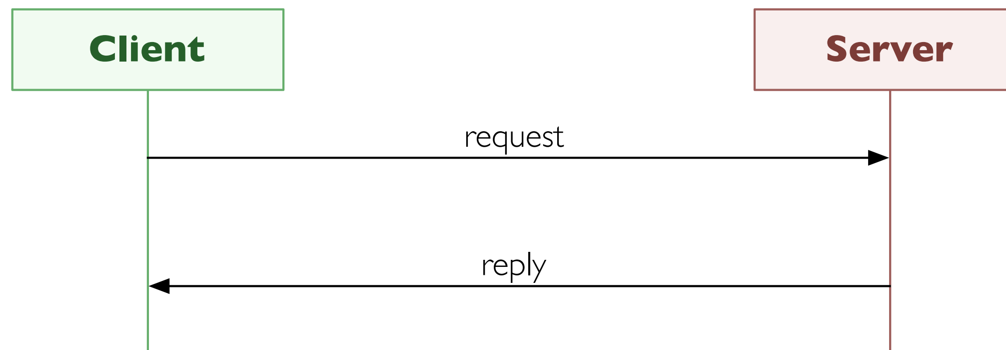
Università di Roma "La Sapienza"
Dipartimento di Informatica e Sistemistica

Publish/Subscribe Systems

Distributed Systems(AA 10/11)

Leonardo Querzoni
querzoni@dis.uniroma1.it

- *Client/server* is the most used architecture for distributed applications.
- Communications take place following the *request/reply* (pull) interaction model.



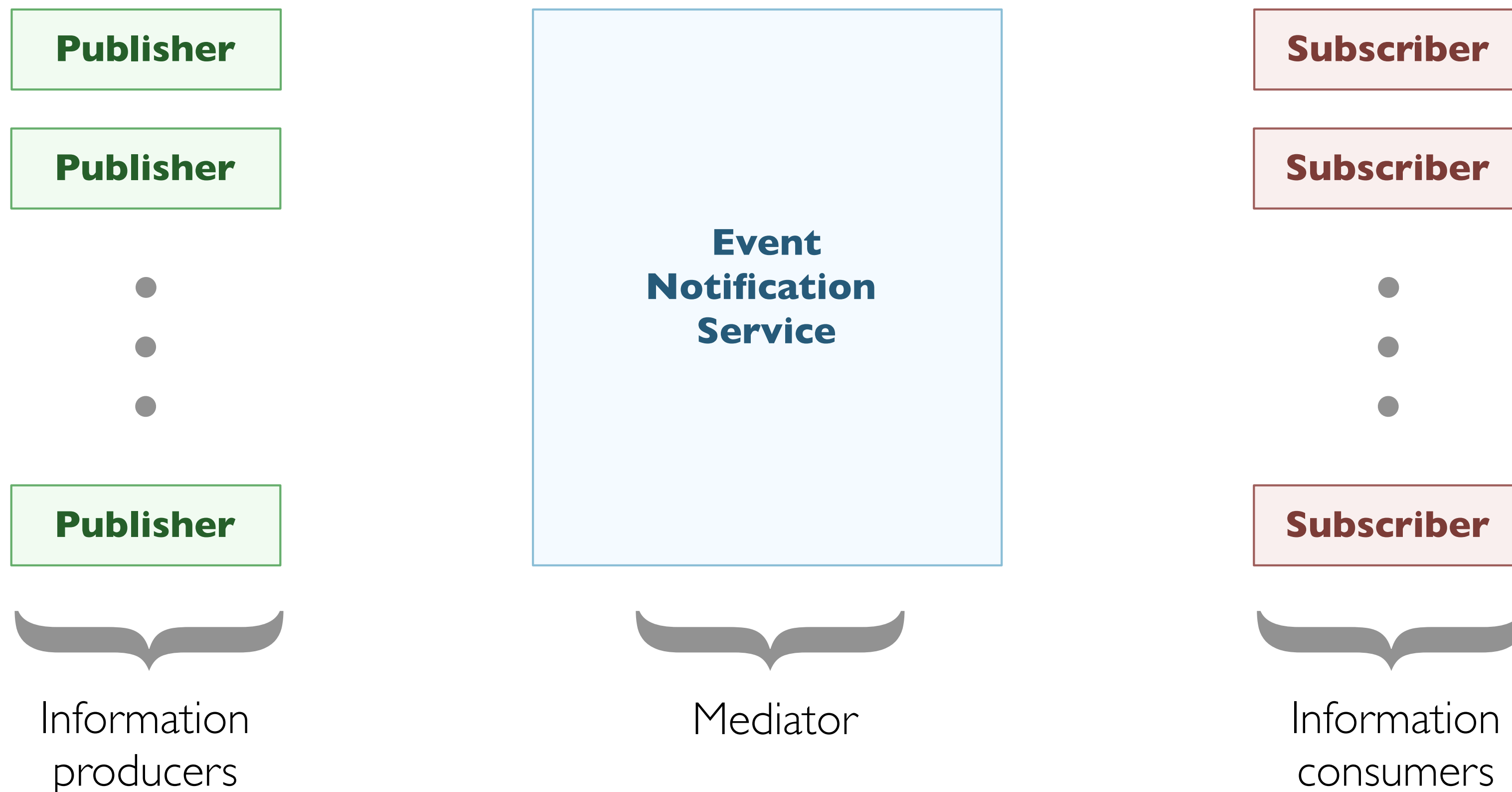
- Drawbacks:
  - interaction is limited to two entities (one-to-one);
  - each entity must know how to address its partner in the communication;
  - the two entities must be available at the same time in order to communicate;
  - communication is inherently synchronous;
  - Communication is only pull-based.
- To address these issues many alternative models were introduced: RPC, Shared memories, Message queues, Publish/subscribe

Publish/subscribe was thought as a comprehensive solution for those problems:

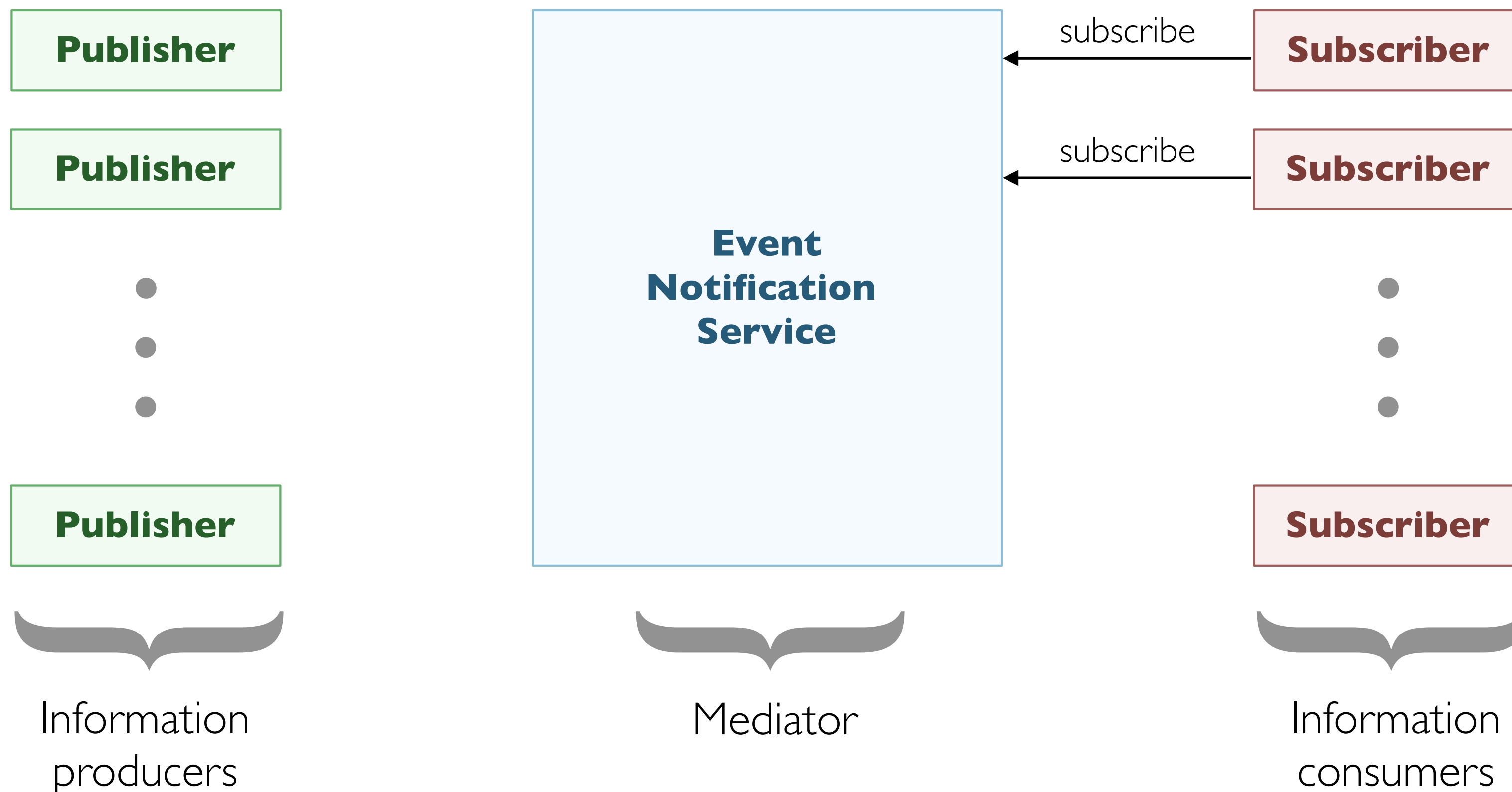
- **Many-to-many communication model** - Interactions take place in an environment where various information producers and consumers can communicate, all at the same time. Each piece of information can be delivered at the same time to various consumers. Each consumer receives information from various producers.
- **Space decoupling** - Interacting parties do not need to know each other. Message addressing is based on their content.
- **Time decoupling** - Interacting parties do not need to be actively participating in the interaction at the same time. Information delivery is mediated through a third party.
- **Synchronization decoupling** - Information flow from producers to consumers is also mediated, thus synchronization among interacting parties is not needed.
- **Push/Pull interactions** - both methods are allowed.

These characteristics make pub/sub perfectly suited for distributed applications relying on document-centric communication.

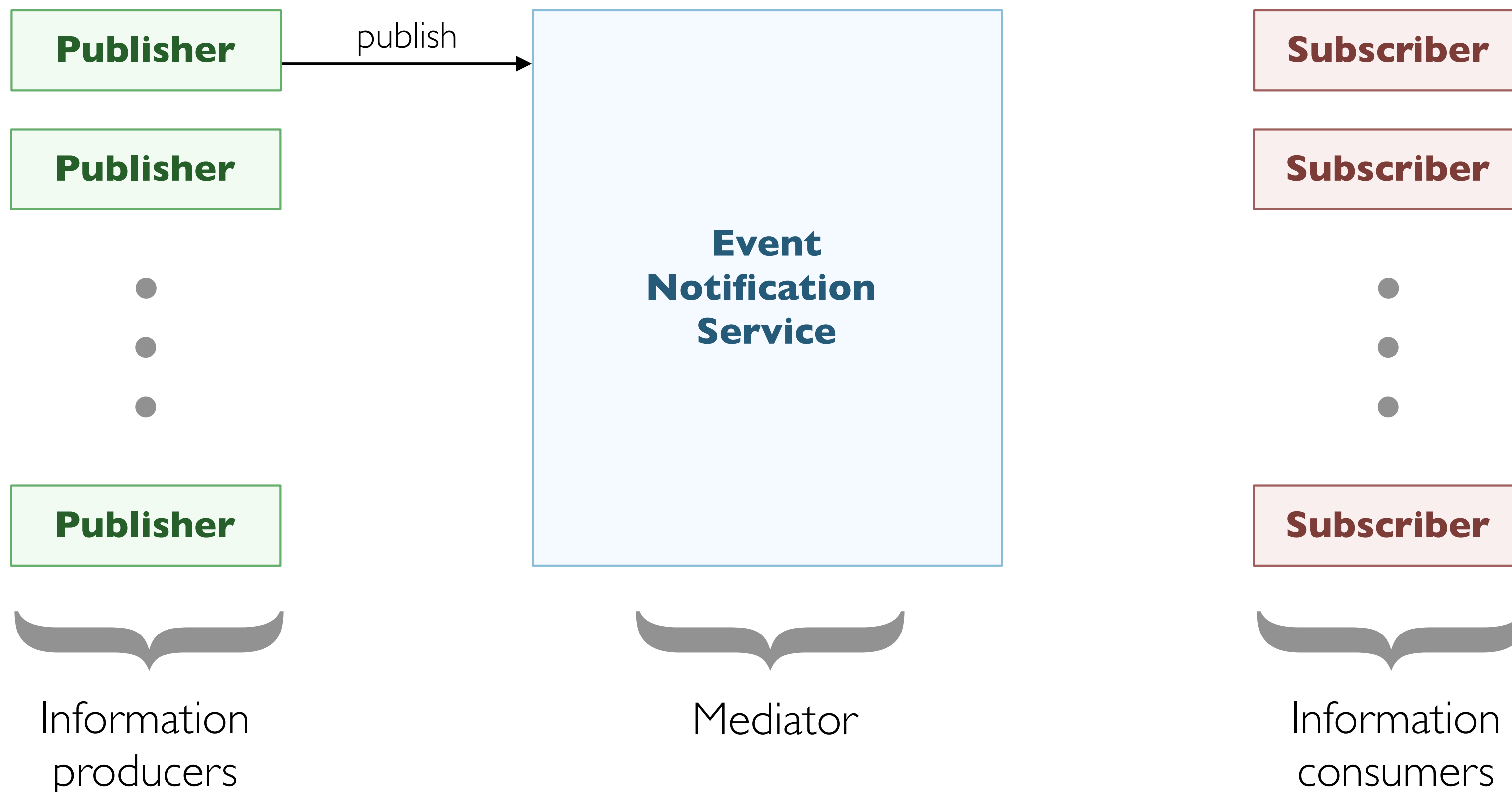
- The publish/subscribe communication paradigm:
  - **Publishers**: produce data in the form of **events**.
  - **Subscribers**: declare interests on published data with subscriptions.
  - Each **subscription** is a filter on the set of published events.
  - An **Event Notification Service** (ENS) notifies to each subscriber every published event that matches at least one of its subscriptions.



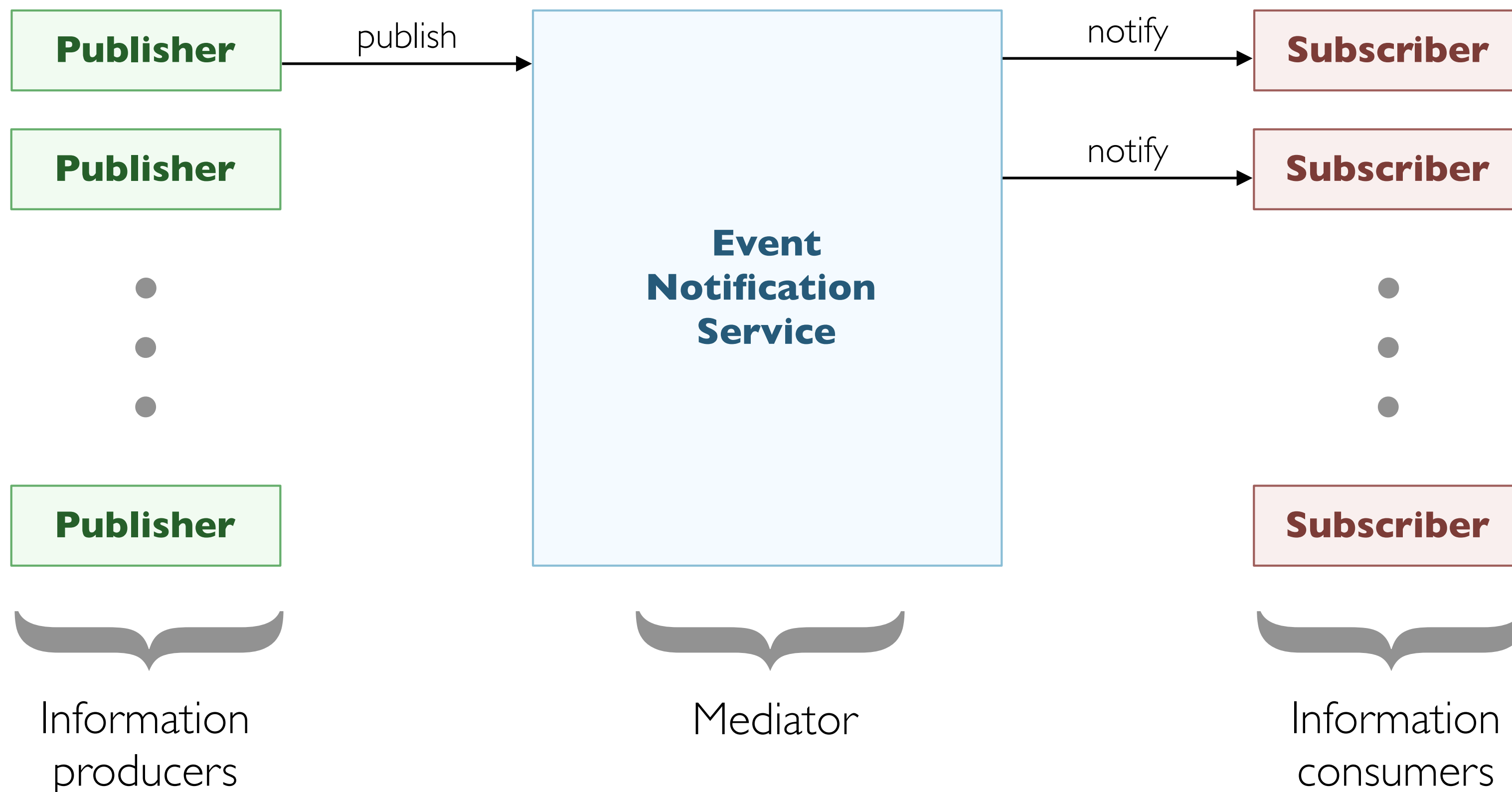
- The publish/subscribe communication paradigm:
  - **Publishers**: produce data in the form of **events**.
  - **Subscribers**: declare interests on published data with subscriptions.
  - Each **subscription** is a filter on the set of published events.
  - An **Event Notification Service** (ENS) notifies to each subscriber every published event that matches at least one of its subscriptions.



