Technische Universität Darmstadt

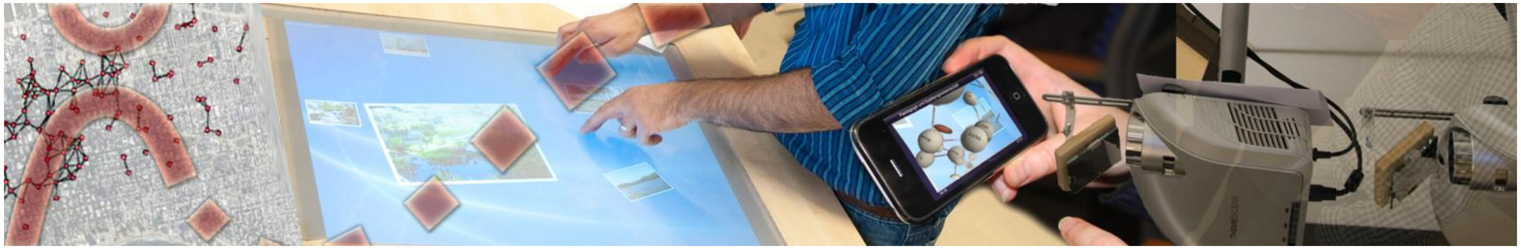
Telecooperation Lab
Prof. Dr. Max Mühlhäuser

TK1: Distributed Systems - Programming & Algorithms

Chapter 2: Distributed Programming

Section 2: Advanced Paradigms

Lecturer: Prof. Dr. Max Mühlhäuser



2.2: ADVANCED PARADIGMS

(1) Event based (Publish/Subscribe)

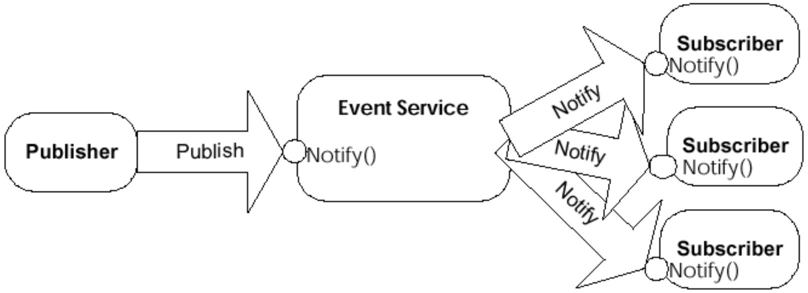
Outline

- Interaction models in distributed systems
  - Callbacks
  - Message Queues
  - Publish/Subscribe

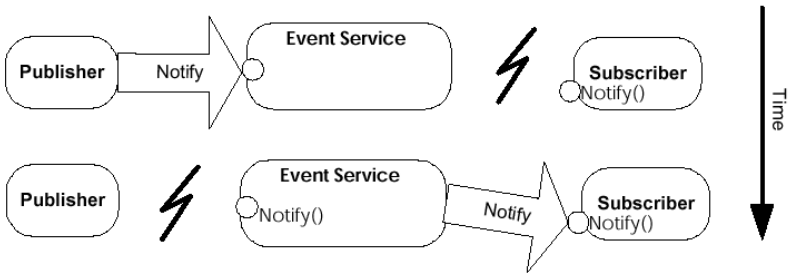
- Publish/Subscribe Systems
  - Classification
  - Addressing
  - Subscription Mechanisms
  - Data and Filter Models
  - Filter covering, overlapping, and merging
  - Distributed Event Systems
  - Routing

Towards loosely coupled systems

1. Space decoupling
   - parties don‘t know each other
   - 1-to-many comm. possible

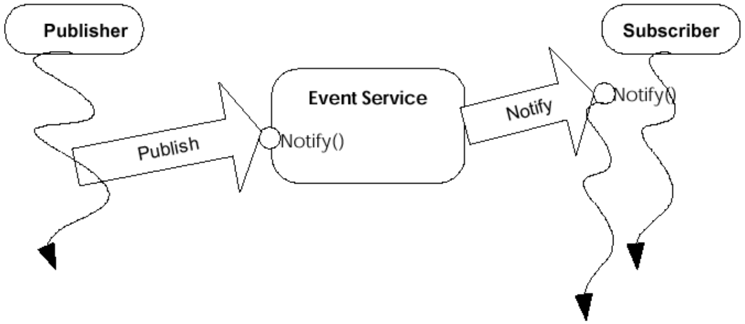


2. Time decoupling:
   - parties not (necessarily) active at same time



3. Flow decoupling
   - event production & consumption ∉ main control flow

- 1, 2, 3: coordination & synchronization drastically reduced



Interaction Models

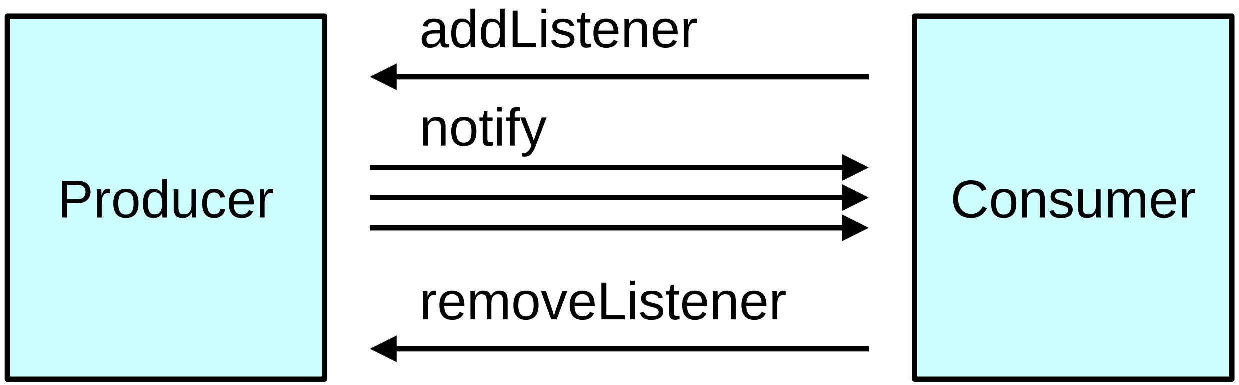
- Interaction Models in Distributed Systems can be classified according to
  - Who **initiated** the interaction
  - How the communication partner is **addressed**

	Consumer-initiated („pull“)	Provider-initiated („push“)
Direct Addressing	Request/Reply	Callback
Indirect Addressing	Anonymous Request/Reply	Event-based

- Provider: provides data or functionality
- Anonymous Request/Reply (rare): provider is selected by communication system and not specified directly (e.g., IP Anycast)

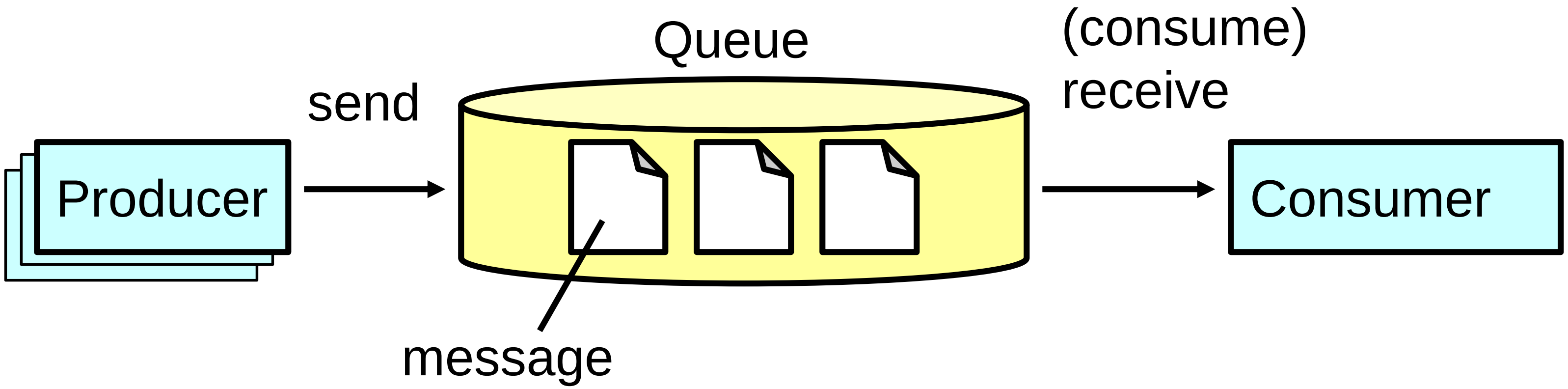
Concepts: Callbacks

- Synchronous (remote) method calls often used to emulate behavior of event based systems
  - See also: Observer Design Pattern
  - Frequently used in GUI toolkits; example:



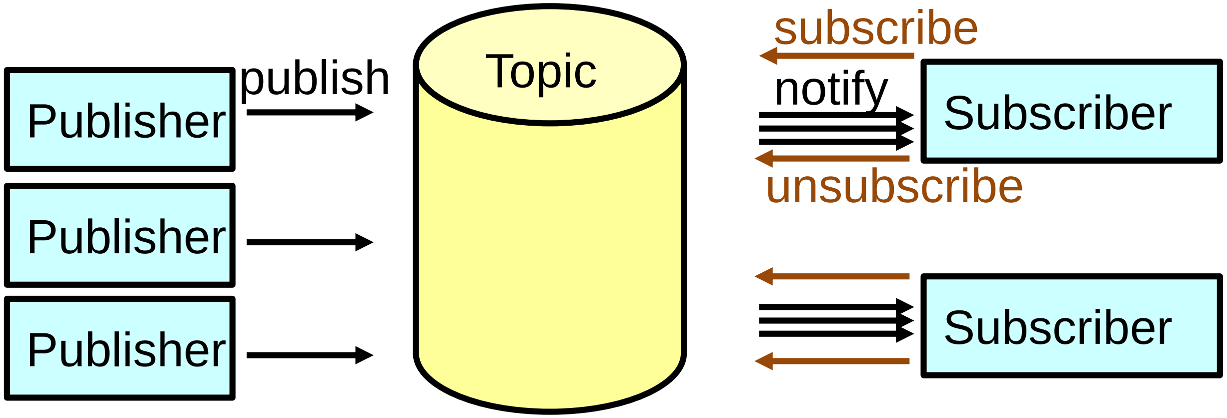
  - Producer & Consumer coupled in space and time, decoupled in flow
  - Producers have to take care of subscription management and error handling

Concepts: Message Queues



- No timing dependencies between sender and receiver
- Queue stores message (persistently), until
  - It is read by a consumer
  - The message expires (Leases)

Concepts: Publish/Subscribe



- Here: *Topic based* Publish/Subscribe
  - Interested parties can “subscribe” to a topic (channel)
  - Applications “publish” i.e. post messages explicitly to specific topics
- Each message may have multiple receivers
- Full decoupling in space, time, and flow
- Terminology:
  - **Event:** *observable “*happening in real world” or “state change inside information system”
  - **Notification:** The reification of an event as a data structure
  - **Message:** Transport container for notifications and control messages

Outline

- Interaction models in distributed systems
  - Callbacks
  - Message Queues
  - Publish/Subscribe

- Publish/Subscribe Systems
  - Classification
  - Addressing
  - Subscription Mechanisms
  - Data and Filter Models
  - Filter covering, overlapping, and merging
  - Distributed Event Systems
  - Routing

Publish/Subscribe - Classification (1)

- Messaging Domain
  - Subscription based Pub/Sub
  - Advertisement based Pub/Sub

- Subscription [or: Addressing] Mechanism (3 or 6 – depends on how you count)
  - Channel based (‘extensions’: topic based, type based)
  - Subject based
  - Content based (‘abstraction’: concept based)

Publish/Subscribe - Classification (2)

- Server Topology
  - Single Server (Elvin3)
  - Hierarchical (TIB/Rendezvous, JEDI, Keryx)
  - Peer-to-Peer (‘Acyclic’ or ‘Generic’ as in SIENA)
  - Generic Peer-to-Peer (SIENA)
  - Hybrid
- Event Data Model (major categories):
  - Untyped
  - Typed
  - Object-oriented
- Event Filtering
  - Expressiveness and flexibility of subscription language
    - Simple Expressions – SQL-like Query Language – (Mobile) Code
  - Pattern Monitoring: Temporal sequence of events
  - Evaluated in router network

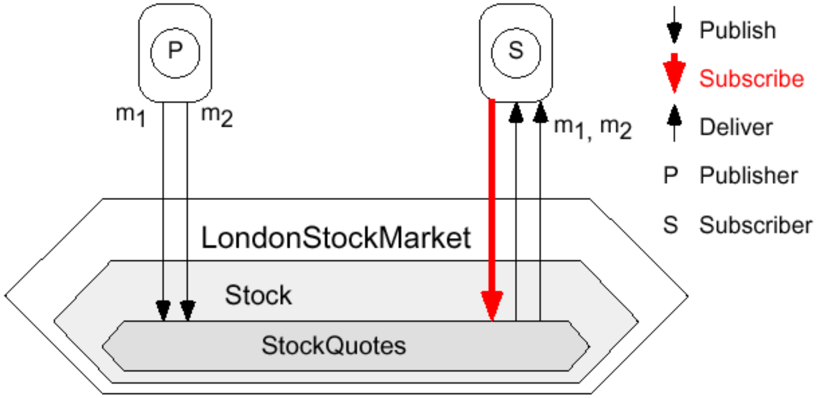
Publish/Subscribe - Classification (3)

- Note: Scalability <-> Expressiveness Tradeoff
  - Simple Expressions permit Filter Merging → better scalability

- Note: Further description wrt. useful features:
  - Scalability
  - Security
  - Client Mobility
    - Transparent
    - Native / external
  - Disconnection
  - QoS
    - Reliability
      - Response Time (Real-Time Constraints)
  - Transactions
  - Exception Handling

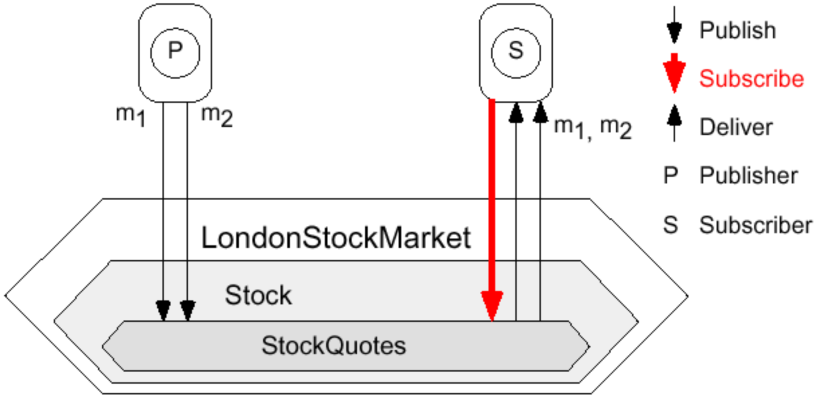
Addressing: Channels & More

- Channel based addressing (≈ topic based)
  - Interested parties can subscribe to a channel
  - Application posts messages explicitly to a specific channel
  - Channel ID is only part of message visible to event service
  - There is no interplay between two different channels



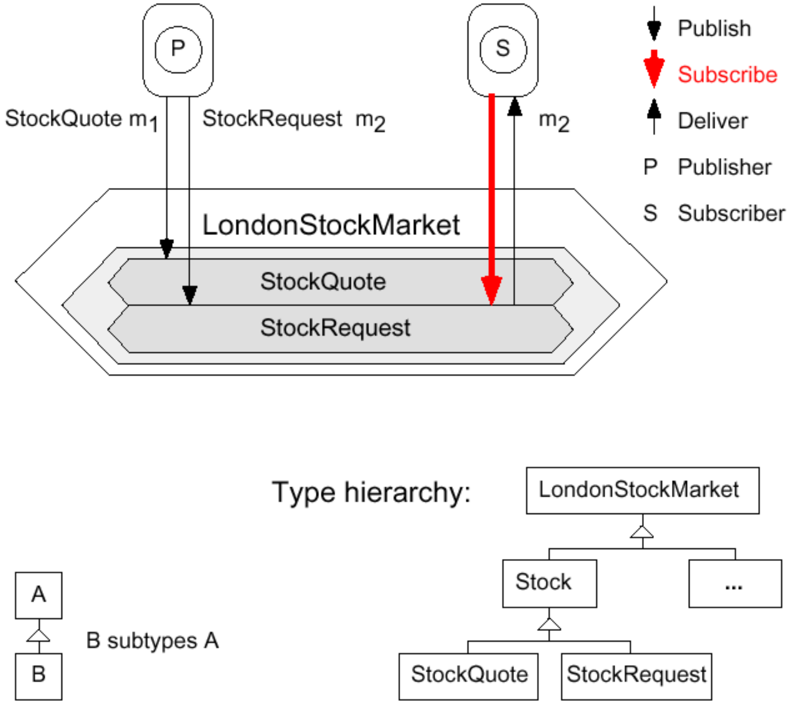
Addressing: Channels & More

- Topic based can be viewed as extension:
  - Cf. topic hierarchies, e.g.,in SwiftMQ: <roottopic>.<subtopic>.<subsubtopic>
  - Messages published to addressed node and all subnodes
    - iit.sales -> iit.sales.US, iit.sales.EU
  - Subscribing means receiving messages addressed to
    this node, all parent nodes and all sub nodes (!):
    - Subscription to iit.sales
    - … receives from: iit, iit.sales, iit.sales.US, iit.sales.EU
    - … but not from: iit.projects
  - Subscriber receives each message only once



