Characteristics

- Hard to realize with RPC
  - Sender needs to know identities of all receivers
  - Sender communicates with a large and dynamically changing set of receivers
  - Senders and receivers may not be up simultaneously
  - Information dissemination blocks sender

- Many other applications
  - Stock information systems, auction systems, air traffic control, news ticker, alerting services, …
Pub/Sub Architecture

- Publishers
  - Entities that produce information
- Subscribers
  - Entities that consume information
- Broker
  - Delivers information from producers to consumers
- Events
  - Basic units of information
- Subscription
  - Expression of interest in certain events
Application Revisited

IBM: 100
UBS: 20

Ubiquitous subs
broker

subs
IBM?
UBS?

subscribers

events

IBMs
UBS
Events & Notifications

- Events
  - Asynchronous state transitions
  - Have a type: "stockvalue"
  - May have parameters: "IBM", "100"
    - Parameter names: "company", "value"
    - Parameter types: string, integer
    - Special parameters: time, location

- Representation
  - Plain text, XML, binary encoding, ...
  - Example: stockvalue(string company="IBM", int value=100)

- Notifications
  - Messages carrying event representations
Broker

- **Main tasks**
  - (De)registration of publishers / subscribers
  - Matching of events with subscriptions
  - Routing of events from publishers to matching subscribers

- **Different implementations**
  - Centralized vs. Distributed
  - Simplicity vs. Scalability
Benefits

- Decoupling of publishers and subscribers
  - **Space**: no need to be connected or even know each other (anonymous)
  - **Time**: no need to be up at the same time
  - **Synchronization**: publishing and receiving events are asynchronous operations

- Suitable for large, dynamic systems
  - Many participating entities
  - Entities frequently enter and leave the system
Event Matching

- Key operation to identify subscribers to which an event must be delivered
  - Input: event $e$, set of subscriptions $S$
  - Output: all $s \in S$ that match $e$

- Depends on the semantics of subscriptions
  - Type-based: type of the event
  - Channel-based: cf. multicast group
  - Topic-based: cf. news group
  - Content-based: parameters of events
Type-Based Subscriptions

- Subscriptions specify the type of the requested events
  - E.g., „stockvalue“

- Event matching
  - Comparison of event type and subscription type
Channel-based Subscriptions

- Multiple, named broadcasts channels can be created
  - E.g., news, stock, ...
- Publishers publish events to a specific channel
  - Events of different types may be published to a single channel
- Subscriptions specify the desired channel
- Event matching
  - E.g., a multicast group for each channel

![Diagram of channel-based subscriptions]
Topic-Based Subscriptions

- Each event has a „topic“ parameter
- Topics are organized in a hierarchy
- Subscriptions specify a topic of interest
- Event matching:
  - $s$ matches $e$ if topic of $e$ is descendent of topic of $s$ in the hierarchy

Diagram:

```
stock
  ↓
IT
  ↓
IBM
  ↓
UBS

financial
  ↓
banks
  ↓
insurance
  ↓
CS
```
Content-Based Subscriptions

- Subscriptions refer to type and parameters values of events
  - E.g., type=„stockvalue“ \( \land \) company=„UBS“ \( \land \) value < 100
  - Often a conjunction over atomic predicates:
    \((p_1 \ op_1 \ v_1) \land (p_2 \ op_2 \ v_2) \land \ldots\)

- Event matching
  - s matches e if the predicate of s holds on the parameter values of e
  - Algorithms to process multiple subscriptions with much lower overhead
    - If „value < 100“ doesn‘t match, then „value < 20“ won‘t match as well
Event Routing

- Nontrivial problem with distributed brokers for content-based subscriptions
  - Cannot use address-based routing!
- Many different approaches
  - Hierarchical vs. peer-to-peer
  - Flooding vs. overlay
A Distributed Broker Model

- Subscribers and publishers connect to a local broker
- Brokers maintain connections to some, but not all other brokers
  - Form connected (acyclic?) network
- Routing problem:
  - Broker receives an event from either a publisher or another broker
  - To which brokers (and subscribers) should the event be sent?
Event Flooding

- If a broker receives an event $e$ from $n$
  - Do nothing if $e$ has been received before
  - Send $e$ to all neighboring brokers except $n$
  - Send $e$ to all matching subscribers
- Ensures that all brokers will receive each event
Example

- Event: $e(v=\ldots)$
- Content-based subscription: $v<100$
Subscription Flooding

- Idea: Use subscriptions to compute routing paths for events
  - This causes overhead whenever a subscriptions is added / removed / changed
  - But pays off if subscriptions change rarely compared to number of events published

- Approach: each broker maintains a routing table of entries \((n_i, s_i)\)
  - \(n\) = neighboring broker, \(s\) = subscription
  - Send \(e\) to all \(n_i\) where \(s_i\) matches \(e\)

- Setup by subscription flooding: When a broker receives \(s\) from \(n\)
  - Do nothing if \(s\) has been received before
  - Create routing table entry \((n, s)\)
  - Send \(s\) to all neighboring brokers
Example

- Event: \( e(v=...) \)
- Content-based subscription: \( v<100 \)
Covered Subscriptions

- **Observations**
  - Flooding the complete network for each subscription
  - Each broker has one routing table entry per subscription

- **Idea: exploit redundancy of subscriptions**
  - E.g., events matching "v<20" also match "v<100"
  - We say: "v<100" covers "v<20"

- **Modified route setup: When a broker receives s from n**
  - Do nothing if s has been received before
  - Do nothing if there is an entry \((n, s_i)\) and s is covered \(s_i\)
  - Create routing table entry \((n, s)\)
  - Send s to all neighboring brokers
Example

How to handle Unsubscriptions?

Beware of loops
Subscription Merging

- Goal: reduce size of the routing table
- Approach: Merge two routing table entries with same destination into one
  - Perfect: (1, v<10), (1, v>10) -> (1, v≠10)
  - Imperfect: (1, v<10), (1, v>11) -> (1, v≠10)
Advertisements

- Idea: Publishers announce details on the events they will generate, e.g. \( \nu < 500 \)
  - Assuming this won’t change over time

- Approach: Flood advertisements, building a second routing table
  - Subscriptions are no longer flooded, but routed using the advertisement routing table
  - Instead of checking if an event matches a subscription, need to check if a subscription overlaps an advertisement
    - E.g., \( \nu < 500 \) and \( \nu < 100 \) overlap, but not \( \nu < 500 \) and \( \nu > 600 \)
Example

- Advertisement: $v<500$
- Subscription: $v<100$

What if advertisement after subscription?
What if almost concurrently?
Further Issues

- Dynamic networks
  - Joining, leaving, crashing brokers

- Intermittent connectivity
  - Mobile networks with roaming publishers / subscribers
  - Event buffering

- Quality of Service
  - Event ordering, latency, guaranteed delivery, security, ...

- Composite events
  - A and B, A followed by B, ...