Message reliability and caching for publish/subscribe systems

Andreas Tanner, Gero Mühl

Technische Universität Berlin
Einsteinufer 17, 10587 Berlin, Germany
{tanner,gmuehl}@ivs.tu-berlin.de

March 5, 2004

Publish/subscribe

Message completeness

Temporal logic

Safety and Liveness

Axioms for p/s systems

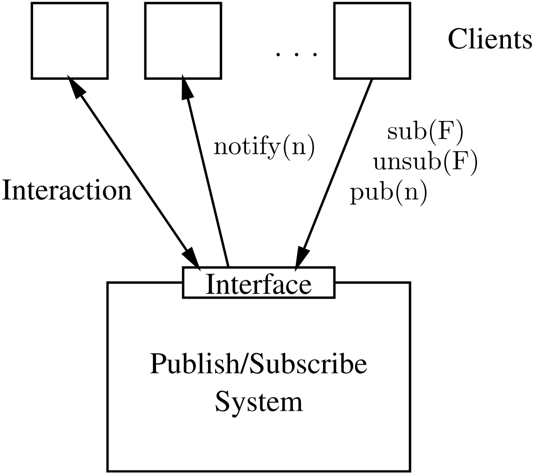
Distributed Implementaion

Rebeca

Publish/Subscribe

- Enables loosely coupled communication using notifications.
- Two kinds of "clients"
  - Producers publish notifications
  - Consumers subscribe to notifications
- Notification service
  - Decouples producers from consumers
  - Delivers a published notification to all consumers with a matching subscription

Interface



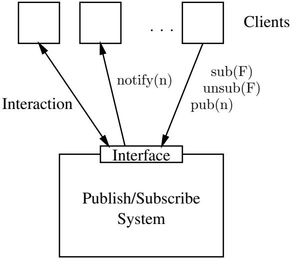
Clients
. . .
notify(n)
sub(F)
unsub(F)
pub(n)
Interaction
Interface
Publish/Subscribe
System

Basic Definitions and Assumptions

- A filter F is a mapping from the set of notifications $\mathcal{N}$ to the boolean values *true* and *false*.
- A notification n matches a filter F iff $F(n) = \textit{true}$.
- Set of notifications matched by a filter F and by a set of filters $\mathcal{A}$ is denoted by $N(F)$ and $N(\mathcal{A})$, respectively.
- Notifications are unique and can be published only once.

Black box view of a publish/subscribe system

- Describes the system behavior by solely looking at its interface.



- Interface Operations (set of all actions A)

$sub(Y, F)$	Client Y subscribes to filter F
$unsub(Y, F)$	Client Y unsubscribes to filter F
$ack(Y, F)$	System notifies client Y about message completeness guarantee
$notify(Y, n)$	System notifies client Y notified about n
$pub(X, n)$	Client X publishes n

Multicast

- publish/subscribe is special type of multicast
  - event-based
  - dynamically created groups
- problems classically considered for multicast:
  - message completeness
  - message order
  - etc.

Message completeness

How can *message completeness* be defined in a publish/subscribe system?

- *first try:* once a client subscribes to a filter, he gets all matching messages published thereafter

Message completeness

How can *message completeness* be defined in a publish/subscribe system?

- *first try:* once a client subscribes to a filter, he gets all matching messages published thereafter
- in an asynchronous system, this is not achievable if messages are not cached (subscription must first be propagated through the network)

Message completeness

How can *message completeness* be defined in a publish/subscribe system?

- *first try:* once a client subscribes to a filter, he gets all matching messages published thereafter
- in an asynchronous system, this is not achievable if messages are not cached (subscription must first be propagated through the network)
- *better:* after a client subscribes to a filter, there is a future time at which he is guaranteed to see all messages published thereafter

Message completeness

How can *message completeness* be defined in a publish/subscribe system?