Addressing: Channels & More

- Type based addressing
  - Similar to channel based Pub/Sub with hierarchies, i.e. to topic-based
  - Supports subtype tests (instance of)
  - Good integration of middleware & language; type safety

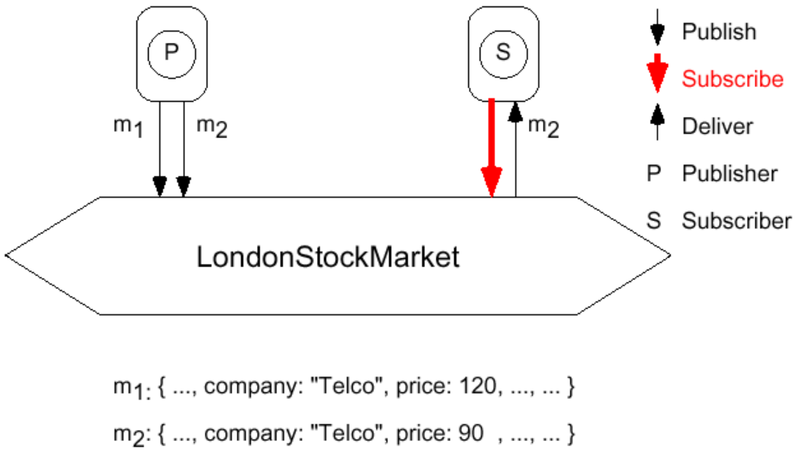


Addressing: Subjects

- Subject based addressing (think of classified eMail subjects):
  - Notifications contain a well-known attribute – the subject
  - Subject determines ‘address’ (not unlike channels, types)
  - Subscriptions express interest in subjects by some form of expressions to be evaluated against the subject
  - Subject may be, e.g.:
    - List of strings (TIB/Rendezvous, JEDI)
    - Properties: Typed Key/Value-Pairs (JMS)
  - Subject (= header of notification) is visible to event service, body is opaque
  - Subscription is
    - (limited form of) Regular Expressions over Strings (TIB, JEDI)
    - (limited form of) SQL92 Queries (JMS)
  - Filtering is done in the Router Network!
  - Limited form of Content based Subscription

Addressing: Content & More

- Content based subscription
  - Domain of filters extended to the whole content of notification
  - More freedom in encoding data upon which filters can be applied
  - More information for event service to set up routing information


P
S
m1
m2
m2
LondonStockMarket
Publish
Subscribe
Deliver
P Publisher
S Subscriber
m1: { ..., company: "Telco", price: 120, ..., ... }
m2: { ..., company: "Telco", price: 90 , ..., ... }

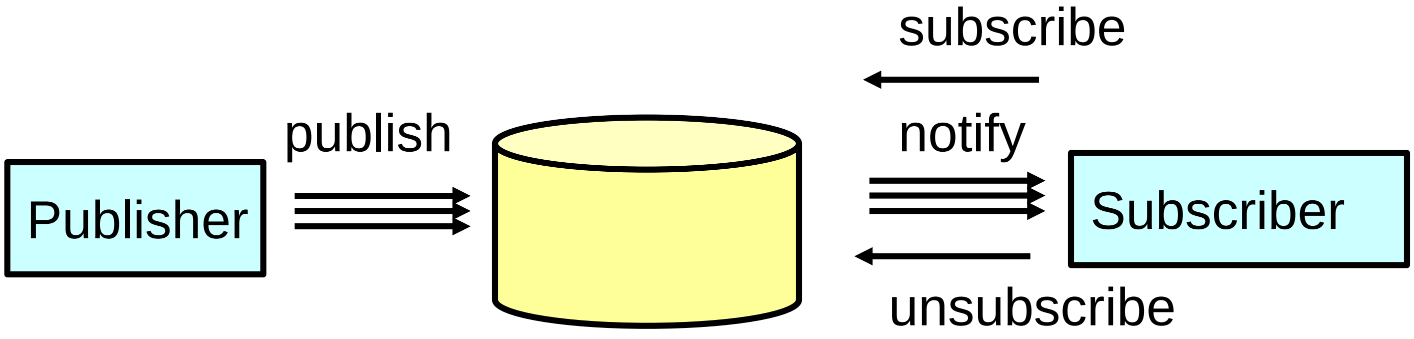
Adressing: Content & More

- Notes: content based vs. subject based
  - Subject based: requires some preprocessing by publisher
    - Information that might be relevant for subscribers for filtering must be placed in header fields
    - Hence, producer makes assumptions about subscribers‘ interests
  - Content based, on the contrary:
    - Subscribers exclusively describe their interests in filter expressions
  - However: this comes at a high cost:
    - Parsing entire content
    - Heavy load on routers for routing topology setup & forwarding

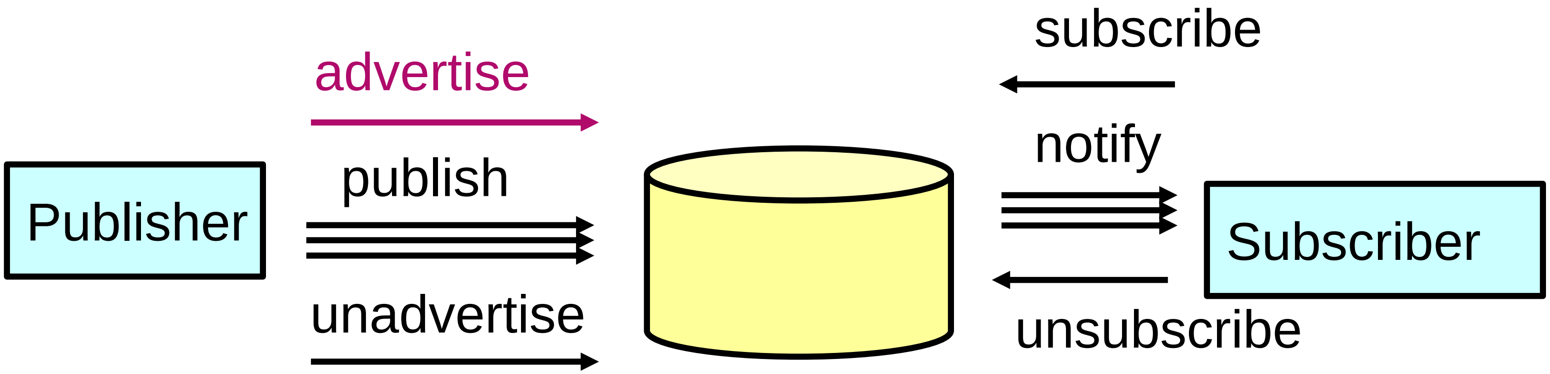
- Concept based addressing
  - Provides higher level of abstraction for description of subscribers’ interests
  - Matching of notifications and transformation of notifications based on ontologies

Subscription Mechanisms

- **Subscription based**



- **Advertisement based**



Data Model

- Notification

  consists of a nonempty set of attributes $\{a_1, \ldots, a_n\}$

  An attribute is a triple $a_i = (n_i, t_i, v_i)$ where

  - n_i is the attribute name
  - t_i is the attribute type, and
  - v_i is the value

- All data models can be mapped to this representation
  - Hierarchical messages in which attributes may be nested are flattened by using a dotted naming scheme, e.g.,

    $\{(pos, set, \{(x, int, 1), (y, int, 2)\})\}$ can be rewritten as

    $\{(pos.x, int, 1), (pos.y, int, 2)\}$

  - Object structures: e.g.,
    - Assign IDs to objects: (obj1.id, int, 1)
    - Introduce *reference type* to describe references: (obj2.p, ref, 1)

Attribute Filters

- An *attribute filter* is a simple filter that imposes a constraint on the value and type of a single attribute. It is defined as a tuple

$$A = (n, t, op, c)$$

- Where
  - *n* is the name of the attribute to test
  - *t* is the expected value type,
  - *op* is the test operator, and
  - *c* is a constant that serves as parameter for the operator

- An attribute *a* matches an attribute filter *A*, iff

$$a \sqsubseteq A :\Leftrightarrow n_A = n_a \wedge t_A = t_a \wedge op_A(v_a, c_A)$$

Filters

- A **filter** is composed of one or more *attribute filters*. While attribute filters are applied to single attributes, filters are applied to whole notifications

- Filters that only consist of a single attribute filter are called **simple filters**, i.e., $F = \{ A_1 \}$

- Filters containing multiple attribute filters are called **compound filters**, i.e. $F = \{ A_1, \ldots, A_n \}$
  - In the following, we only consider compound filters that only use conjunctions (also called conjunctive filters).

- Arbitrary logic expressions can be written as conjunctive filters in one or multiple subscriptions

- A notification *n* **matches** a filter *F*, iff it satisfies all attribute filters of *F*:

$$n \sqsubseteq F :\Leftrightarrow \forall A \in F : \exists a \in n : a \sqsubseteq A$$

Matching: Example

Message		Filter
String event=alarm	*matches*	String event=alarm
Time date=02:40:03		
String event=alarm	*not matches*	String event=alarm
Time date=02:40:03		Integer level>3

Covering

- Covering between attribute filters:
  - An attribute filter A_1 *covers* another attribute filter A_2, iff

$$A_1 \sqsupseteq A_2 :\Leftrightarrow n_1 = n_2 \wedge t_1 = t_2 \wedge L_A(A_1) \supseteq L_A(A_2)$$

  - where L_A is the set of all values that cause an attribute filter to match

$$L_A(A_i) = \{v \mid op_i(v, c_i) = true\}$$

- Covering between filters:
  - A filter F_1 *covers* another filter F_2, iff for each attribute filter in F_1 there exists an attribute filter in F_2 that is covered by the attribute filter in F_1:

$$F_1 \sqsupseteq F_2 :\Leftrightarrow \forall i \, \exists j : A_{1,i} \sqsupseteq A_{2,j}$$

- The covering relation is required for identifying and merging similar filters

Overlapping

- The filters F_1 and F_2 are *overlapping*, iff

$$F_1 \sqcap F_2 :\Leftrightarrow$$

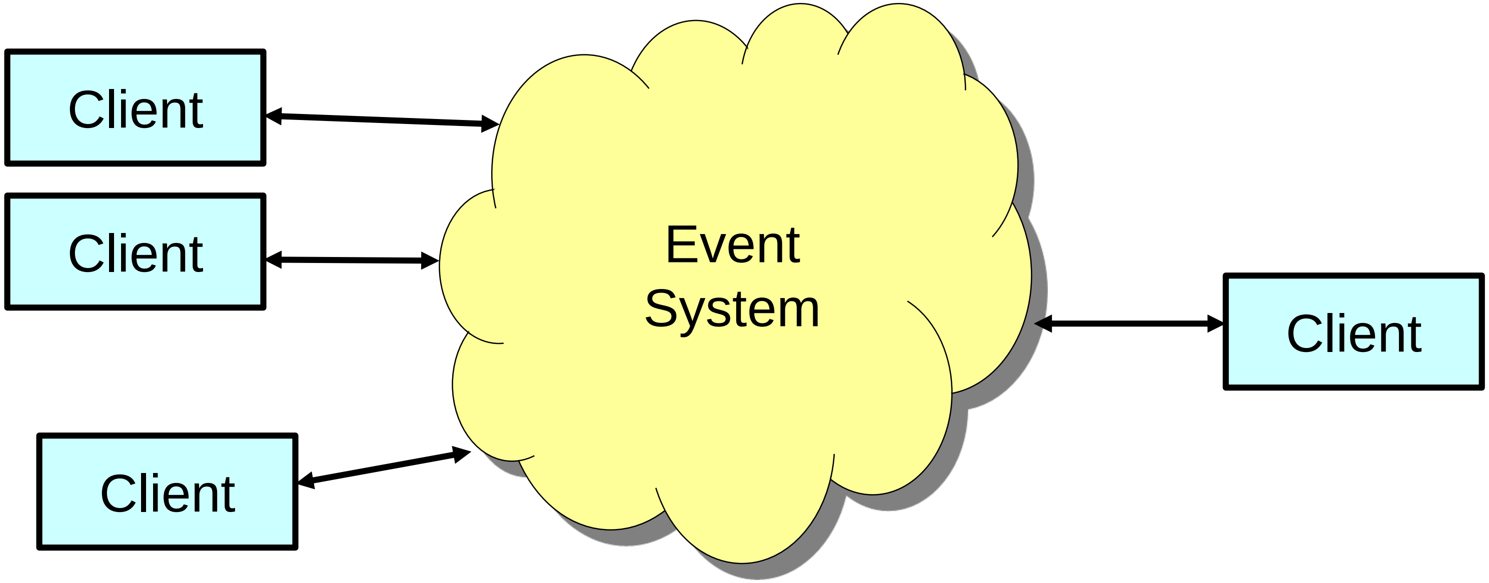
$$\neg\exists A_{1,i}, A_{2,j} : (n_{1,i} = n_{2,j} \land (t_{1,i} \neq t_{2,j} \lor L_A(A_{1,i}) \cap L_A(A_{2,j}) = \emptyset))$$

- The overlapping relation is required to implement advertisements.

- When an advertisement A overlaps with a subscription S, we say that A **is relevant** for S.
- As a consequence, all notifications published by the client that issued A must be forwarded to the clients that issued S.

Distributed Event Systems

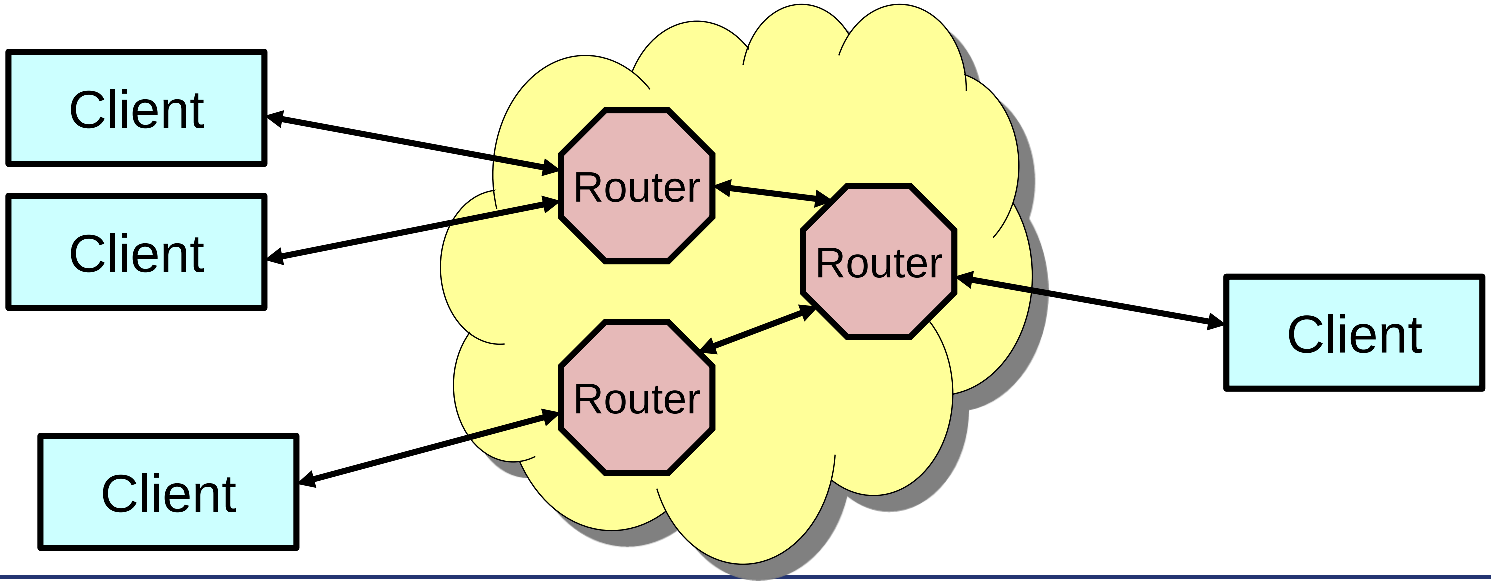
- (Distributed) Event Systems
  - Permit loosely coupled, asynchronous point-to-multipoint communication patterns
  - Represent application independent infrastructures
  - Quasi ‘clients-only’, communicating via a *logically* centralized component



