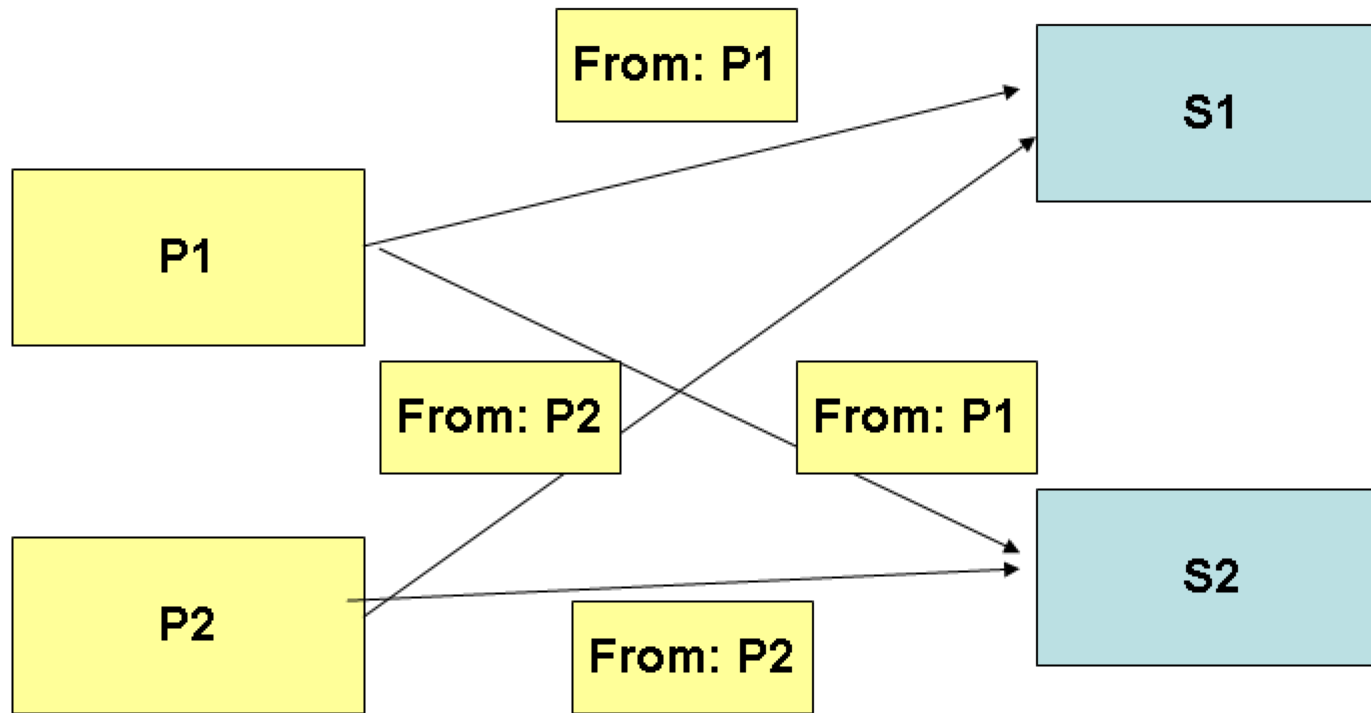
How do subscribers distinguish sources?

Total-distributed: chats, MMPG

Feedback is now possible!