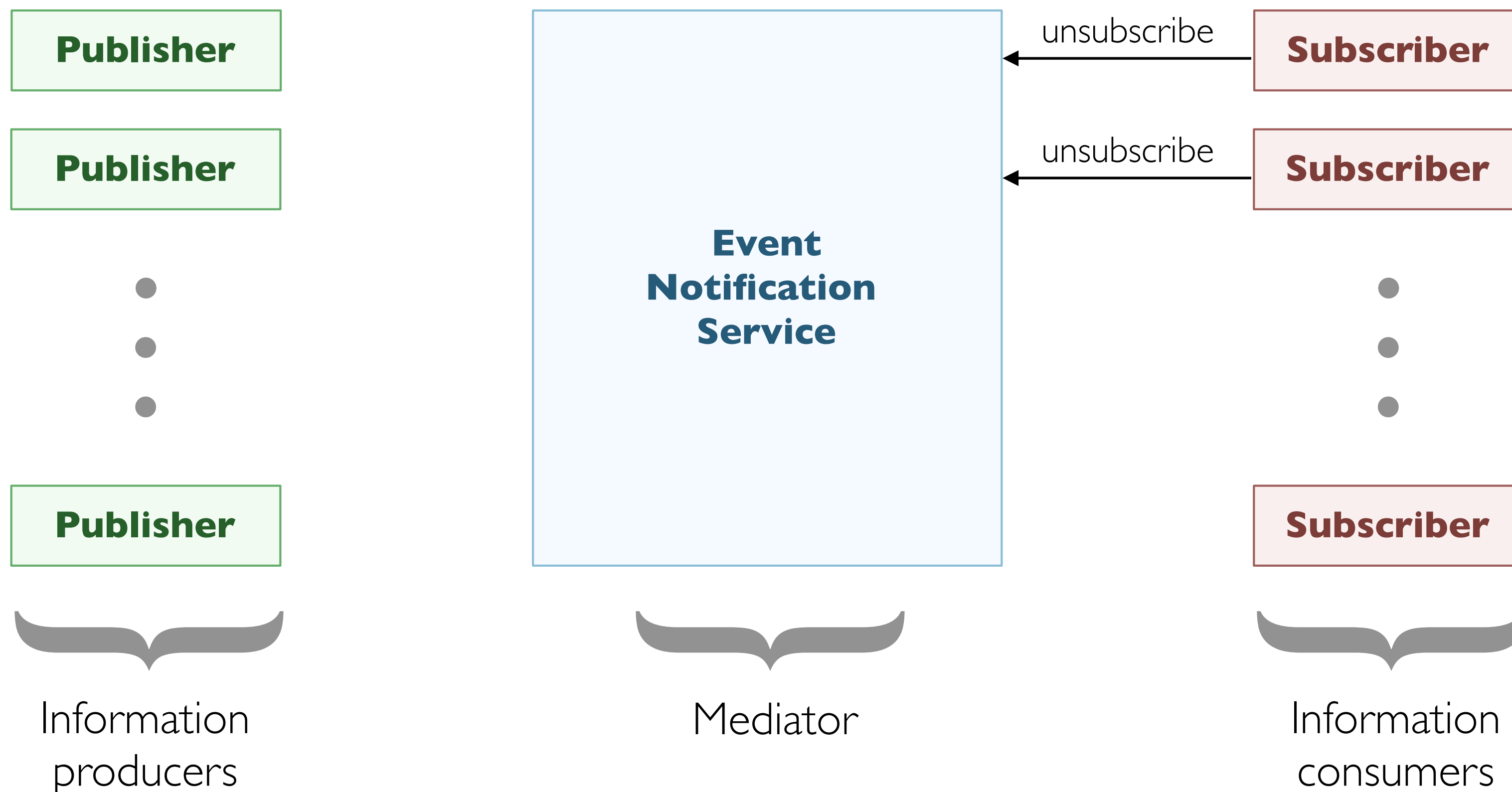
- The publish/subscribe communication paradigm:
  - **Publishers**: produce data in the form of **events**.
  - **Subscribers**: declare interests on published data with subscriptions.
  - Each **subscription** is a filter on the set of published events.
  - An **Event Notification Service** (ENS) notifies to each subscriber every published event that matches at least one of its subscriptions.



- The publish/subscribe communication paradigm:
  - **Publishers**: produce data in the form of **events**.
  - **Subscribers**: declare interests on published data with subscriptions.
  - Each **subscription** is a filter on the set of published events.
  - An **Event Notification Service** (ENS) notifies to each subscriber every published event that matches at least one of its subscriptions.



- The publish/subscribe communication paradigm:
  - **Publishers**: produce data in the form of **events**.
  - **Subscribers**: declare interests on published data with subscriptions.
  - Each **subscription** is a filter on the set of published events.
  - An **Event Notification Service** (ENS) notifies to each subscriber every published event that matches at least one of its subscriptions.



- Events represent information structured following an *event schema*.
- The event schema is fixed, defined a-priori, and known to all the participants.
- It defines a set of fields or attributes, each constituted by a name and a type. The types allowed depend on the specific implementation, but basic types (like integers, floats, booleans, strings) are usually available.
- Given an event schema, an event is a collection of values, one for each attribute defined in the schema.

<u>Example</u>: suppose we are dealing with an application whose purpose is to distribute updates about computer-related blogs.

name	type	allowed values
blog_name	string	ANY
address	URL	ANY
genre	enumeration	[hardware, software, peripherals, development]
author	string	ANY
abstract	string	ANY
rating	integer	[1-5]
update_date	date	>1-1-1970 00:00

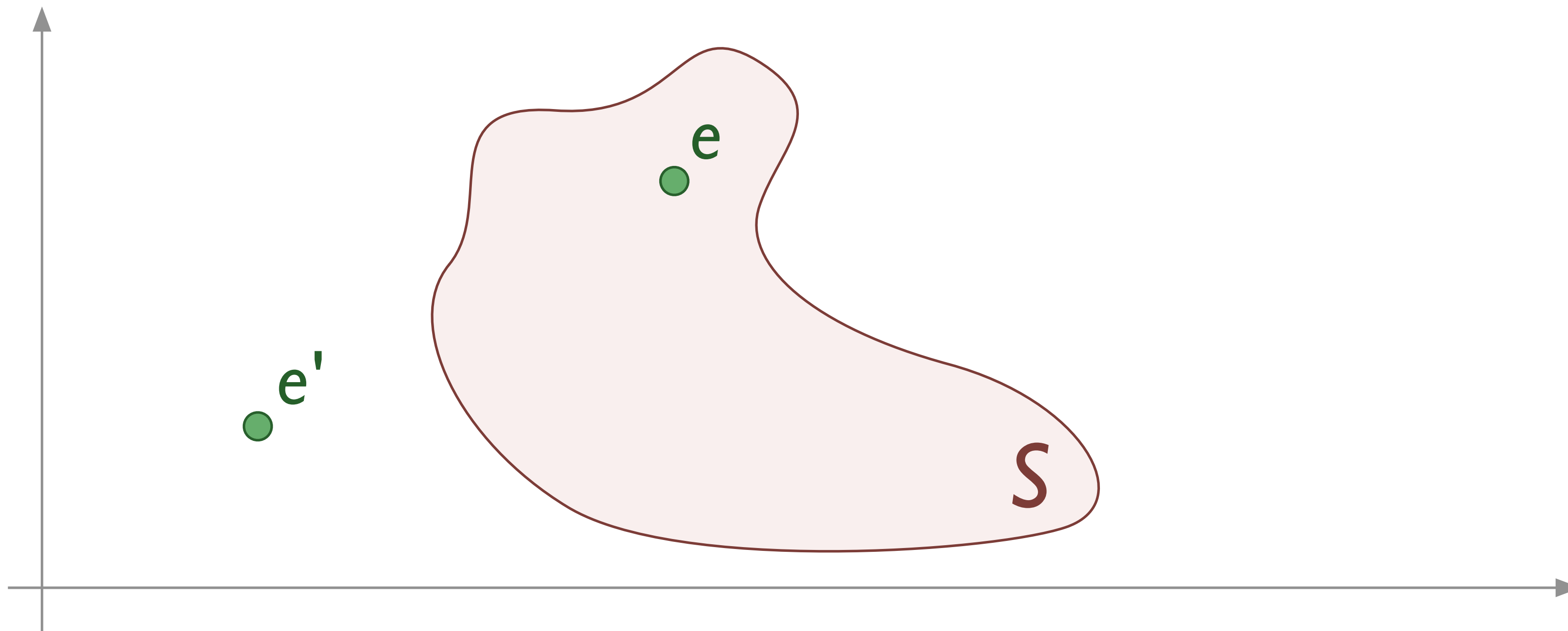
Event Schema

Event

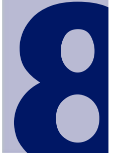
name	value
blog_name	Prad.de
address	http://www.prad.de/en/index.html
genre	peripherals
author	Mark Hansen
abstract	"The review of the new TFT panel..."
rating	4
update_date	26-4-2006 17:58

- Subscribers express their interests in specific events issuing subscriptions.
- A subscription is, generally speaking, a *constraint* expressed on the event schema.
- The Event Notification Service will notify an event e to a subscriber *x* only if the values that define the event satisfy the constraint defined by one of the subscriptions *s* issued by x. In this case we say that **e matches *s***.
- Subscriptions can take various forms, depending on the subscription language and model employed by each specific implementation.
- Example: a subscription can be a conjunction of constraints each expressed on a single attribute. Each constraint in this case can be as simple as a >=< operator applied on an integer attribute, or complex as a regular expression applied to a string.

- From an abstract point of view the event schema defines an n-dimensional **event space** (where n is the number of attributes).
- In this space each event e represents a point.
- Each subscription *s* identifies a subspace.
- An event e matches the subscription *s* if, and only if, the corresponding point is included in the portion of the event space delimited by *s*.

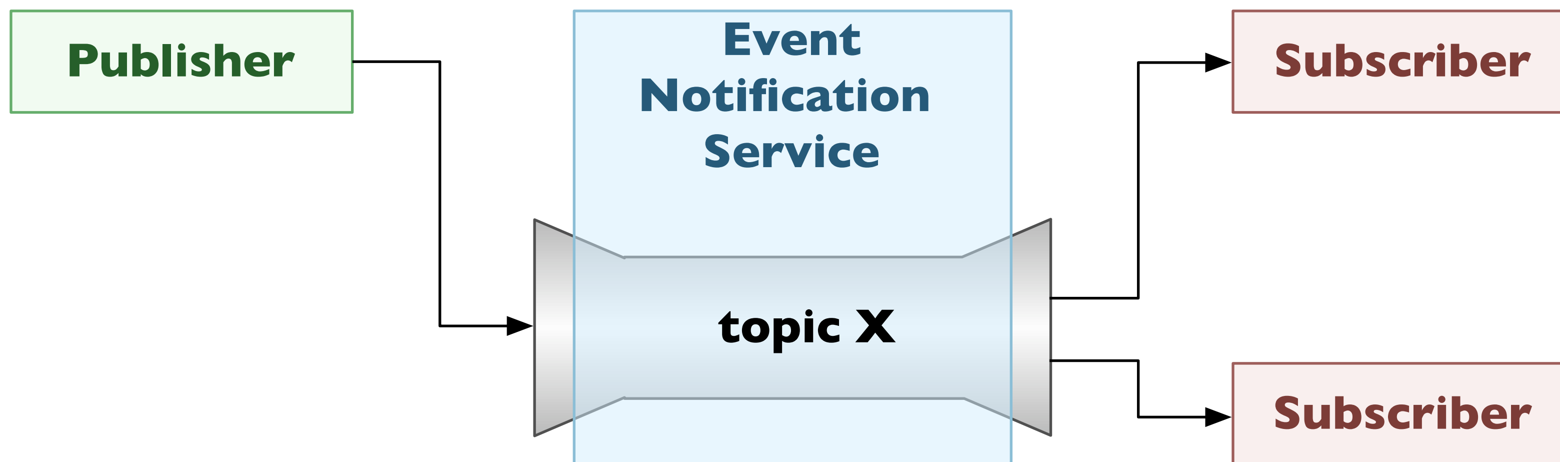


- Depending on the subscription model used we distinguish various flavors of publish/subscribe:
  - Topic-based
  - Hierarchy-based
  - Content-based
  - Type-based
  - Concept-based
  - XML-based
  - ???-based



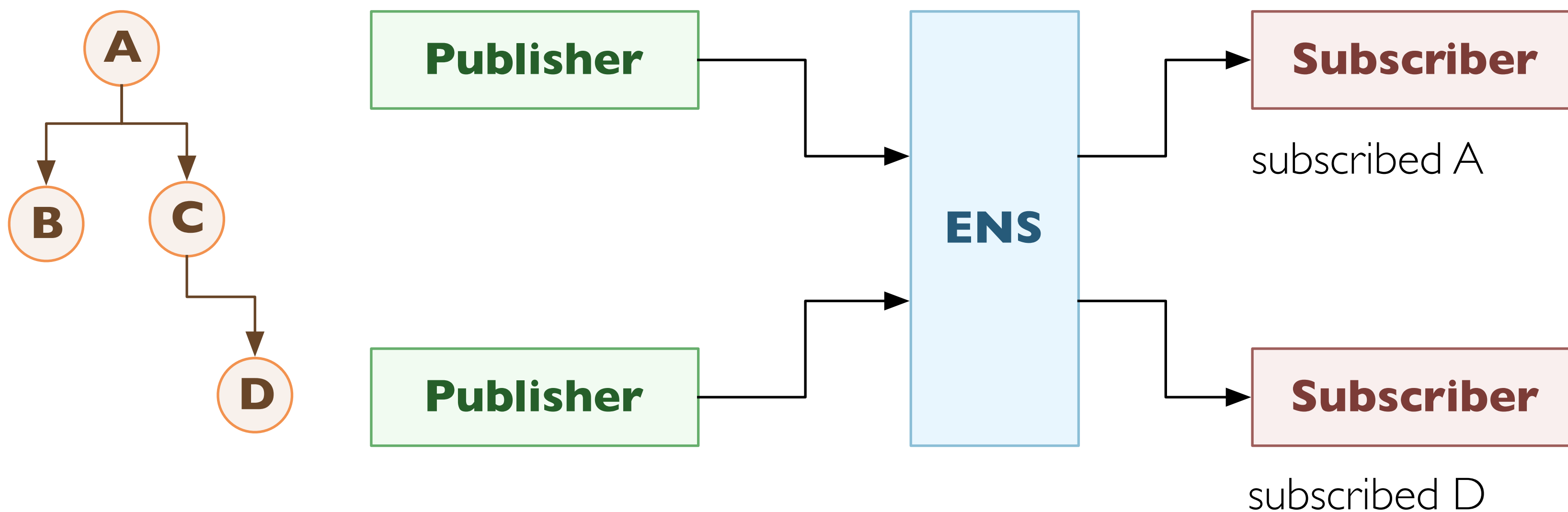
Topic-based selection: data published in the system is mostly unstructured, but each event is "tagged" with the identifier of a *topic* it is published in. Subscribers issue subscriptions containing the topics they are interested in.

A topic can be thus represented as a "virtual channel" connecting producers to consumers. For this reason the problem of data distribution in topic-based publish/subscribe systems is considered quite close to group communications.



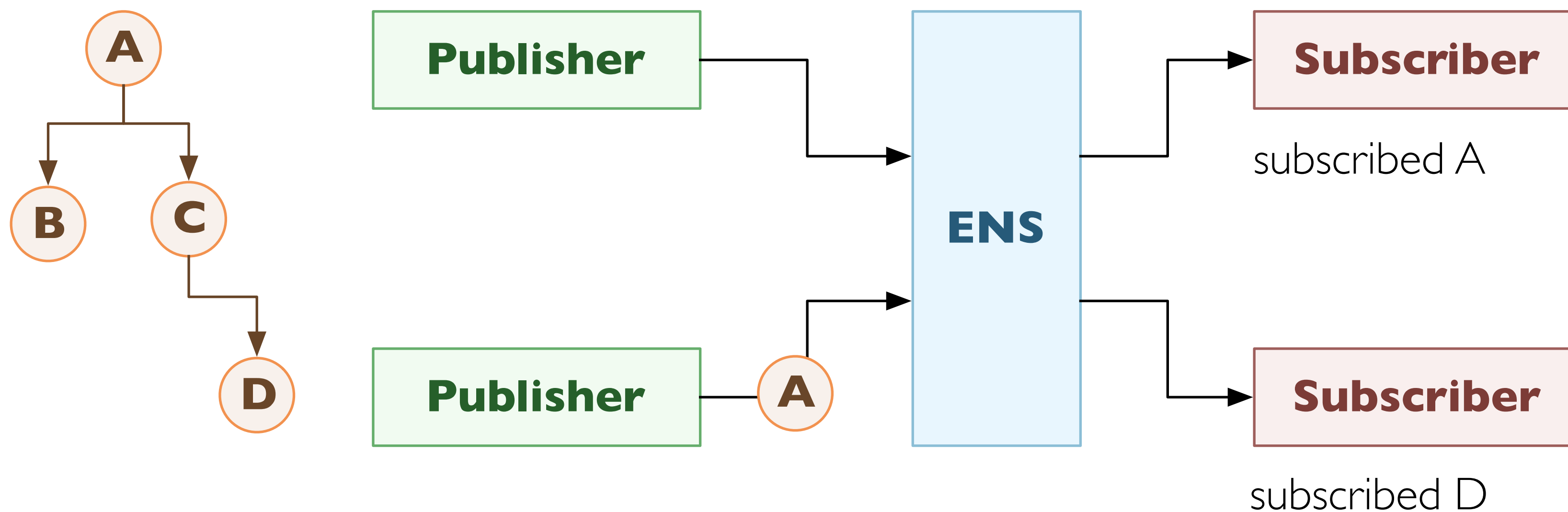
Hierarchy-based selection: even in this case each event is "tagged" with the *topic* it is published in, and Subscribers issue subscriptions containing the topics they are interested in.

Contrarily to the previous model, here topics are organized in a hierarchical structure which express a notion of containment between topics. When a subscriber subscribe a topic, it will receive all the events published in that topic and in all the topics present in the corresponding sub-tree.



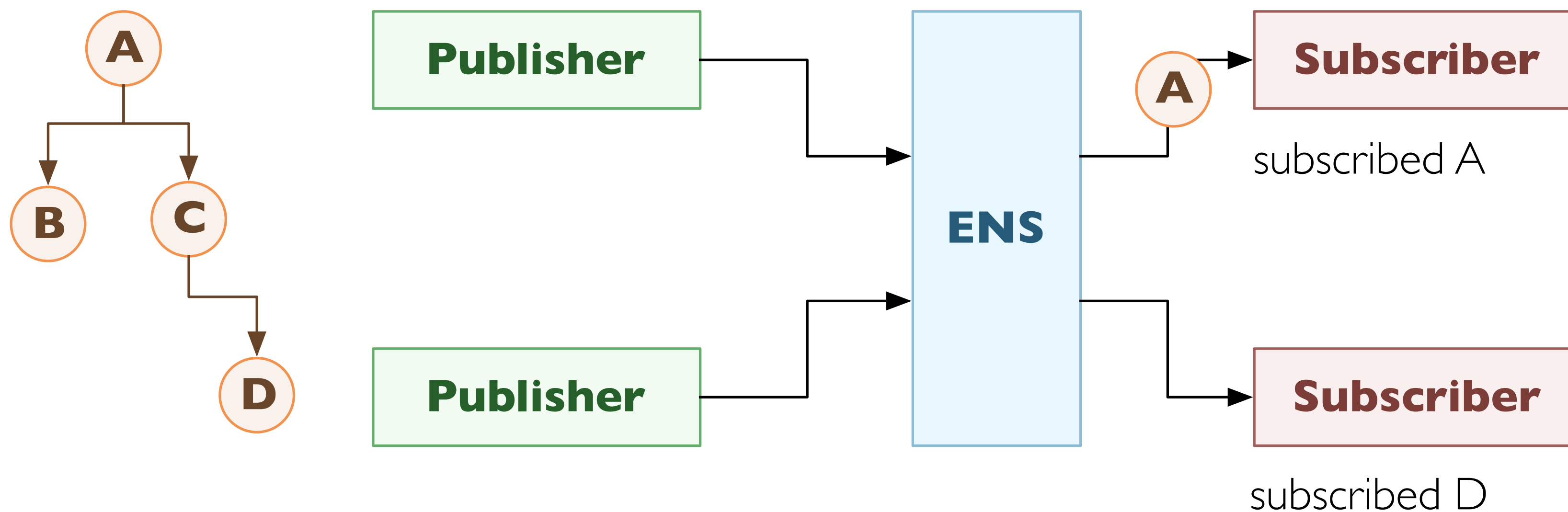
Hierarchy-based selection: even in this case each event is "tagged" with the *topic* it is published in, and Subscribers issue subscriptions containing the topics they are interested in.

