Introduction to Communication Networks and Distributed Systems

Unit 7 – Complex Communication Patterns: Pub/Sub -

Fachgebiet Telekommunikationsnetze

Prof. Dr. A. Zubow

zubow@tkn.tu-berlin.de

Technische Universität Berlin

Acknowledgements

- We acknowledge the use of slides from: **Prof. Adam Wolisz**

- "Handbook of Research: Ubiquitous Computing Technology for Real Time Enterprises“, Max Mühlhäuser and Iryna Gurevych.
- Eugster, Patrick Th, et al. "The many faces of publish/subscribe." *ACM computing surveys (CSUR)* 35.2 (2003): 114-131.

Motivation

- Traditional client/server communication model (using e.g. RPC, message queue, shared memory, etc.)
  - Synchronous, tightly-coupled request invocations.
  - Very restrictive for distributed applications, especially for Wide Area Network (WAN) and mobile environments.
  - When nodes/links fail, system is affected.

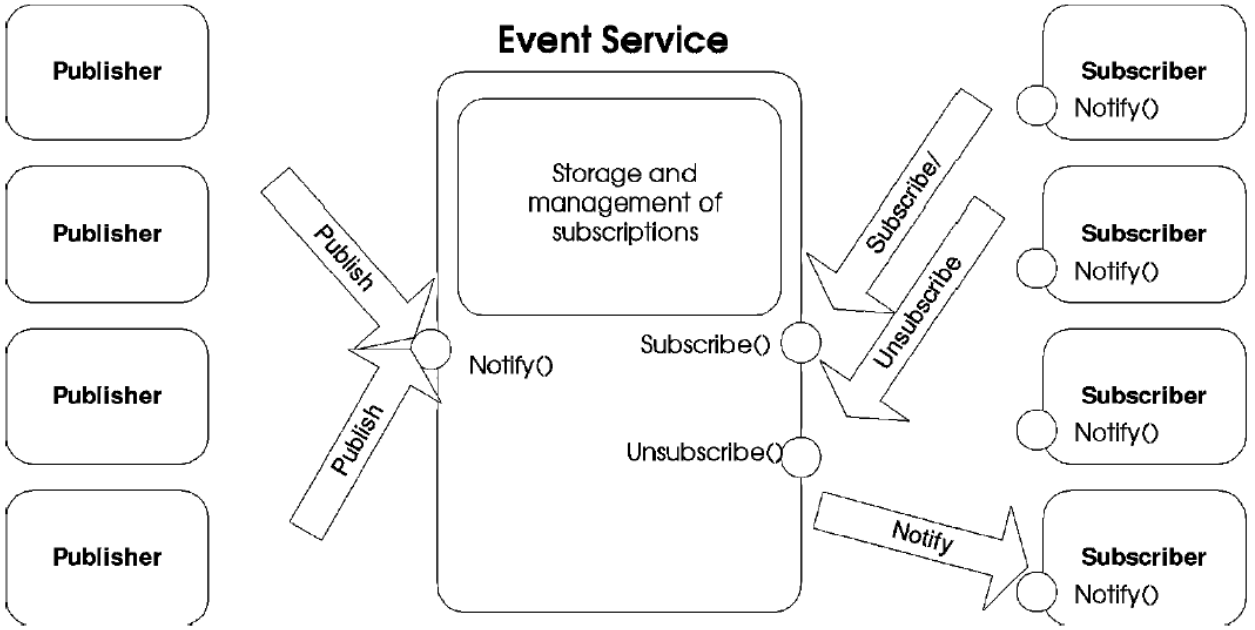
    → **Fault tolerance** must be built in to support this.
- Require a more **flexible** and **de-coupled communication** style that offers anonymous and asynchronous mechanisms.

What is a publish/subscribe system?

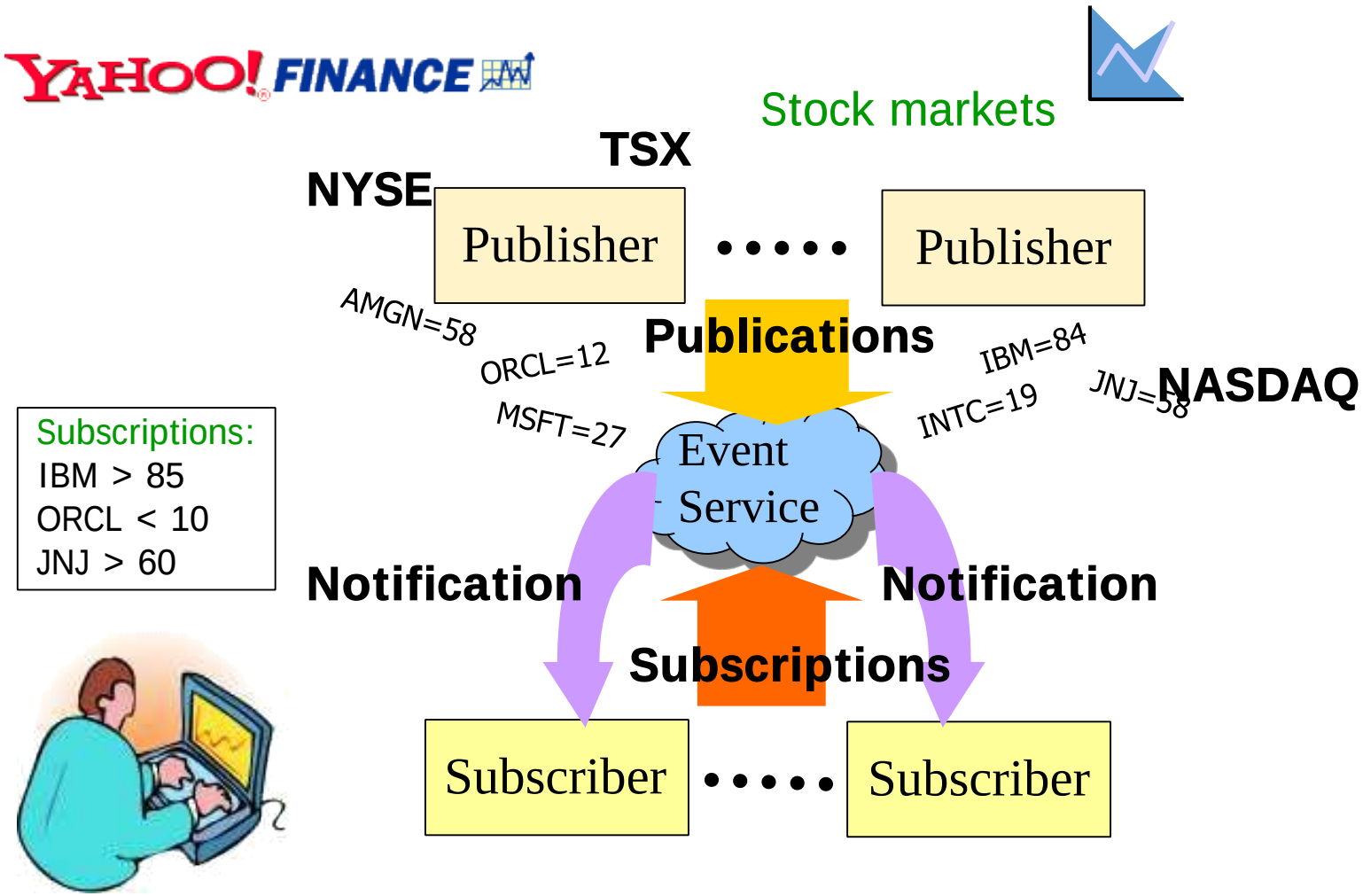
- Distributed pub/sub system is a **communication paradigm** that allows freedom in the distributed system by the **decoupling** of communication entities in terms of **time**, **space** and **synchronization**.
- An **event service system** that is asynchronous, anonymous and loosely-coupled.
- Ability to quickly adapt in a dynamic environment.

Basic system model for publish/subscribe

- Key components of pub/sub:
  - **Publishers**: Publishers generate event data and publishes them.
  - **Subscribers**: Subscribers submit their subscriptions and process the events received.
  - **Event Service**: It's the mediator/broker that filters and routes events from publishers to interested subscribers.



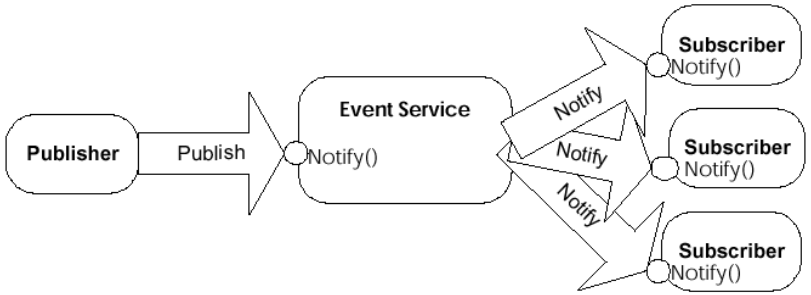
Example of pub/sub – Stock Quote Service

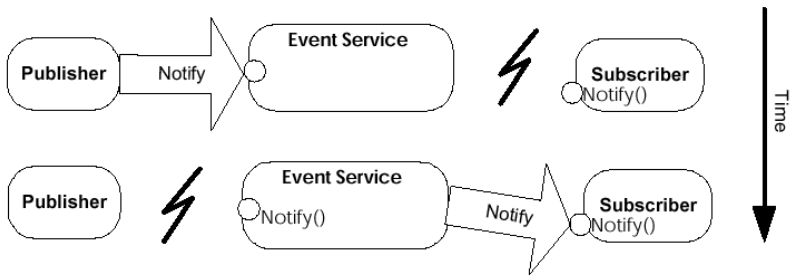


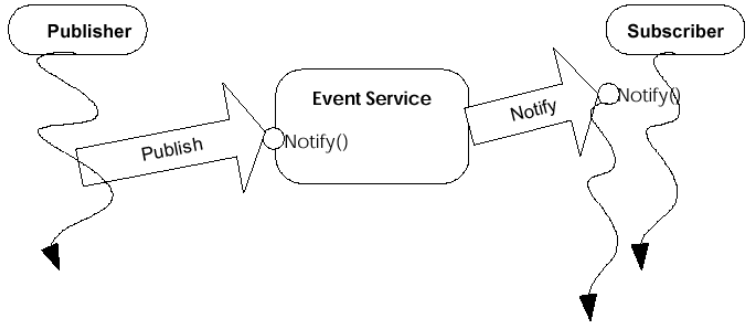
Towards loosely coupled systems

[Eugster, op. cit.]