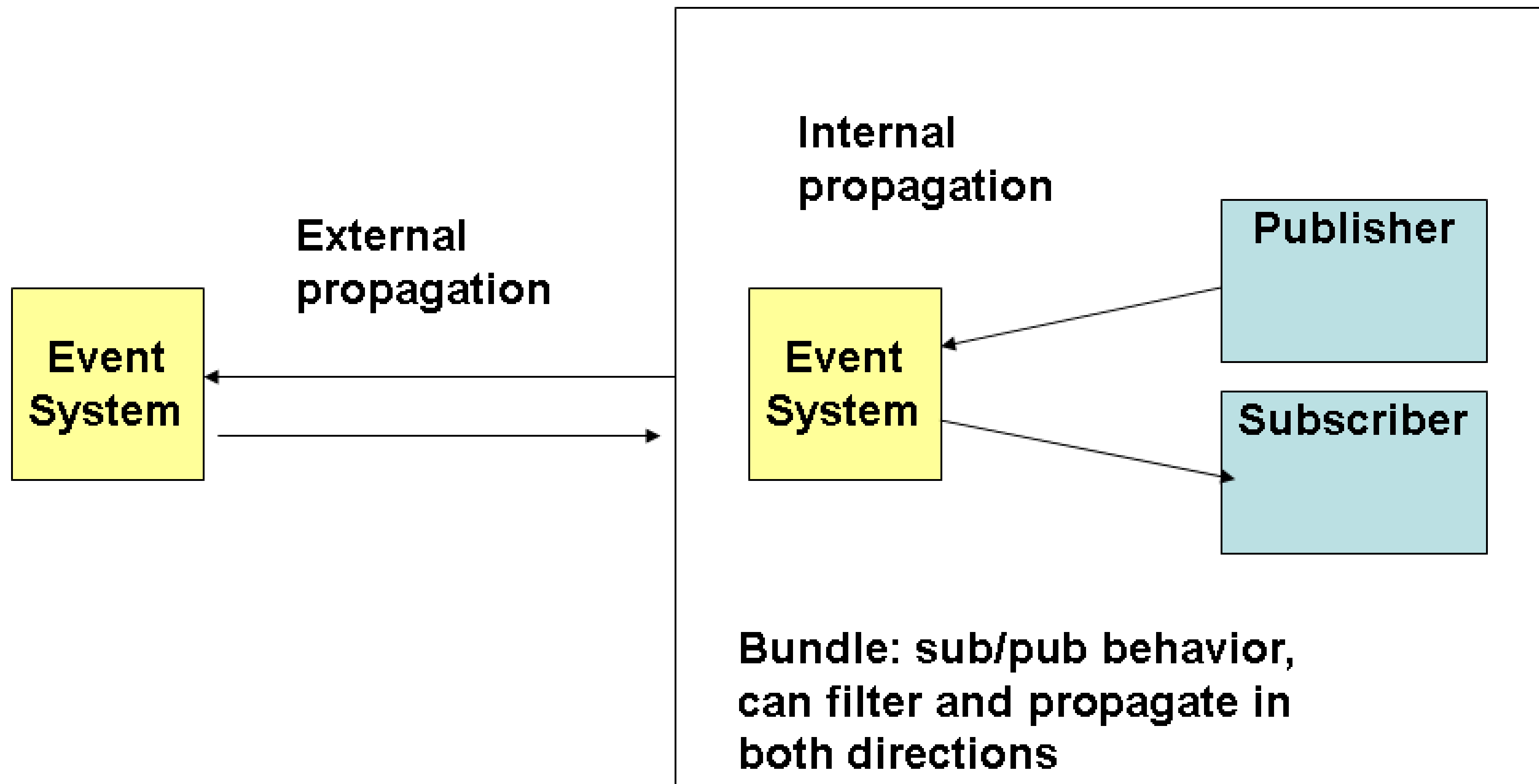
(see Mühl et.al. Pg. 130)

Multi-Source Information Distribution with Event-Encoding



Subscribers learn about the communication infrastructure. If the publisher topology changes (embedding into larger organization) the naming system needs to change. This shows how event tagging with source IDs create coupling between components.