Contrarily to the previous model, here topics are organized in a hierarchical structure which express a notion of containment between topics. When a subscriber subscribe a topic, it will receive all the events published in that topic and in all the topics present in the corresponding sub-tree.



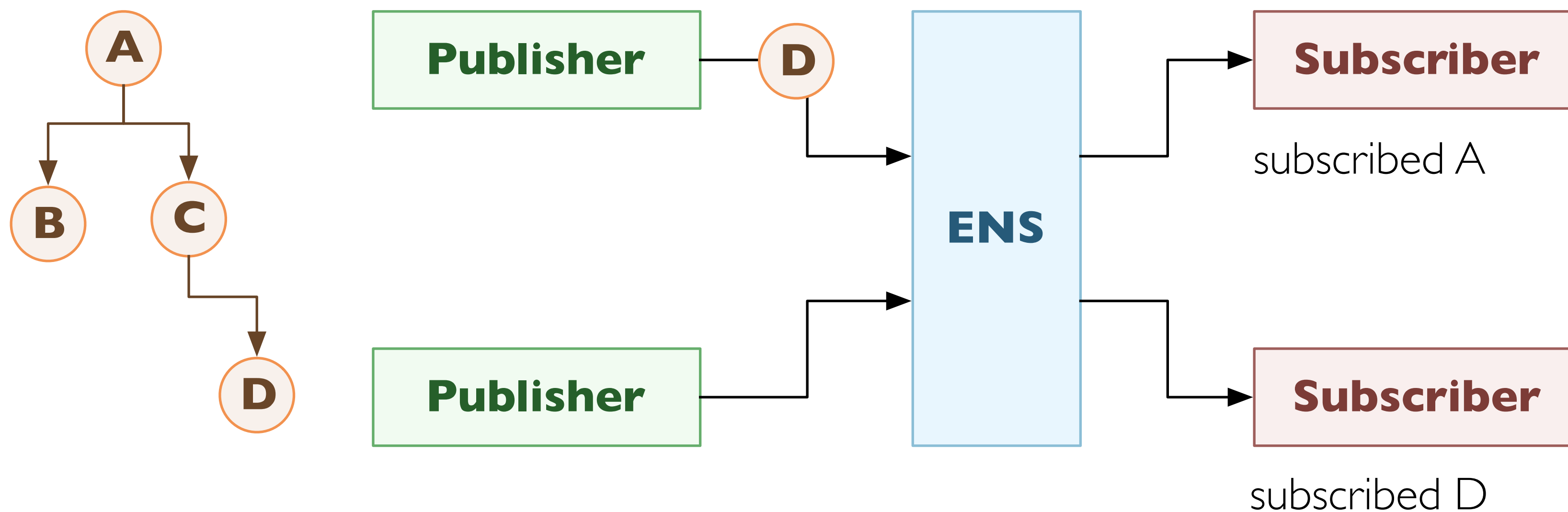
Hierarchy-based selection: even in this case each event is "tagged" with the *topic* it is published in, and Subscribers issue subscriptions containing the topics they are interested in.

Contrarily to the previous model, here topics are organized in a hierarchical structure which express a notion of containment between topics. When a subscriber subscribe a topic, it will receive all the events published in that topic and in all the topics present in the corresponding sub-tree.



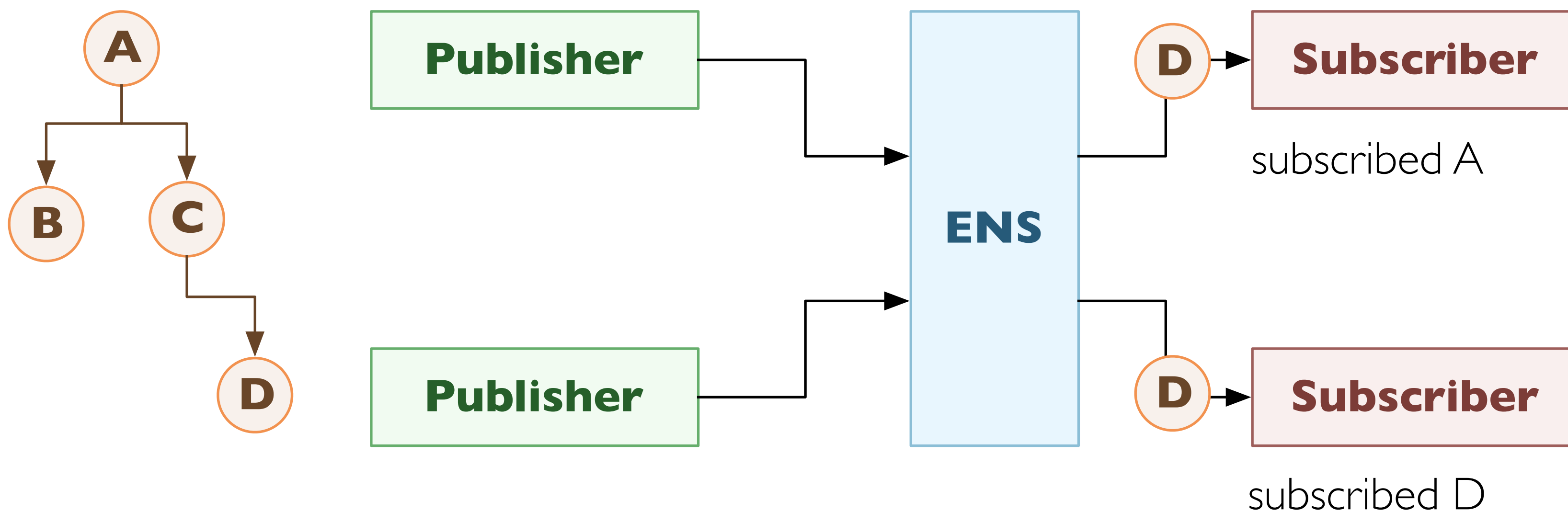
Hierarchy-based selection: even in this case each event is "tagged" with the *topic* it is published in, and Subscribers issue subscriptions containing the topics they are interested in.

Contrarily to the previous model, here topics are organized in a hierarchical structure which express a notion of containment between topics. When a subscriber subscribe a topic, it will receive all the events published in that topic and in all the topics present in the corresponding sub-tree.

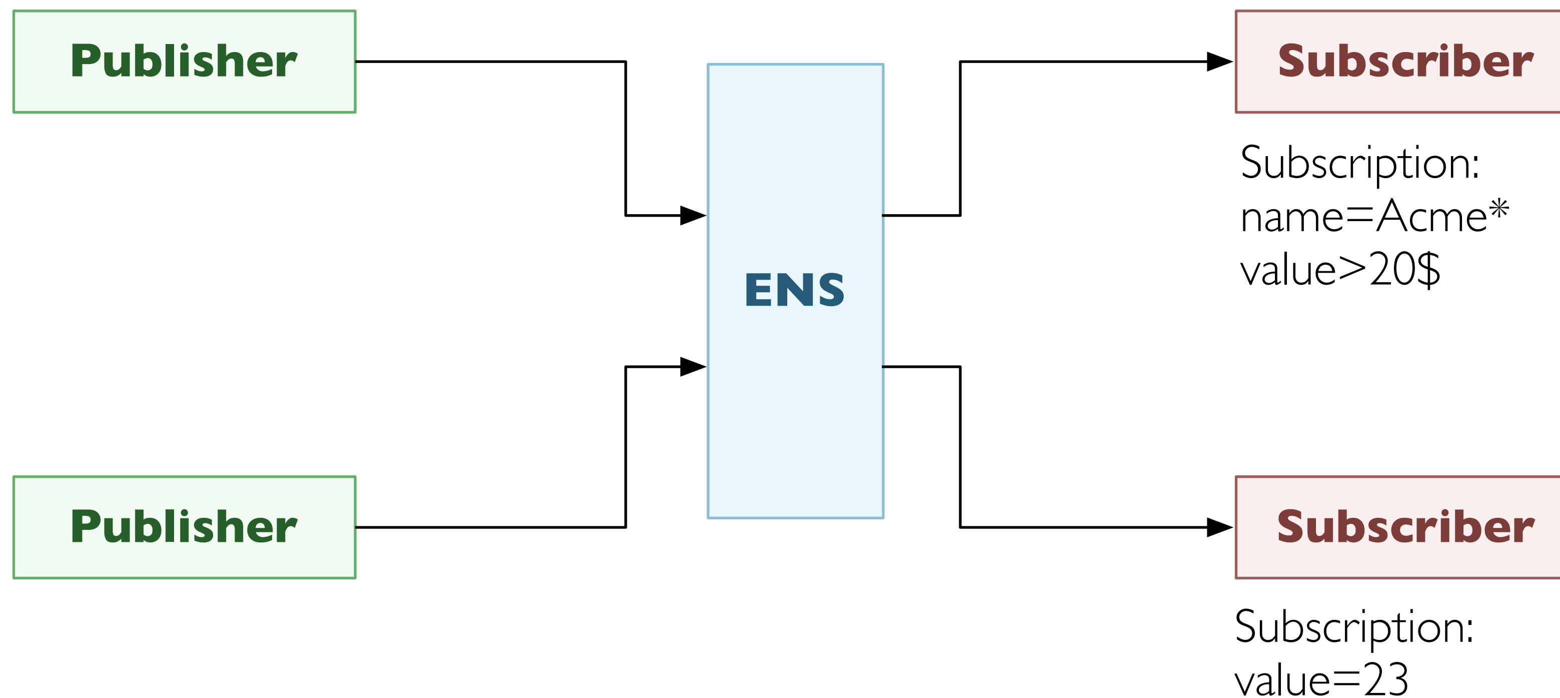


Hierarchy-based selection: even in this case each event is "tagged" with the *topic* it is published in, and Subscribers issue subscriptions containing the topics they are interested in.

Contrarily to the previous model, here topics are organized in a hierarchical structure which express a notion of containment between topics. When a subscriber subscribe a topic, it will receive all the events published in that topic and in all the topics present in the corresponding sub-tree.

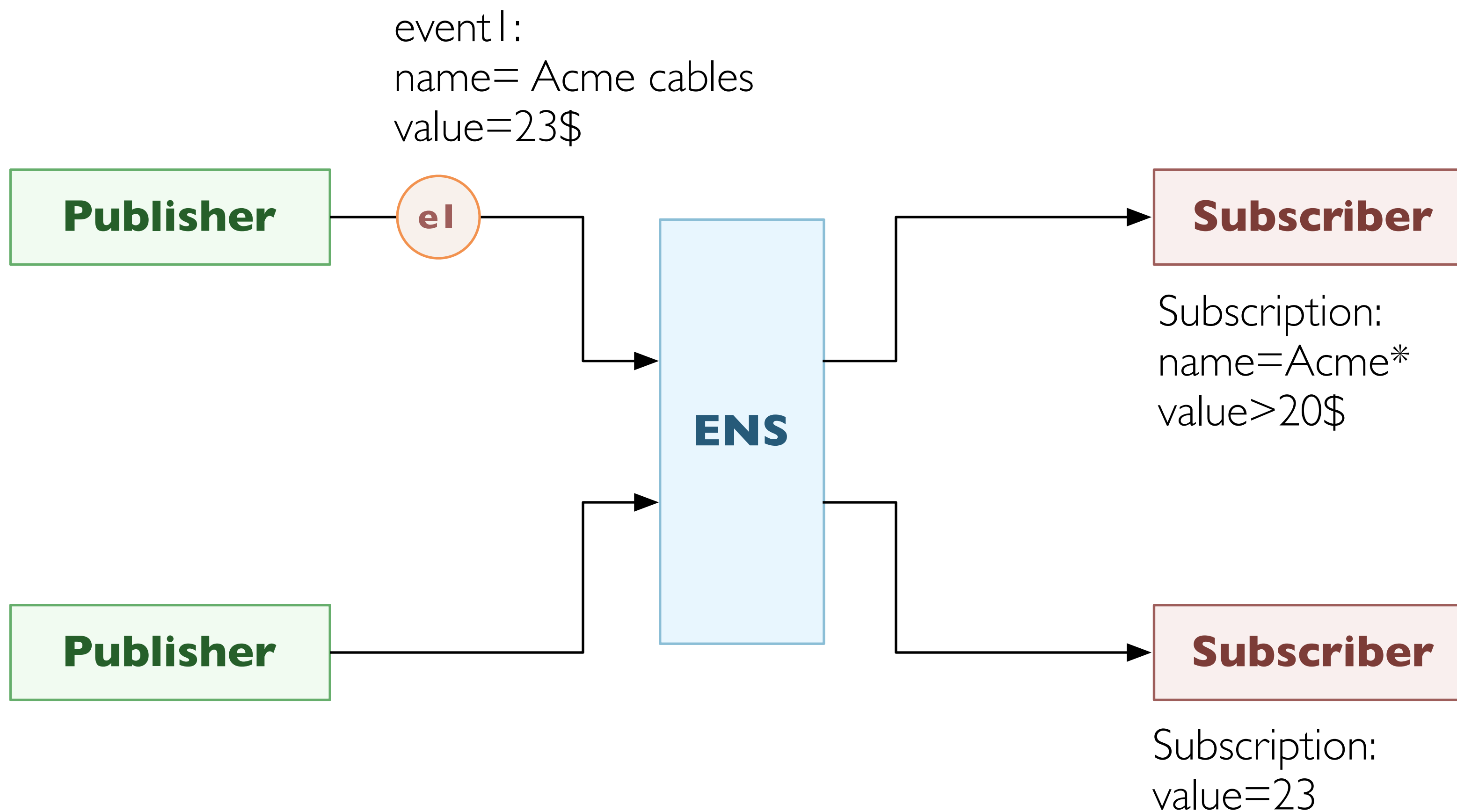


Content-based selection: all the data published in the system is mostly structured. Each subscription can be expressed as a conjunction of constrains expressed on attributes. The Event Notification Service filters out useless events before notifying a subscriber.



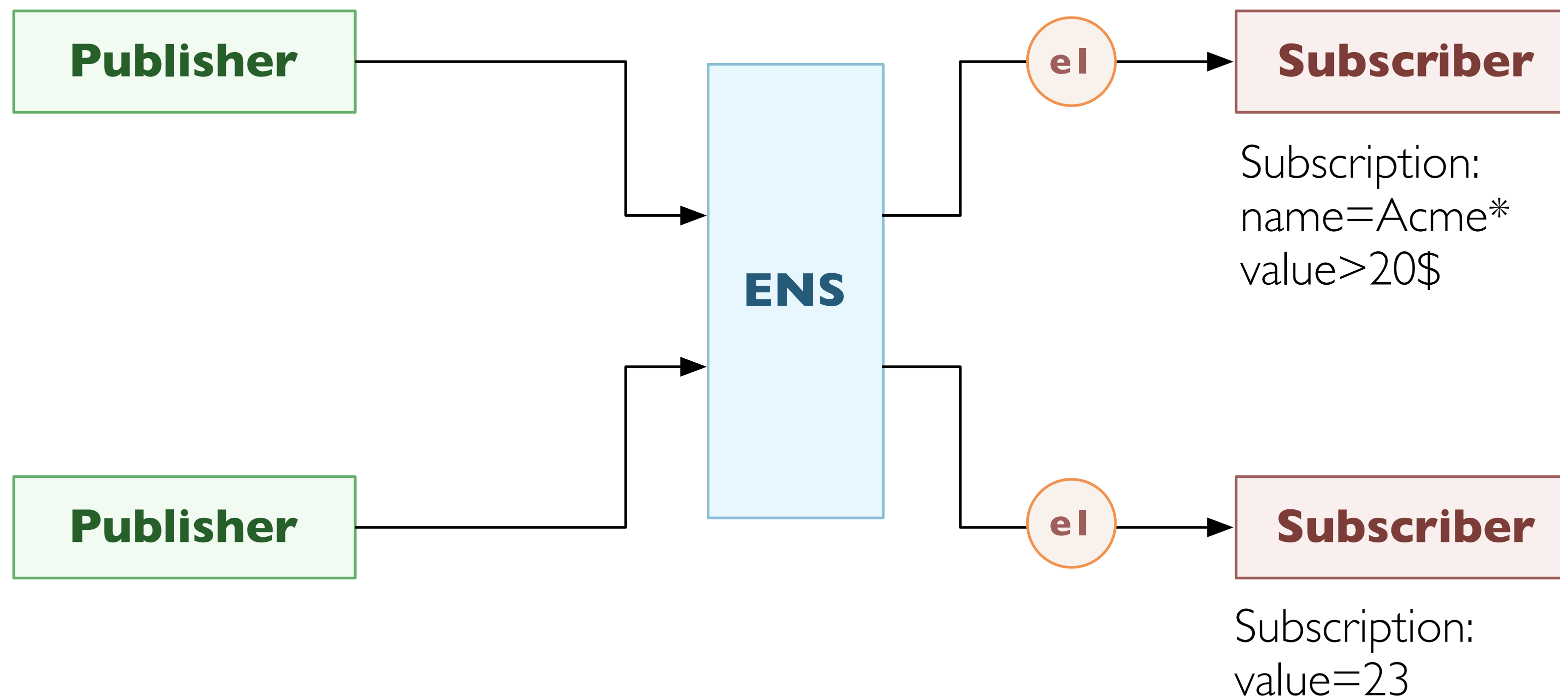
Event schema and subscription models

Content-based selection: all the data published in the system is mostly structured. Each subscription can be expressed as a conjunction of constrains expressed on attributes. The Event Notification Service filters out useless events before notifying a subscriber.