1. **Space** decoupling
   - parties don‘t know each other
   - 1-to-many communication 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 –> **scalability**



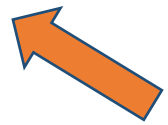




Interaction models

- 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

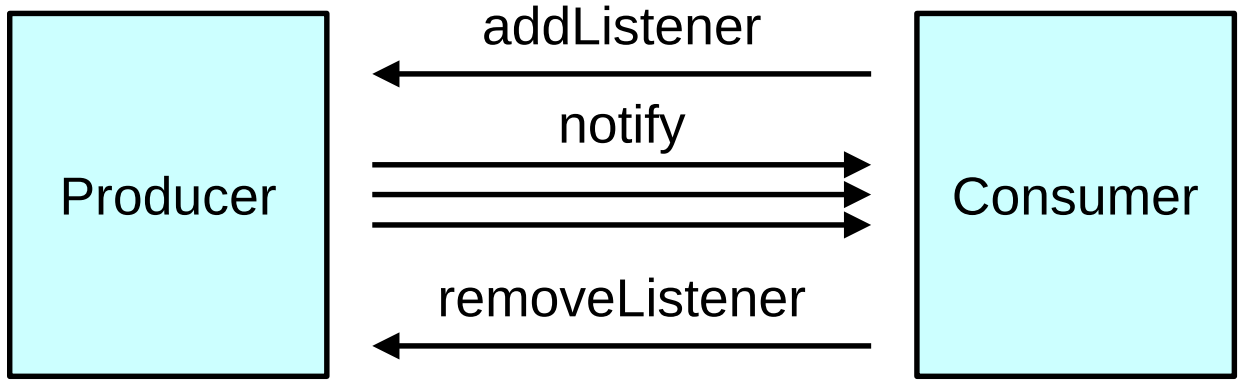


Today‘s focus

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

Concepts: Callbacks

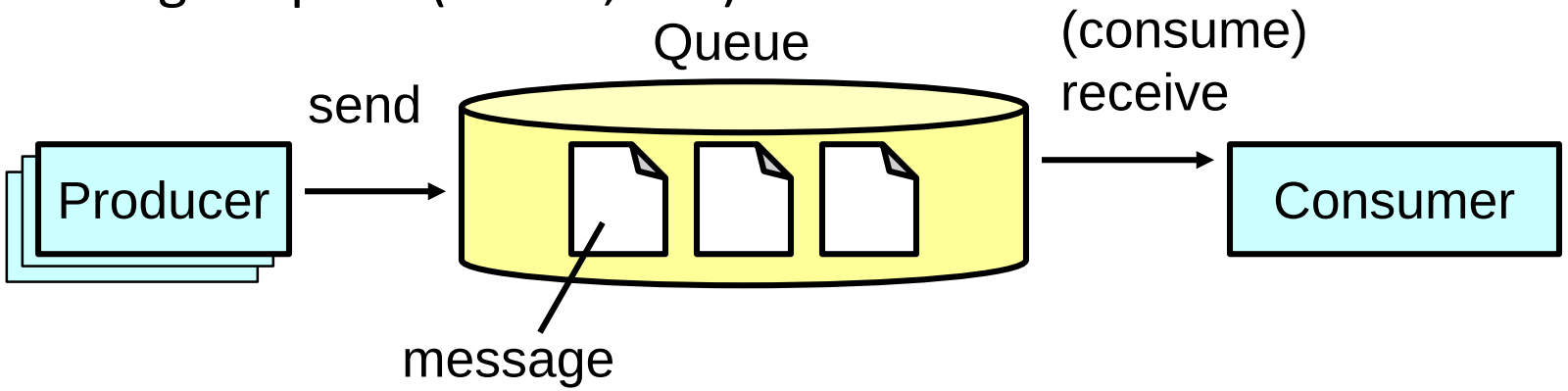
- Synchronous (remote) method calls often used to emulate behavior of event-based systems
  - See also: Observer Design Pattern
  - Frequently used in UI toolkits; example:



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

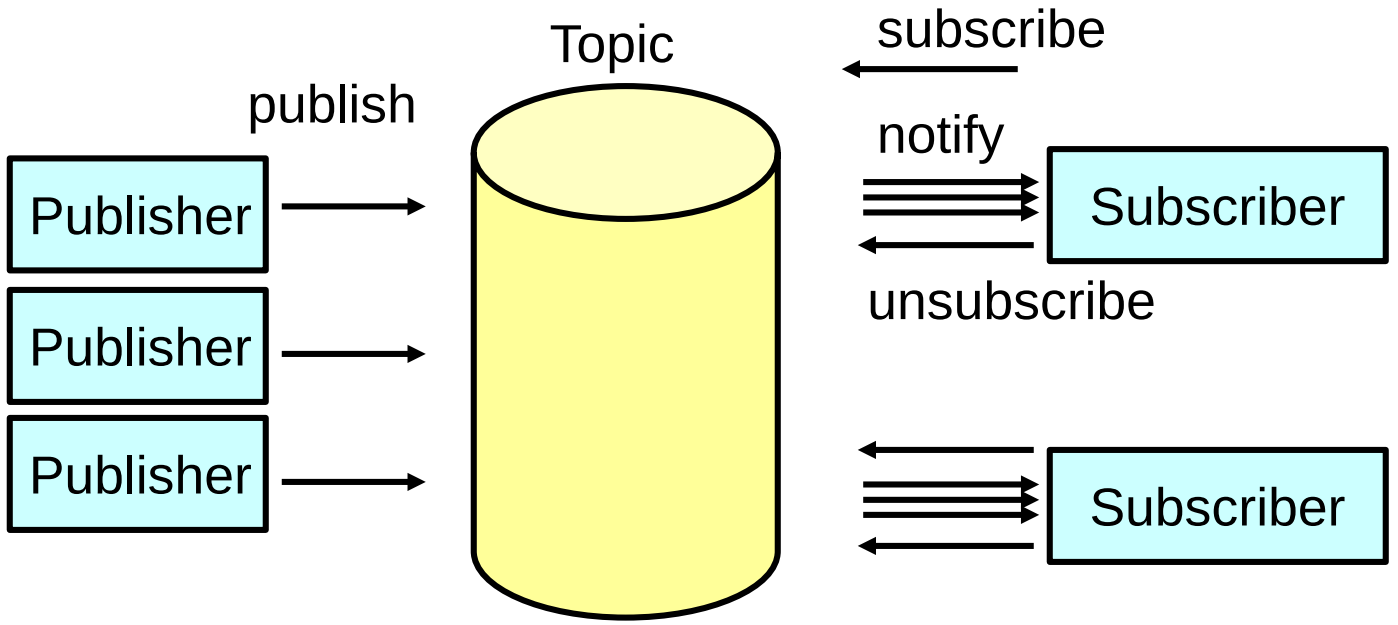
Concepts: Message Queues

- Each message has only one consumer
- Receiver acknowledges successful processing of message
- No timing dependencies between sender and receiver
- Queue stores message (persistently), until
  - It is read by a consumer
  - The message expires (leases, TTL)



Concepts: Publish/Subscribe

- Here: **Topic-based** Publish/Subscribe
  - Interested parties can subscribe to a topic (channel)
  - Applications post messages explicitly to specific topics
- Each message may have multiple receivers
- **Full decoupling** in space, time, and flow



Terms

- **Event**: Any happening in the real world or any kind of state change inside an information system that is observable
- **Notification**: The reification (ger. *Verdinglichung*) of an event as a data structure
- **Message**: Transport container for notifications and control messages

Classification (I)

- Messaging domain
  - Point-to-point (producer → consumer)
  - Subscription-based pub/sub
  - Advertisement-based pub/sub
- Subscription mechanism
  - Channel-based (=topic-based) subscription
  - Content-based subscription
  - Subject-based subscription (limited form of content-based sub.)
- Server topology
  - Single server, hierarchical, acyclic peer-to-peer, generic peer-to-peer

Classification (II)

- Event data model
  - Untyped vs. typed vs. object-oriented
- Event filters
  - Expressiveness and flexibility of subscription language
    - Simple expressions
    - SQL-like query language
  - Evaluated in event system (router/broker network)
- **Note**: **scalability ↔ expressiveness tradeoff**
  - Simple expressions permit filter merging ↔ better scalability

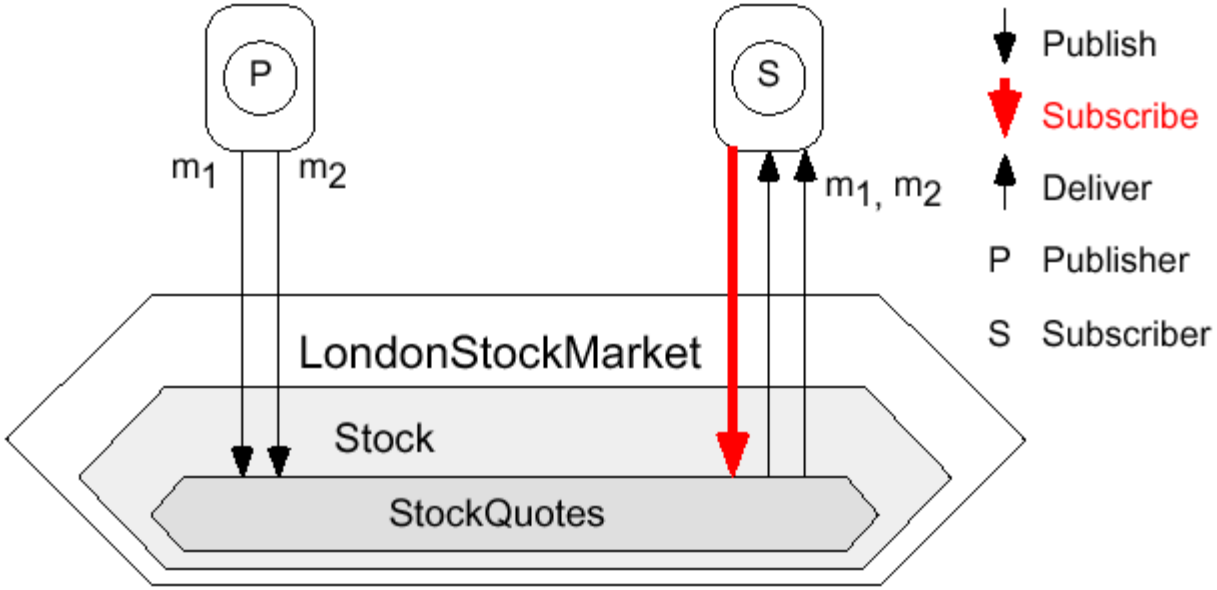
Classification (III)

- Features
  - Scalability
  - Security
  - Client mobility
    - Transparent vs. native vs. external
  - Disconnection
  - Quality of Service (QoS)
    - Reliability & response time (real-time constraints)
  - Transactions
  - Exception handling

Addressing

[Eugster, op. cit.]