Distributed Event Systems

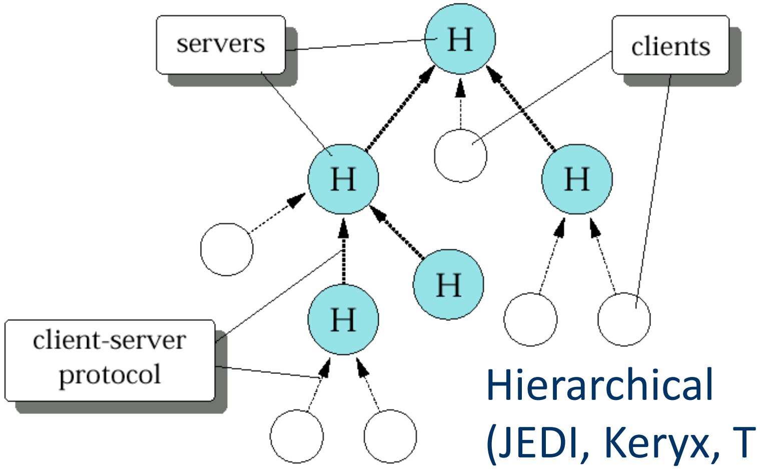
- Logically centralized component
  - Single server *or* network of event routers
  - Transparent for application (=Client)
  - Router network can be reconfigured independently and without changes to the application

    => Scalability

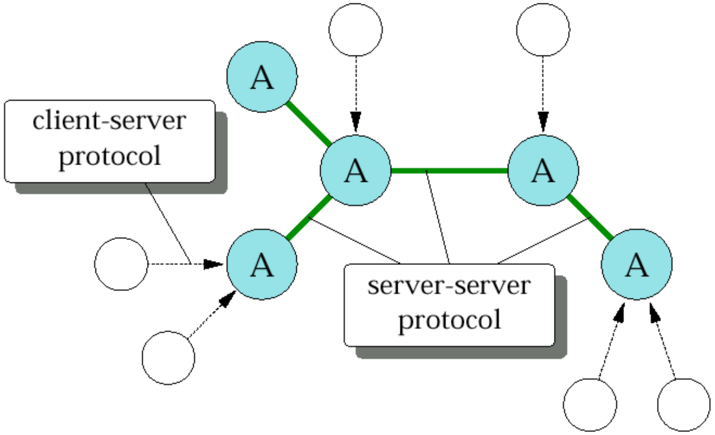


Router Topologies

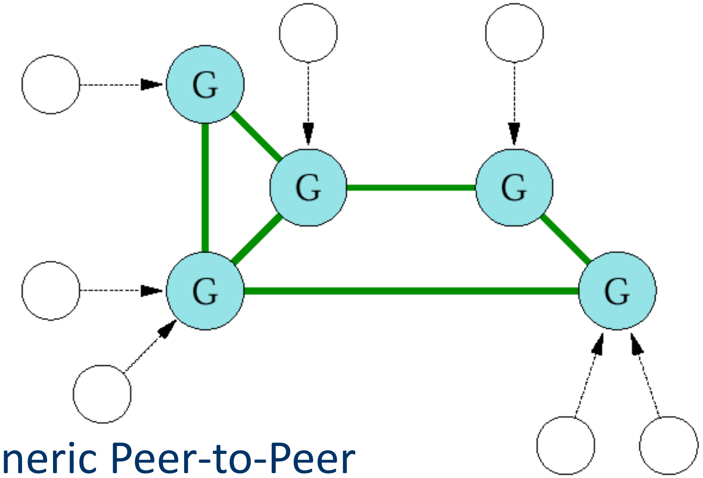
Centralized Server
(Elvin3)



Hierarchical
(JEDI, Keryx, TIB)



Acyclic Peer-to-Peer



Generic Peer-to-Peer
(SIENA)

Routing of Requests

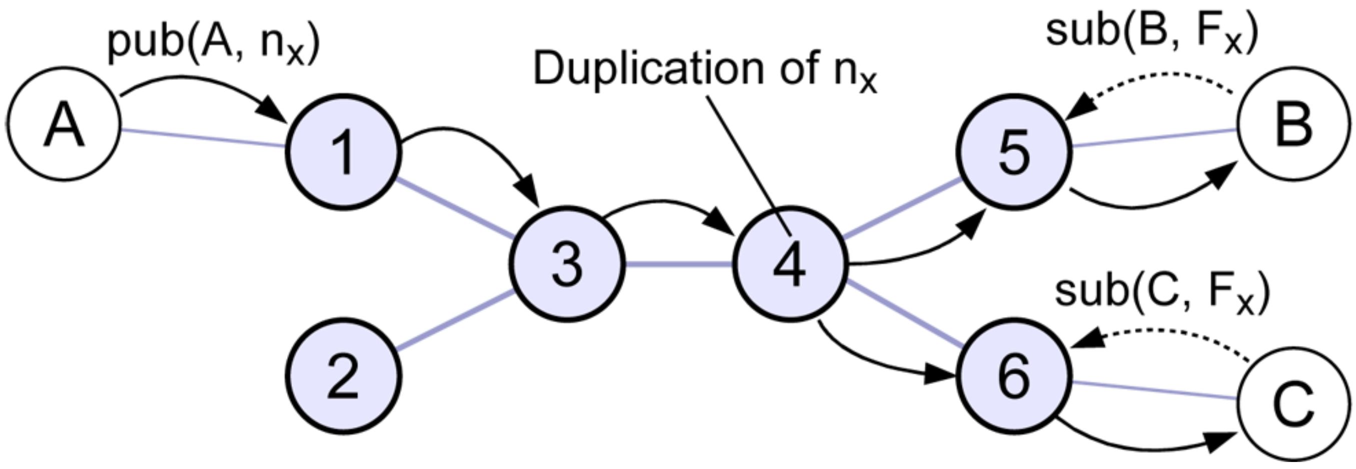
- The network of brokers forms an overlay network

- Routing can be split up into two layers
  - At the lower level, requests, i.e. control and data **messages** must be routed between brokers
  - At the higher level, **notifications** must be routed according to subscriptions and advertisements

- Routing algorithm depends on overlay structure
  - Unstructured, generic peer-to-peer networks must avoid routing messages in cycles, e.g., use
    - Variants of Distance Vector Routing
    - Spanning Tree
  - Structured peer-to-peer networks, e.g., use
    - Distributed Hash Tables

Routing: Principles

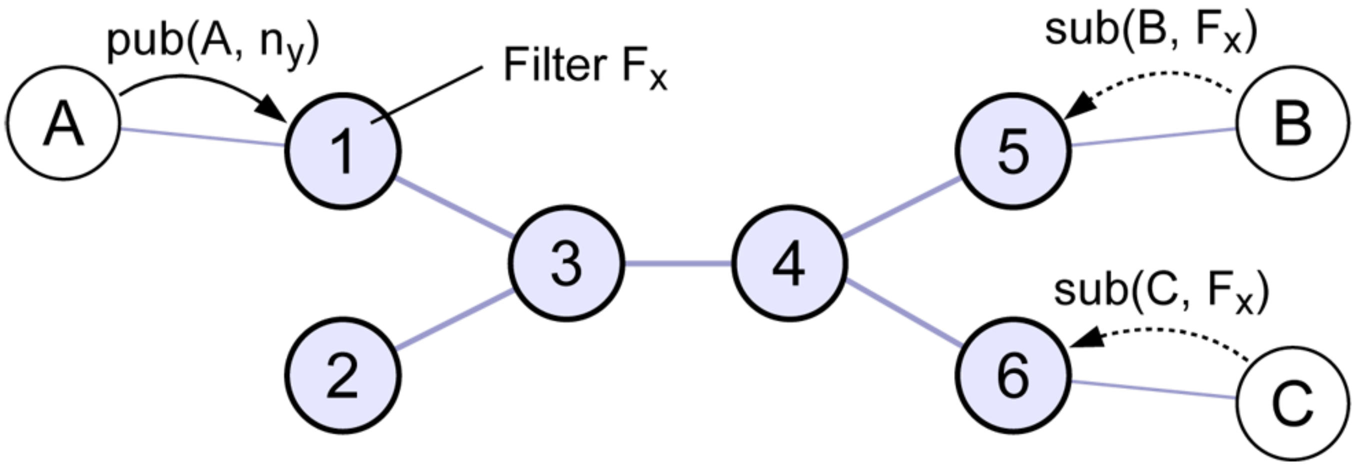
- Downstream duplication
  - Route notification as a single copy as far as possible



  - Clients B, C subscribe at routers 5, 6 with filter F_x
  - Client A publishes notification n_x (which is covered by F_x) to router 1
  - The notification is replicated not before router 4

Routing: Principles

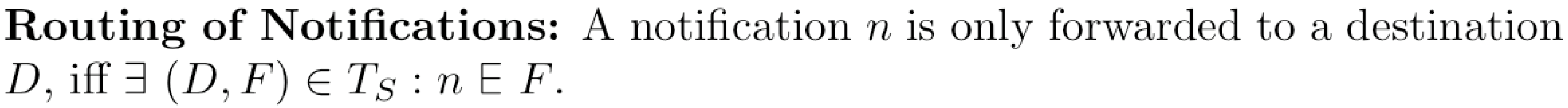
- Upstream filtering
  - Apply filters upstream (as close as possible to source)



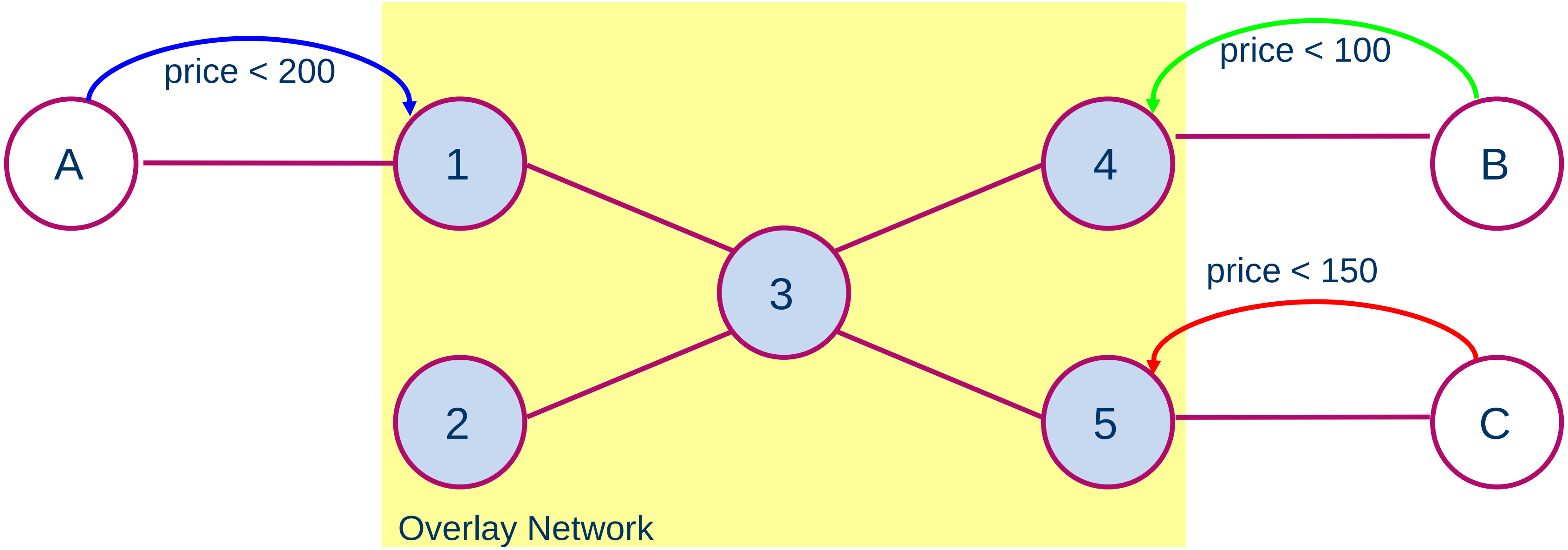
  - Clients B, C subscribe at routers 5, 6 with filter F_x
  - Client A publishes notification ny (not covered by F_x) to router 1
  - The notification is discarded at router 1

Routing with Subscriptions

- Each broker maintains a routing table T_S to route notifications based on subscriptions

- Routing of Subscriptions
  - On subscribe by destination D with filter F: $T_S = T_S \cup (D, F)$
  - On unsubscribe: $T_S = T_S - (D, F)$
  - Request is then forwarded to all neighbors except source (=flooding)
- 
  Routing of Notifications: A notification n is only forwarded to a destination D, iff $\exists\ (D, F) \in T_S : n \sqsubseteq F$.
  - (= application of **matching**)
- Hence,
  - Routing paths for notifications are set by subscriptions
  - Subscription creates a tree originating from the subscriber
  - Notifications are routed towards subscriber following reverse path