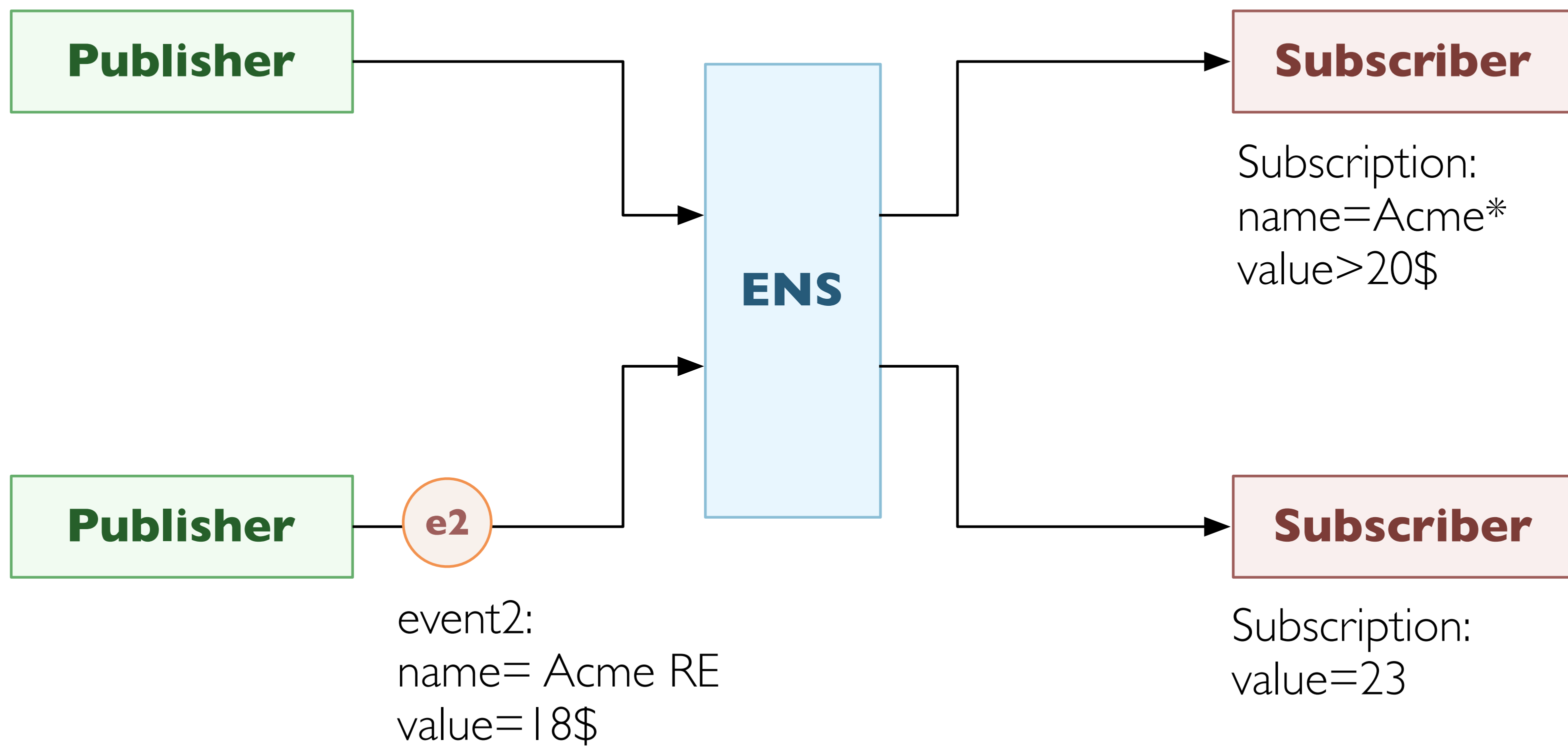


Event schema and subscription models

Content-based selection: all the data published in the system is mostly structured. Each subscription can be expressed as a conjunction of constrains expressed on attributes. The Event Notification Service filters out useless events before notifying a subscriber.

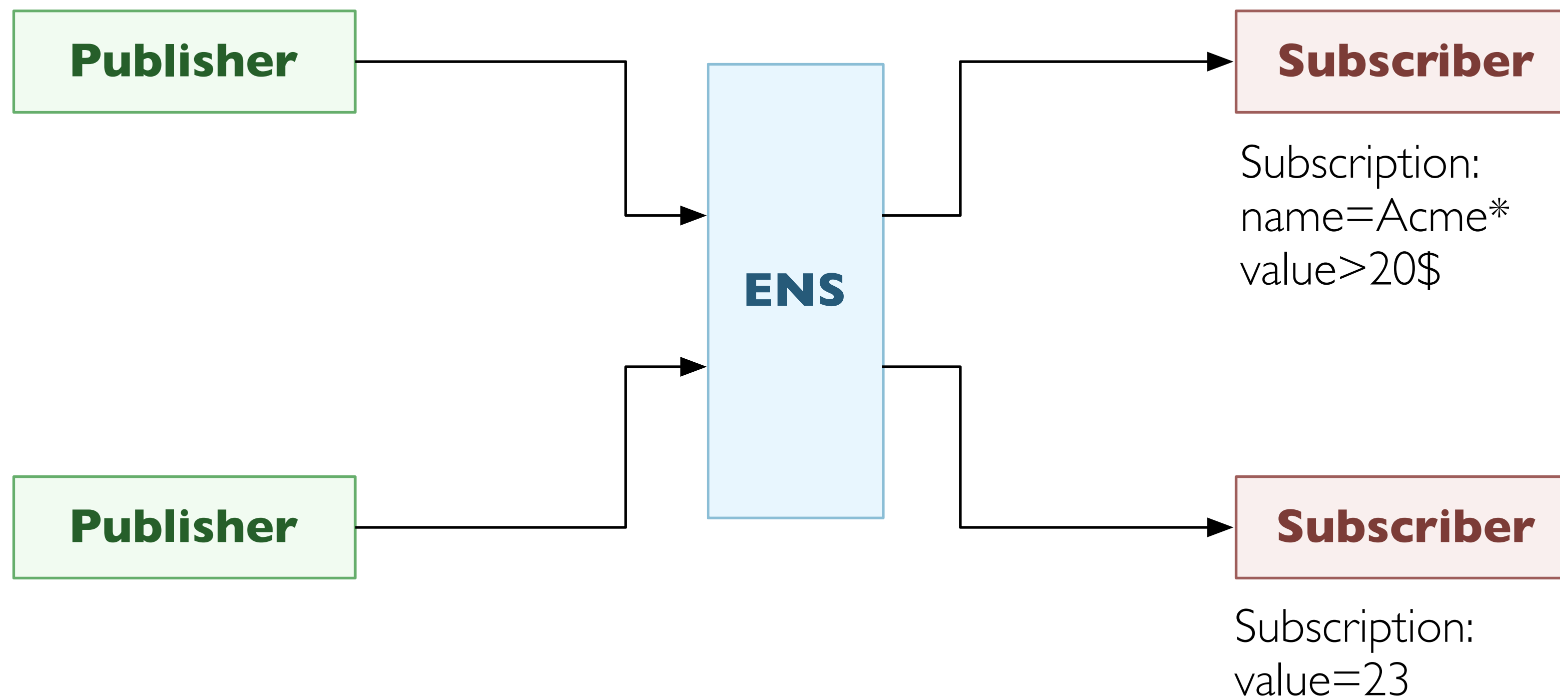


Content-based selection: all the data published in the system is mostly structured. Each subscription can be expressed as a conjunction of constrains expressed on attributes. The Event Notification Service filters out useless events before notifying a subscriber.

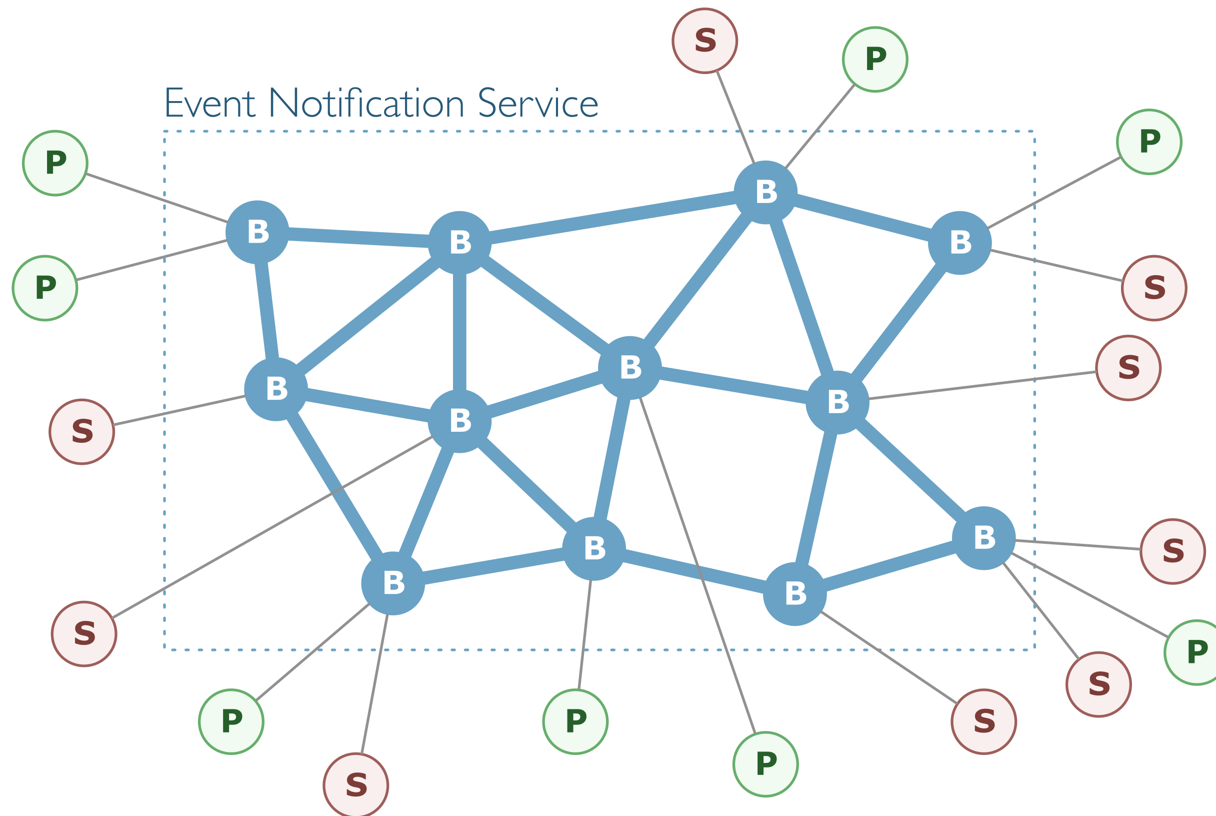


Event schema and subscription models

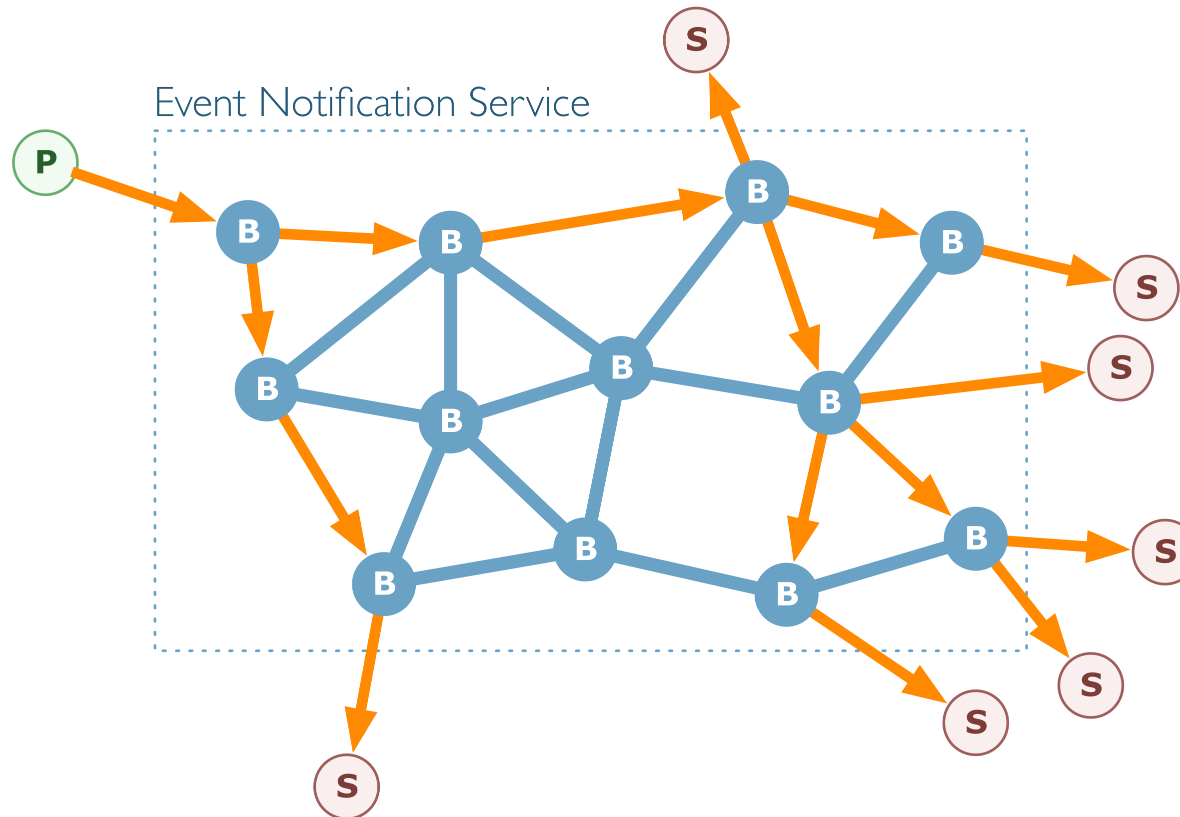
Content-based selection: all the data published in the system is mostly structured. Each subscription can be expressed as a conjunction of constrains expressed on attributes. The Event Notification Service filters out useless events before notifying a subscriber.



- The Event Notification Service is usually implemented as a:
  - **Centralized service**: the ENS is implemented on a single server.
  - **Distributed service**: the ENS is constituted by a set of nodes, event brokers, which cooperate to implement the service.
- The latter is usually preferred for large settings where scalability is a fundamental issue.



- Modern ENSs are implemented through a set of processes, called *event brokers*, forming an overlay network.
- Each client (publisher or subscriber) accesses the service through a broker that masks the system complexity.

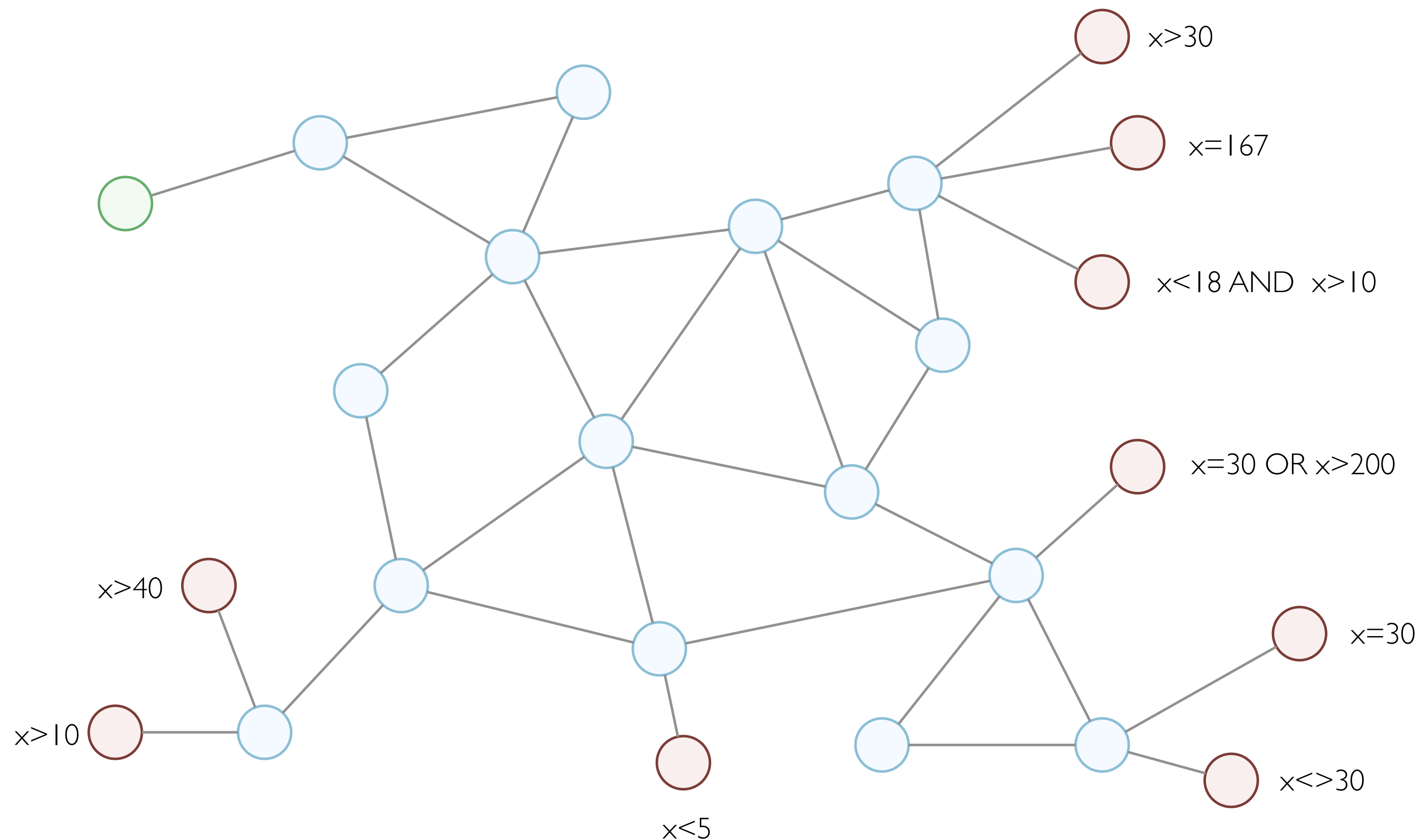


- An **event routing** mechanism routes each event inside the ENS from the broker where it is published to the broker(s) where it must be notified.

Event flooding: each event is broadcast from the publisher in the whole system.

The implementation is straightforward but very expensive.

This solution has the highest message overhead with no memory overhead.



14

Event routing

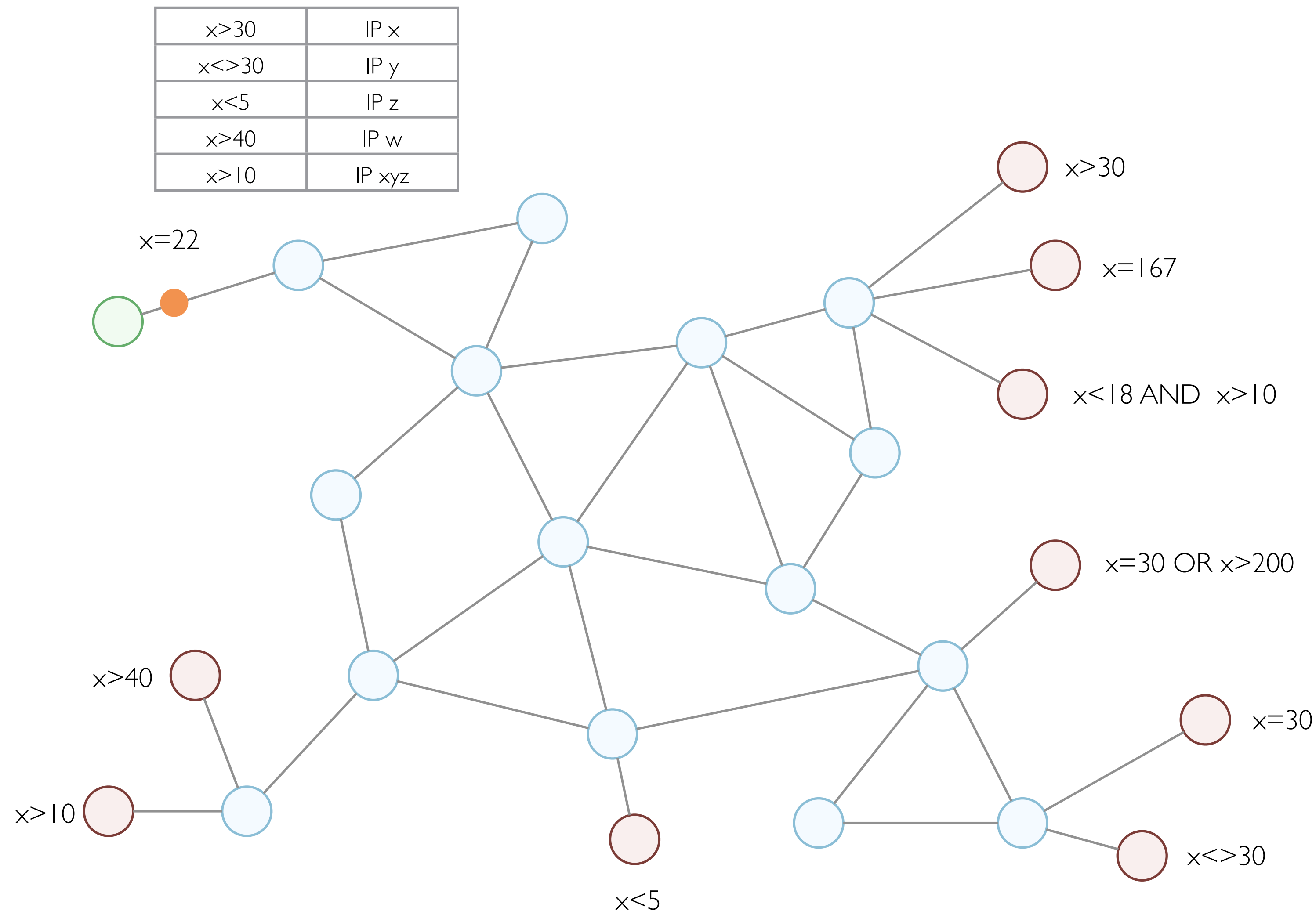
Event flooding: each event is broadcast from the publisher in the whole system.