Bundles/Scopes of Components



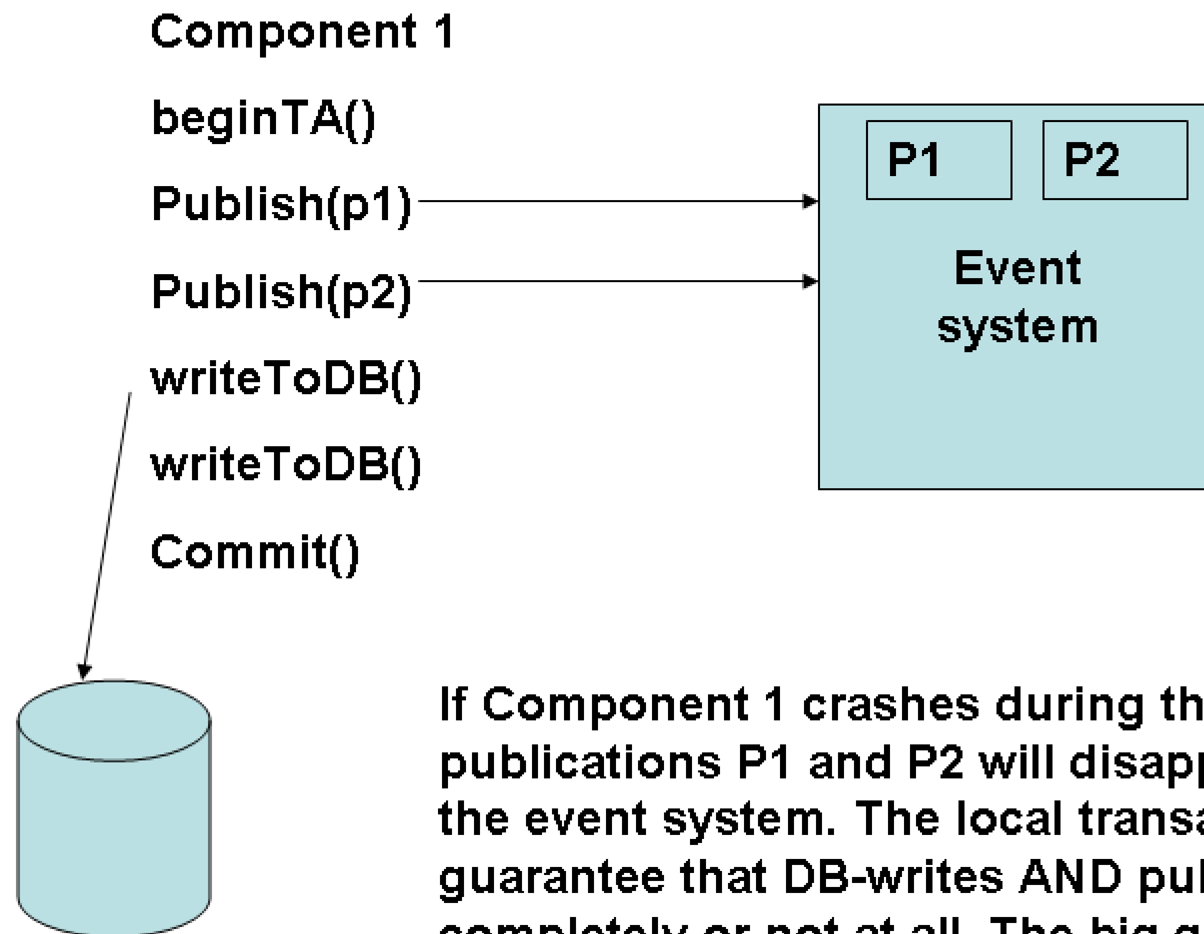
The bundle can use different mechanisms and policies internally. It controls propagation of events to and from the outside system. Scopes are a generalization of the bundle concept.

Unsolved problems: Sessions and Activities

Sessions need a way to relate notifications. Sessions should not be maintained by publishers or subscribers. Subscribers need a way to identify related, consecutive notifications