Subscription Routing



Router 1	
D	Filter

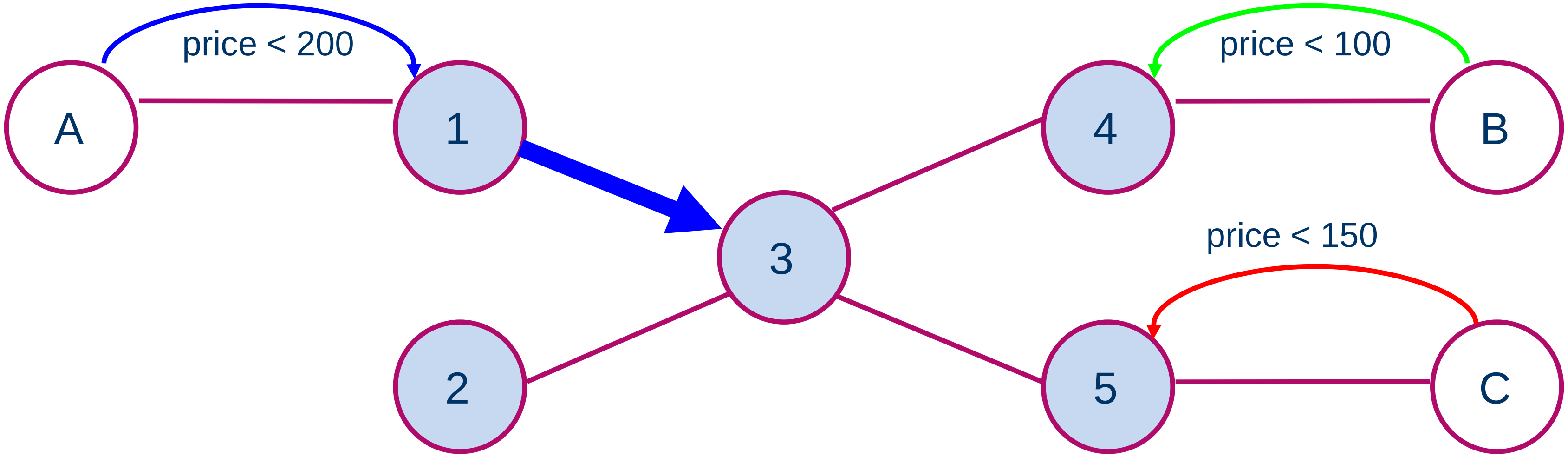
Router 2	
D	Filter

Router 3	
D	Filter

Router 4	
D	Filter

Router 5	
D	Filter

Subscription Routing



Router 1	
D	Filter

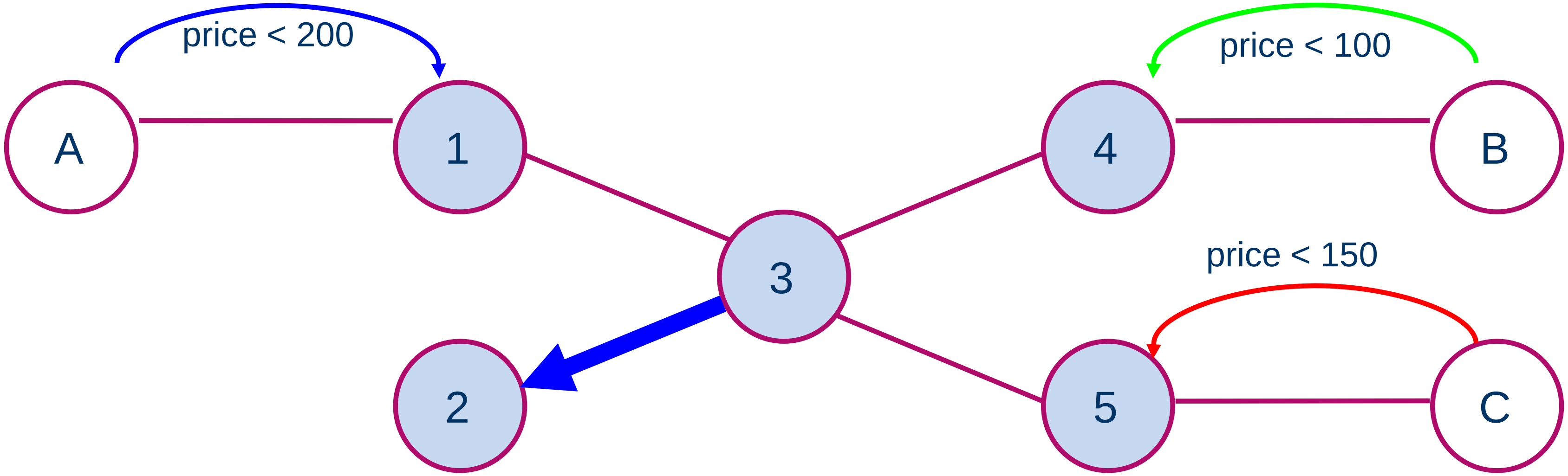
Router 2	
D	Filter

Router 3	
D	Filter
1	price<200

Router 4	
D	Filter

Router 5	
D	Filter

Subscription Routing



Router 1	
D	Filter

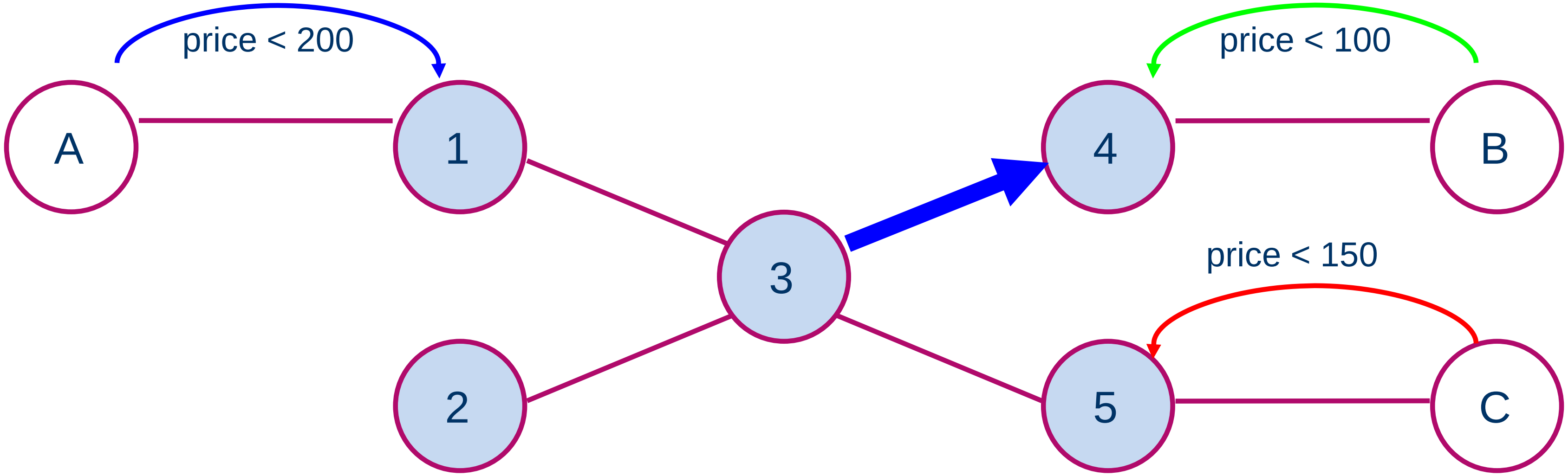
Router 2	
D	Filter
3	price<200

Router 3	
D	Filter
1	price<200

Router 4	
D	Filter

Router 5	
D	Filter

Subscription Routing



Router 1	
D	Filter

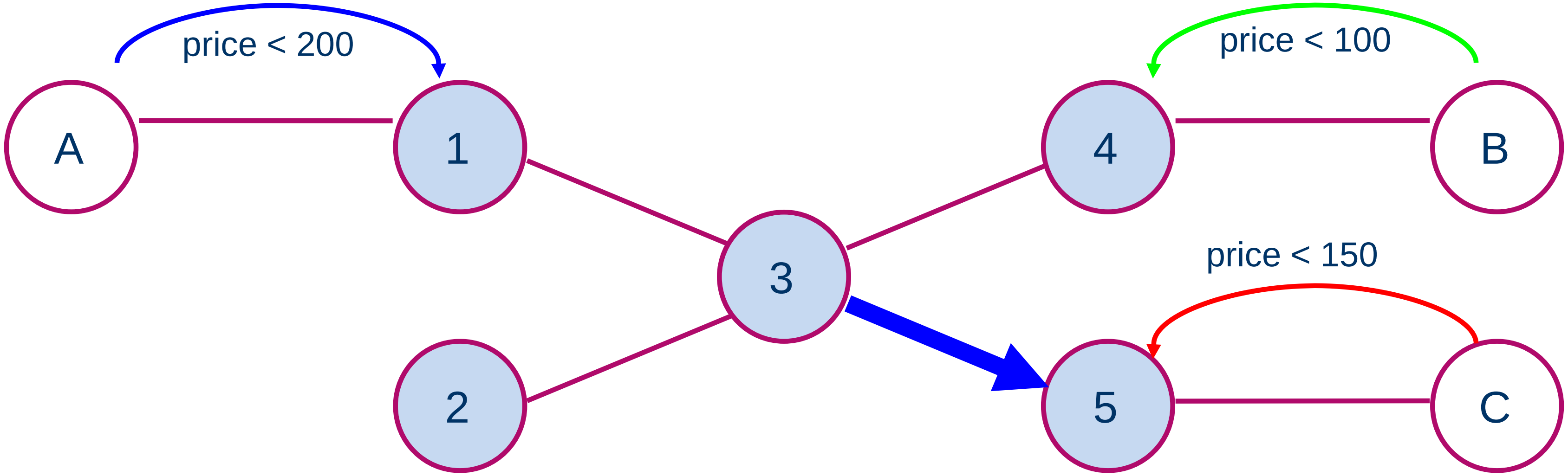
Router 2	
D	Filter
3	price<200

Router 3	
D	Filter
1	price<200

Router 4	
D	Filter
3	price<200

Router 5	
D	Filter

Subscription Routing



Router 1	
D	Filter

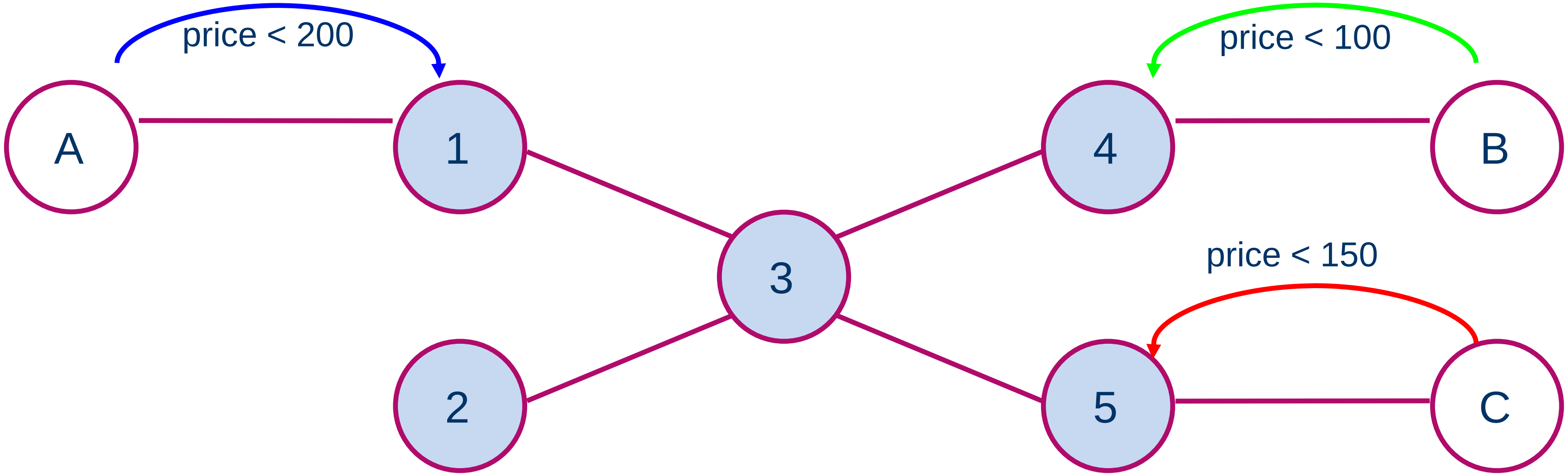
Router 2	
D	Filter
3	price<200

Router 3	
D	Filter
1	price<200

Router 4	
D	Filter
3	price<200

Router 5	
D	Filter
3	price<200

Subscription Routing



Router 1	
D	Filter
3	price<100
3	price<150

Router 2	
D	Filter
3	price<200
3	price<100
3	price<150

Router 3	
D	Filter
1	price<200
4	price<100
5	price<150

Router 4	
D	Filter
3	price<200
3	price<150

Router 5	
D	Filter
3	price<200
3	price<100

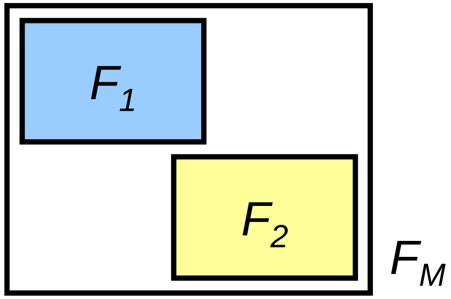
Filter Merging

- Inexact Merging

  F_M is an inexact merge of F_1 and F_2 iff

  $$F_M \sqsupseteq F_1 \wedge F_M \sqsupseteq F_2$$

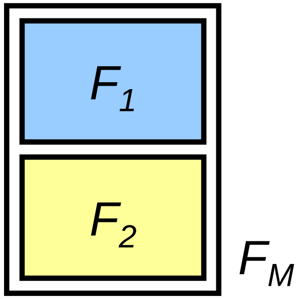
  - (= application of **covering**)



- Exact Merging

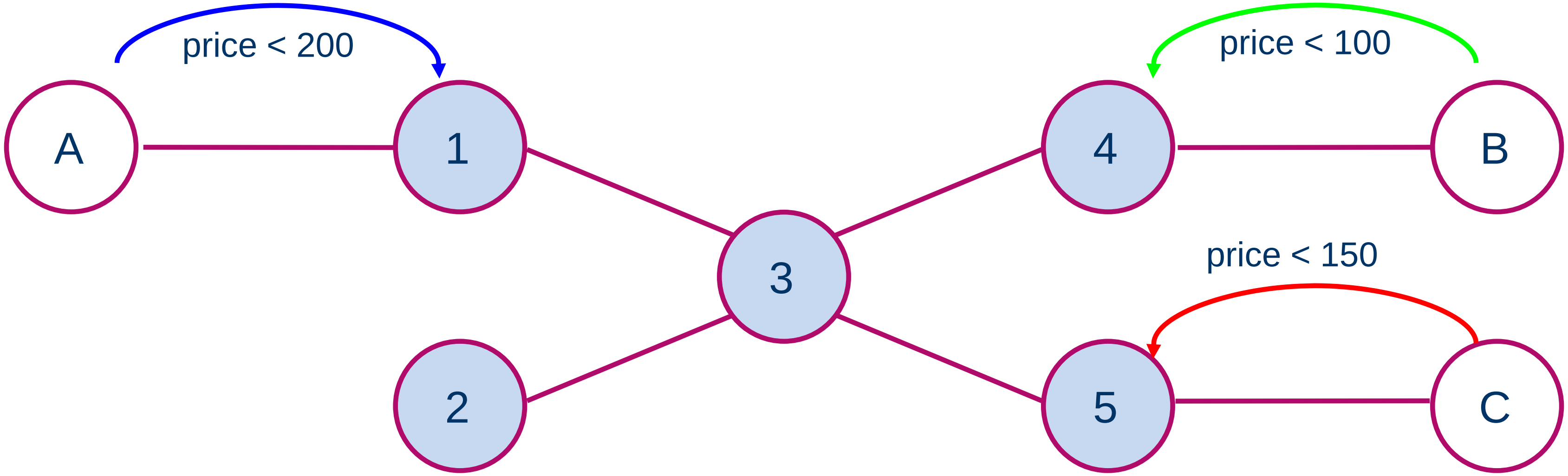
  F_M is an exact merge of F_1 and F_2 iff

  $$F_M \sqsupseteq F_1 \wedge F_M \sqsupseteq F_2 \wedge \neg\exists F_3 : (F_3 \not\sqcap F_1 \wedge F_3 \not\sqcap F_2 \wedge F_M \sqsupseteq F_3)$$



- Application in Subscription Routing:
  Filter merging is used to reduce subscribe requests