The implementation is straightforward but very expensive.

This solution has the highest message overhead with no memory overhead.

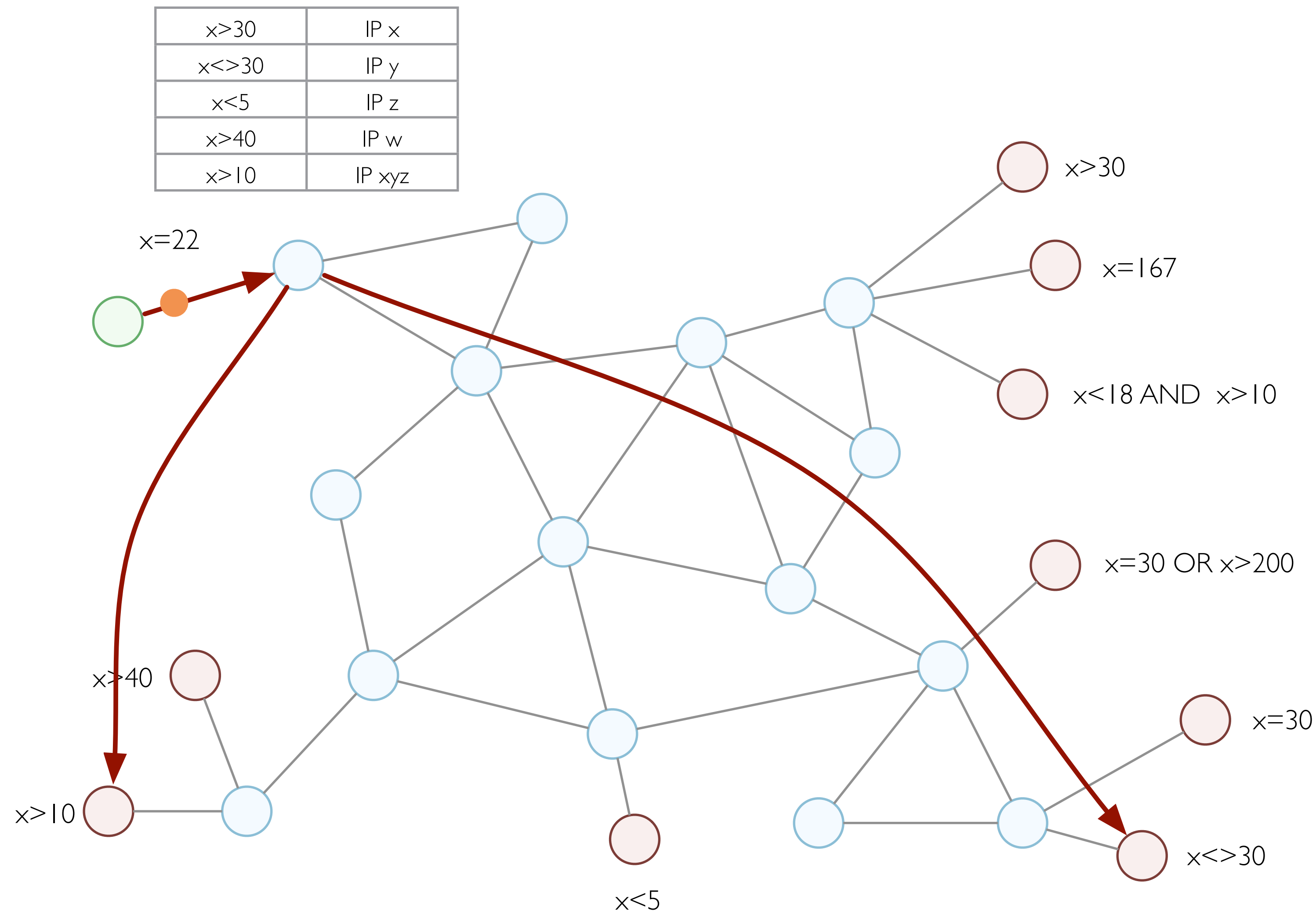
x=22
x>30
x=167
x<18 AND x>10
x=30 OR x>200
x>40
x>10
x<5
x=30
x<>30

Subscription flooding: each subscription is copied on every broker, in order to build locally complete subscription tables. These tables are then used to locally match events and directly notify interested subscribers. This approach suffers from a large memory overhead, but event diffusion is optimal. It is impractical in applications where subscriptions change frequently.



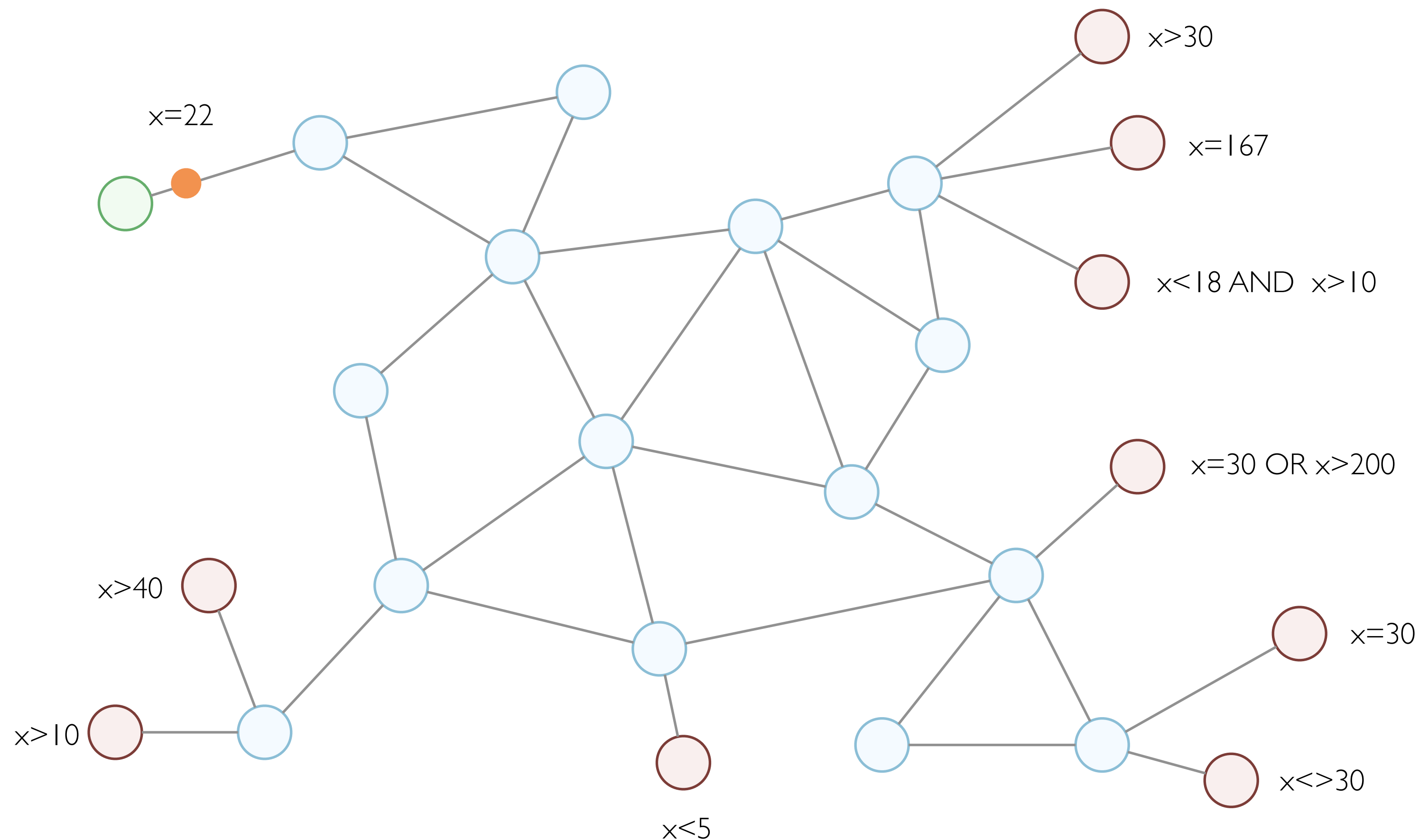
x>30	IP x
x<>30	IP y
x<5	IP z
x>40	IP w
x>10	IP xyz

Subscription flooding: each subscription is copied on every broker, in order to build locally complete subscription tables. These tables are then used to locally match events and directly notify interested subscribers. This approach suffers from a large memory overhead, but event diffusion is optimal. It is impractical in applications where subscriptions change frequently.

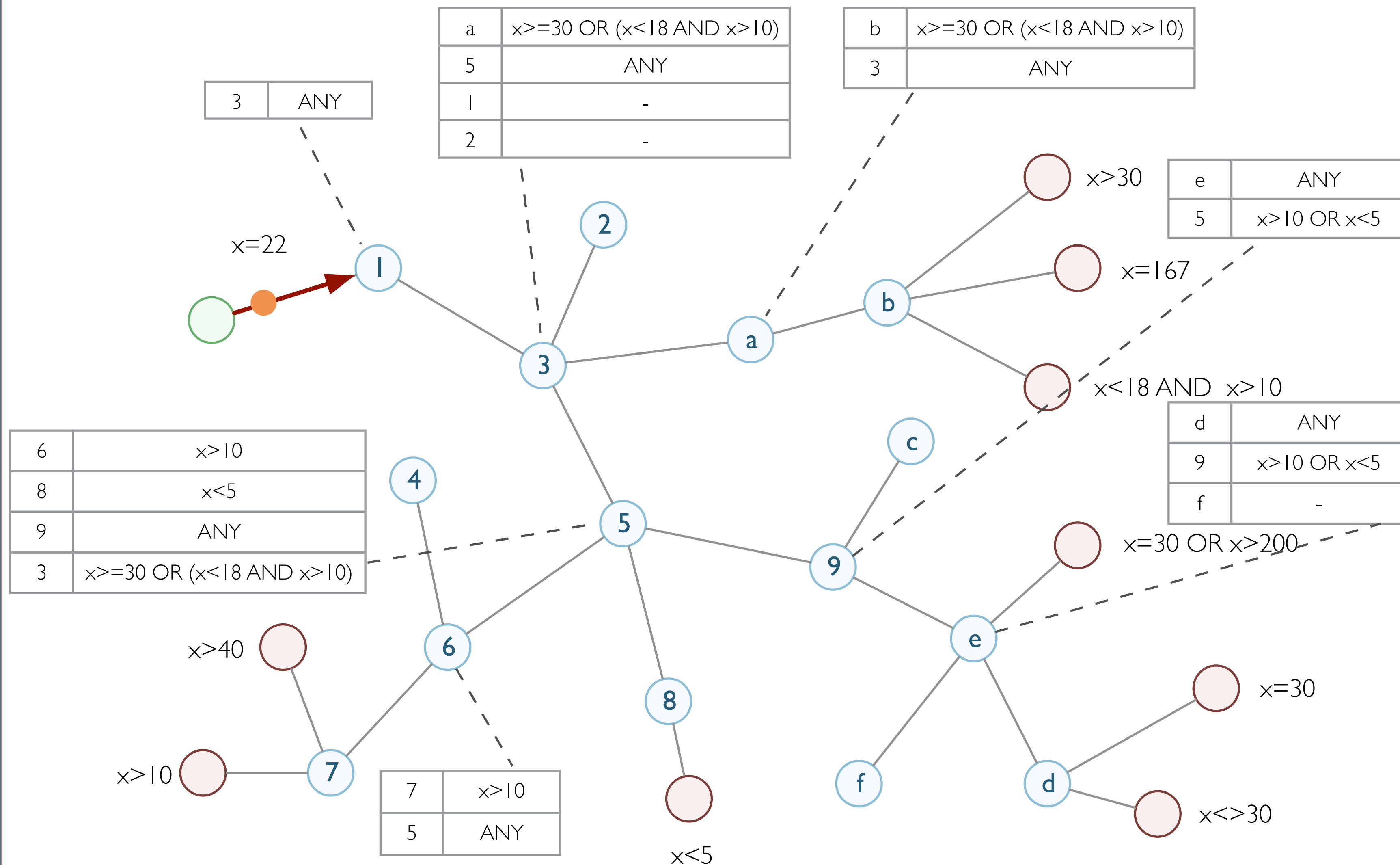


x>30	IP x
x<>30	IP y
x<5	IP z
x>40	IP w
x>10	IP xyz

Filter-based routing: subscriptions are partially diffused in the system and used to build *routing tables*. These tables, are then exploited during event diffusion to dynamically build a multicast tree that (hopefully) connects the publisher to all, and only, the interested subscribers.



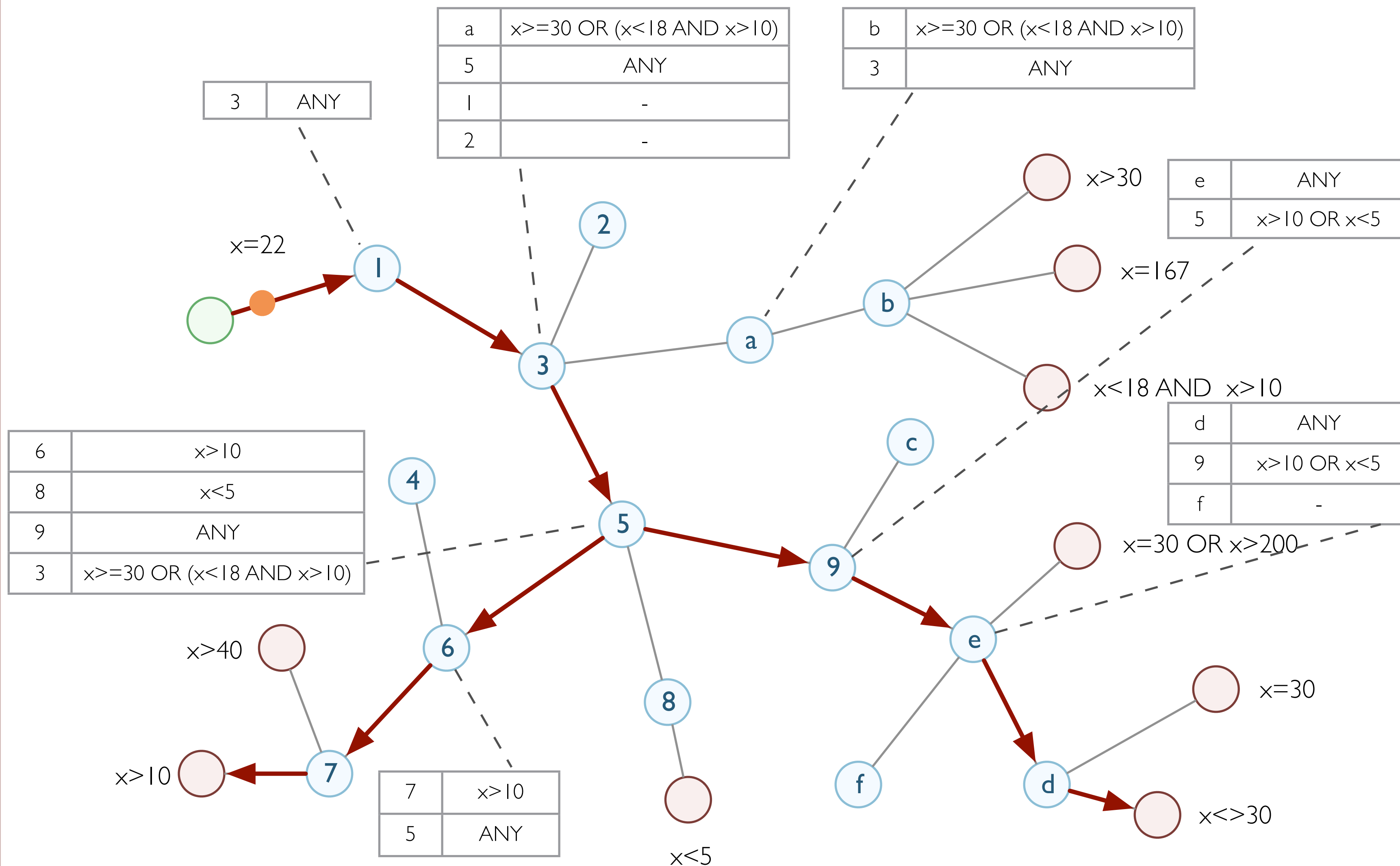
Filter-based routing: subscriptions are partially diffused in the system and used to build *routing tables*. These tables, are then exploited during event diffusion to dynamically build a multicast tree that (hopefully) connects the publisher to all, and only, the interested subscribers.



16

Event routing

Filter-based routing: subscriptions are partially diffused in the system and used to build *routing tables*. These tables, are then exploited during event diffusion to dynamically build a multicast tree that (hopefully) connects the publisher to all, and only, the interested subscribers.



16

Event routing

Rendez-Vous routing: it is based on two functions, namely *SN* and *EN*, used to associate respectively subscriptions and events to brokers in the system.

Given a subscription *s*, *SN(s)* returns a set of nodes which are responsible for storing *s* and forwarding received events matching *s* to all those subscribers that subscribed it.