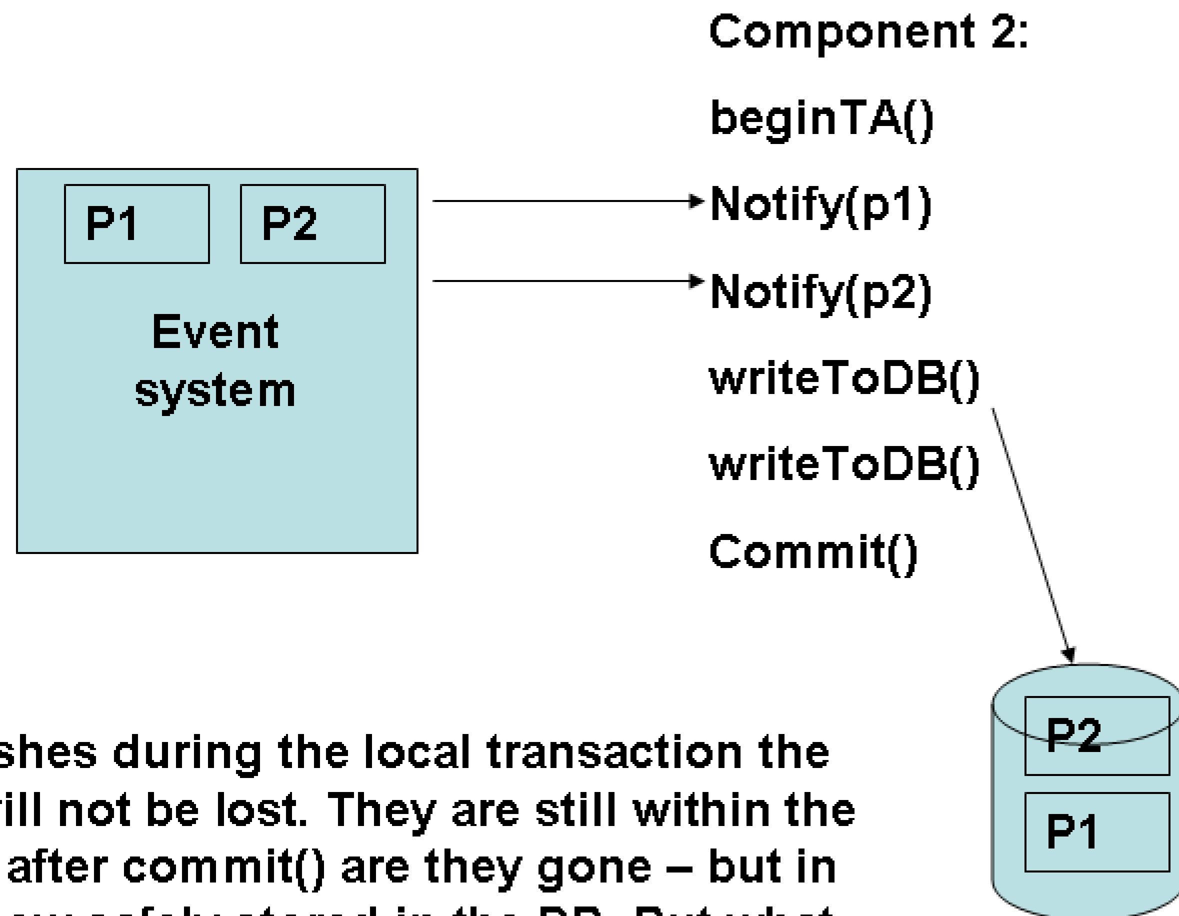
Activities are a representation of related events/notifications, like a transactions groups calls against resources into a unit of work.