- Channel-based addressing (=**topic**-based)
  - Interested parties can subscribe to a channel (analogous to mailing list)
  - Application (producer) posts messages explicitly to a specific channel
  - Channel Identifier is only part of message visible to event service
  - There is no interplay between two different channels



Addressing (II)

- Channel-based addressing (=**topic**-based)
  - Extension: **topic hierarchies** (e.g., SwiftMQ)

    <roottopic>.<subtopic>.<subsubtopic>
  - Messages are 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

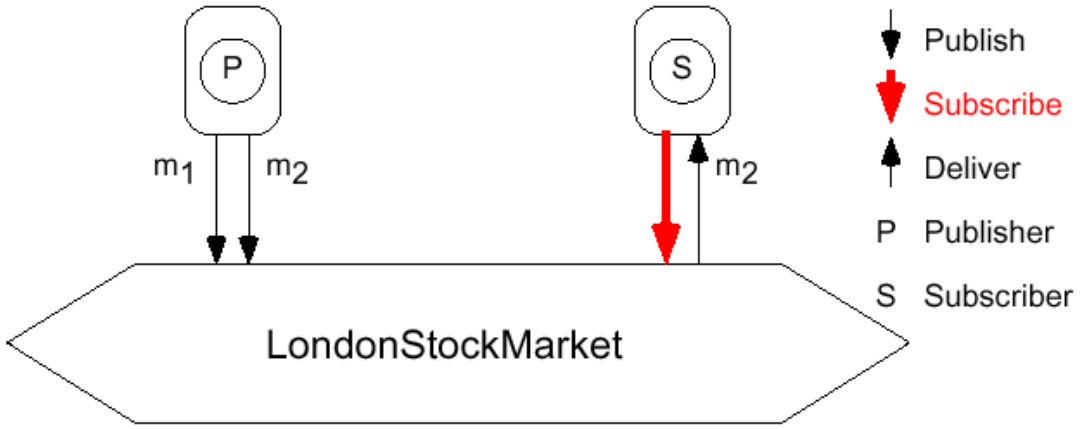
Addressing (III)

- **Subject**-based addressing
  - Limited form of content-based subscription
  - Notifications contain a **well-known attribute** – the **subject** – that determines their address
  - Subscriptions **express interest** in subjects by some form of **expressions** to be evaluated against the subject
  - Subject is
    - List of strings (e.g., TIB/Rendezvous, JEDI)
    - Properties: typed key/value-pairs (e.g., Java Messaging Service (JMS))
  - Subject (= header of notification) is visible to event service, remaining information is opaque
  - Subscription is
    - (Limited form of) regular expressions over strings (TIB, JEDI) or subset of SQL92 queries (JMS)
  - Filtering is done in the event system (router/broker network)!

Addressing (IV)

[Eugster, op. cit.]

- **Content**-based subscription
  - Domain of filters extended to the whole content of notification (i.e., payload)
  - More freedom in encoding data upon which filters can be applied
  - More information for event service to set up routing information



m_1: { ..., company: "Telco", price: 120, ..., ... }

m_2: { ..., company: "Telco", price: 90 , ..., ... }

Addressing (V)

- Content-based addressing

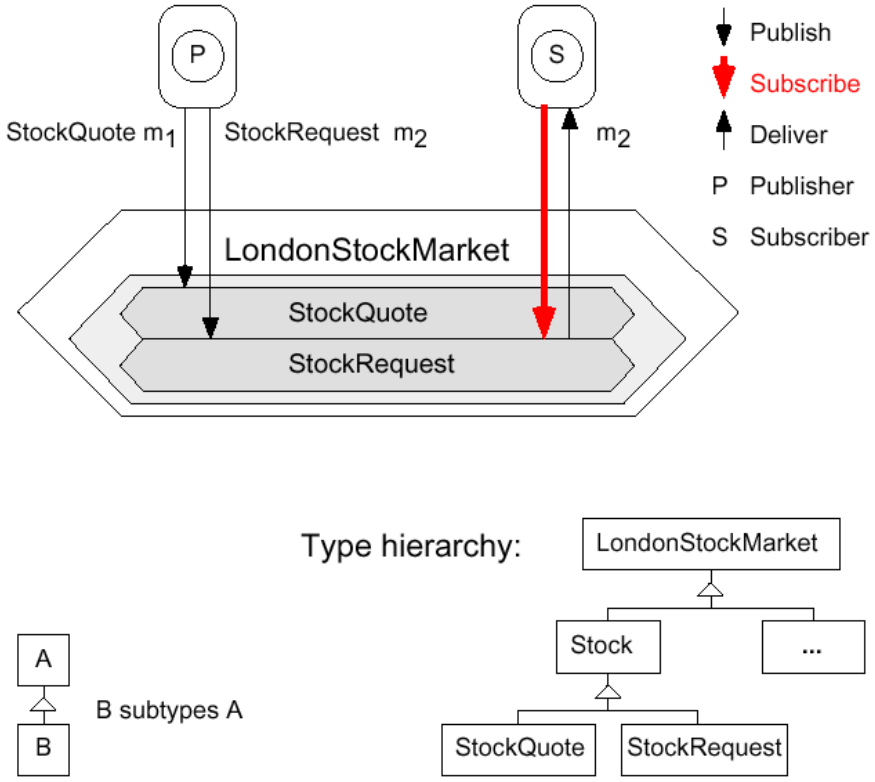
  Content-based ↔ subject-based

  - Subject-based requires some preprocessing by publisher
    - Information that might be used by subscribers for filtering must be placed in header fields
    - Thus producer makes assumptions about subscribers‘ interests
  - Content-based
    - Subscribers exclusively describe their interests in filter expressions
- **Concept**-based addressing
  - Provides higher level of abstraction for description of subscribers’ interests
  - Matching of notifications and transformation of notifications based on ontologies

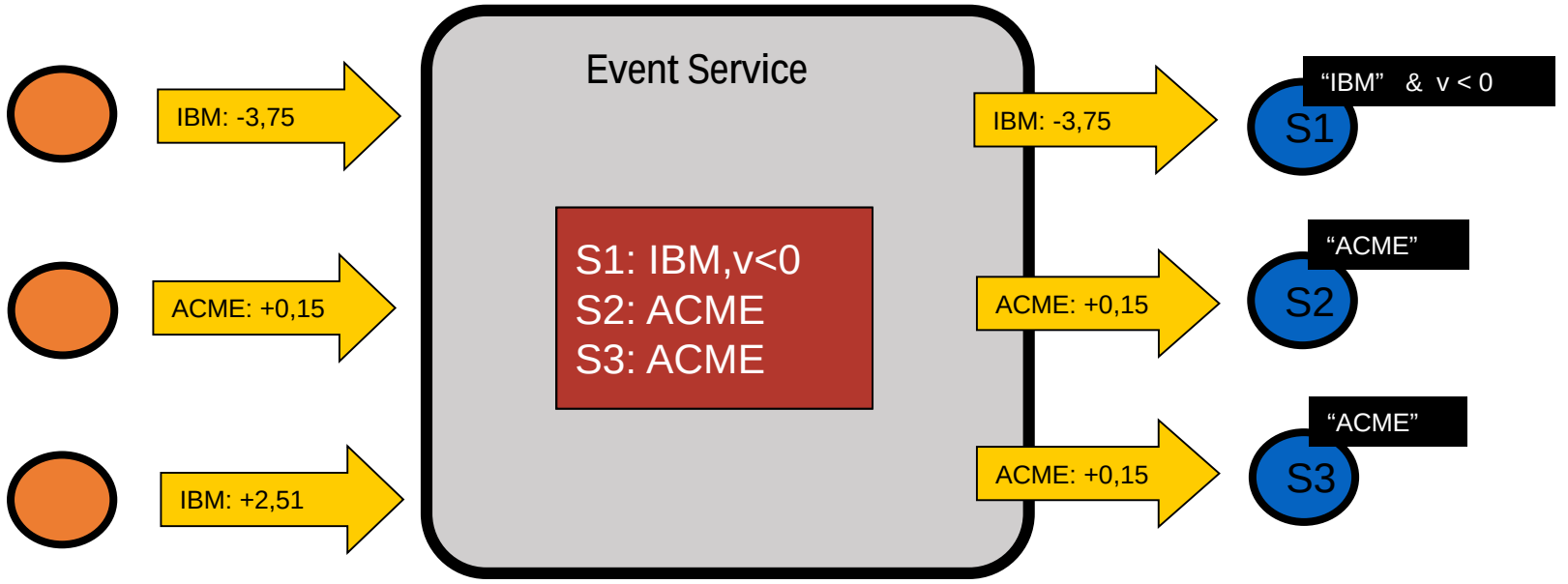
Addressing

[Eugster, op. cit.]