Subscription Routing with Merging



Router 1	
D	Filter

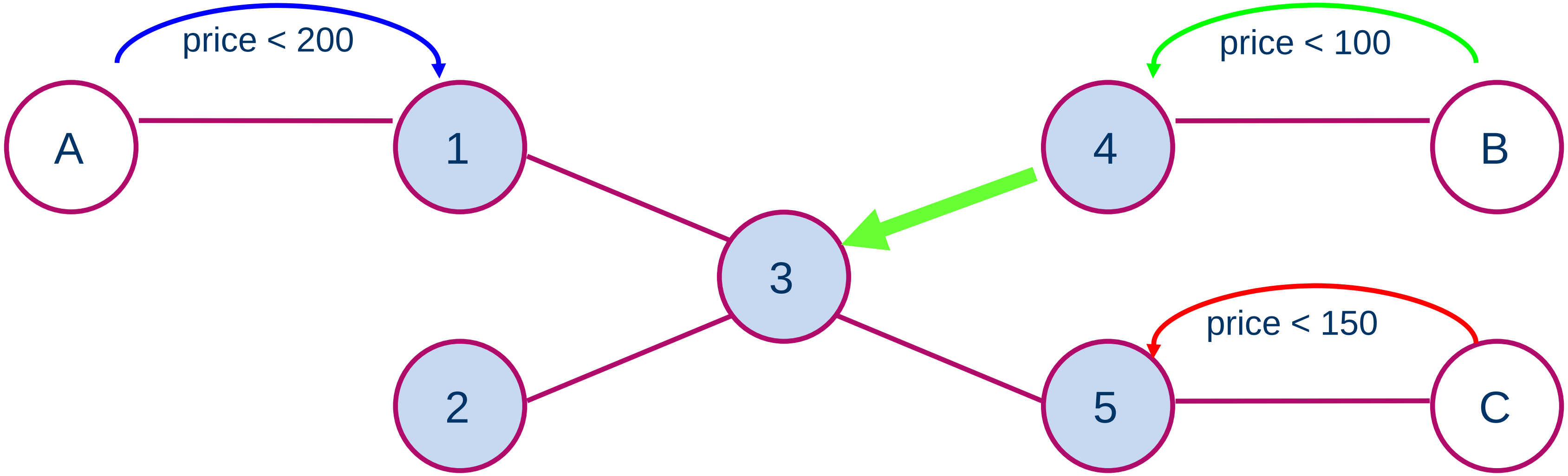
Router 2	
D	Filter
3	price<200

Router 3	
D	Filter
1	price<200

Router 4	
D	Filter
3	price<200

Router 5	
D	Filter
3	price<200

Subscription Routing with Merging



Router 1	
D	Filter

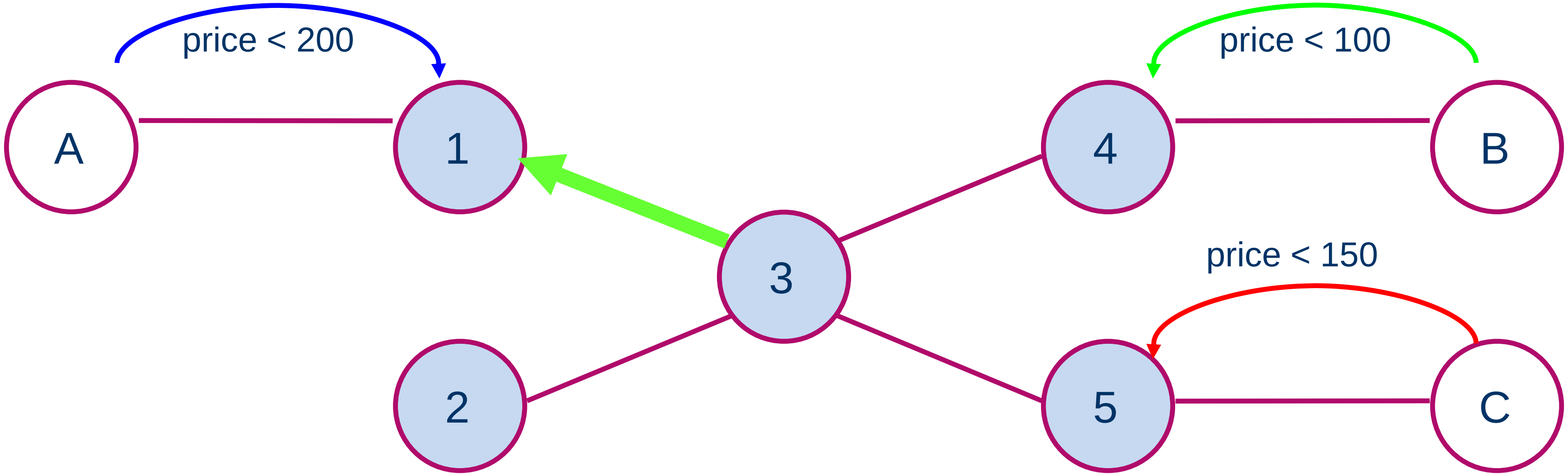
Router 2	
D	Filter
3	price<200

Router 3	
D	Filter
1	price<200
4	price<100

Router 4	
D	Filter
3	price<200

Router 5	
D	Filter
3	price<200

Subscription Routing with Merging



Router 1	
D	Filter
3	price<100

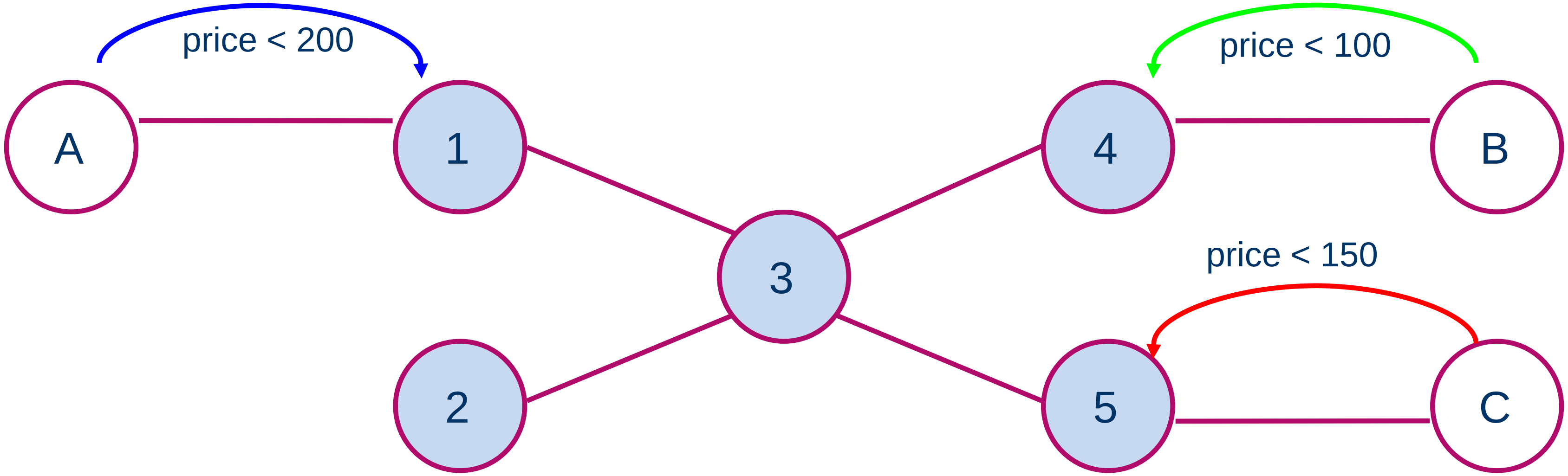
Router 2	
D	Filter
3	price<200

Router 3	
D	Filter
1	price<200
4	price<100

Router 4	
D	Filter
3	price<200

Router 5	
D	Filter
3	price<200

Subscription Routing with Merging



Router 1	
D	Filter
3	price<100

Router 2	
D	Filter
3	price<200

Router 3	
D	Filter
1	price<200
4	price<100

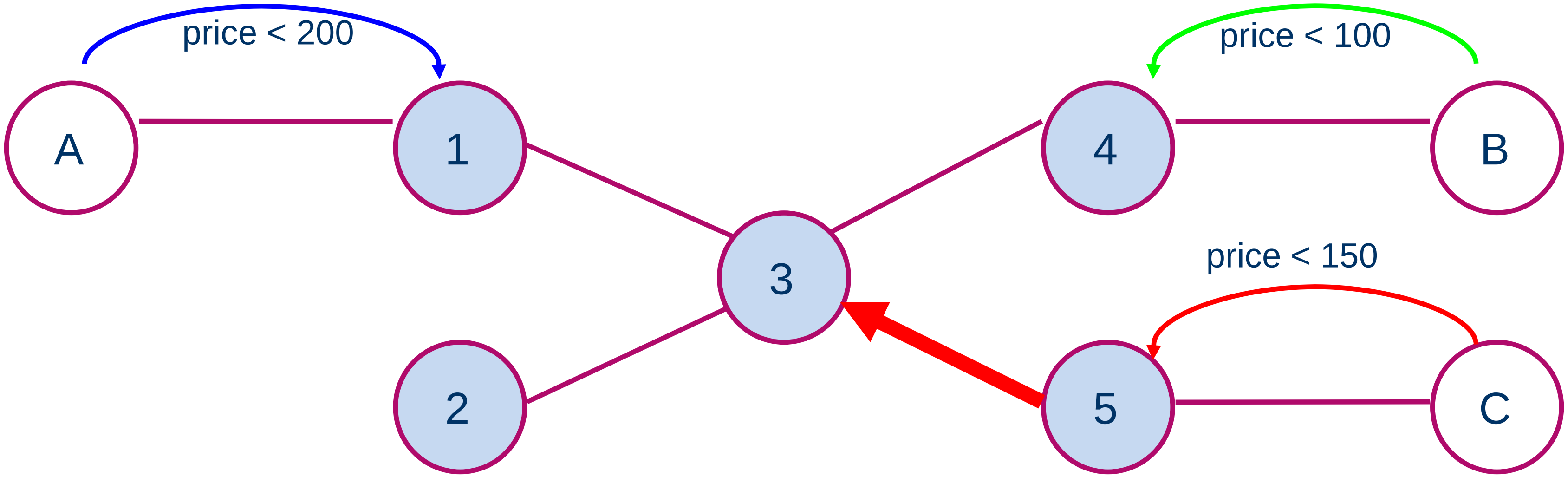
Router 4	
D	Filter
3	price<200

Router 5	
D	Filter
3	price<200

Router 3

- Does not send subscription to 2, because **price < 200** *covers* **price < 100**
- Does not send subscription to 5, because **price < 200** *covers* **price < 100**
- Does not send subscription to 4, because the subscribe request came from 4

Subscription Routing with Merging



Router 1	
D	Filter
3	price<100

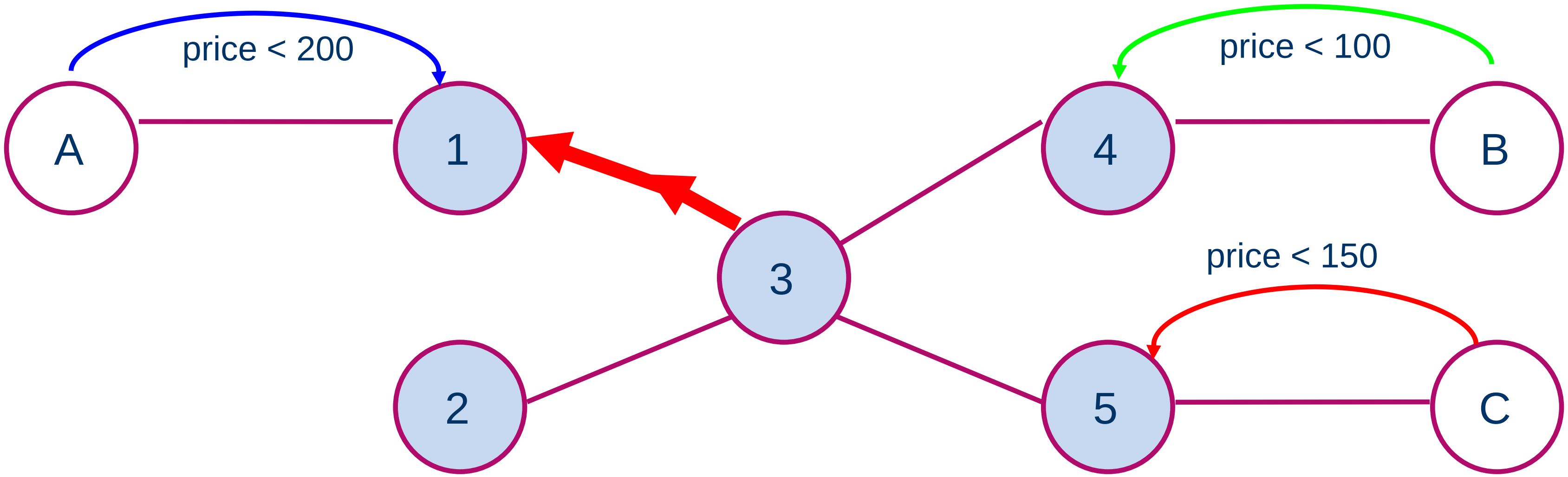
Router 2	
D	Filter
3	price<200

Router 3	
D	Filter
1	price<200
4	price<100
5	price<150

Router 4	
D	Filter
3	price<200

Router 5	
D	Filter
3	price<200

Subscription Routing with Merging



Router 1	
D	Filter
~~3~~	~~price<100~~
3	price<150

Router 2	
D	Filter
3	price<200

Router 3	
D	Filter
1	price<200
4	price<100
5	price<150

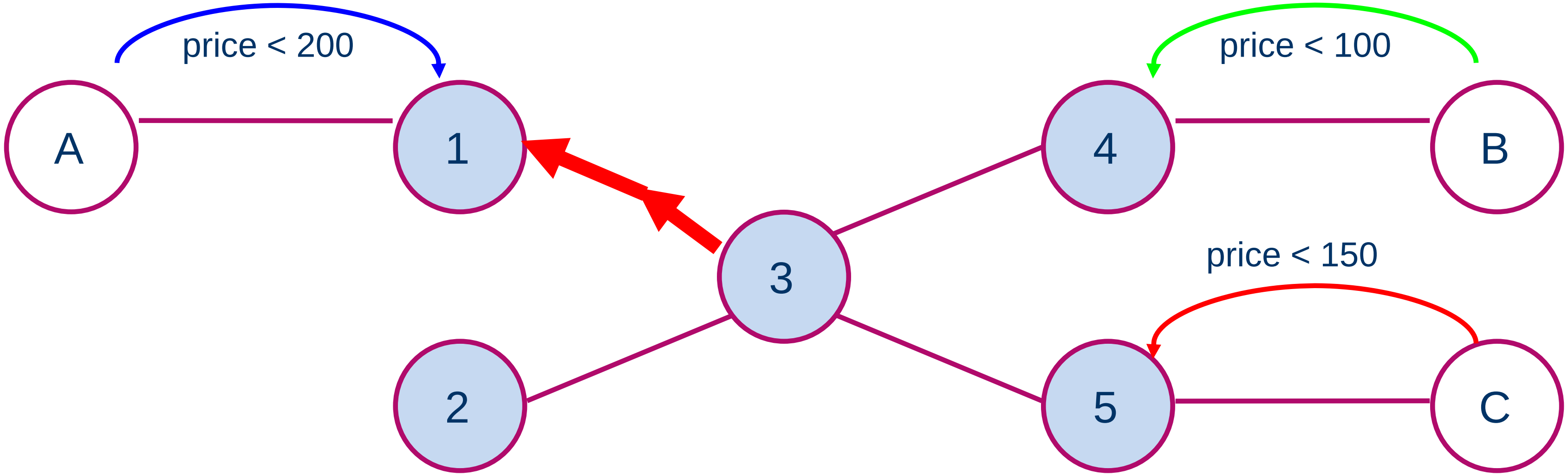
Router 4	
D	Filter
3	price<200

Router 5	
D	Filter
3	price<200

Router 3

- sends unsubscribe **price < 100** to 1
- Sends subscribe **price < 150** to 1

Subscription Routing with Merging



Router 1	
D	Filter
~~3~~	~~price<100~~
3	price<150

Router 2	
D	Filter
3	price<200

Router 3	
D	Filter
1	price<200
4	price<100
5	price<150

Router 4	
D	Filter
3	price<200

Router 5	
D	Filter
3	price<200

Router 3

- Does not send subscription to 2, because **price < 200** *covers* **price < 150**
- Does not send subscription to 4, because **price < 200** *covers* **price < 150**
- Does not send subscription to 5, because it came from 5

Routing with Advertisements

- Terms (again, informal)
  - Notification: “Event X has occurred”
  - Subscription: “I am interested in Event X”
  - Advertisement: “In the future I intend to publish Event X”
- Routing with Advertisements: Basic Idea
  - Subscriptions are only forwarded towards publishers that intend to generate notifications that are potentially relevant to this subscription
  - Every advertisement is forwarded throughout the network, thereby forming a tree that reaches every server
  - Subscriptions are propagated in reverse, along the path to the advertiser, thereby activating the path
  - Notifications are then forwarded only through activated paths.

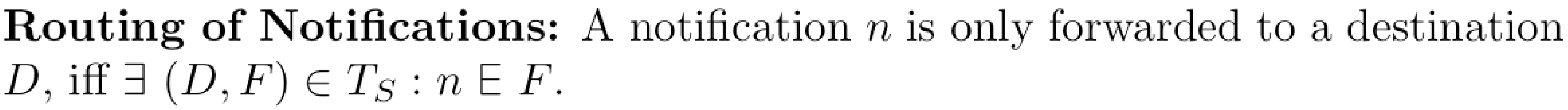
Routing with Advertisements

- Each broker maintains
  - a routing table T_S to route notifications based on subscriptions
  - a routing table T_A to route subscriptions based on advertisements

- Routing of Subscriptions
  - On subscribe by destination D with filter F: $T_S = T_S \cup (D, F)$
  - On unsubscribe: $T_S = T_S - (D, F)$
  - Request with a filter F_S is only forwarded to a neighbor D, iff

$$\exists\ (D, F_A) \in T_A : F_S \sqcap F_A$$

(= application of **overlapping**)

- 

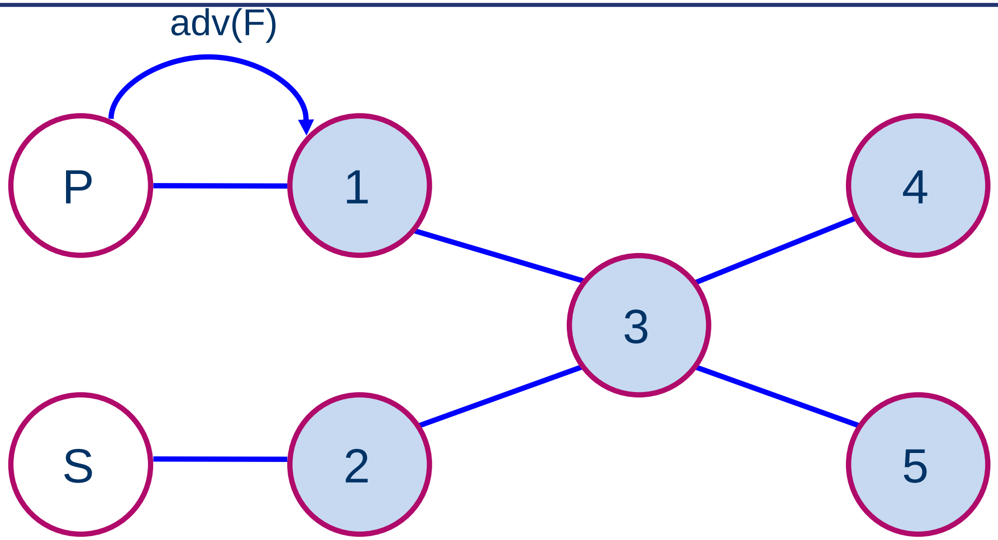
Routing of Notifications: A notification n is only forwarded to a destination D, iff $\exists\ (D, F) \in T_S : n \sqsubseteq F$.