- *first try:* once a client subscribes to a filter, he gets all matching messages published thereafter
- in an asynchronous system, this is not achievable if messages are not cached (subscription must first be propagated through the network)
- *better:* after a client subscribes to a filter, there is a future time at which he is guaranteed to see all messages published thereafter
- *even better:* after a client subscribes to a filter, there is a future time *the client is notified about* at which he gets message completeness guarantee

The Need for a formal treatment

- A formal specification
  - defines precisely what is expected from a correct system and
  - allows to reason about the correctness of an implementation.
- A formal treatment gives new insights that could otherwise be overlooked!
- *Linear Temporal Logic* offers a formalizm suitable for characterization of the behaviour of distributed systems

State

- *state* s is assignment $s = s : \mathcal{V} \ni v \mapsto v_s$ of the state variables $v \in \mathcal{V}$ to values in their domains

State

- *state* s is assignment $s = s : \mathcal{V} \ni v \mapsto v_s$ of the state variables $v \in \mathcal{V}$ to values in their domains
- *interface operations op* trigger state transitions

State

- *state* s is assignment $s = s : \mathcal{V} \ni v \mapsto v_s$ of the state variables $v \in \mathcal{V}$ to values in their domains
- *interface operations op* trigger state transitions
- a *trace* is a sequence of initial state s_0, followed by interface operations:

$$\sigma = s_0, op_1, op_2, \ldots, op_n \tag{1}$$

State

- *state* s is assignment $s = s : \mathcal{V} \ni v \mapsto v_s$ of the state variables $v \in \mathcal{V}$ to values in their domains
- *interface operations op* trigger state transitions
- a *trace* is a sequence of initial state s_0, followed by interface operations:
$$\sigma = s_0, op_1, op_2, \ldots, op_n \tag{1}$$
- predicate applied to trace σ refers to *first state* s_0 or *first operation* op_1

Temporal quantifiers

For some formula ϕ and $\sigma = s_0, op_1, op_2, \ldots, op_n$,

- $\Diamond\phi(\sigma)$ holds iff there exists i such that ϕ holds for the trace $s_i^\sigma, op_{i+1}, \ldots$,

Temporal quantifiers

For some formula ϕ and $\sigma = s_0, op_1, op_2, \ldots, op_n$,

- $\Diamond\phi(\sigma)$ holds iff there exists i such that ϕ holds for the trace $s_i^\sigma, op_{i+1}, \ldots$,
- $\Box\phi(\sigma)$ holds iff for all i, ϕ holds for the trace $s_i^\sigma, op_{i+1}, \ldots$,

Temporal quantifiers

For some formula ϕ and $\sigma = s_0, op_1, op_2, \ldots, op_n$,

- $\Diamond\phi(\sigma)$ holds iff there exists i such that ϕ holds for the trace $s_i^\sigma, op_{i+1}, \ldots$,
- $\Box\phi(\sigma)$ holds iff for all i, ϕ holds for the trace $s_i^\sigma, op_{i+1}, \ldots$,
- $\bigcirc\phi(\sigma)$ holds iff ϕ holds for the trace $s_1^\sigma, op_2, \ldots$.

LTL and Concurrency

- LTL is well-suited to describe concurrent systems
- A concurrent system is replaced by a nondeterministic sequential one.
- The concurrent execution of two operations in the real system is replaced in the model by the nondeterminism of which one occurs first.
- This type of nondeterminism is conceptually different from that studied in the area of automata theory ($\Rightarrow$ branching time).

LTL Examples

- $\Diamond\Box A$
- $\Box\Diamond A$
- $A \Rightarrow \Diamond B$
- $\Box[A \Rightarrow \Diamond B]$
- $\Box[A \Rightarrow \Box A]$
- $\Box[\Box A \Rightarrow \Diamond\Box B]$
- $\Box[A \vee \Box\neg A]$

State Variables in p/s systems

P_X	set of *published notifications*
S_Y^{ack}	set of *acknowledged subscriptions*
S_Y^{pend}	set of *pending subscriptions*
D_Y	multiset of *delivered notifications*

Initial values of state variables: $\emptyset$ for all of them

Effect of interface operations on state variables

$pub(X, n)$	$P'_X = P_X \cup \{n\}$
$sub(Y, F)$	$S_Y^{\mathrm{pend}'} = S_Y^{\mathrm{pend}} \cup \{F\}$
$unsub(Y, F)$	$S_Y^{\mathrm{ack}'} = S_Y^{\mathrm{ack}} \setminus \{F\}$; $S_Y^{\mathrm{pend}'} = S_Y^{\mathrm{pend}} \setminus \{F\}$
$ack(Y.F)$	$S_Y^{\mathrm{pend}'} = S_Y^{\mathrm{pend}'} \setminus \{F\}$; $S_Y^{\mathrm{ack}'} = S_Y^{\mathrm{ack}} \cup \{F\}$
$notify(Y, n)$	$D'_Y = D_Y \cup \{n\}$