- **Type**-based addressing
  - Similar to channel-based pub/sub with **hierarchies**
  - Supports subtype tests (*instanceof*)
  - Good integration of middleware & language, type safety



Addressing - Example



Topic-Based [Oki et al. 93]:
- events are divided in topics
- subscribers subscribe for a single topic

Content-Based [Carzaniga et al 2001]:
- subscriptions are generic queries, i.e. SQL-like on the event schema

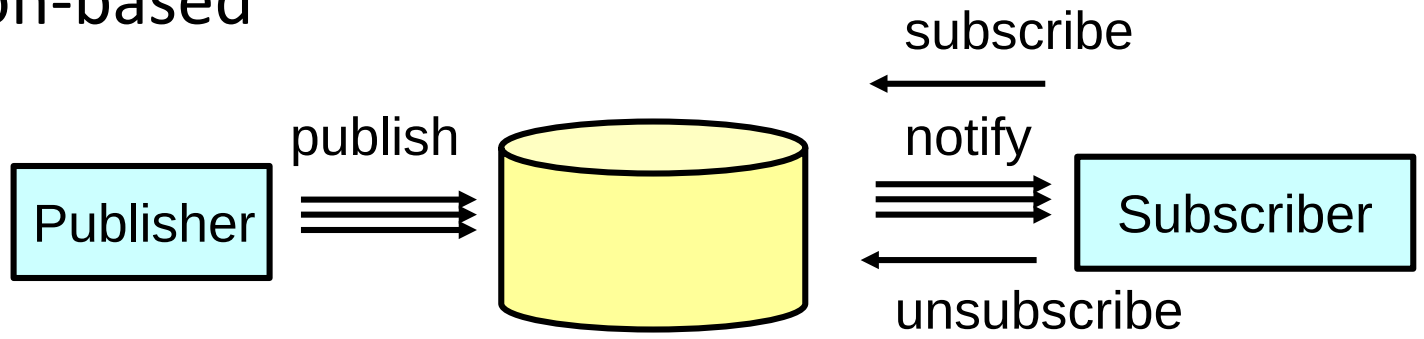
Topic vs. Content-based Subscription

- Topic-based
  - Recipients (consumer) are known a-priori
  - Many efficient implementations exist
  - Limited expressiveness

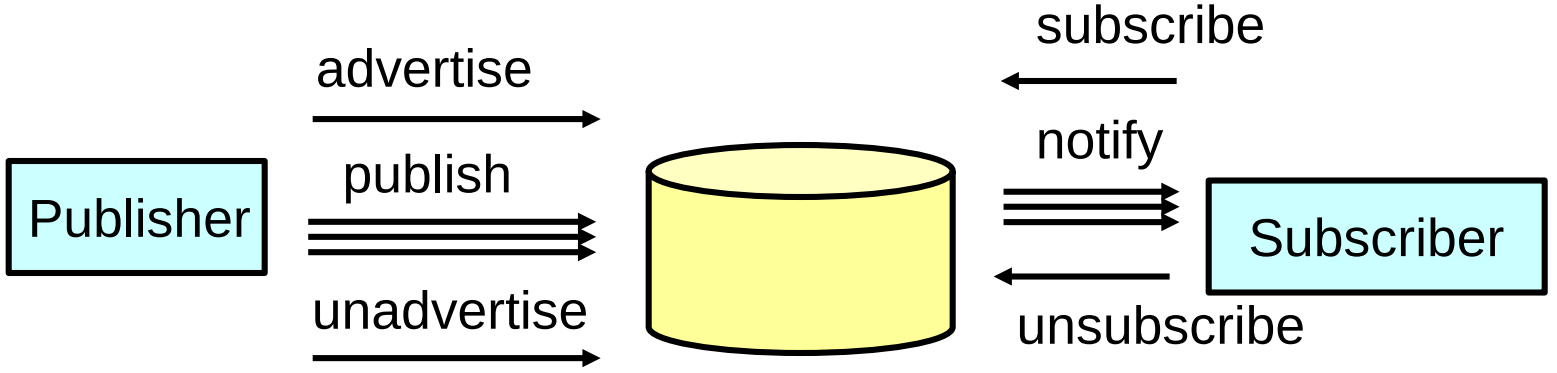
- Content-based
  - Cannot determine recipients before publication
  - More flexible
  - More general
  - Much more difficult to implement efficiently

Subscription Mechanisms

- Subscription-based

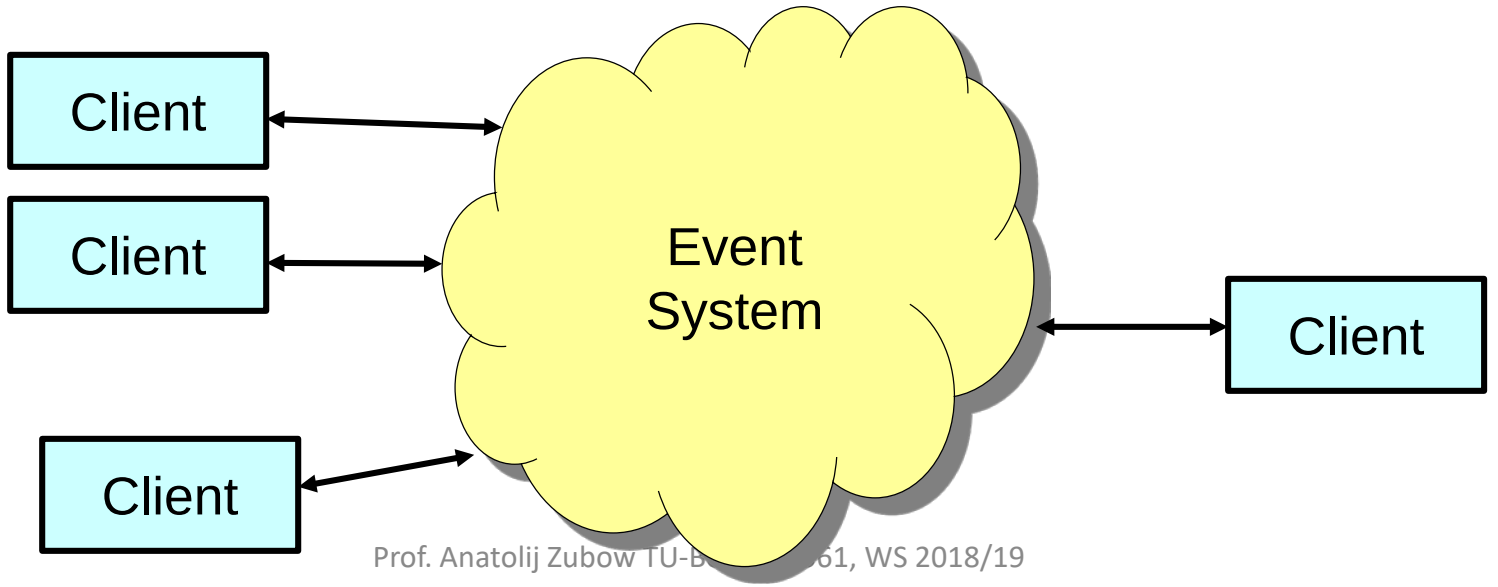


- Advertisement-based



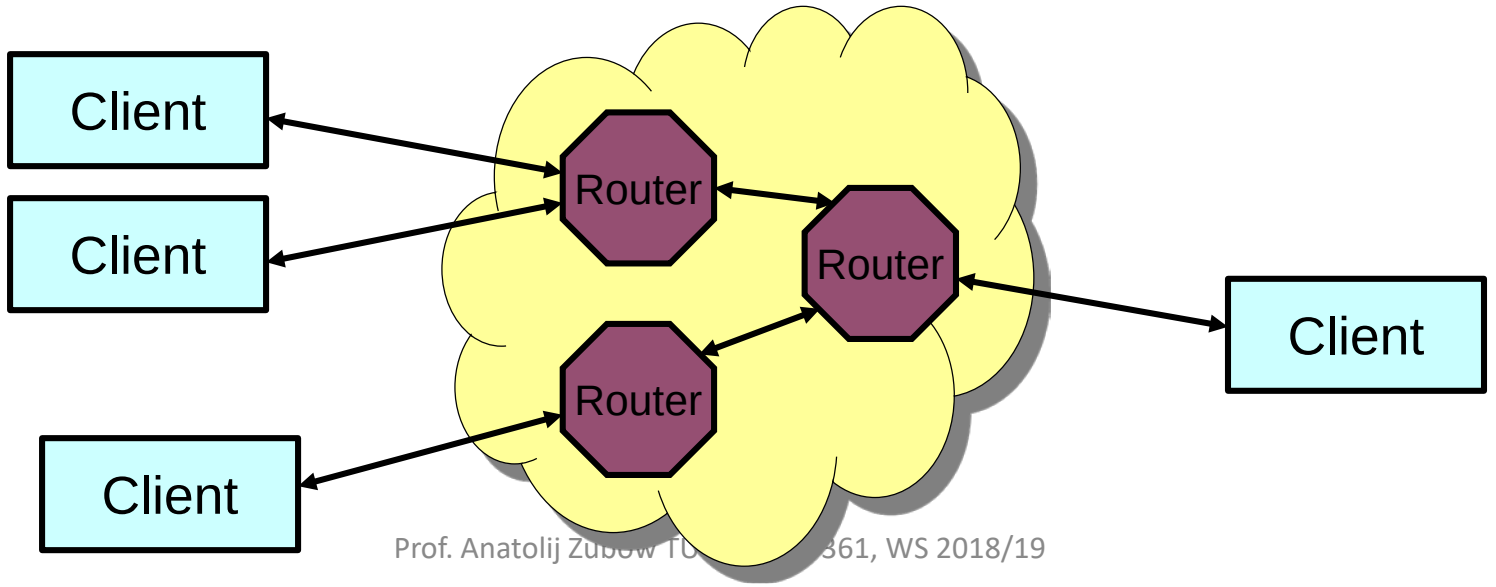
Distributed Event Systems

- (Distributed) event systems
  - … permit loosely coupled, asynchronous point-to-multipoint communication patterns
  - … are application independent infrastructures
  - Only clients communicating via/with a **logically centralized** component



Distributed Event Systems

- Logically centralized component
  - Single server or network of event routers (brokers)
  - Transparent (i.e. not visible) for application (=client)
    Router network can be reconfigured independently and without changes to the application → **Scalability**



Data Model

- **Notification**
  - Consists of a nonempty **set of attributes** $\{a_1, \ldots, a_n\}$
  - An **attribute** is a **triple** $a_i=(n_1, t_1, v_1)$, 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 written as {(*pos.x*, int*,* 1), (*pos.y,* int*, 2*)}
  - Objects can be externalized (serialized) into a tree structure

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 serve as parameter for the operator
- An attribute *a* **matches** an attribute filter *A*, iff (if and only if, gdw.)