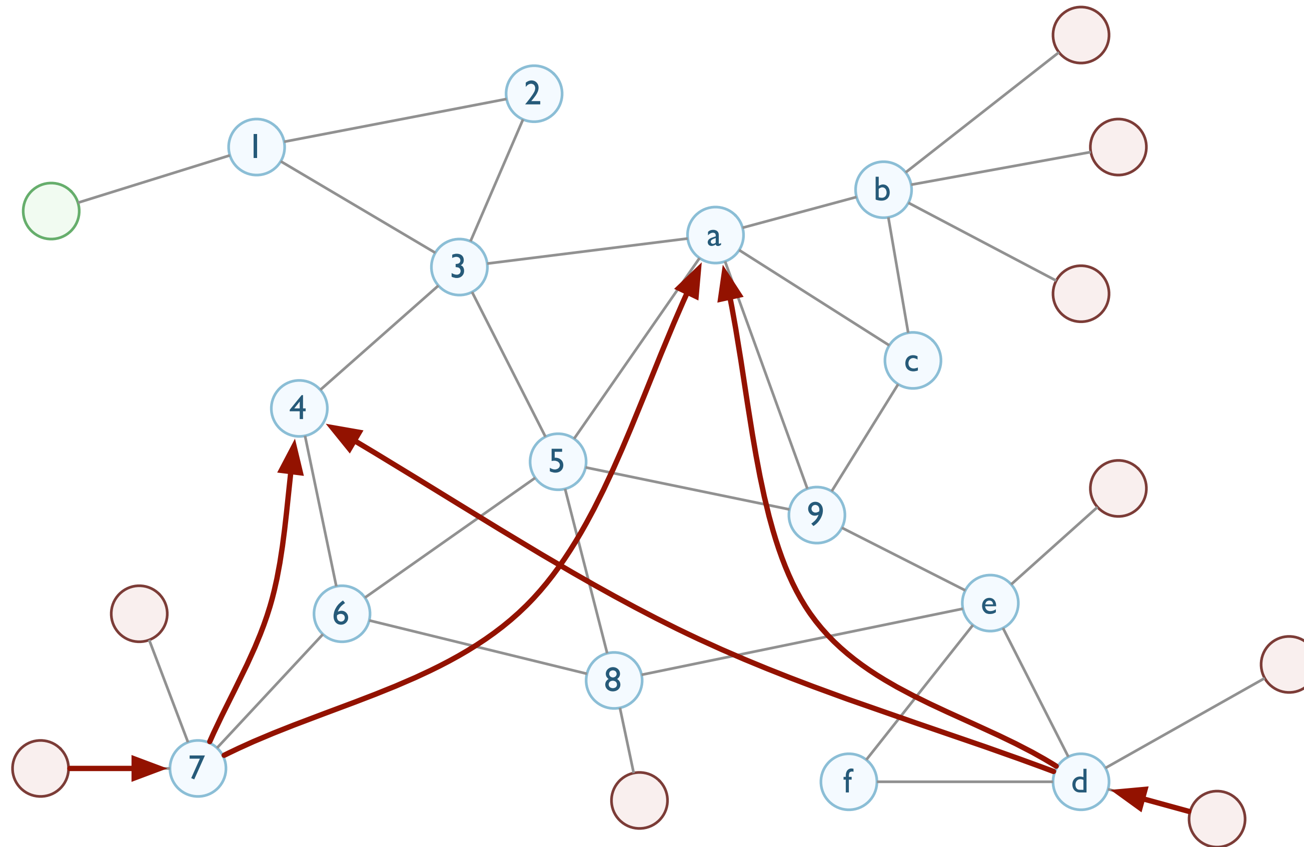
Given an event e, *EN(e)* returns a set of nodes which must receive e to match it against the subscriptions they store.

Event routing is a two-phases process: first an event e is sent to all brokers returned by *EN*(e), then those brokers match it against the subscriptions they store and notify the corresponding subscribers.

This approach works only if for each subscription *s* and event e, such that e matches *s*, the intersection between *EN*(e) and *SN(s)* is not empty (*mapping intersection rule*).

Rendez-Vous routing: example.

Phase 1: two nodes issue the same subscription S.



SN(S) = {4,a}

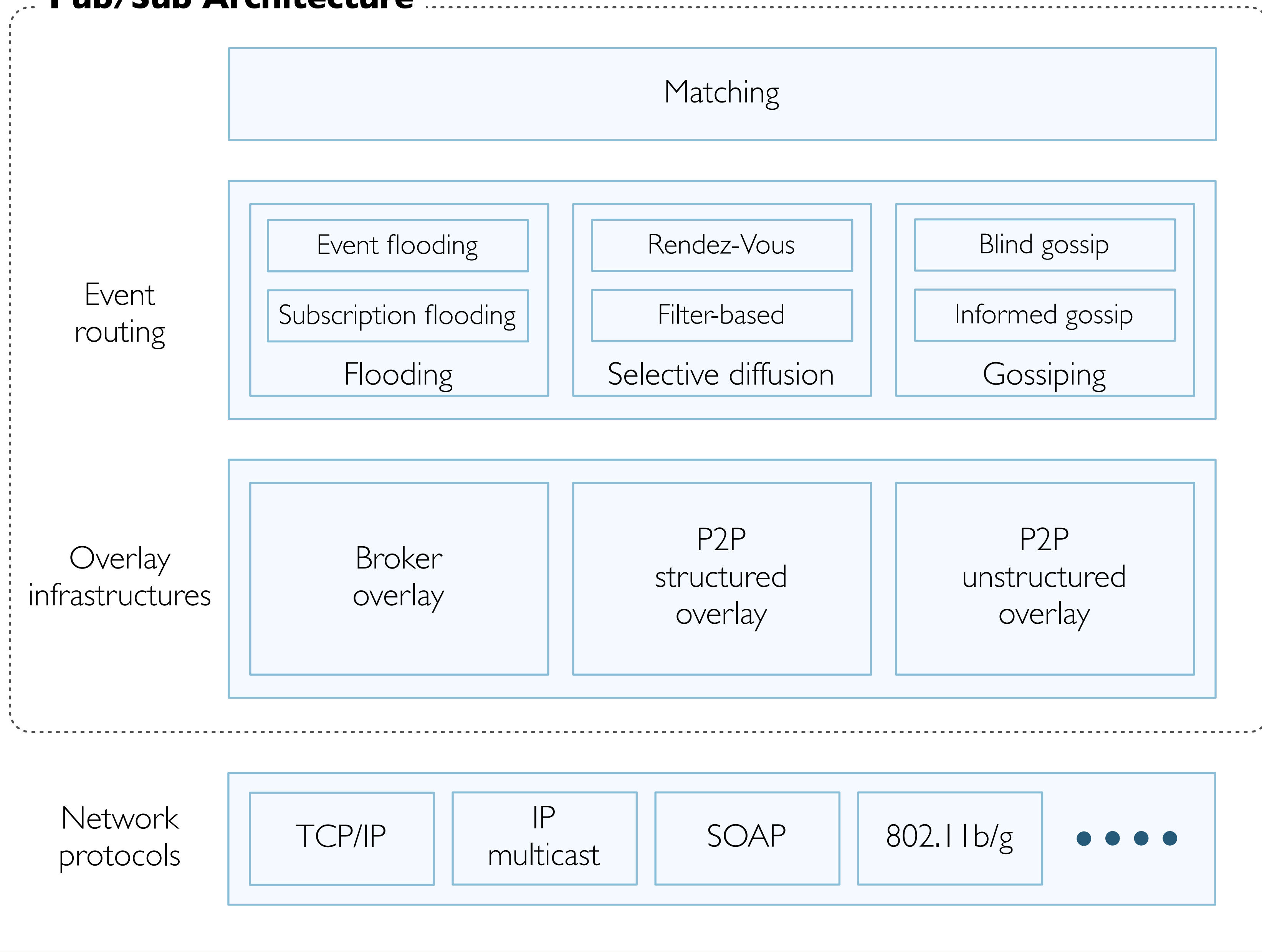
Rendez-Vous routing: example.

Phase II: an event e matching S is routed toward the rendez-vous node where it is matched against S.

EN(e) = {5,6,a}

Broker **a** is the rendez-vous point between event e and subscription S.

A generic architecture of a publish/subscribe system:



Antonio Carzaniga, Matthew J. Rutherford, Alexander J. Wolf

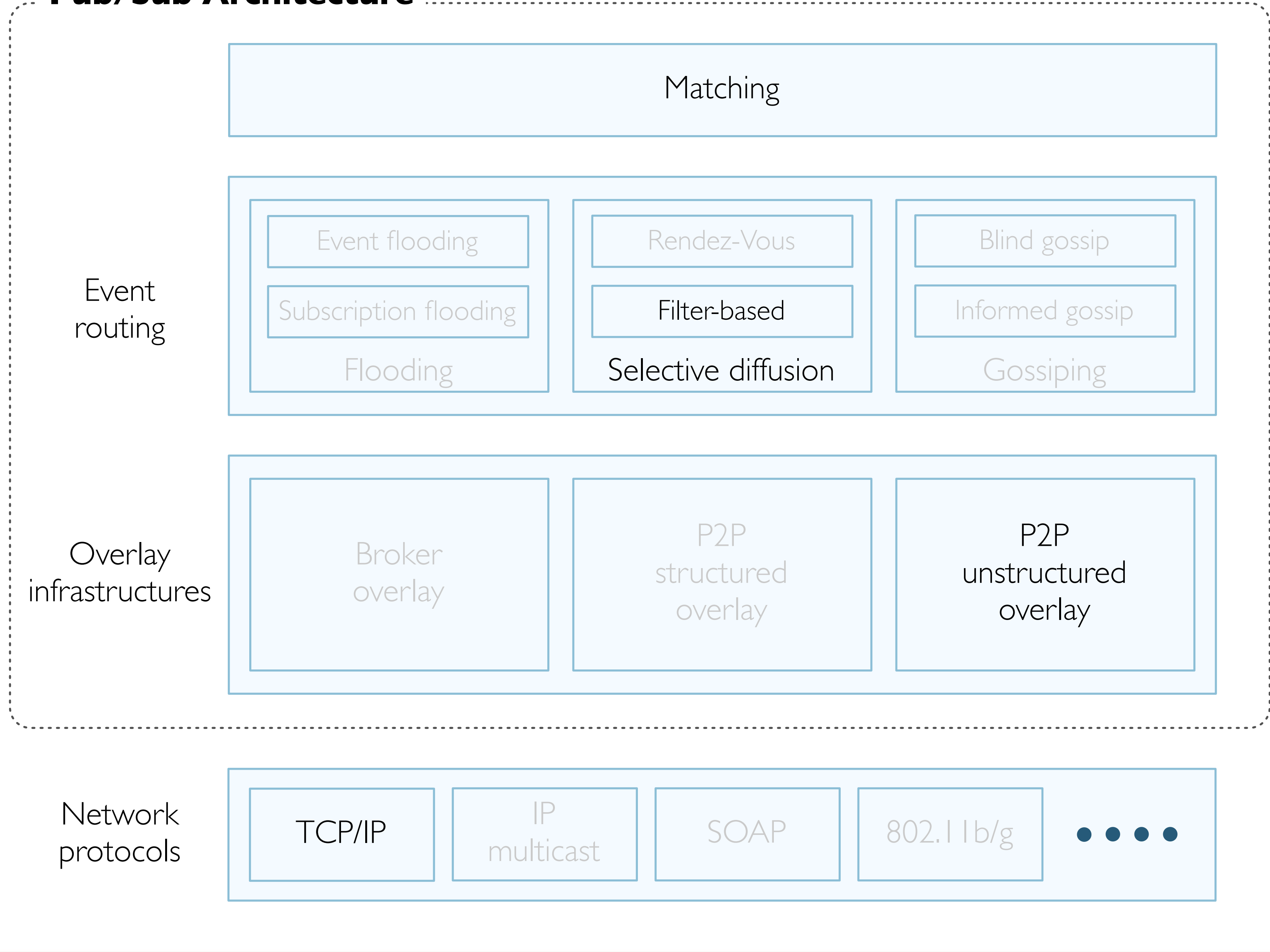
"A Routing Scheme for Content-Based Networking" (SIENA)

in Proceedings of IEEE INFOCOM 2004.

Abstract:

"This paper proposes a routing scheme for content-based networking. A content-based network is a communication network that features a new advanced communication model where messages are not given explicit destination addresses, and where the destinations of a message are determined by matching the content of the message against selection predicates declared by nodes. Routing in a content-based network amounts to propagating predicates and the necessary topological information in order to maintain loop-free and possibly minimal forwarding paths for messages. The routing scheme we propose uses a combination of a traditional broadcast protocol and a content-based routing protocol. We present the combined scheme and its requirements over the broadcast protocol. We then detail the content-based routing protocol, highlighting a set of optimization heuristics. We also present the results of our evaluation, showing that this routing scheme is effective and scalable."

The specific architecture of this system:



SIENA

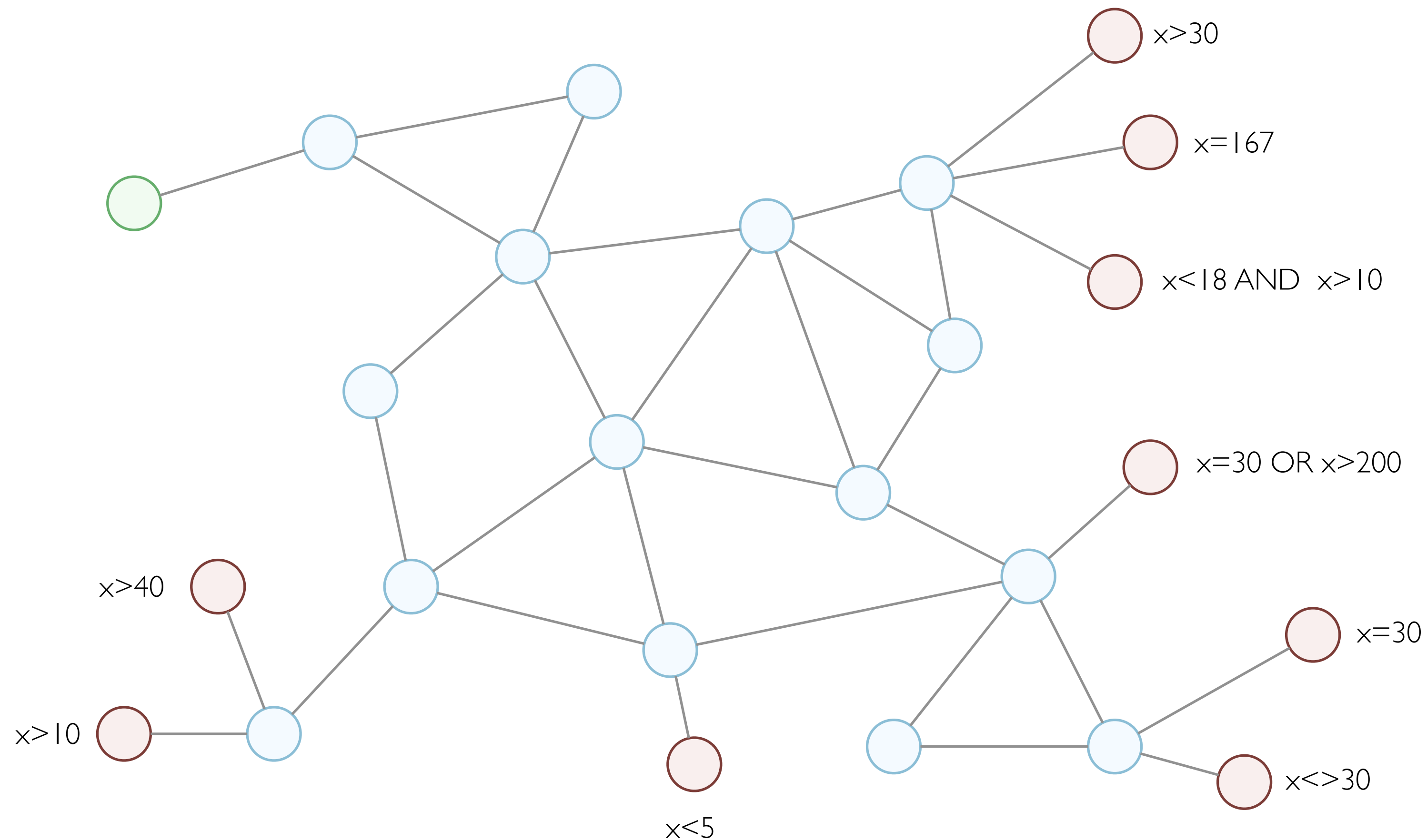
- Each node has a service interface consisting of two operations:
  - send_message(m)
  - set_predicate(p)
- A predicate is a disjunction of conjunctions of constraints of individual attributes.
- A content-based network can be seen as a dynamically-configurable broadcast network, where each message is treated as a broadcast message whose broadcast tree is dynamically pruned using content-based addresses.

Combined Broadcast and Content-Based (CBCB) routing scheme.

Content-based layer: "prunes" broadcast forwarding paths

Broadcast layer: diffuses messages in the network

Overlay point-to-point network: manages connections

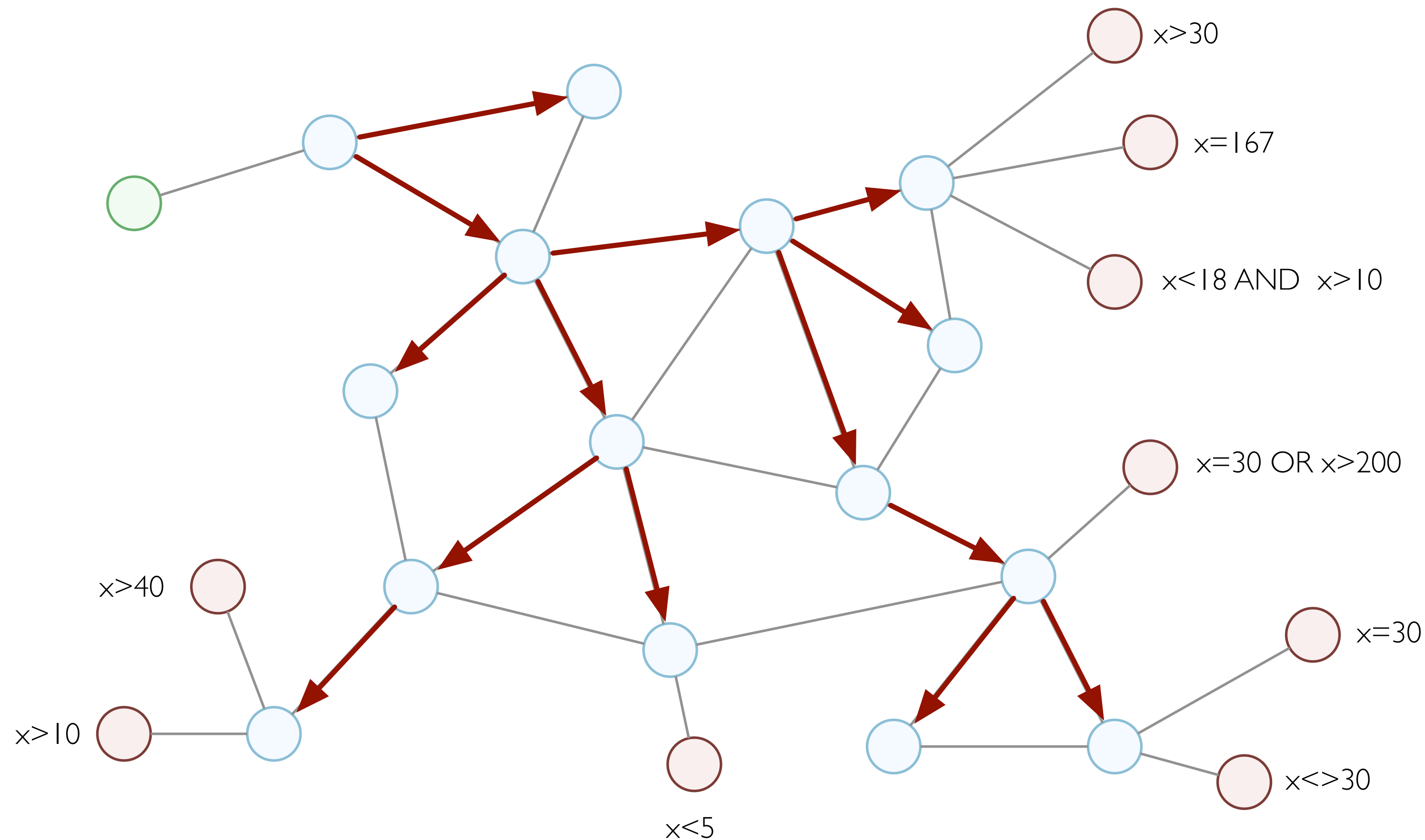


Combined Broadcast and Content-Based (CBCB) routing scheme.