  - (= application of **matching**)

Routing with Advertisements

- **Routing of Advertisements:** If a broker receives a new advertisement with a filter F_A from a neighbor E, it
  - forwards all subscriptions to E that came from a destination $D \neq E$, overlap with F_A, and do not overlap with any previous advertisement from E:
    $\{(D, F_S) \in T_S \mid \mathrm{D{\neq}E} \wedge F_S \sqcap F_A \wedge \neg\exists\ (D', F'_A) \in T_A : D' = E \ \wedge\ F_S \sqcap F'_A\}$
  - adds the advertisement to T_A:
    $T_A = T_A \cap \{(E, F_A)\}$
  - forwards the advertise request potentially to all neighbors $D \neq E$, according to the underlying routing algorithm.
- If a broker receives an unadvertisement request with a filter F_A from a neighbor E, it
  - removes the advertisement from T_A:
    $T_A = \{(D, F'_A) \in T_A \mid \neg(D = E \ \wedge\ F_A = F'_A)\}$
  - removes all routing entries from T_S of all neighbors $U \neq E$, for whose filter there is no other advertisement from any other destination $D \neq U$ that overlaps:
    $T_S = T_S - \{(U, F_S) \in T_S \mid U \neq E \wedge \neg\exists\ (D, F'_A) \in T_A : D \neq U \wedge F_S \sqcap F'_A\}$
  - forwards the unadvertise request potentially to all neighbors $D \neq E$, according to the underlying routing algorithm.

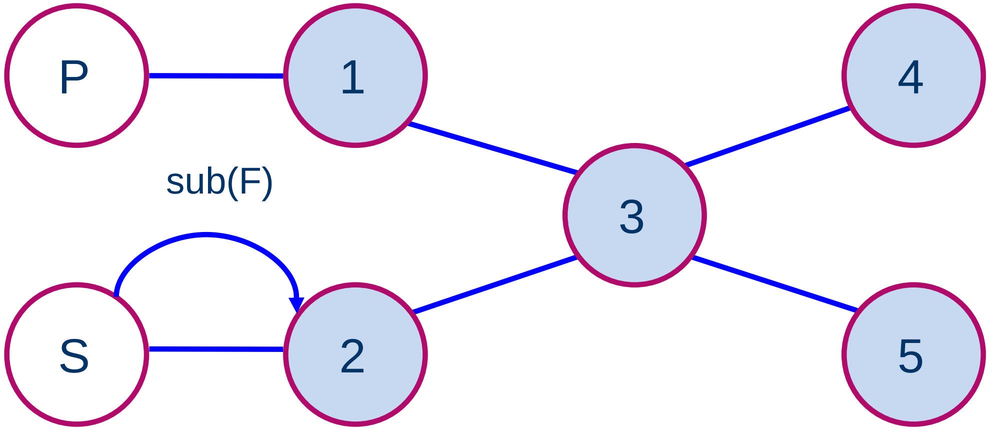
Advertisement Routing



Source	Dest	Request	Filter
1	3	adv	F
3	4	adv	F
3	5	adv	F
3	2	adv	F

Permutation possible

Subscription Routing

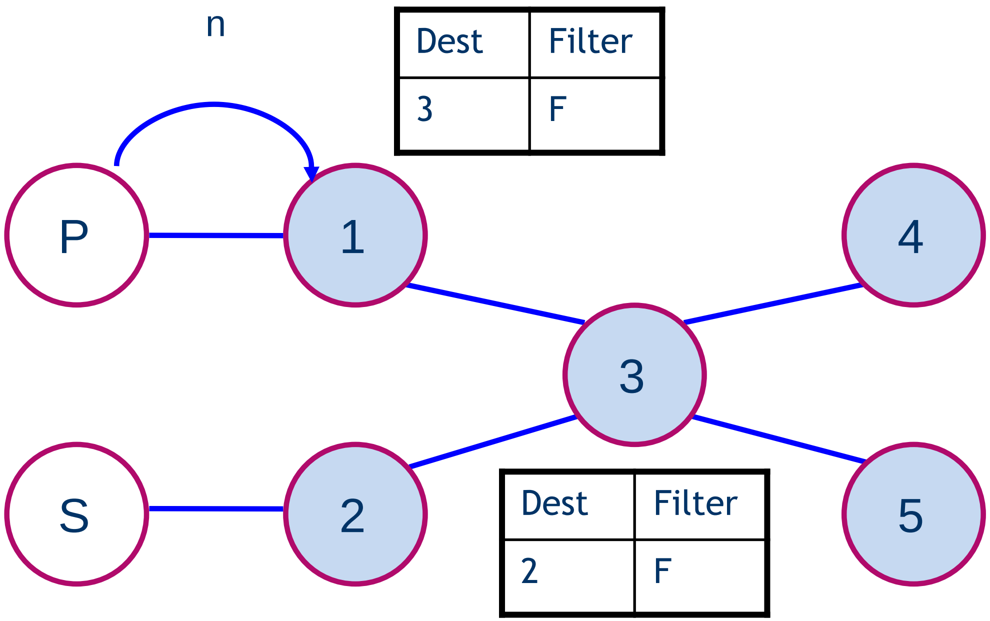


Source	Dest	Request	Filter
2	3	sub	F
3	1	sub	F

Note:

If router 2 receives *sub(F)* from S before it receives *adv(F)* from P, it has to store the subscription. When *adv(F)* is received later, the subscription is forwarded.

Notification Routing



Dest	Filter
3	F

Dest	Filter
2	F

Source	Dest	Message
1	3	notification n (n matches F)
3	2	notification n (n matches F)

System Examples

- Industry-strength
  - JMS
  - CORBA Notification Service
  - Elvin
  - IBM WebSphere MQ Event Broker (Gyphon)
- Research Prototypes
  - REBECA
  - SIENA
  - Gryphon
  - uMundo (MundoCore)

JMS: Java Message Service

- API
  - „Common set of interfaces and associated semantics“
- Domains
  - Point-to-Point
    - Message-Queue
  - Publish/Subscribe
    - Topic based
    - Subject based
- Separated Administration
  - Queues and Topics are created with product-specific administration tools
    - Application independent
  - Java Naming & Directory Interface (JNDI)

JMS: Message Format

- Header: Predefined Fields (ID, Destination, Timestamp, Priority)
  - Properties (optional): Accessible for Filtering
    Values can be boolean, byte, int, ... double and String
  - Body (optional): Five Types
    - TextMessage: String (XML Document)
    - MapMessage: Key/Value-Pairs
    - BytesMessage: Stream of uninterpreted bytes
    - StreamMessage: Stream of primitive values
    - ObjectMessage: A serializeable object

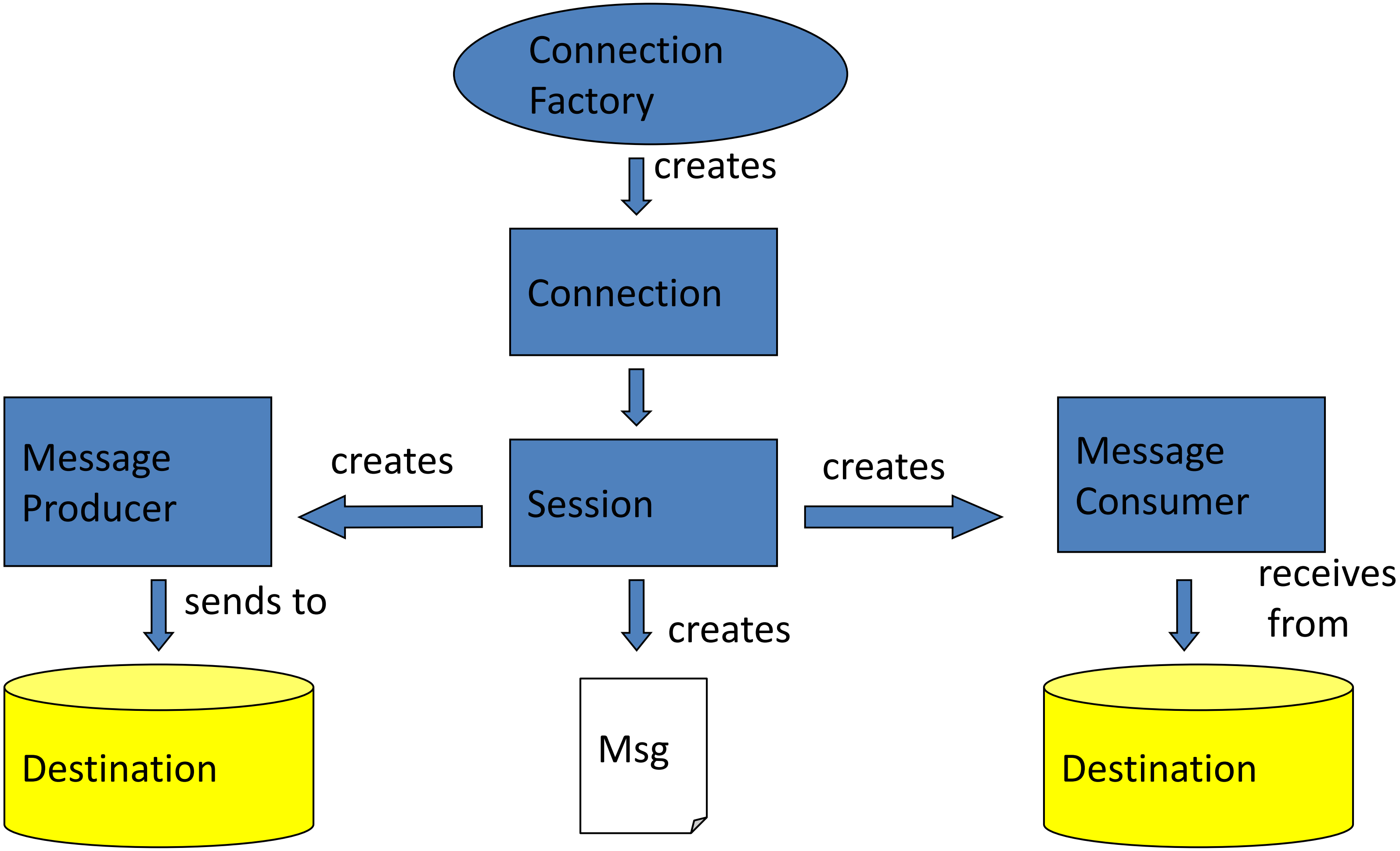
JMS: Message Filtering

- SQL92 conditional expressions (Limited)
  - Logical operators in precedence order: NOT, AND, OR
  - Comparison operators: =, >, >=, <, <=, <> (not equal)
  - Arithmetic operators in precedence order: +, - (unary) *, / (multiplication and division) +, - (addition and subtraction)
  - arithmetic-expr1 [NOT] BETWEEN arithmetic-expr2 AND arithmetic-expr3 (comparison operator)
  - identifier [NOT] IN (string-literal1, string-literal2,...)
    (comparison operator where identifier has a string or NULL value)
  - identifier [NOT] LIKE pattern-value
  - identifier IS [NOT] NULL (comparison operator that tests for a null header field value or a missing property value)
- Examples:
  - NewsType='Opinion' OR NewsType='Sports'
  - phone LIKE '12%3'
  - JMSType='car' AND color='blue' AND weight>2500

JMS: Message Consumptions

- Synchronously
  - A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method.
  - The receive method can *block* until a message arrives or can time out if a message does not arrive within a specified time limit.
- Asynchronously
  - A client can register a *message listener* with a consumer.
  - Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's onMessage() method.

JMS API Programming Model



JMS: Client Example

- Setting up a connection and creating a session

```
InitialContext jndiContext=new InitialContext();
//look up for the connection factory
ConnectionFactory cf=jndiContext.lookup(connectionfactoryname);
//create a connection
Connection connection=cf.createConnection();
//create a session
Session session=connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
//create a destination object
Destination dest1=(Queue) jndiContext.lookup(“/jms/myQueue”); //for PointToPoint
Destination dest2=(Topic)jndiContext.lookup(“/jms/myTopic”); //for publish-subscribe
```

More JMS Features

- Durable subscription
  - By default a subscriber gets only messages published on a topic while a subscriber is alive
  - Durable subscription retains messages until a they are received by a subscriber or expire
- Request/Reply
  - By creating temporary queues and topics
    - Session.createTemporaryQueue()

```
producer=session.createProducer(msg.getJMSReplyTo());
reply= session.createTextMessage(“reply”);
reply.setJMSCorrelationID(msg.getJMSMessageID);
producer.send(reply);
```

JMS: Implementations

J2EE Licensees:

- Allaire Corporation: JRun Server 3.0
- BEA Systems, Inc.: WebLogic Server 6.1
- Brokat Technologies (formely GemStone)
- IBM: MQSeries
- iPlanet (formerly Sun Microsystems, Inc. Java Message Queue)
- Oracle Corporation
- SilverStream Software, Inc.
- Sonic Software
- SpiritSoft, Inc. (formerly Push Technologies Ltd.)
- Talarian Corp.

Open source:

- **Apache ActiveMQ**
- objectCube, Inc.
- OpenJMS
- ObjectWeb – Joram
- …

Selected other companies:

- **SwiftMQ**
- Fiorano Software
- Nirvana (PCB Systems)
- Orion
- SeeBeyond
- Software AG, Inc.
- SoftWired Inc.
- Sunopsis
- Venue Software Corp.

- A more exhaustive listing is available at
  - http://java.sun.com/products/jms/vendors.html

Implementing Pub/Sub

- Why would you implement your own pub/sub middleware?
  - Hardly any of the implementations are truly cross-platform
    - Linux, Windows, Mac OSX, Android, iOS, Windows Phone
  - Virtually all implementations need some central infrastructure
    - You can’t just connect two mobile phones in an ad-hoc network
  - Additional deployment and maintenance effort
  - Support for multiple language bindings
    - C++, Java, C#/Mono and a scripting language
  - Licensing issues
- Missed opportunity for integration
  - Offer services on a mobile phone
  - Use them in a desktop application
  - And vice versa

Implementing Pub/Sub: Requirements

- R_{infra}: Assume minimal Infrastructure
  - Nodes in the system ought to find each other without a central instance
  - Support use-case with two mobile phones in an ad-hoc network
- $R_{platforms}$: Support a wide selection of platforms
  - Try to run on as many platforms as is reasonable
- $R_{languages}$: Do not require users to use language X
  - C++ difficult in server based environments
  - Java unsuited when minimal latency is required
  - Many technologies not even available in some languages
    - No mature multi-touch APIs in Java
    - MS Kinect SDK only available in C#
    - Raw hardware access only in C

Implementing Pub/Sub: Requirements

- $R_{codebase}$: Maintain only a single code-base
  - Naïve approach for multiple language bindings are multiple implementations
  - They will converge to the point of non-interoperability
  - In academic settings, Java is most prominent
  - Other implementations will lack behind in features
- R_{loc}: Try to get away with writing as little code as possible
  - Every line of code you write is a line of code to maintain!
- $R_{license}$: Keep license commercially viable
  - Industrial partners will not use GPL code
  - LGPL still difficult due to static-linking in app-stores
    - Some projects have a “static-linking exception” for this case
  - BSD, MIT, Apache, MPL, CDDL …
    - Most often a white-list exists in companies

Implementing Pub/Sub: API

```
Publisher pub(“topic”);
Message msg;
msg.setData(data); // whatever that means for now
pub.send(msg)
// meanwhile, somewhere else
Subscriber sub(“topic”)
while(Message msg = sub.getNextMsg()) {
  …
}
```

Implementing Pub/Sub: API

```
Node n1(“domain”); // allow parallel deployments: several nodes+domains possible
Publisher pub(“to.pic”); // straightforward hierarchy: subscription to prefix allowed
n1.addPublisher(pub);
Message msg;
msg.setData(data);
pub.send(msg)

// meanwhile, somewhere else
Node n2(“domain”); // we are in same domain as n1 above
Subscriber sub(“to”) // just prefix used in subscription (syntax of “.” not really
   observed)
n2.addSubscriber(sub);
while(Message msg = sub.getNextMsg()) {
  …
}
```

Implementing Pub/Sub: API

```
Node n1(“domain”);
Publisher pub(“to.pic”);
pub.setGreeter(greeter);    // greeter is object provided by publisher, implements
                            // welcome + farewell methods that are ‘called back’ by system
                                        // (e.g., welcome new subscribers with past events or state stored)
n1.addPublisher(pub);
Message msg;
msg.setData(data);
pub.send(msg)

// meanwhile, somewhere else
Node n2(“domain”);
Subscriber sub(“to”, this)    // register ‘receive’ method --> called back at notification
   reception
n2.addSubscriber(sub);

void receive(Message msg) {
  …
}
```

Implementing Pub/Sub: Approach

- Choosing an implementation language
  - $R_{languages}$ demands multiple language bindings
  - $R_{codebase}$ demands a single codebase
  - $R_{platforms}$ demands an implementation that runs on many platforms
- We need a language that can be accessed from all target languages and has runtime support on most platforms
  - Really only C/C++ qualifies
    - Bindings to Java via JNI, C# via DLLInvoke, Perl via XS, Scheme via FFI, …
    - Virtually all languages provide access to C/C++
    - In fact, most languages are even *implemented* in C/C++
    - Runs everywhere (even Windows Phone since end of 2012)
  - Caveat: User-defined types cannot reasonable be wrapped for target languages – only core pub/sub abstractions.