Safety and Liveness

- Safety Conditions
  - Something "irremediably" bad will never happen.
  - Usually, phrased as an invariant of the system.
  - Usually, trivially satisfied by doing *nothing*.
  - General form: $\mathit{Init} \Rightarrow \Box\neg A$.
  - Violation can be detected after *finite* time.
  - E.g. partial correctness (Program never halts with wrong result)

Safety and Liveness (2)

- Liveness Conditions
  - Something “good” that should happen eventually happens.
  - Usually, trivially satisfied by doing *everything*.
  - General form: $\mathit{Init} \Rightarrow \Box[A \Rightarrow \Diamond B]$.
  - Violation can be detected after *infinite* time only.
  - Example: Termination (Program eventually halts)
- Many useful system properties (e.g., total correctness) can be expressed as the intersection of safety and liveness conditions.

Safaty axioms

A *publish/subscribe system* satisfies *message complete safety* if

- $$\Box\Big[\mathit{notify}(Y, n, X) \Rightarrow [\bigcirc\Box\neg\mathit{notify}(Y, n, X)]\Big] \qquad (2)$$

Safaty axioms

A *publish/subscribe system* satisfies *message complete safety* if

- $$\Box\Big[notify(Y, n, X) \Rightarrow \big[\bigcirc\Box\neg notify(Y, n, X)\big]\Big] \tag{2}$$

- $$\Box\Big[notify(Y, n, X) \Rightarrow \big[\exists X.\ n \in P_X\big]\Big] \tag{3}$$

Safaty axioms

A *publish/subscribe system* satisfies *message complete safety* if

- $$\Box\Big[notify(Y, n, X) \Rightarrow \big[\bigcirc\Box\neg notify(Y, n, X)\big]\Big] \tag{2}$$

- $$\Box\Big[notify(Y, n, X) \Rightarrow \big[\exists X.\ n \in P_X\big]\Big] \tag{3}$$

- $$\Box\Big[notify(Y, n, X) \Rightarrow \exists F \in S_Y^{pend} \cup S_Y^{ack}.\ n \in N(F)\Big] \tag{4}$$

Liveness axioms

A *publish/subscribe system* satisfies *message complete liveness* if

- $$\Box\Big[(sub(Y,F) \wedge \neg\Diamond unsub(Y,F)) \Rightarrow \Diamond[ack(Y,F)\Big] \qquad (5)$$

Liveness axioms

A *publish/subscribe system* satisfies *message complete liveness* if

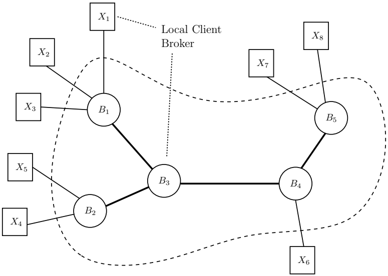
- $$\Box\Big[(sub(Y,F) \wedge \neg\Diamond unsub(Y,F)) \Rightarrow \Diamond[ack(Y,F)\Big] \qquad (5)$$

- $$\Box\Big[(ack(Y,F) \wedge \neg\Diamond unsub(Y,F)) \Rightarrow \big(\Diamond pub(X,n) \wedge n \in N(F) \Rightarrow \Diamond notify(Y,n,X)\big)\Big] \qquad (6)$$

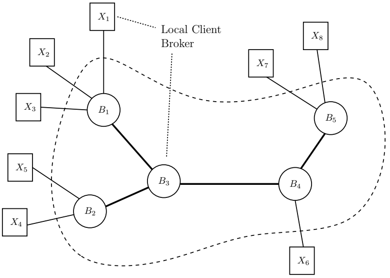
Correctness

A publish/subscribe system is *correct*, if it satisfies safety and liveness.