Content-based layer: "prunes" broadcast forwarding paths

Broadcast layer: diffuses messages in the network

Overlay point-to-point network: manages connections



SIENA

Combined Broadcast and Content-Based (CBCB) routing scheme.

- Content-based layer: “prunes” broadcast forwarding paths
- Broadcast layer: diffuses messages in the network
- Overlay point-to-point network: manages connections

x=22

x>30

x=167

x<18 AND x>10

x=30 OR x>200

x>40

x>10

x=30

x<>30

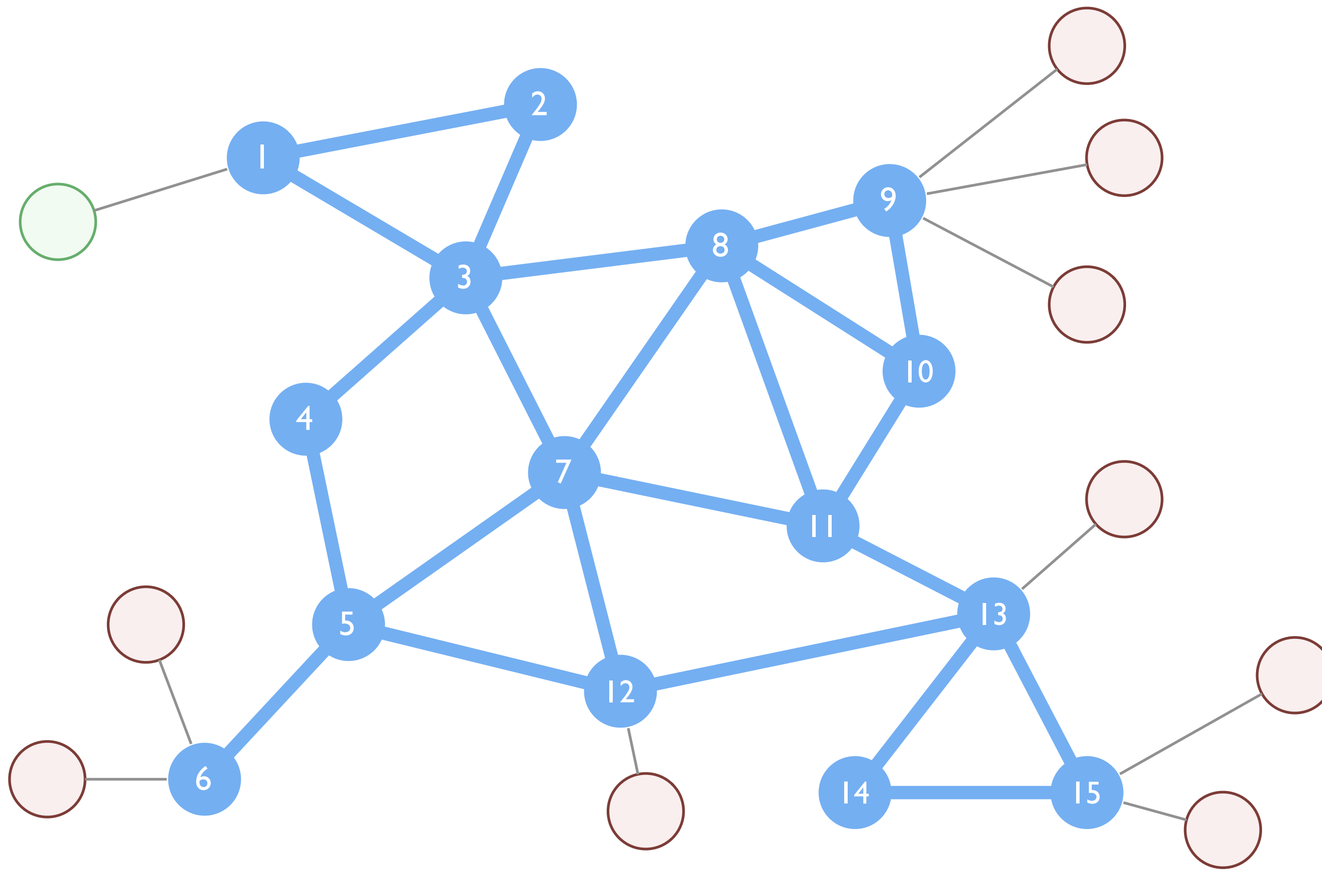
x<5

The broadcast layer:

- A *broadcast function* $B : N \times I \rightarrow I^*$ is available at each router. Given a source node *s* and an input interface *i*, it returns a set of output interfaces.
- The broadcast function defines a broadcast tree routed at each source node.
- The broadcast function satisfies the *all-pairs path symmetry* property: for each pair of nodes *x* and *y*, the broadcast function defines two broadcast trees T_x and T_y, rooted at nodes *x* and *y* respectively, such that the path $x \rightsquigarrow y$ in T_x is congruent to the reverse of the path $y \rightsquigarrow x$ in T_y.

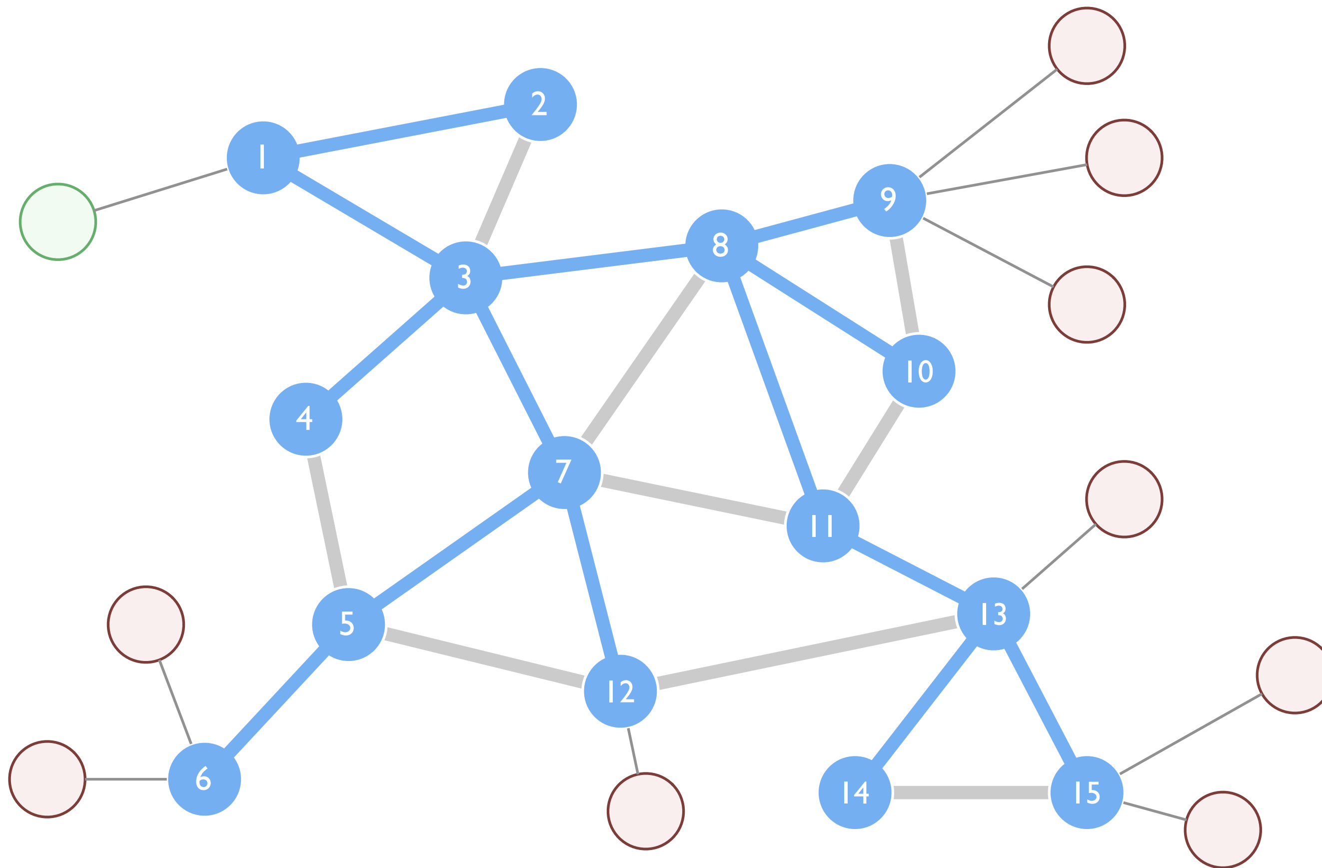
SIENA

Example:



SIENA

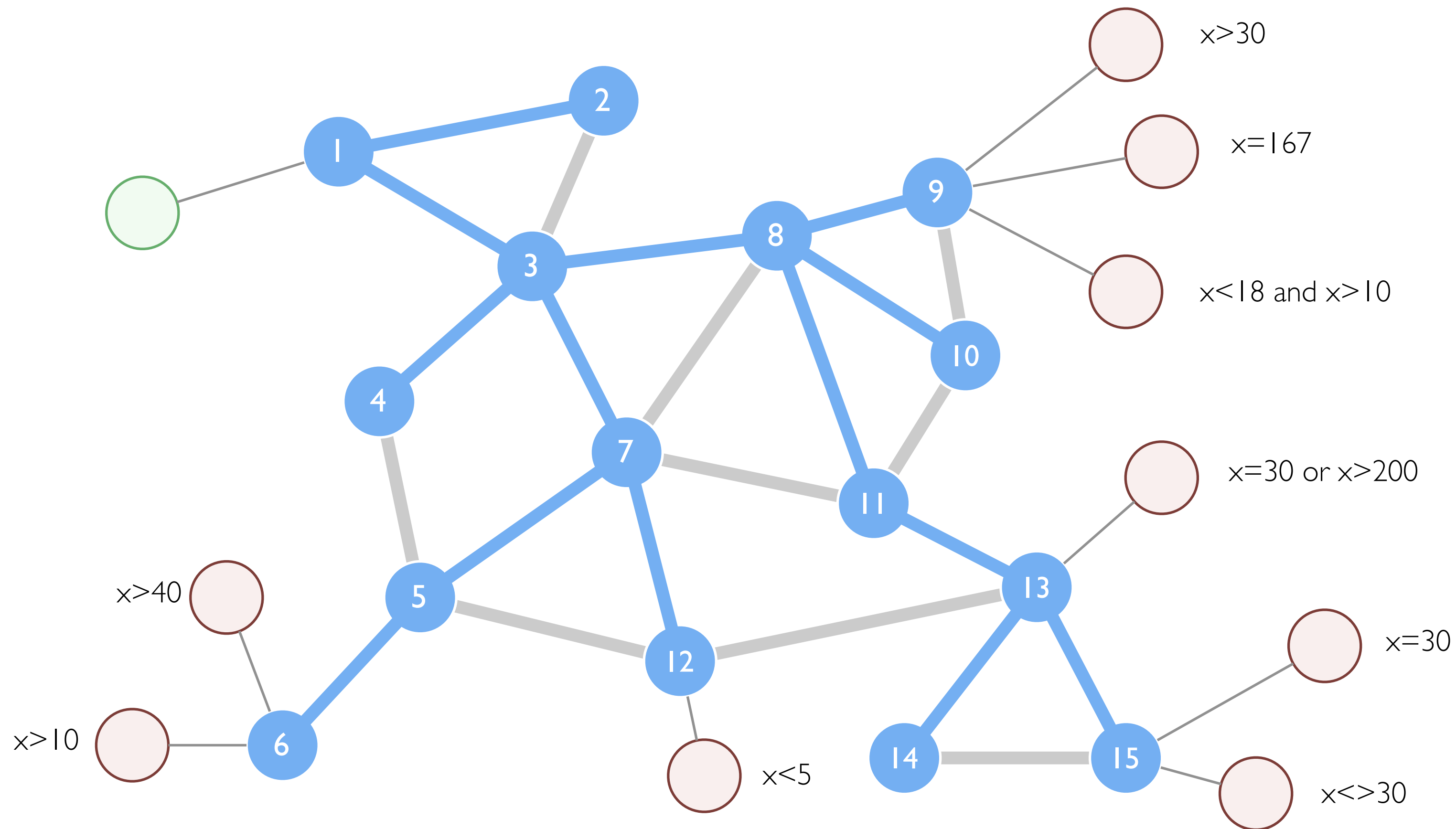
Example:



SIENA

The content-based layer:

- Maintains forwarding state in the form of a *content-based forwarding table*. The table, for each node, associates a content-based address to each interface.



SIENA

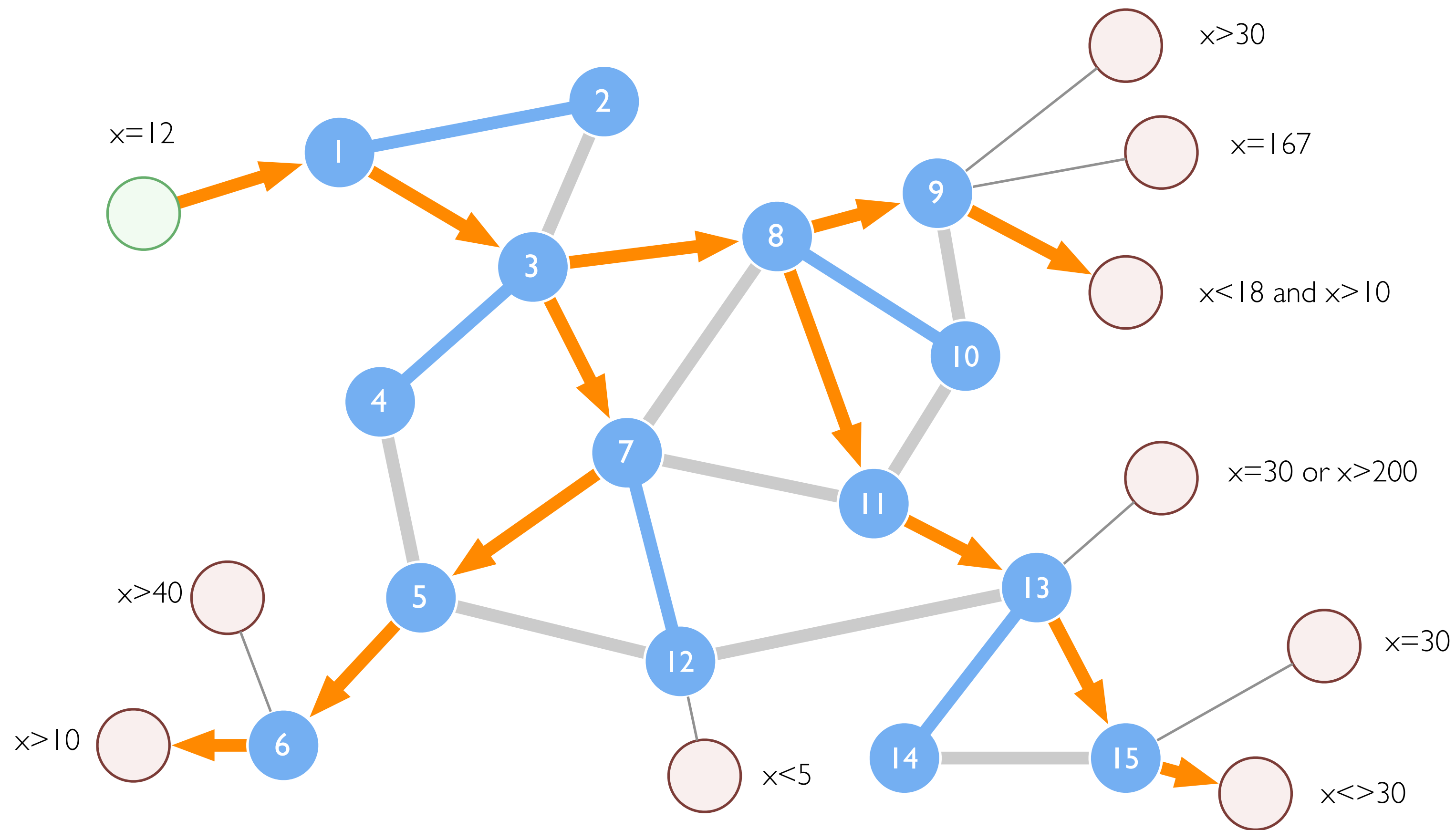
The message forwarding mechanism:

- The content-based forwarding table is used by a forwarding function F_c that, given a message *m*, selects the subset of interfaces associated with predicates matching *m*.
- The result of F_c is then combined with the broadcast function *B*, computed for the original source of *m*.
- A message is therefore forwarded along the set of interfaces returned by the following formula:

$$(B(source(m), incoming_if(m)) \cup \{I_0\}) \cap F_c(m)$$

SIENA

Example:



Forwarding tables maintenance:

- *Push* mechanism based on *receiver advertisements.*
- *Pull* mechanism based on *sender requests* and *update replies.*

Receiver advertisements:

- are issued by nodes periodically and/or when the node changes its local content-based address p_0.
- *Content-based RA ingress filtering*: a router receiving through interface *i* an RA issued by node *r* and carrying content-based address p_{RA} first verifies whether or not the content-based address p_i associated with interface *i* covers p_{RA}. If p_i covers p_{RA}, then the router simply drops the RA.
- *Broadcast RA propagation*: if p_i does not cover p_{RA}, then the router computes the set of next-hop links on the broadcast tree rooted in *r* (i.e., *B(r, i)*) and forwards the RA along those links.
- *Routing table update*: if p_i does not cover p_{RA}, then the router also updates its routing table, adding p_{RA} to p_i, computing $p_i \leftarrow p_i \vee p_{RA}$.