Local Transactions in event-driven systems: Sender Side



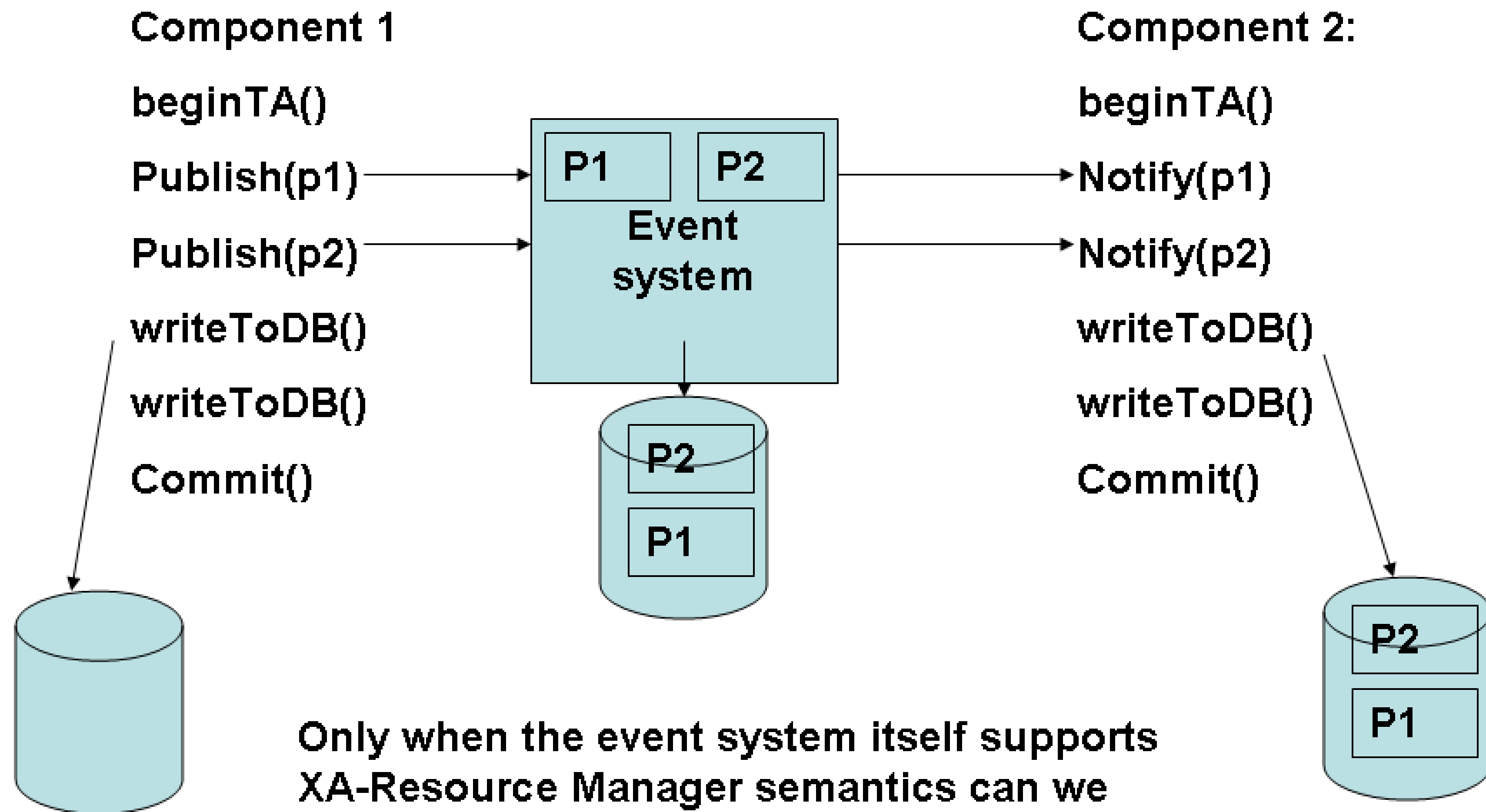
If Component 1 crashes during the transaction the publications P1 and P2 will disappear automatically from the event system. The local transaction semantics guarantee that DB-writes AND publications either finish completely or not at all. The big question is: what happens if the event system crashes AFTER the commit()?