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

Filters

- A **filter** is a stateless boolean predicate $F(n) \longrightarrow \{true, false\}$ that is applied to a notification n
- If $F(n)$ evaluates to *true*, we say that notification *n* matches filter $F(n)$
- Filters that only consist of a single attribute filter are called simple filters, and filters containing multiple attribute filters are called compound filters
- **Compound filters** of the form $F = A_1 \wedge \ldots \wedge A_n$ that only contain conjunctions, are called conjunctive filters
- A notification *n* **matches** a filter F, iff it satisfies all attribute filters of F:

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

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

Matching: Example

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

Covering (for interested reader)

- 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 relations are required to **identify and merge similar filters**

Overlapping (for interested reader)

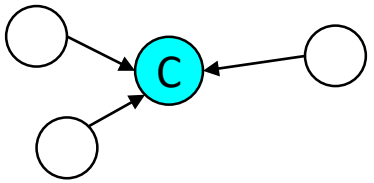
- The filters F_1 and F_2 are ***overlapping***, iff

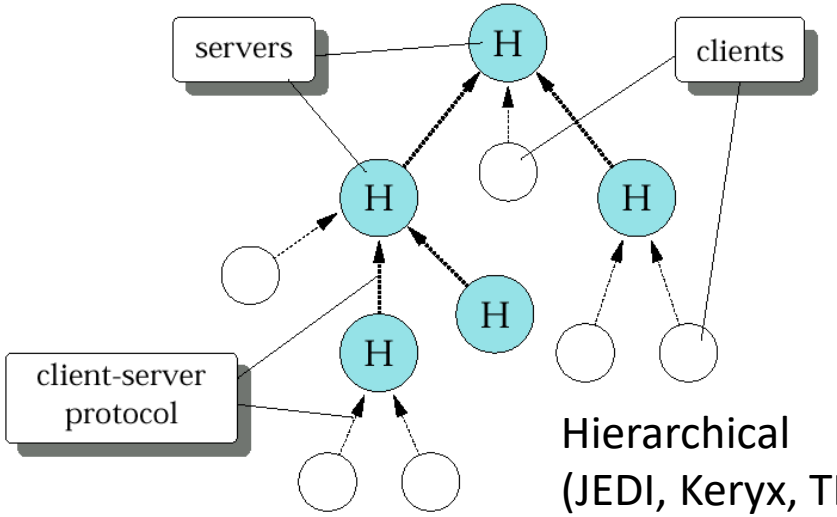
$$F_1 \sqcap F_2 :\Leftrightarrow$$
$$\neg\exists A_{1,i}, A_{2,j} : (n_{1,i} = n_{1,j} \wedge (t_{1,i} \neq t_{1,j} \vee L_A(A_{1,i}) \cap L_A(A_{2,j}) = \emptyset))$$

- 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.

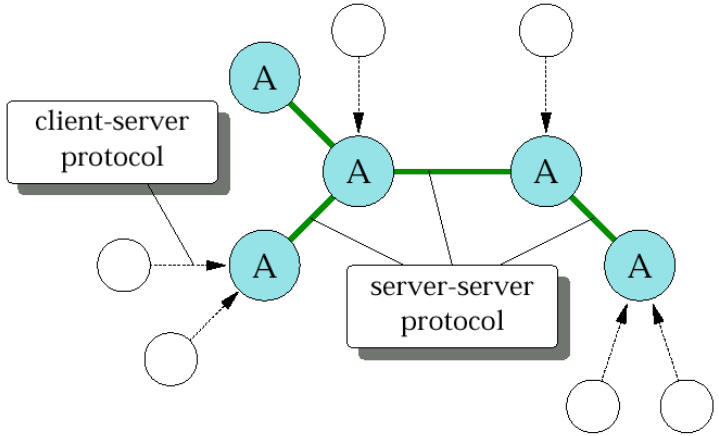
Router Topologies

Centralized Server
(Elvin3)

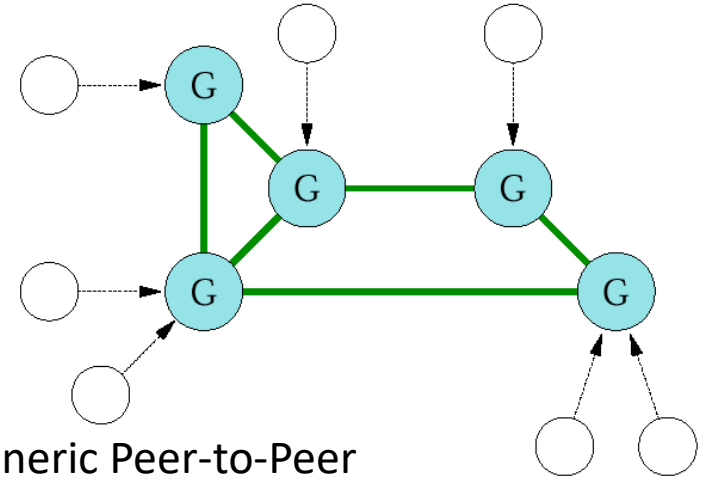




Hierarchical
(JEDI, Keryx, TIB)



Acyclic Peer-to-Peer



Generic Peer-to-Peer
(SIENA)

Routing of Requests

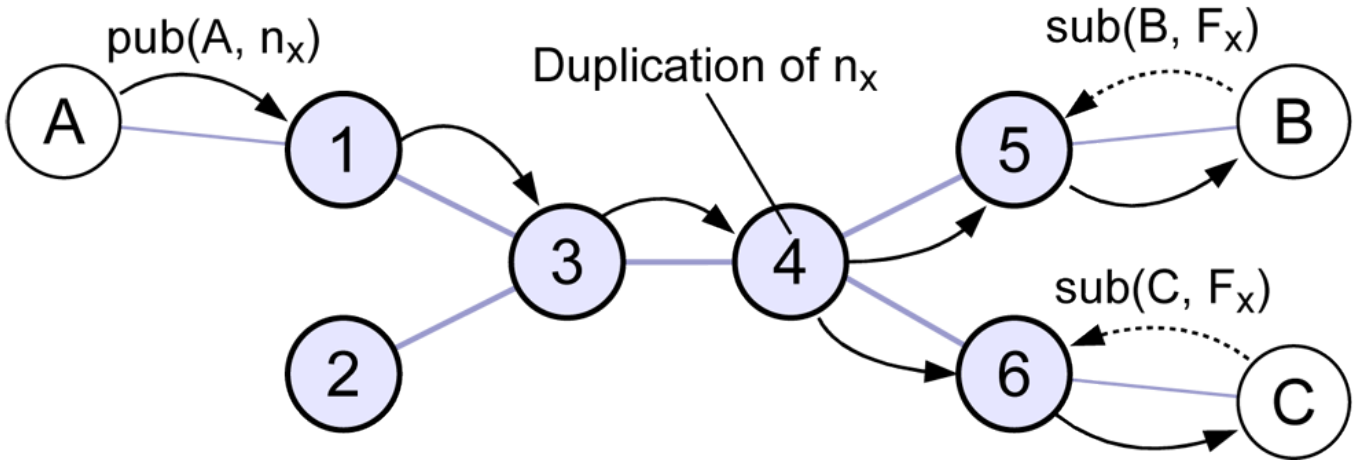
- The network of brokers forms an **overlay network**
- Routing can be split up into two layers
  - At the lower level, requests, i.e. control messages and notifications must be routed between brokers
  - At the higher level, notifications must be routed according to subscriptions and advertisements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

- Naive approaches:
  - **Notification flooding**
    - Flooding of notifications to all brokers in overlay network
    - Each subscription stored only in one place within the broker overlay network
    - Matching operations equal to the number of brokers
  - **Subscription flooding**
    - Each subscription stored at any place within the broker overlay network
    - Each notification matched directly at the broker where the notification enters the broker overlay network

Routing: Principles

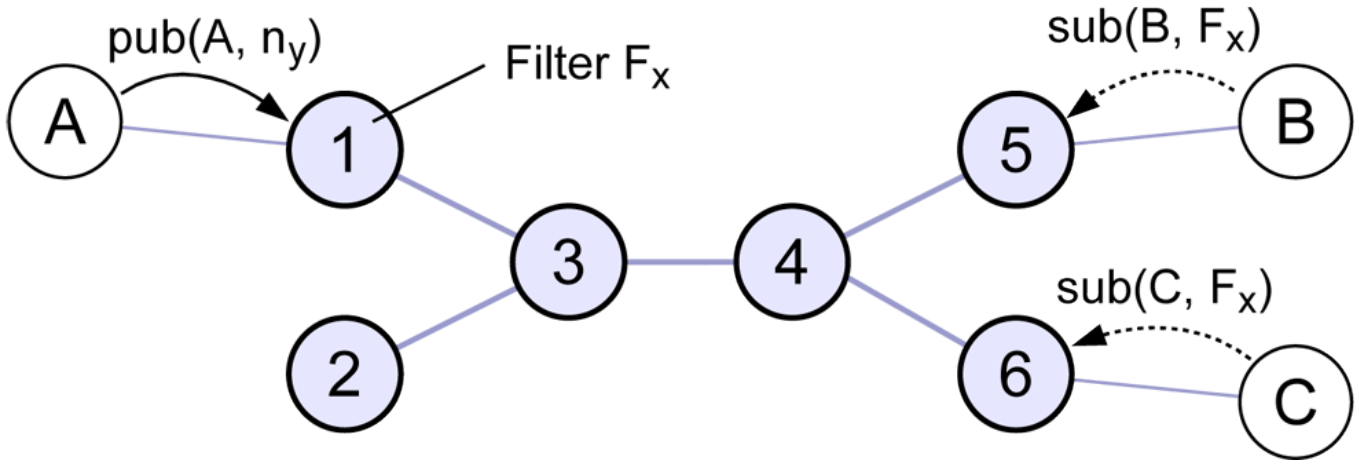
- **Downstream duplication**
  - Route notification as a single copy as far as possible



  - 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 n_y (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 notifications: A notification n is only forwarded to a destination D, iff

$$\exists\, (D, F) \in T_S : n \sqsubseteq F$$

- Routing of subscriptions: If a subscribe or unsubscribe request is received, the table T_S is updated accordingly.
  - Subscribe or unsubscribe requests are potentially forwarded to all neighbors according to the underlying routing algorithm

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.