Implementing Pub/Sub: A note on language bindings

- Most of the language bindings are repetitive boilerplate code
  - CSharp:

```
[DllImport(”DLLName", EntryPoint=”methodName")]
public static extern void Class_method(HandleRef arg1, byte[] arg2, uint arg3);
```

  - Java:

```
public final static native void Class_method(long jarg1, Message jarg1_, byte[] jarg2);
void Class_method JNIEnv *jenv, jclass jcls, jlong jarg1, jobject jarg1_, jbyteArray jarg2);
```

- Hard to write and maintain, but easy to generate
  - Simplified Wrapper and Interface Generator (SWIG) does just that!
  - SWIG will generate language bindings to C/C++ for many languages:
    - C#, Java, Tcl/TK, Perl, Python, Ruby, PHP, Lua, R, Scheme/Lisp
  - Generated bindings will work but are rather awkward at times
  - Fine-tune type mappings and generated code with *.i files

Implementing Pub/Sub: Approach

- Identifying and Choosing Related Technologies
  - R_{infra} demands to deploy without central infrastructure
  - Main issue is for Subscribers to find matching Publishers
    - Standard network sockets are available after endpoints are known
- Multicast DNS offers just that
  - Part of the ZeroConf IETF standards
  - Only requires multicast packages to be transmitted
  - Given on every WiFi network if not explicitly deactivated
  - Limits scope of the pub/sub system to a multicast domain
    - Domains can be joined explicitly (Extranets)
  - Browse for services offered by endpoints
    - A service here is a node that provides publishers
- We can connect Subscribers to matching Publishers

Implementing Pub/Sub: Approach

- Identifying and Choosing Related Technologies
  - R_{loc} demands to write as few code as we can get away with
- We still need:
  - A multicast DNS implementation
  - A message broker
    - To get messages from publishers to subscribers
    - One with support for the pub/sub paradigm would be great
  - A serialization implementation
    - To transform objects into a byte-arrays and back
  - An RPC implementation
    - To support remote procedure calls on top of pub/sub
- Just glue FOSS components together!

Implementing Pub/Sub: Approach

- Multicast DNS Implementations:
  - Avahi
    - Available on many Unixes
    - Installed per default on many Linux distributions
  - Bonjour
    - Available on every Mac and iOS device
    - Installed on many embedded devices (printers / routers)
    - Available for Windows (e.g., as part of iTunes)
    - Compatible with Avahi
  - uPNP
    - Microsofts approach available on WinXP and above
    - Incompatible with any other mDNS implementation
    - Linux port available but last updated in march 2006

Implementing Pub/Sub: Approach

- Message Broker Implementations:

Name	Comments
RabbitMQ	Central message broker in Erlang
ActiveMQ	Central message broker in Java
StormMQ	Central message broker in Java
QPid	Central message broker in Java
ZeroMQ	Abstraction above BSD sockets
OpenAMQ	Central message broker in Java
XMPP	Distributed but rather old, XML messages
OpenMQ	JMS implementation with central message broker
Corba	Central message broker, non-trivial to get on mobiles

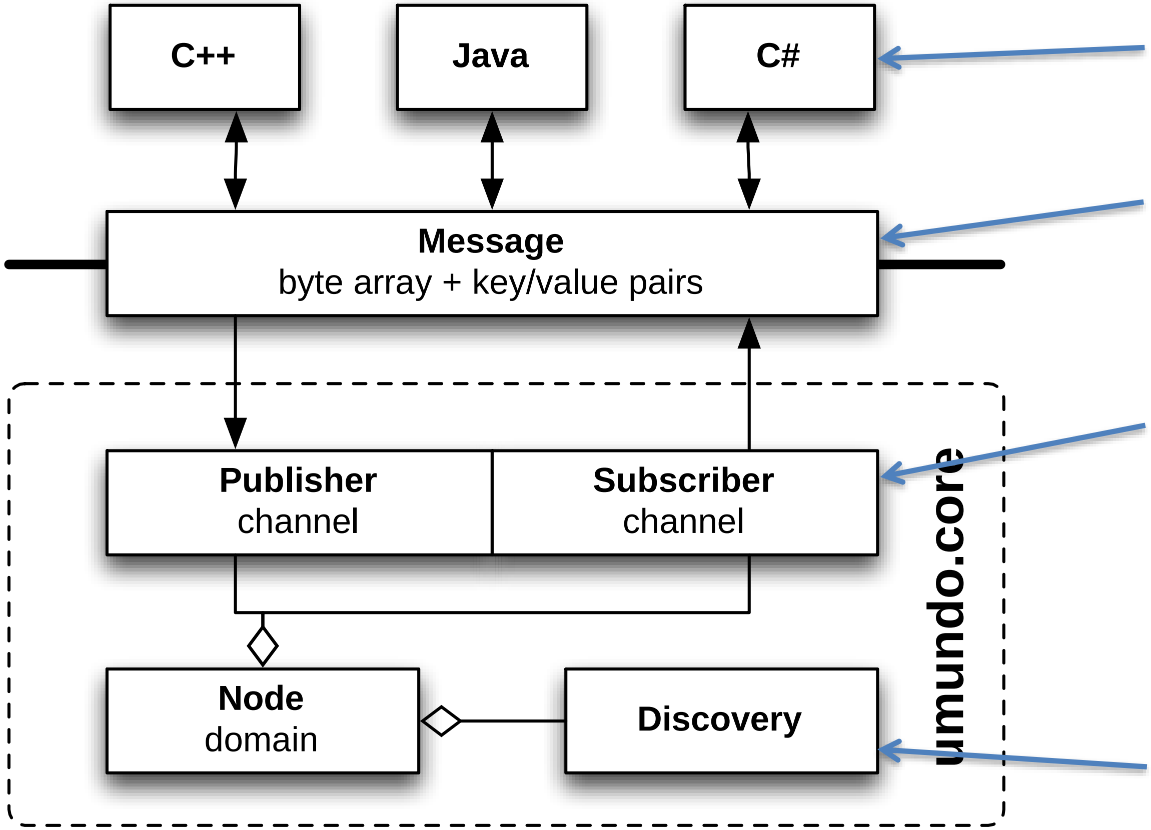
Implementing Pub/Sub: Approach

- Serialization Implementations
- Virtually every language brings its own that is incompatible with others
  - ASN.1
    - Standardized by ISO and IEC
    - Mature but hardly any implementations for anything but C
  - Apache Thrift
    - Lots of language bindings
    - Dictates RPC implementation
  - Protocol Buffers
    - Somewhat fewer language bindings than thrift out-of-the-box
    - Somewhat faster than thrift (when optimizing for speed)
  - XML / JSON
    - Way slower (20 - 100X) than either Thrift of Protobuf
    - Larger messages (3 - 10X) than either Thrift or Protobuf

Implementing Pub/Sub: Approach

- RPC Implementation
  - Different RPC Paradigms
    - Synchronous, Asynchronous, Broadcast Calls
- RPC has to integrate with the serialization implementation
  - RPC: Request -> Reply
    - With both request and reply being objects from serialization
- Protocol Buffers
  - Only defines syntax for service description and its AST within a Plugin
  - Actual implementation is generated from Plugin you wrote
  - Used to provide RPC framework as well (abandoned for plugin approach)
- Thrift provides more support for RPC
  - Less flexible as RPC implementation is fixed by Thrift

Implementing Pub/Sub: Bringing it all together



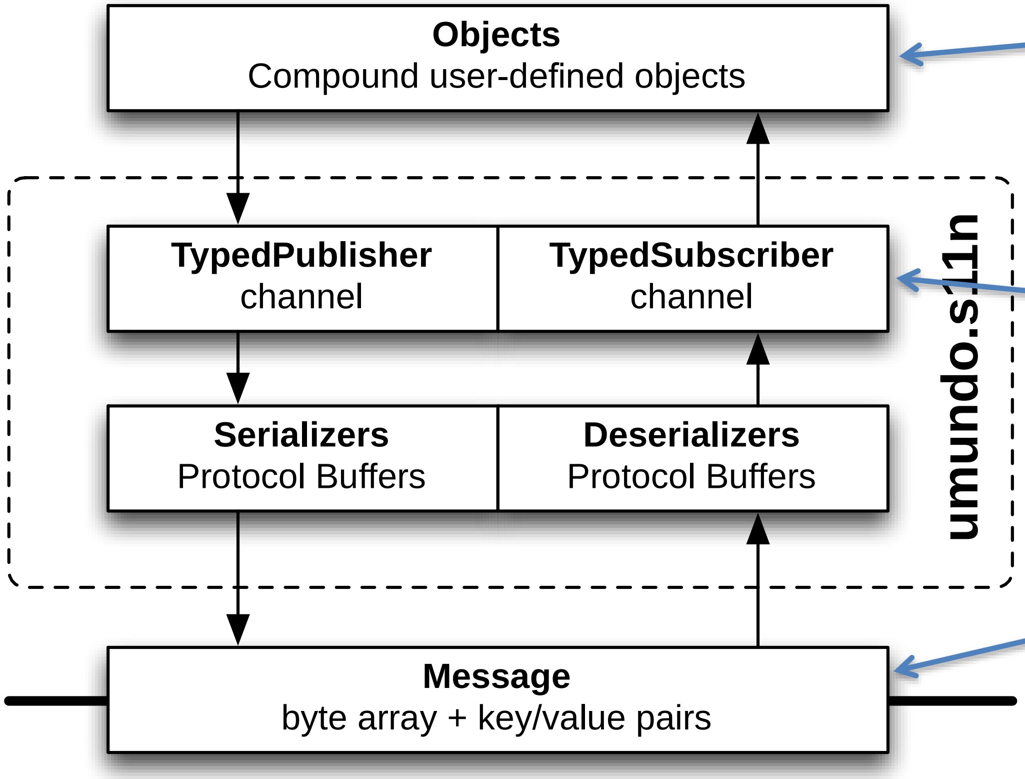
Language Bindings

A message is a raw byte-array with optional key/value pairs

Connect Subscribers to Publishers via **ZeroMQ**

Find other nodes via **Multicast DNS**.
Nodes will continuously report their publishers to all other nodes.

Implementing Pub/Sub: Bringing it all together

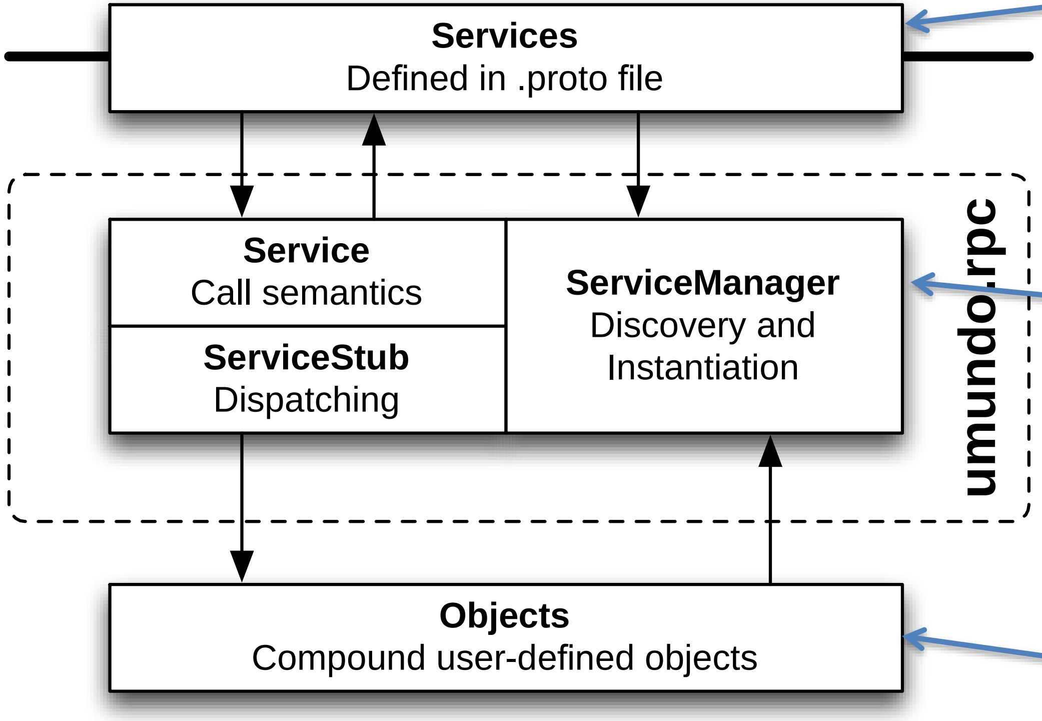


Classes generated by protoc Compiler or generic classes with **protobuf** reflection

TypedPublisher and TypedSubscriber provide abstraction for serialization

Serialized object in raw byte array, key/value pairs describe objects type for other nodes

Implementing Pub/Sub: Bringing it all together



Protoc plugins for **protobuf** will generate abstract base classes to be implemented

Continuous service discovery, service filters and local service stubs for remote services.

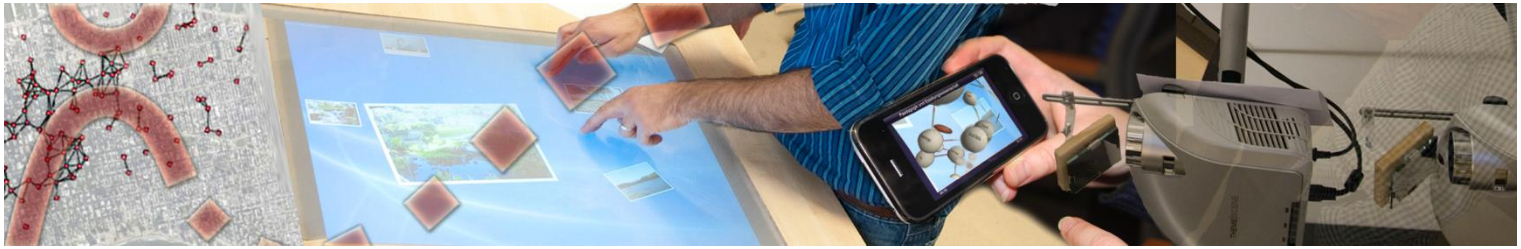
Invoking a service is to publish a request object and receive a reply object.
ZeroMQ allows to address single subscribers with reply.