Distributed Implementation

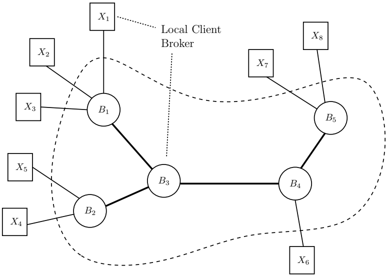


Distributed Implementation



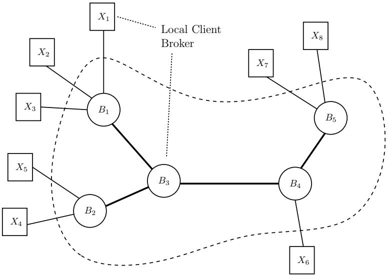
- Set of brokers $B_1, \ldots, B_n$ serving as access points.

Distributed Implementation



- Set of brokers $B_1, \ldots, B_n$ serving as access points.
- Brokers are concurrent processes cooperating by message passing.

Distributed Implementation



- Set of brokers $B_1, \ldots, B_n$ serving as access points.
- Brokers are concurrent processes cooperating by message passing.
- Each broker B manages a mutually exclusive set of local clients L_B and only communicates directly with its neighbor brokers N_B.

Assumptions

- Broker topology assumed to be acyclic.
- Channels are reliable (no corrupted, duplicated, lost, or spurious messages).
- Message latency is bounded.
- Communication with clients conceptually treated as message passing but assumed to be instantaneous.

Content-Based Routing Framework

- Each broker B manages a routing table T_B comprising a set of routing entries.

Content-Based Routing Framework

- Each broker B manages a routing table T_B comprising a set of routing entries.
- A routing entry is a pair (F, D) of a filter F and a destination $D \in L_B \cup N_B$.

Content-Based Routing Framework

- Each broker B manages a routing table T_B comprising a set of routing entries.
- A routing entry is a pair (F, D) of a filter F and a destination $D \in L_B \cup N_B$.
- Brokers exchange three kinds of messages:

Content-Based Routing Framework

- Each broker B manages a routing table T_B comprising a set of routing entries.
- A routing entry is a pair (F, D) of a filter F and a destination $D \in L_B \cup N_B$.
- Brokers exchange three kinds of messages:
  - *forward*(n) (used to disseminate notifications)

Content-Based Routing Framework

- Each broker B manages a routing table T_B comprising a set of routing entries.
- A routing entry is a pair (F, D) of a filter F and a destination $D \in L_B \cup N_B$.
- Brokers exchange three kinds of messages:
  - *forward*(n) (used to disseminate notifications)
  - *admin*$(\mathcal{S}, \mathcal{U})$ (used to update routing tables)

Content-Based Routing Framework

- Each broker B manages a routing table T_B comprising a set of routing entries.
- A routing entry is a pair (F, D) of a filter F and a destination $D \in L_B \cup N_B$.
- Brokers exchange three kinds of messages:
  - *forward*(n) (used to disseminate notifications)
  - *admin*($\mathcal{S}, \mathcal{U}$) (used to update routing tables)
  - *admin_ack*($\mathcal{S}, \mathcal{U}$) (used for acknowledgement between brokers)

Content-Based Routing Framework

- Each broker B manages a routing table T_B comprising a set of routing entries.
- A routing entry is a pair (F, D) of a filter F and a destination $D \in L_B \cup N_B$.
- Brokers exchange three kinds of messages:
  - *forward*(n) (used to disseminate notifications)
  - *admin*($\mathcal{S}, \mathcal{U}$) (used to update routing tables)
  - *admin_ack*($\mathcal{S}, \mathcal{U}$) (used for acknowledgement between brokers)
- Processing of *forward* and *pub* messages hardwired.

Content-Based Routing Framework

- Each broker B manages a routing table T_B comprising a set of routing entries.
- A routing entry is a pair (F, D) of a filter F and a destination $D \in L_B \cup N_B$.
- Brokers exchange three kinds of messages:
  - $forward(n)$ (used to disseminate notifications)
  - $admin(\mathcal{S}, \mathcal{U})$ (used to update routing tables)
  - $admin\_ack(\mathcal{S}, \mathcal{U})$ (used for acknowledgement between brokers)
- Processing of *forward* and *pub* messages hardwired.
- Processing of *admin* messages is customized by implementing an instance of the `administer` procedure.