Scalability

- System should be **scalable** in terms of
  - the number of clients (i.e., producers and consumers),
  - the number of event routers (brokers),
  - the number of subscriptions and advertisements, and
  - the amount of traffic (e.g., number of notifications/second)
- Problems in unstructured peer-to-peer overlays
  - Either subscriptions or advertisements forwarded to each node
    - Assumption (for Internet-based services): advertisements are rather static, subscriptions are dynamic → use routing with advertisements
  - Routing tables grow proportionally with the size of the network

    → use filter merging

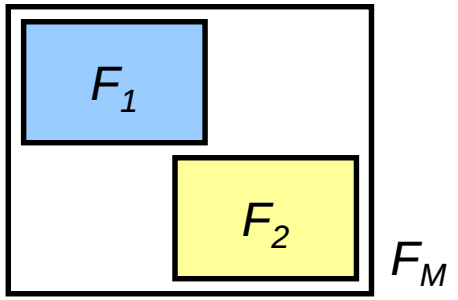
    → use structured overlays

Filter Merging (for interested reader)

- Inexact Merging

F_M is an inexact merge of F_1 and F_2 iff

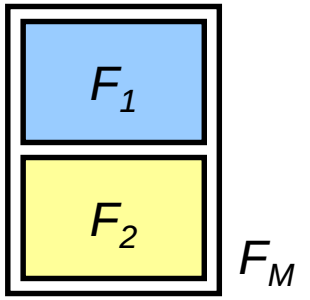
$$F_M \sqsupseteq F_1 \wedge F_M \sqsupseteq F_2$$



- Exact Merging

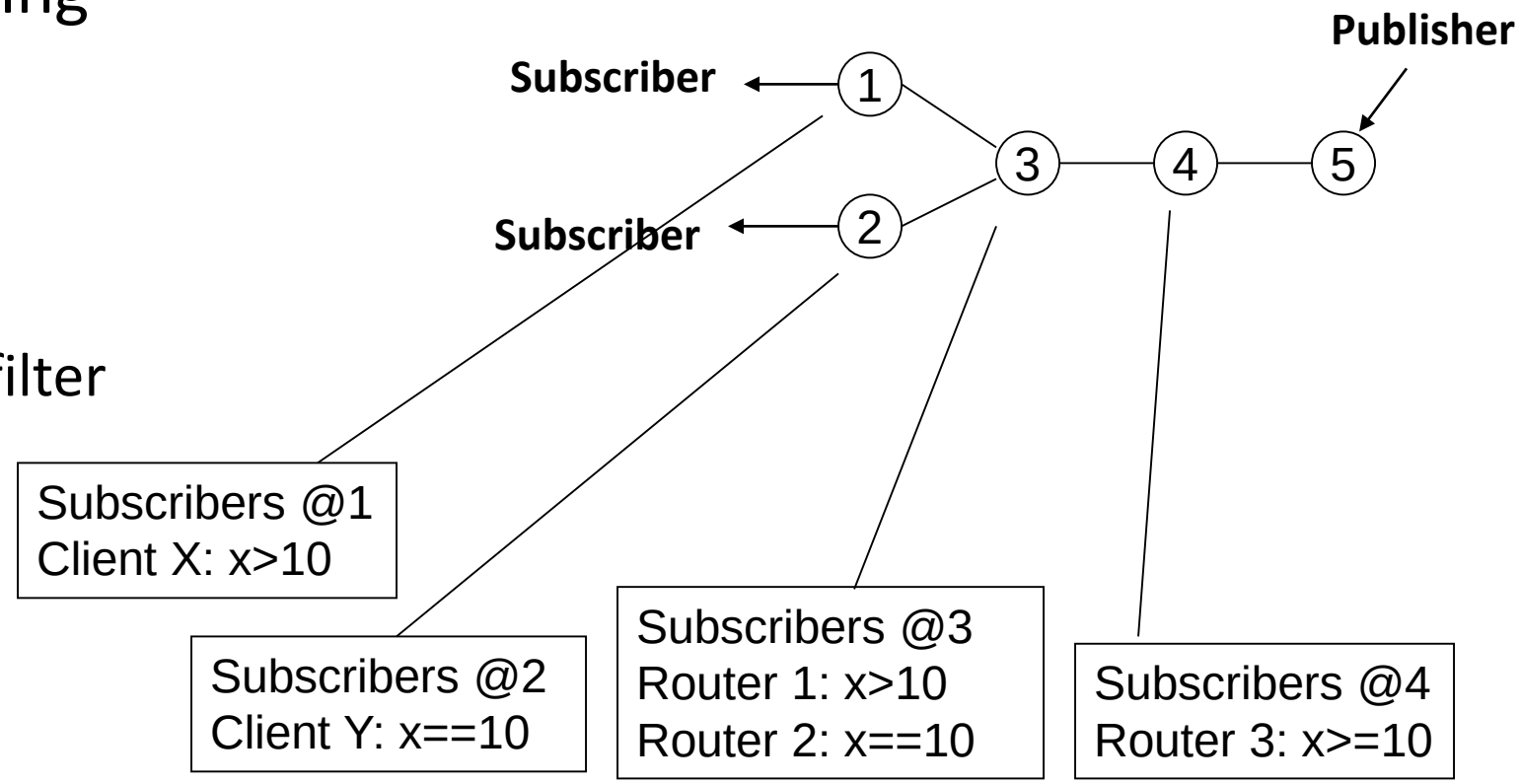
F_M is an exact merge of F_1 and F_2 iff

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



Filter Merging: Example

- Filter merging
  - Filter X
    x>10
  - Filter Y
    x==10
  - Merged filter
    x>=10



Structured Overlays

- Systems based on **distributed hash tables** (e.g. SCRIBE)
- In a DHT, the storage location of an information item is defined by its hash value
  - Channel-based addressing: calculate hash value from channel name
  - Content-based addressing: no general solution
    → „Channelization“: calculate hash from selected attributes, e.g. message type
- The (global) subscription table is distributed over the network
  - A broker is responsible for specific subscriptions
  - The broker is the rendezvous point for publishers and subscribers

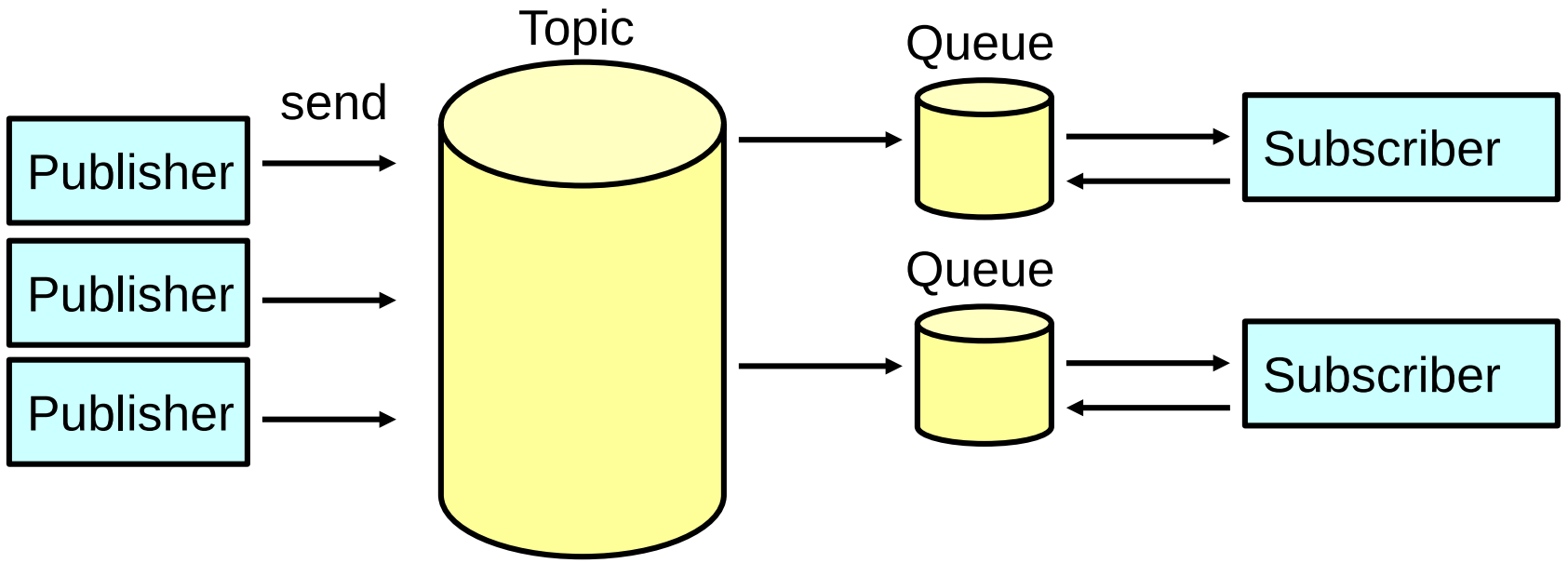
Structured Overlays (II)

- Routing of subscriptions
  - Subscriber calculates hash of subscription h(S) and sends it to the broker with hash h(B) closest to h(S). The subscription is stored at B.
- Routing of notifications
  - Publisher calculates hash of notification h(n) and sends it to the broker with h(B) closest to h(n). Broker B has a list of all relevant subscribers.

QoS and Transactions

- Quality of Service
  - Guaranteed delivery
    - Logistics
    - Stock quotes
  - Low latency
    - Sensor, audio, or video data streams
- Local Transactions
  - between the publisher and the event service, or between the event service and the subscriber
  - groups a series of operations into an atomic unit of work

Mobility Support: Durable Subscriptions



- Messages are stored for each subscriber
  - Permits disconnection of subscriber
  - But: subscriber bombarded with messages on reconnect (remedy: use TTL)

Generic Pub/Sub Interface

- **subscribe():**
  - register interest in events to event service (without specifying the effective sources of these events)
  - usually not forwarded to publishers
- **unsubscribe():**
  - terminates a subscription
- **publish():**
  - generates an event at the event service
  - usually propagates event to all relevant subscribers
- **advertise():**
  - expresses the intent to publish a particular kind of event
  - can be used for discovery of publishers or to simplify matching
  - not strictly needed and often not implemented

System Examples

- Industry-strength
  - **JMS**
  - CORBA Notification Service
  - **ZeroMQ**
  - Elvin
  - IBM WebSphere MQ Event Broker (Gyphon)

- Academic prototypes
  - REBECA
  - SIENA

JMS: Java Message Service

- API - „Common set of interfaces and associated semantics“
- Domains
  - Point-to-point: message-queue
  - Publish/subscribe
    - Topic-based
    - Subject-based
    - Durable subscribers
- Separated administration
  - Queues and topics are created with product-specific administration tools
  - Application independent
  - Support of local transactions

JMS: Java Message Service

- 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
- Event consumption
  - Synchronously: subscriber explicitly fetches message from destination
  - Asynchronously: subscriber registers a message listener

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)
  - identifier [NOT] LIKE pattern-value [ESCAPE escape-character]
  - identifier IS [NOT] NULL (comparison operator that tests for a null header field value or a missing property value)

JMS: Message Filtering (II)