Local Transactions in event-driven systems: Receiver Side



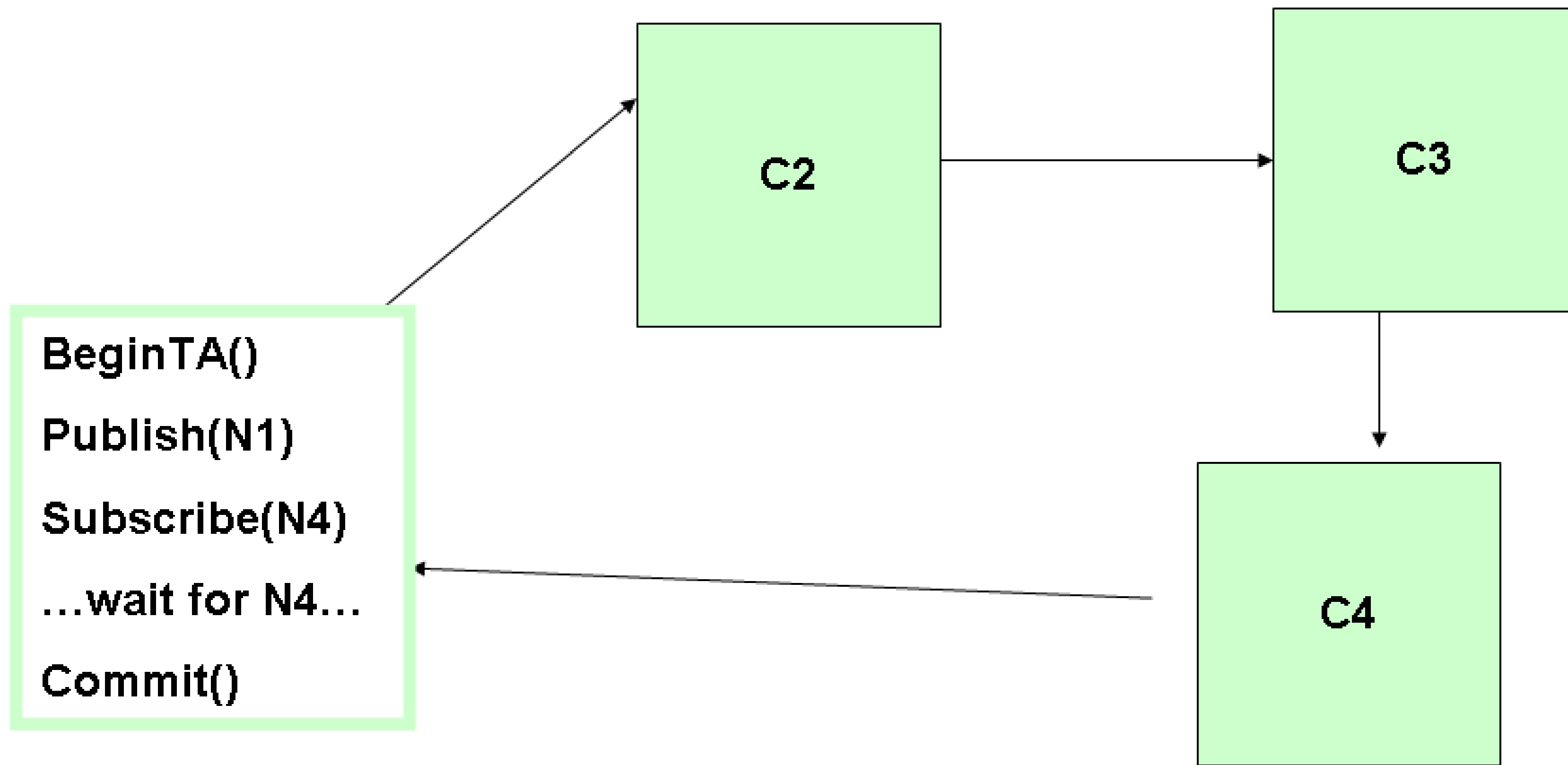
If Component 2 crashes during the local transaction the events P1 and P2 will not be lost. They are still within the event system. Only after commit() are they gone – but in that case they are now safely stored in the DB. But what happens if the event system crashes during the transaction?

External Transactions in event-driven systems



Only when the event system itself supports XA-Resource Manager semantics can we guarantee a once and only once semantics. The event system needs to store the events in safe storage like a DB.

Wrong feedback expectations within a transaction



Code that will not work. N4 causally depends on N1 being published by C1 and received by C2. But as publishing by C1 is done within a transaction the notification N1 does not become visible until the end of the transaction – which will not be reached because of the wait for N4 – which will never come.

End-to-End Transactions in event-driven systems?

Once a component commits it cannot recall the publishing of an event. Delivery of the event happens in a different transaction.

What if there are several events involved and together they form an activity? Much like a long running transaction there is always the option of compensation.

How is the concept of activity or unit-of-work expressed in event-driven systems? And how is such a thing revoked? CEP? Hierarchy? Scope?

Resources

Mühl et.al., Event-Driven-Systems