Notification Forwarding

- If a broker receives a *forward*(n) / *pub*(n) message from a neighbor / local client, it sends a *forward*(n) / *notify*(n) message to all neighbors / local clients D for which there is a routing entry (F, D) with $n \in N(F)$.

Notification Forwarding

- If a broker receives a *forward*(n) / *pub*(n) message from a neighbor / local client, it sends a *forward*(n) / *notify*(n) message to all neighbors / local clients D for which there is a routing entry (F, D) with $n \in N(F)$.
- But a notification is never passed back to the neighbor it was received from.

Notification Forwarding

- If a broker receives a *forward*(n) / *pub*(n) message from a neighbor / local client, it sends a *forward*(n) / *notify*(n) message to all neighbors / local clients D for which there is a routing entry (F, D) with $n \in N(F)$.
- But a notification is never passed back to the neighbor it was received from.
- As the topology is acyclic, duplicate notifications are avoided.

Notification Forwarding

- If a broker receives a *forward*(n) / *pub*(n) message from a neighbor / local client, it sends a *forward*(n) / *notify*(n) message to all neighbors / local clients D for which there is a routing entry (F, D) with $n \in N(F)$.
- But a notification is never passed back to the neighbor it was received from.
- As the topology is acyclic, duplicate notifications are avoided.
- As every *notify*(Y, n) has a preceding *pub*(X, n), no spurious notifications are delivered.

Handling of *sub* and *unsub* messages

- Handling of subscription changes is parametrized on `administer` procedure.

Handling of *sub* and *unsub* messages

- Handling of subscription changes is parametrized on `administer` procedure.
- Paper gives criterias (*valid routing algorithms*) that when met, lead to correct system

Handling of *sub* and *unsub* messages

- Handling of subscription changes is parametrized on `administer` procedure.
- Paper gives criterias (*valid routing algorithms*) that when met, lead to correct system
- Various routing algorithms can be implemented via `administer`:

Handling of *sub* and *unsub* messages

- Handling of subscription changes is parametrized on `administer` procedure.
- Paper gives criterias (*valid routing algorithms*) that when met, lead to correct system
- Various routing algorithms can be implemented via `administer`:
  - Flooding

Handling of *sub* and *unsub* messages

- Handling of subscription changes is parametrized on `administer` procedure.
- Paper gives criterias (*valid routing algorithms*) that when met, lead to correct system
- Various routing algorithms can be implemented via `administer`:
  - Flooding
  - Simple routing

Handling of *sub* and *unsub* messages

- Handling of subscription changes is parametrized on `administer` procedure.
- Paper gives criterias (*valid routing algorithms*) that when met, lead to correct system
- Various routing algorithms can be implemented via `administer`:
  - Flooding
  - Simple routing
  - Routing with filter merging

Rebeca

- started by Ludger Fiege and Gero Mühl, TU Darmstadt as Java-based implementation of p/s system
- implemented along the lines of formal framework
- based on microkernel architecture with routing component as kernel
- ported to C# by Andreas Ulbrich
- uses events and delegates, seemlessly integrating into .NET framework
- extended (acknowledgements, caching) in student project winter 2003/2004 at TU Berlin

Rebeca

- started by Ludger Fiege and Gero Mühl, TU Darmstadt as Java-based implementation of p/s system
- implemented along the lines of formal framework
- based on microkernel architecture with routing component as kernel
- ported to C# by Andreas Ulbrich
- uses events and delegates, seemlessly integrating into .NET framework
- extended (acknowledgements, caching) in student project winter 2003/2004 at TU Berlin
- ...Rebeca promises to become the first ever publish/subscribe system whose correctness can be formally proven!

Future Work

- Routing in cyclic topologies
- Funnel functions
- Fault tolerance
- Adaptivity
- Integration of routing and composite event detection
- Streaming operators
- Quality of Service

Thank You.

Questions?

Andreas Tanner
FG Intelligent Networks and Management of Distributed Systems
Technische Universität Berlin
tanner@ivs.tu-berlin.de