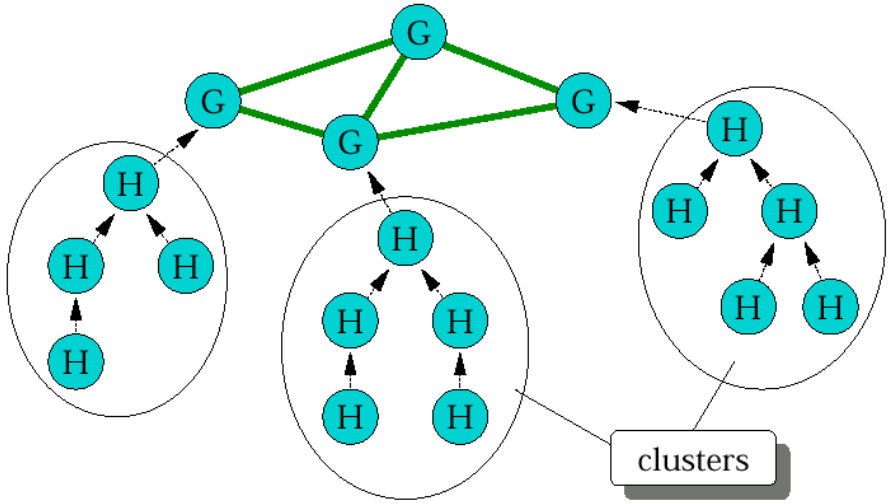
- Examples:
  - NewsType='Opinion' OR NewsType='Sports'
  - phone LIKE '12%3'
  - JMSType='car' AND color='blue' AND weight>2500

JMS: SwiftMQ

- Domain
  - Point-to-point
  - Topic- and subject-based publish/subscribe
- Server topology
  - Generic peer-to-peer: federated router network
- Features
  - Fully implements JMS 1.0.2 specification
  - Topic hierarchies
  - SQL-Like predicate topic addressing
    Permits subscription with topic name wildcard. Example:
    iit.s%s._S matches iit.sales.US
  - File based persistent message store

SIENA

- SIENA = Scalable Internet Event Notification Architecture
- Domain
  - Advertisement-based publish/subscribe
  - Content-based subscriptions
- Server topology
  - Generic peer-to-peer
  - Hybrid topology
    - LAN: hierarchical
    - WAN: generic peer-to-peer
- Data model
  - Notification is set of attribute=(name, type, value)
  - Limited set of types (string, time, date, integer, float, ...)
- Subscription language
  - Filter is set of attr_filter=(name, type, operator, value)
  - Operators: any, =, <, >, >* (prefix), *< (postfix)

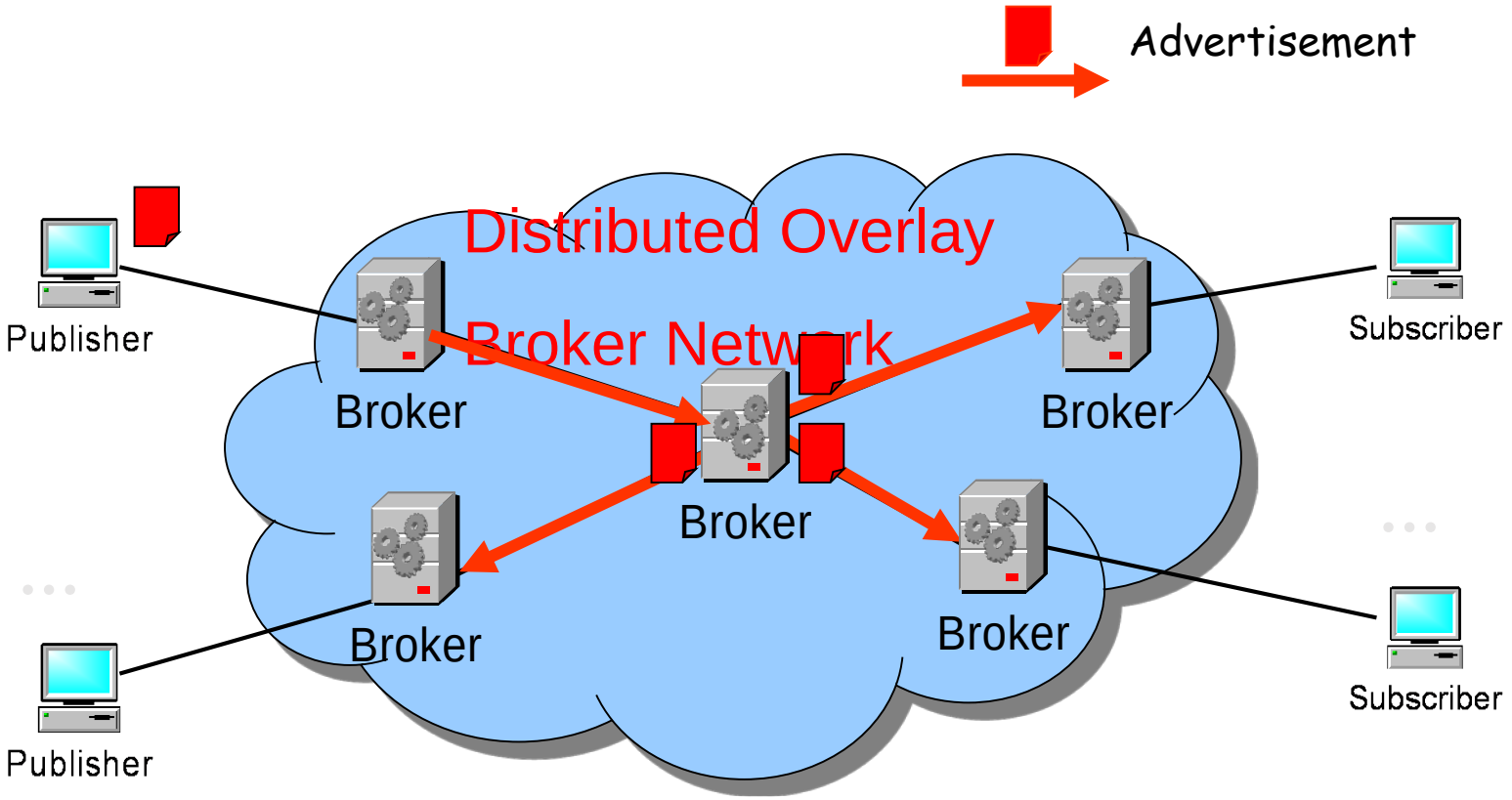


PADRES

- PADRES
  - Uses **advertisement based routing** which is adopted from SIENA and REBECA system.
  - Consists of a broker network.
  - Publishers and subscribers are connected to brokers as clients.
  - A publisher advertises before it publishes. The advertisement is flooded in the broker network.

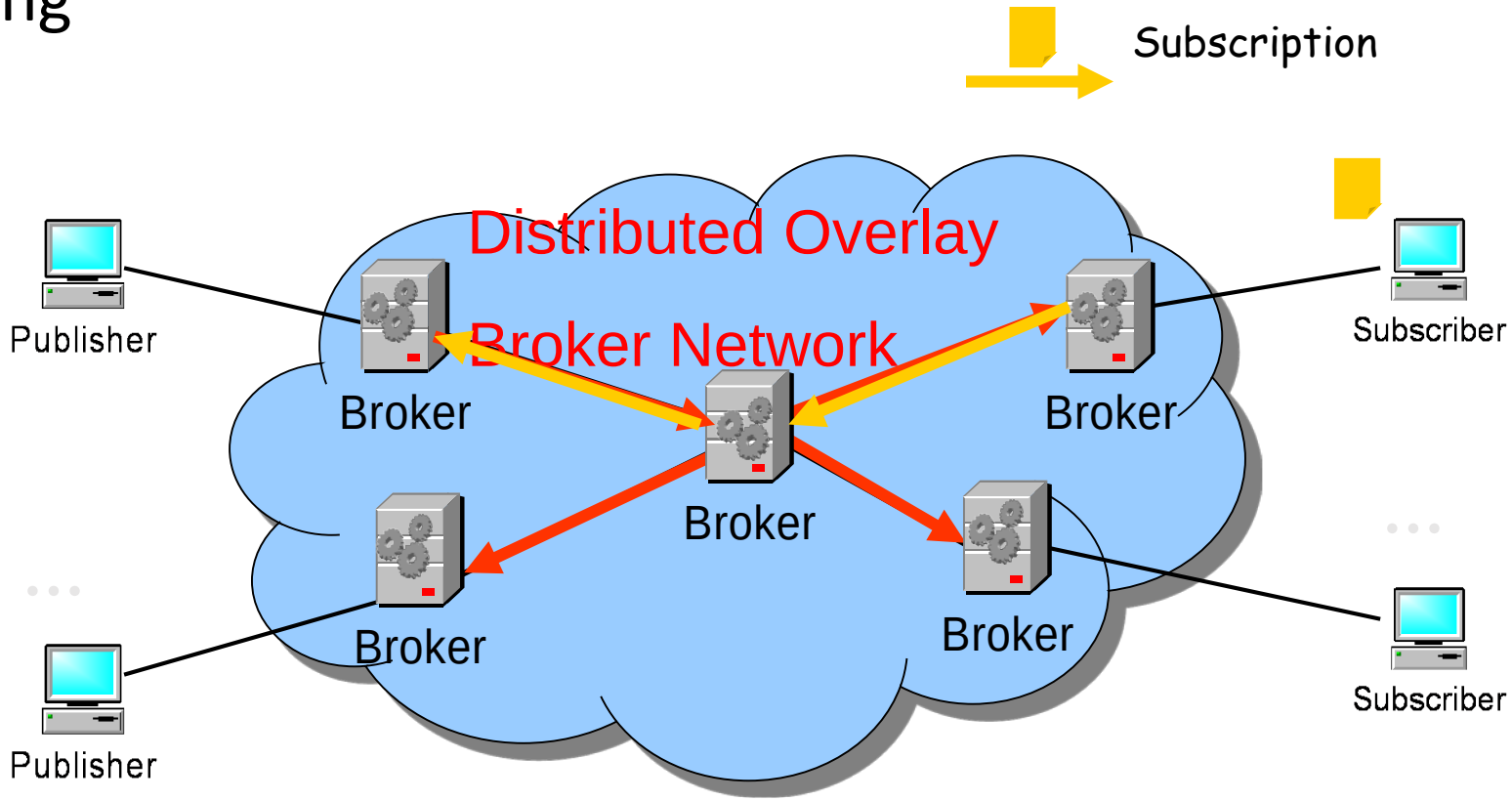
PADRES (II)