Implementing Pub/Sub: Seeing it in Action

- All of these concepts are implemented as part of uMundo
  - Refactoring of MundoCore with focus on maintainability and deployment
  - Available on github
    - **https://github.com/tklab-tud/umundo**
  - Installer packages for the SDK
    - **http://umundo.tk.informatik.tu-darmstadt.de/installer/**
    - To develop on Windows, Linux, Mac OSX
    - With binaries for all supported platforms
  - Maven repository available for the Java bindings
    - **http://umundo.tk.informatik.tu-darmstadt.de/maven2**
  - Video of uMundo on mobile devices
    - **http://www.youtube.com/watch?v=GrZA2ONhYb8**

Pub/Sub: Summary

- Loosely coupled systems
  - Space decoupling
  - Time decoupling
  - Control flow decoupling
- Publish/Subscribe
  - Powerful and scalable abstraction for decoupled interaction
  - Problems are at the algorithm & implementation level
  - Research Challenges: Scalability/Expressiveness-Tradeoff, Fault Tolerance, Integration w. P2P, Security, Reliability, …
- Has specific application areas
  - E.g., RFID middleware, sensor systems in Ubicomp
  - Will not replace request/reply



2.2: ADVANCED PARADIGMS

(1) Event based & Publish/Subscribe Communication

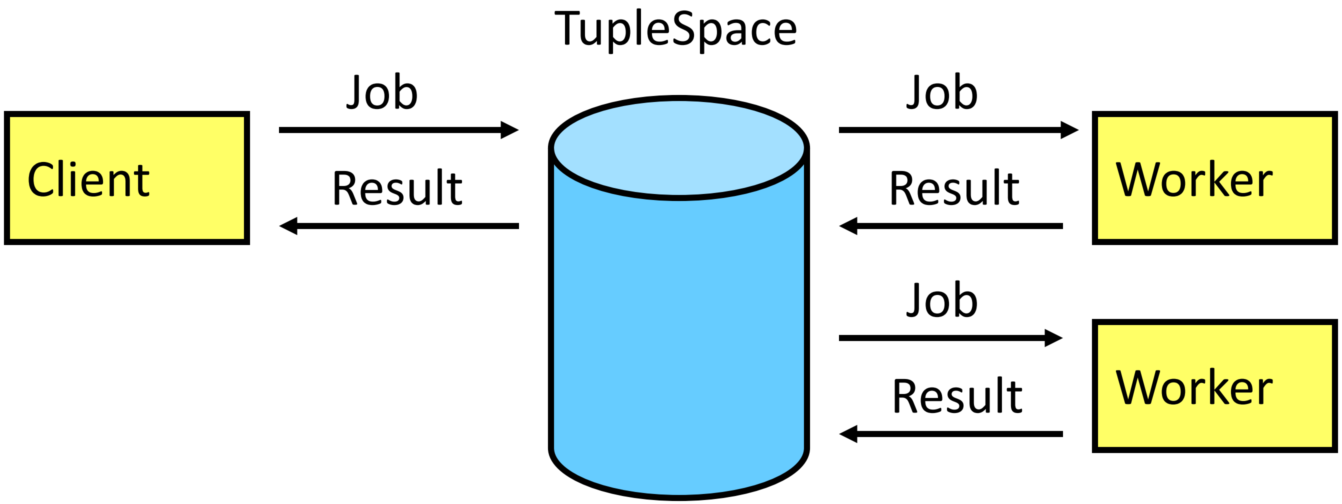
(2) Tuple Spaces

Tuple Spaces

- A tuple space is a
  - **globally shared**
  - **associatively addressed** memory space
  - organized as a **bag of tuples**
- Originally proposed by Gelernter as part of the Linda coordination language (1985)
  - Programming language + coordination language -> full parallel programming language (e.g., C-Linda or FORTRAN-Linda)
- Terms
  - Tuple: vector of typed values, or (actual) **fields**
  - Template: used to associatively address tuples via **matching**; some fields may be replaced by typed placeholders (**formal fields**)
- Example
  - Tuple: (2.24, „hello world“, 345)
  - Matches template (float, „hello world“, int)
  - Does not match template (float, String, 345.0)   (int vs. float)

Tuple Spaces

- Operations in Linda
  - **out**(tuple): inserts tuple into TupleSpace
  - **in**(template): withdraws tuple from TupleSpace
  - **read**(template): like in, but tuple is not removed
- Tuplespace concept extends message-passing systems with a simple data repository providing space, time, and flow decoupling
- Tuple spaces are a good basis for implementing **Master-Worker** or **Producer-Consumer** patterns; e.g.,



Tuple Spaces

- Operations in Linda
  - **out**(tuple): inserts tuple into TupleSpace
  - **in**(template): withdraws tuple from TupleSpace
  - **read**(template): like in, but tuple is not removed
- Note (for the „classical“ version):
  - Nondeterminism is inherent in these definitions
    - „first“ matching tuple encountered is used (indeterministic!)
    - 10 consecutive reads may all yield same result
  - no matching tuple ⇨ call blocks
  - read operation is problematic ⇨ race conditions
- Java based systems (JavaSpaces, TSpaces)
  - Tuples are vectors of **objects**, e.g.,

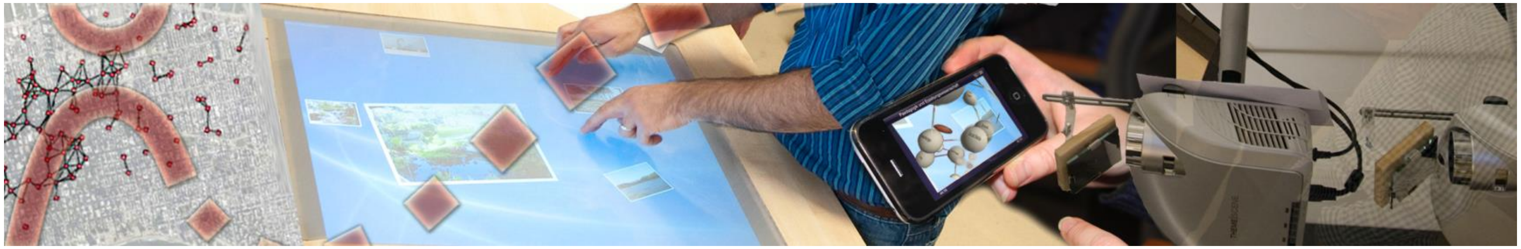
    { new Float(2.24), „hello world“, new Integer(345) }
  - **Actual fields** in templates must exactly match (Object.equals)

    { new Float(2.24), „hello world“, new Integer(345) }
  - **Formal fields** in templates only specify the type

    { Float.class, String.class, Integer.class }

Tuple Spaces

- Operations in JavaSpaces
  - *write* (instead of out)
  - *take*, ***takeIfExists*** (instead of in)
    takeIfExists is a non-blocking variant of *take*
  - *read*, ***readIfExists*** (instead of read)
    readIfExists is a non-blocking variant of *read*
  - ***notify***: registers a listener that is notified when a matching tuple is inserted into the space (cf. subscribe operation in pub/sub!)
- Operations in TSpaces
  - *write*
  - *waitToRead* (blocking), ***read*** (non-blocking)
  - *take* (blocking), ***takeIfExists*** (non-blocking)
  - ***eventRegister*** (cf. notify)
  - ***scan***: returns a list of all matching tuples – read/take only provide single tuples



2.2: ADVANCED PARADIGMS

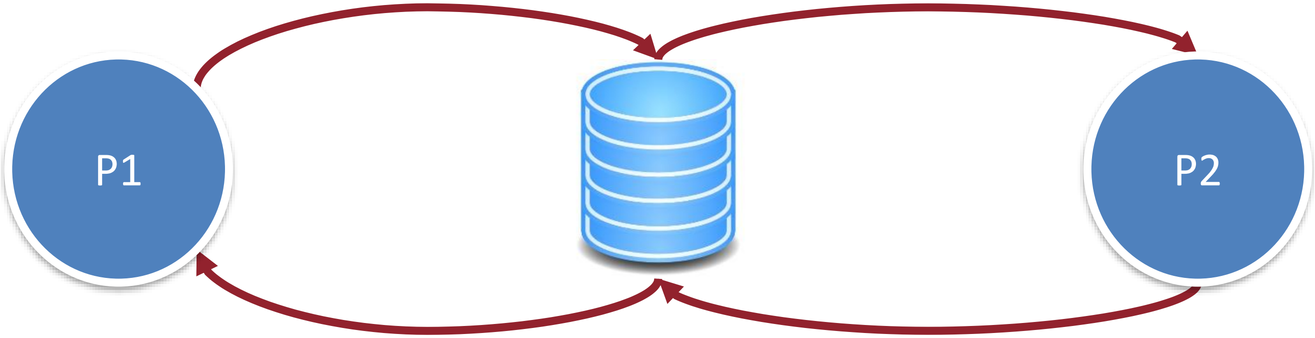
(1) Event based & Publish/Subscribe Communication

(2) Tuple Spaces

(3) Distributed Shared Memory Approach

Interprocess Communication (IPC)

IPC = **Interprocess Communication** = a way of exchanging data between programs running at the same time



Shared Memory:

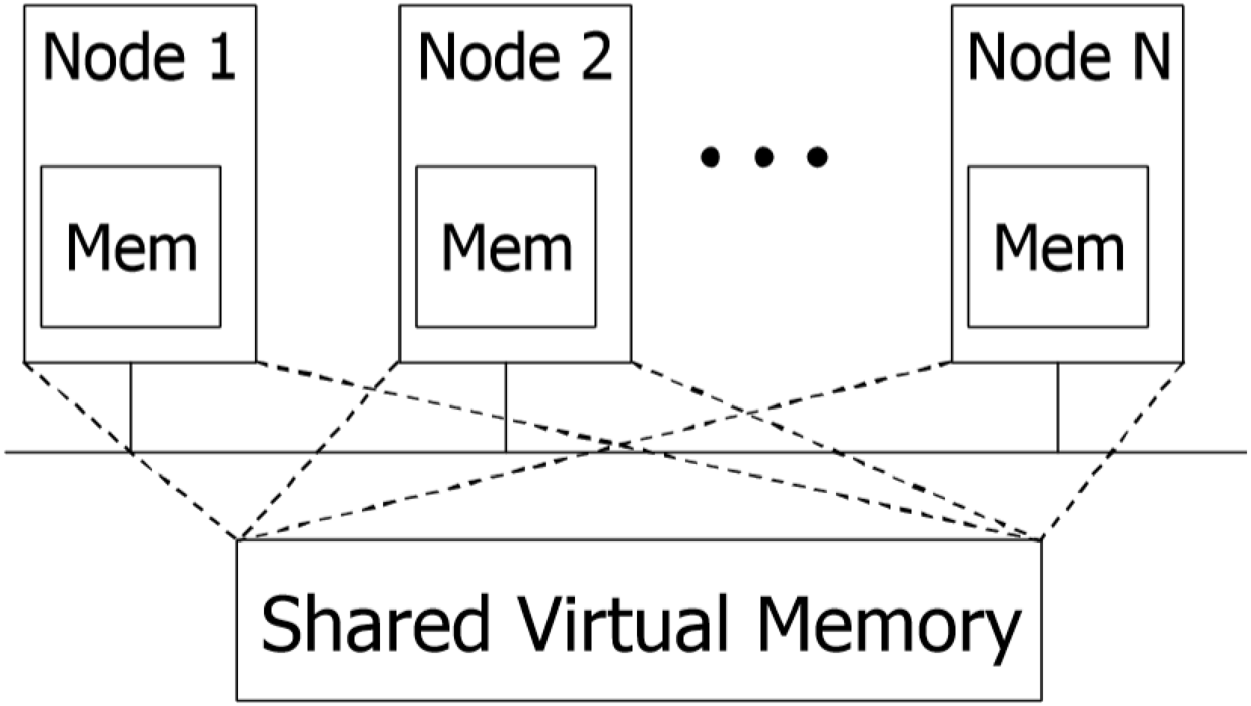
- Memory that can be simultaneously accessed by multiple programs
- Implicit communication

Distributed Shared Memory

- Distinguish: UMA, NUMA
- **UMA: Uniform Memory Access**
  - All processors share the physical memory uniformly
  - Access time to a memory location is independent of requesting processor
  - CPUs and memory on a shared bus ⇨ scales only up to ~10 CPUs
- **NUMA: Non-Uniform Memory Access**
  - Processors (usually 1-4) have local memory
  - Global virtual address space
  - Access to local memory is faster than to remote memory
  - Fast interconnection network between processors, e.g.,
    - Multicore CPUs: HyperTransport
    - SGI NUMAlink, Infiniband, Myrinet, …
    - NUMAlink vs. Ethernet: ~30x lower latency, ~30x higher bandwidth
- **ccNUMA: Cache Coherent NUMA**
  - Based on NUMA + processors can cache global memory
  - Requires complex cache-coherence protocols;
    however, non-CC systems are too complex to program

Distributed Shared Memory

- Abstraction for sharing data between computers that do not share physical memory



Distributed Shared Memory

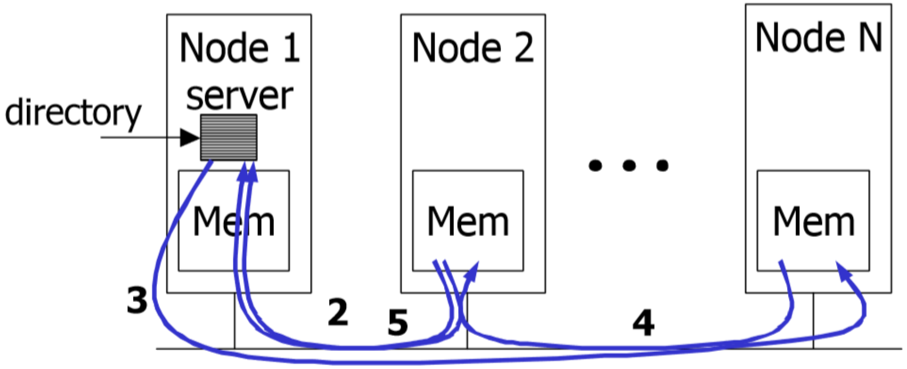
- Shared memory appears to processes like local memory
  - Local pages are present in current node's memory
  - Remote pages are in some other node's memory

- Retrieve pages
  - Local entries are valid
  - Non-local pages cause page fault
  - Handled by DSM runtime

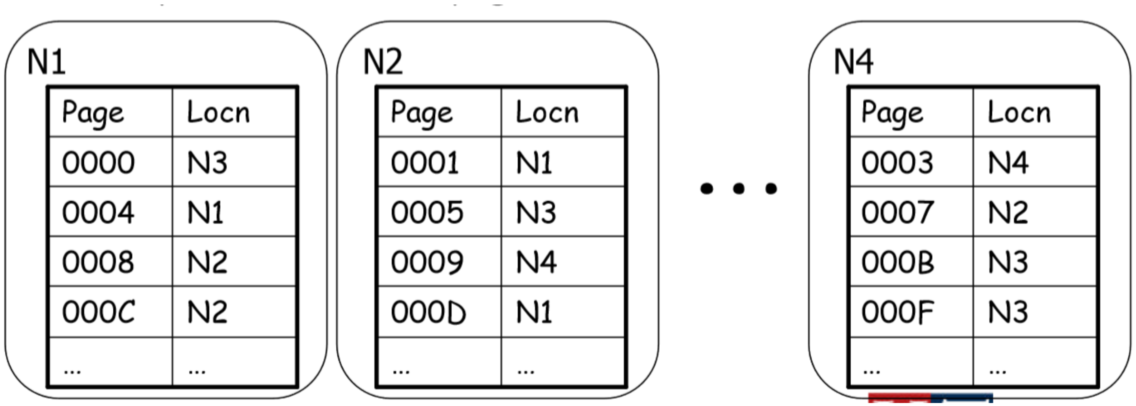
- DSM runtime
  - Relies on virtual memory mechanisms of CPU and OS
  - Uses message passing to synchronize replicas of shared memory regions

Distributed Shared Memory

- Central Server
  - Simple query
  - Central Bottleneck



- Distributed Directory
  - Responsible for part of the address space
    - E.g., Responsible node = (page #) mod (num nodes)
    - Caching?



N1

Page	Locn
0000	N3
0004	N1
0008	N2
000C	N2
…	…

N2

Page	Locn
0001	N1
0005	N3
0009	N4
000D	N1
…	…

• • •

N4

Page	Locn
0003	N4
0007	N2
000B	N3
000F	N3
…	…

Distributed Shared Memory

- DSM vs. Message Passing
  - **Marshalling**
    - Not required, entire memory pages transferred
    - but: same variable representation in memory required! (heterogeneity)
    - Even pointers will still work
  - **Synchronization**
    - No implicit synchronization like with message passing op’s
    - With locks, semaphores, … provided by DSM runtime
  - **Efficiency**
    - In message passing, all remote data accesses are explicit
    - Shared Memory
      - Any particular read/update may involve DSM runtime and communication
      - Programmer not always aware if in-process or remote memory access
      - Copy of page already cached? sharing pattern?

MPI & PVM

- **MPI: Message Passing Interface**
  - Harmonizes different multiprocessor system APIs
  - Programs can be recompiled → run on different vendor's hardware
  - Still, no vendor-spanning programs supported
  - MPI-2 also supports shared memory

- **PVM (parallel virtual machine)**
  - Open SW (any machine!); C/C++/Fortran, msg passing paradigm
    → unique distributed runtime environment; plus XPVM GUI available
  - User configures *host pool* : nodes participating in "parallel" program
  - Translucent access to HW: programs *may* exploit, e.g., multiprocessor subnet
  - Process based computation: appl. composed of tasks (e.g., unix proc's), not nodes
  - Heterogeneity support wrt. machines, networks, applications, data representations
  - Multiprocessor support: optimized to exploit fast local msg passing

Summary

- Event based & Publish/Subscribe Communication
  - Formal data and filter models
  - Routing algorithms
  - Systems: JMS, SIENA, Elvin, Gryphon
- Communication in Ubiquitous Computing
  - RMI on top of Publish/Subscribe
  - Decoupling of components via input & output interfaces
- Tuple Spaces
  - Linda, JavaSpaces, TSpaces
- Communication in Parallel Computers & Clusters
  - (Virtual) Distributed Shared Memory
  - Message passing: MPI, PVM