- Advertising



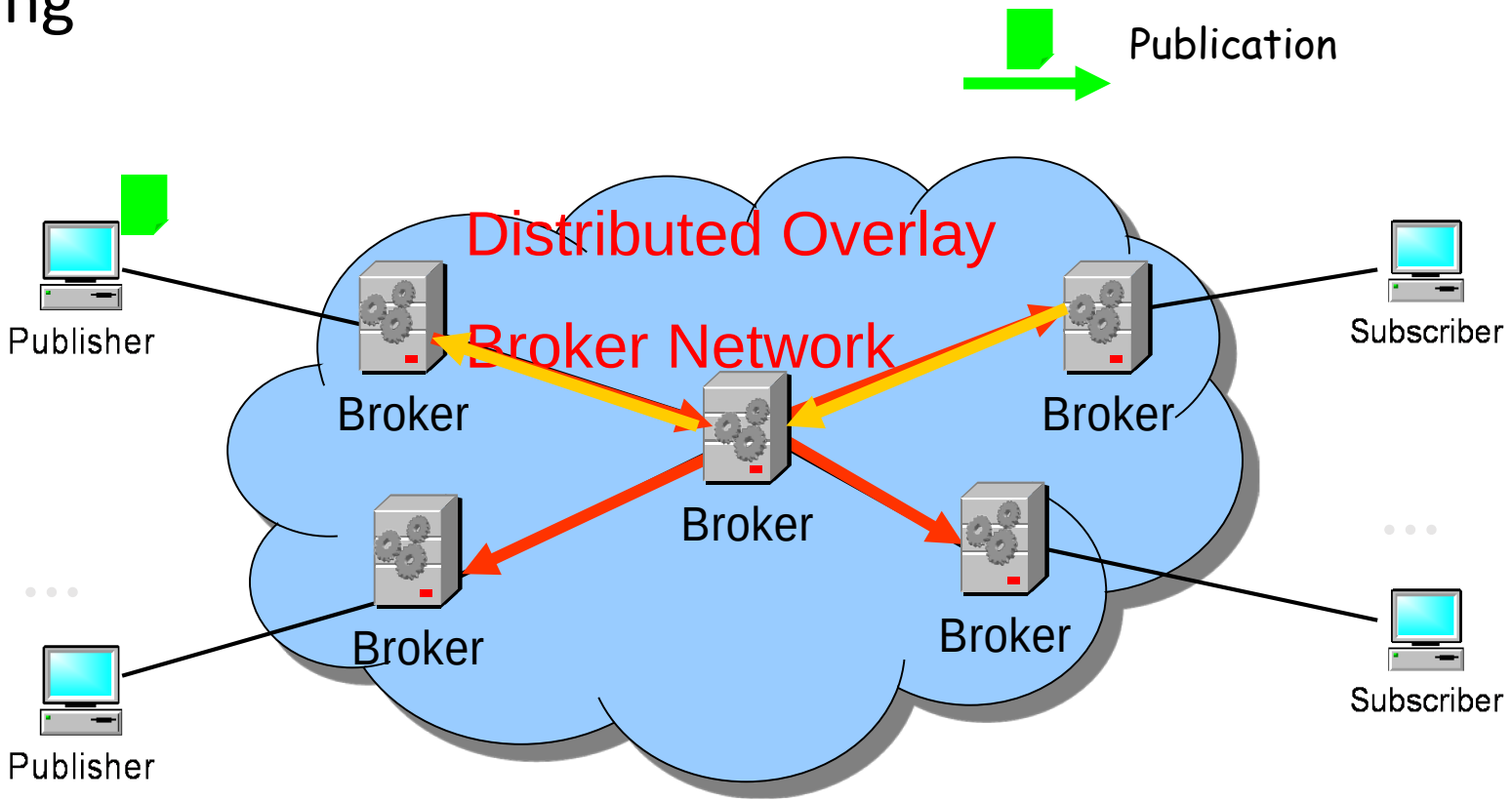
PADRES (II)

- Subscribing



PADRES (II)

- Publishing



Elvin

- Elvin3
  - Subscription-based publish/subscribe
  - Content-based subscriptions
  - Centralized server
- Elvin4
  - Data model: typed key/value-pairs
    Types: integer (32 and 64), string, FP, binary data (opaque)
  - Subscription Language:
    Simple Integer and FP arithmetic
    Strings: POSIX ERE (Extended Regular Expressions),
    begins-with, ends-with, contains (for better optimization)
  - Source code available for non-commercial use
  - Proxy at network boundary to support disconnection

ZeroMQ API

- **An attempt to create a unique interface for different interaction models**
- Open source (http://zeromq.org/)
- ZeroMQ socket: multiple types for different patterns:
  - Request/Response, Publish/Subscribe, etc.
- Uniform API across different programming languages:
  - More than 40 bindings (including C, Erlang, Go, Java, Python, etc.)
- Addressing similar to URL: string [transport]://[address]
  - Automatic DNS resolving

ZeroMQ API (II)

- ZeroMQ offers asynchronous messaging with transparent message queuing:
  - Sending messages does usually not block
  - Messages are queued at sender if destination host is not available or too slow until “high watermark” is reached (queue full), then blocks or discards depending on socket type
  - Can reconnect automatically (e.g. if server temporarily unavailable) when underlying transport signals closed connection

Berkeley Socket API vs. ZeroMQ API

Berkeley Sockets	ZeroMQ equivalent
Socket	zmq_socket()
Bind	zmq_bind()
Listen	-
Accept	-
Connect	zmq_connect()
Send	zmq_send()
Receive	zmg_recv()
Close	zmq_close()

- Most API calls map to Berkeley sockets, except:
  - Sequence Bind/Listen/Accept from Berkeley sockets passive side replaced by the ZeroMQ bind

Publish/subscribe with ZeroMQ

- **Publisher** and **subscriber** use **different sockets types**:
  - PUB for publishers, SUB for subscribers
- Topic based filtering: message prefix matching
- Routing: subscription flooding to publishers (filtering at publishers)
- Topology: PUB and SUB sockets can be connected directly or through brokers (which use XPUB/XSUB sockets and act like a single subscriber or publisher to connecting sockets)
- Subscriptions are aggregated at brokers and only forwarded once

Publish/subscribe Interface using ZeroMQ

General pub/sub API	ZeroMQ equivalent
Socket	zmq_socket()
Bind	zmq_bind()
Connect	zmq_connect()
Publish	zmq_send()
Subscribe	zmq_setsockopt(ZMQ_SUBSCRIBE)
Unsubscribe	zmq_setsockopt(ZMQ_UNSUBSCRIBE)
Close	zmq_close()

Example of pub/sub with ZeroMQ's API

- Usage of C binding:

```
void *context= zmq_ctx_new ();
void *publisher = zmq_socket(context, ZMQ_PUB);
zmq_bind(publisher, "tcp://localhost:5001");

zmsg_send("Hello World", publisher);

zmq_close(publisher );
zmq_ctx_destroy(context);
```

← Publisher

Subscriber →

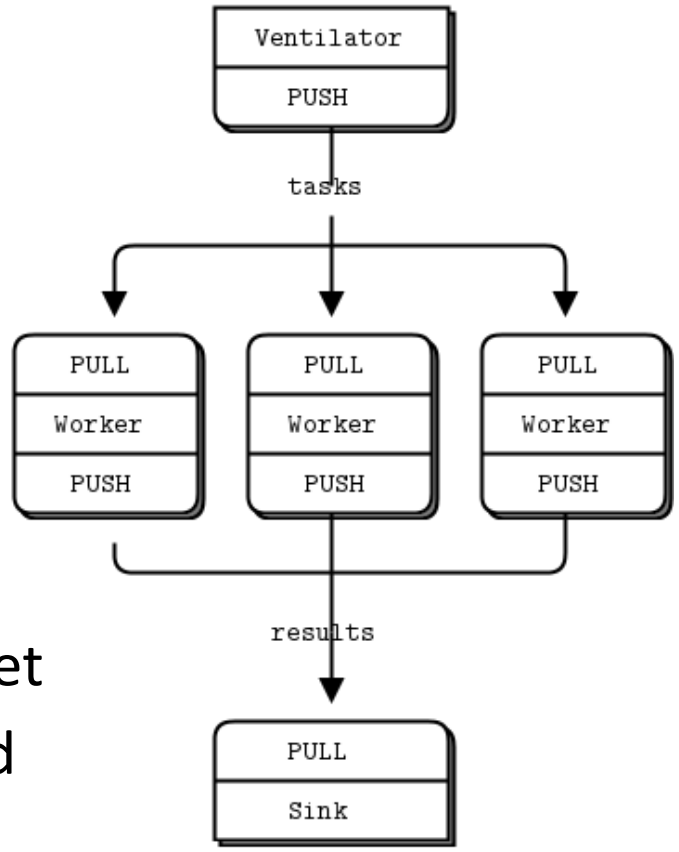
```
void *context = zmq_ctx_new ();
void *subscriber = zmq_socket(context, ZMQ_SUB);
zmq_connect(subscriber, "tcp://localhost:5001");

zmq_setsockopt (socket, ZMQ_SUBSCRIBE, "H", 0);
zmsg_t *msg = zmsg_recv(subscriber);

zmq_close(subscriber);
zmq_ctx_destroy(context);
```

ZeroMQ pipeline messaging pattern

- Intended for **task distribution**
- Tasks are distributed from a ventilator to workers and are aggregated at a sink
- Ventilator uses PUSH socket
  - Sending messages round-robin to workers
- Workers use PULL socket to get tasks
  - Blocking until task is available
  - Queues incoming tasks if busy
- Workers use PUSH Socket to send results to sink's PULL socket
- Because of rigid round-robin distribution only recommended for task that execute quickly



Payload Serialization Formats

- How can we convert complex data objects into a sequence of bits, to be used as part of service request or response messages?
  - Use of “Data Serialization / Encoding Formats”
- Conflicting requirements for data serialization formats
  - Efficient encoding and decoding (time and space/size)
    - Leads to binary encodings
  - Extensibility and backward-compatibility
    - Enables decoupling of the server and client implementations
  - Human readability
    - Simplifies development by increased visibility
  - Type safety
    - Automatic detection of type errors
  - Self-describing schema

Examples of Data Serialization Formats

- JSON
  - Attribute-value pairs
  - Supported types: Number, String, Boolean, Array, Object, null
  - Everything encoded as strings, **not very space efficient**
  - Can be additionally compressed (cJSON, bJSON)
  - **Human readable**, programing language independent
  - Not self describing, no type checking, no schema
- Google Protocol Buffers
  - **Very efficient binary** data serialization format
    - Used as payload serialization for RPC calls and as data persistence format
  - **Strongly typed**, separate schema file (.proto)
  - Requires a compilation step to read and serialize messages
  - **Extensible and backward-compatible**

Examples of Data Serialization Formats (II)

- Apache Avro
  - Efficient binary data serialization format
  - Simple integration in dynamic languages, no compilation needed
  - **Dynamic typing**
  - **Self-contained schema** specification (using JSON)

Summary - Pub/Sub

- **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 with P2P, security, reliability, …