Example: Broker 6 issues subscription s_1

i	*pred*
3	s_1

i	*pred*
4	s_1

i	*pred*
6	s_1

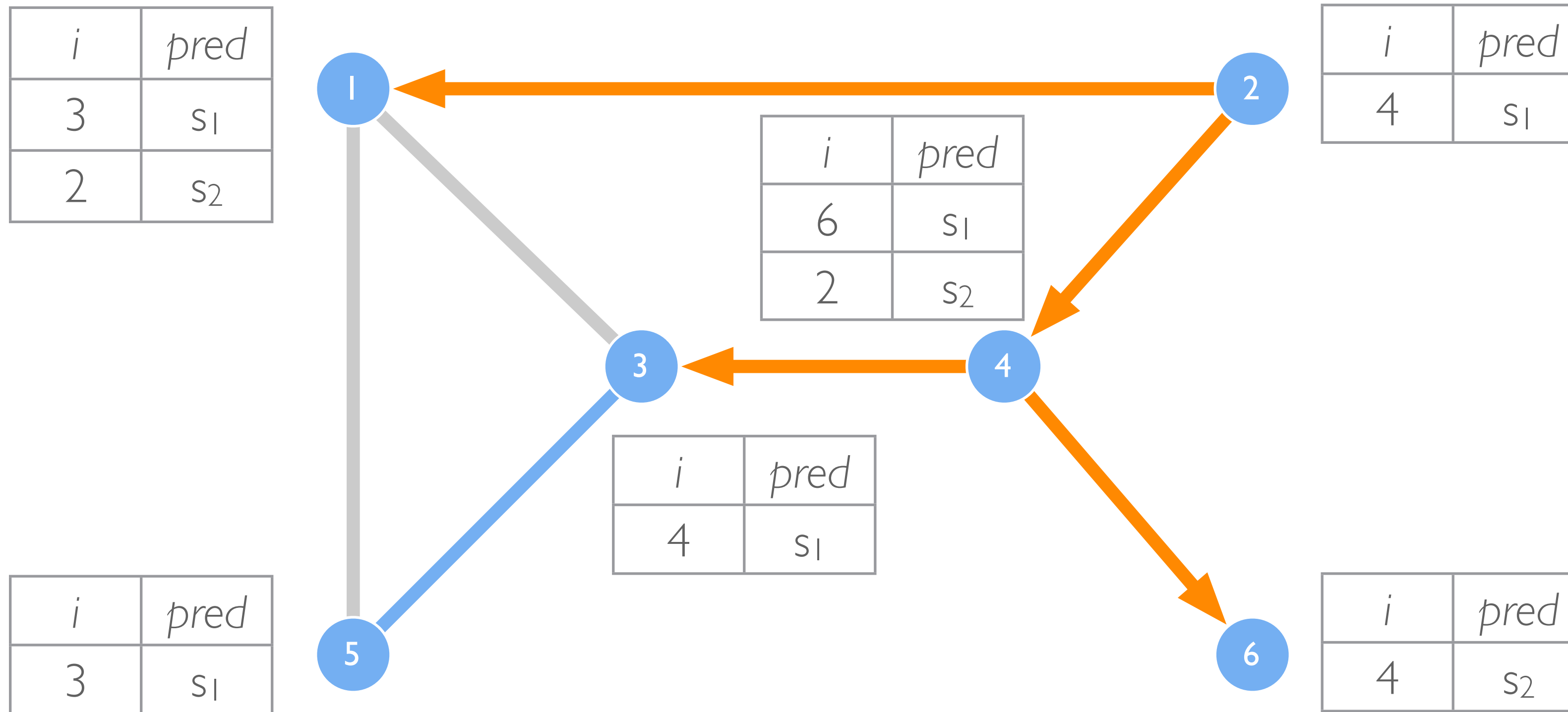
i	*pred*
4	s_1

i	*pred*
3	s_1

1 2 3 4 5 6

3I SIENA

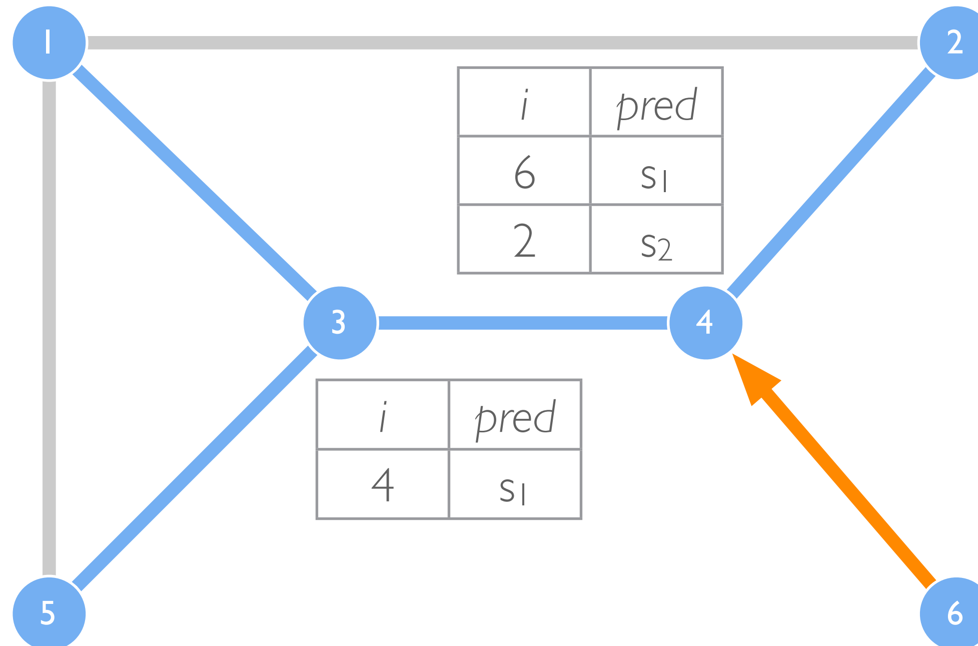
Example: Broker 2 issues subscription $s_2 < s_1$



SIENA

Notice that, because of the ingress filtering rule, the RA protocol can only widen the selection of the content-based addresses stored in routing tables. In the long run, this may cause an "inflation" of those content-based addresses.

Example: Broker 6 substitute its predicate with $s_3 < s_1$

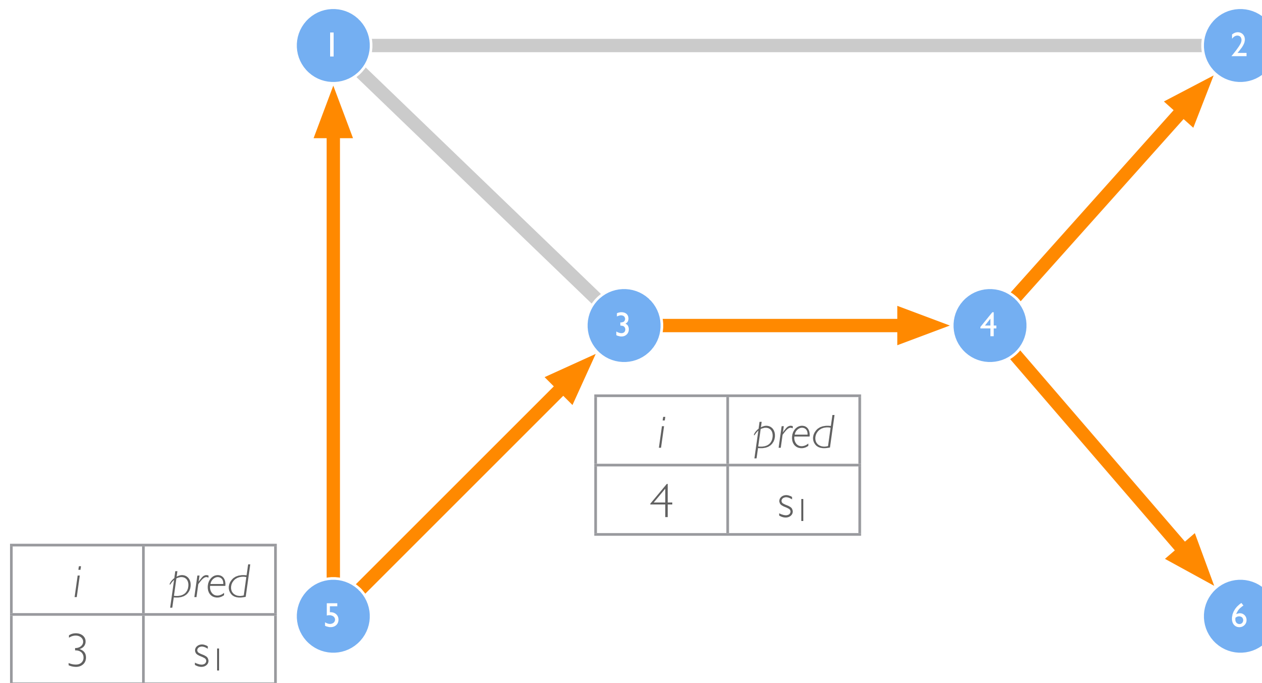


SIENA

Sender Requests and Update Replies:

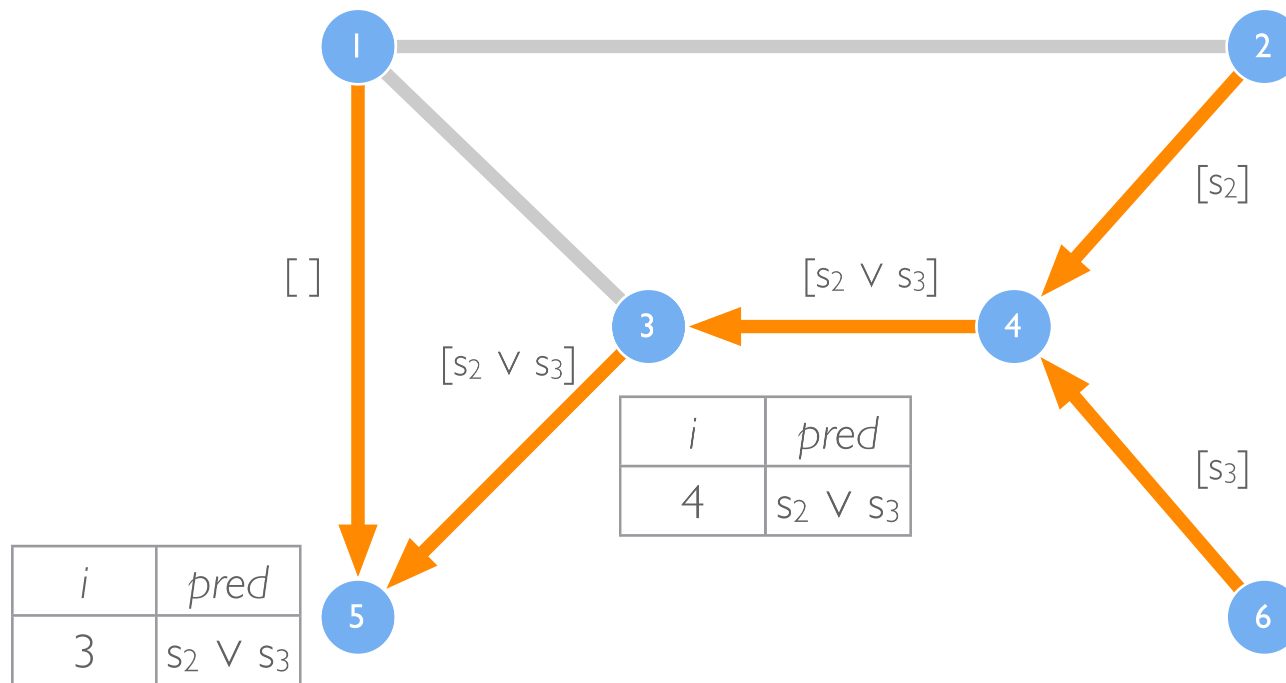
- A router uses sender requests (SRs) to pull content-based addresses from all receivers in order to update its routing table.
- The results of an SR come back to the issuer of the SR through update replies (URs).
- The SR/UR protocol is designed to complement the RA protocol. Specifically, it is intended to balance the effect of the address inflation caused by RAs, and also to compensate for possible losses in the propagation of RAs.
- An SR issued by n is broadcast to all routers, following the broadcast paths defined at each router by the broadcast function $B(n, \cdot)$.
- A leaf router in the broadcast tree immediately replies with a UR containing its content-based address p_0.
- A non-leaf router assembles its UR by combining its own content-based address p_0 with those of the URs received from downstream routers, and then sends its URs upstream.
- The issuer of the SR processes incoming URs by updating its routing table. In particular, an issuer receiving a UR carrying predicate p_{UR} from interface i updates its routing table entry for interface i with $p_i \leftarrow p_{UR}$.

Example: Broker 5 sends a Sender Request (SR) to refresh its forwarding table.

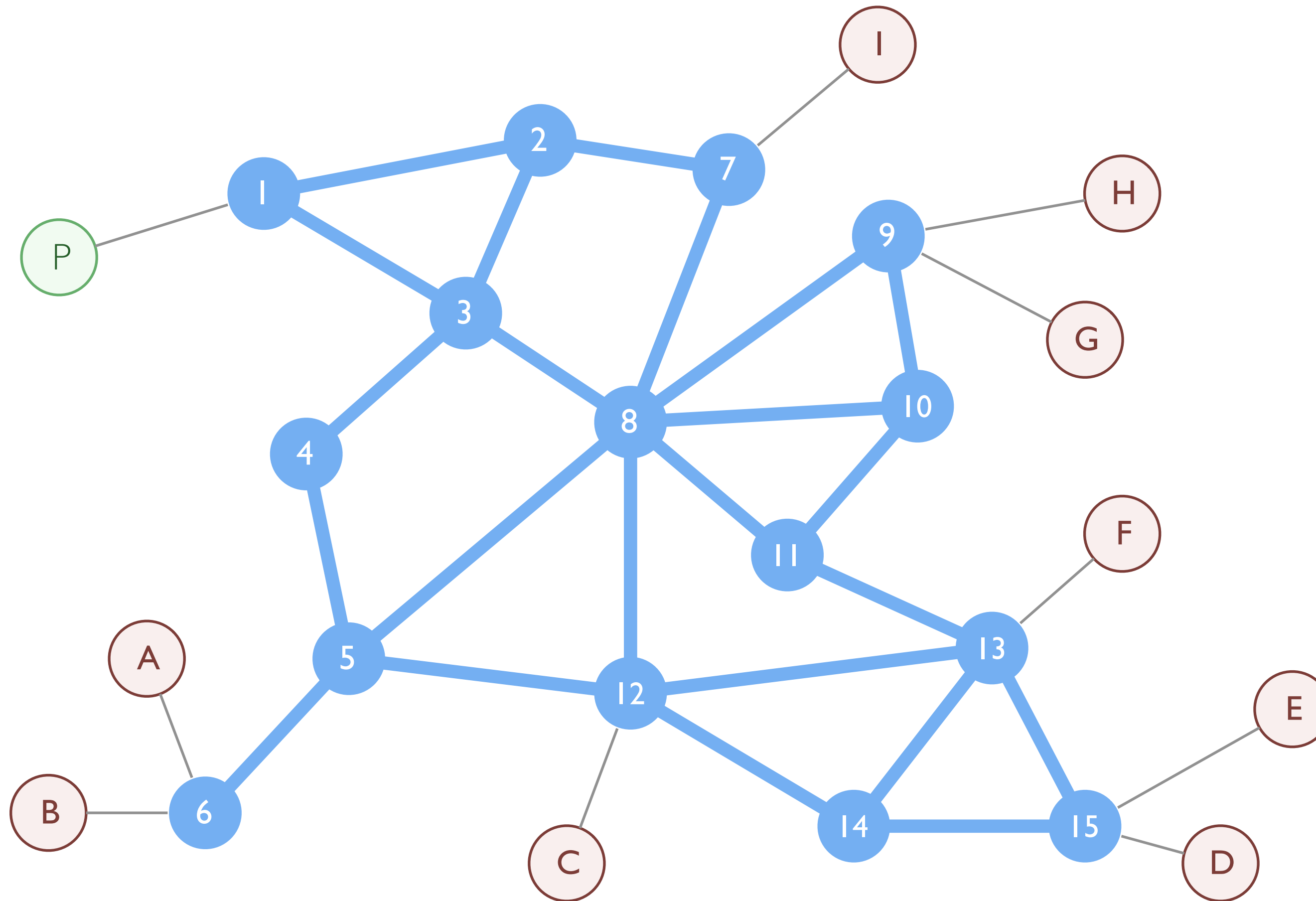


SIENA

Example: Update Replies (URs) are collected on the paths toward broker 5.



- Exercise: consider the following system:



- The event space is represented by a single numerical attribute x which can assume real values. Subscriptions can be expressed using the operators <=>.

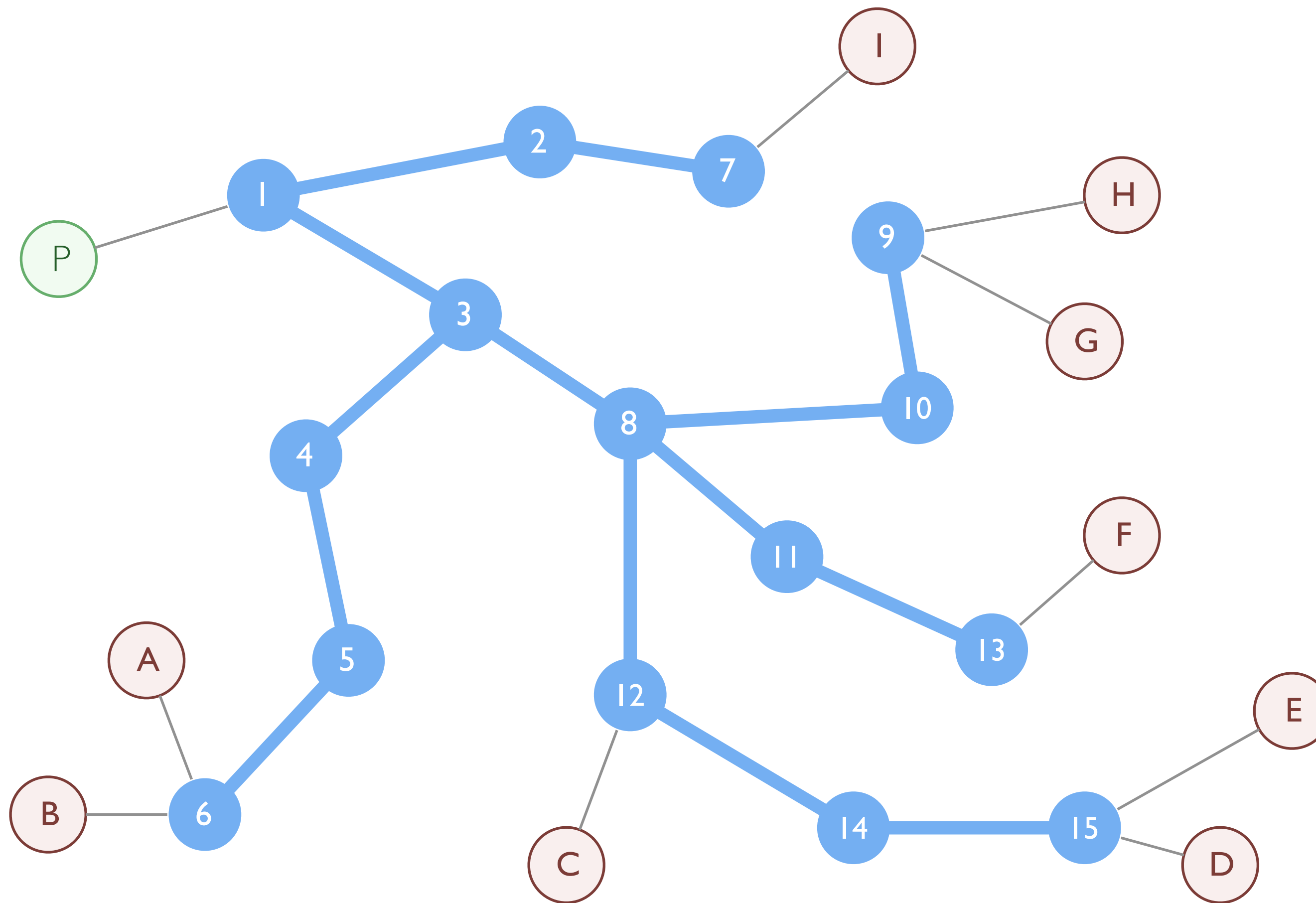
- Subscribers issued the following subscriptions.

Subscriber	Subscription
A	x>23
B	x<0 OR x>90
C	x<40
D	x>25 AND x<60
E	x>5 AND x<18
F	x>5 AND x<10
G	x>15 AND x<20
H	x<12
I	x>50

- Firstly define a spanning tree associated to the broker associated with publisher P. Then, for every broker compute the content-based forwarding table associated to this spanning tree. Finally compute the path followed by event x=16 through the ENS.

SIENA

- 1: define a spanning tree associated to broker 1
- Every tree including all the brokers is ok.



- The content of subscription tables is computed starting from each subscriber and "climbing the tree" toward the root (broker 1).

Broker	Interface	Content-based address
1	2	x>50
1	3	x>23 OR (x<0 OR x>90) OR x<40 OR (x>25 AND x<60)
2	7	x>50
3	4	x>23 OR (x<0 OR x>90)
3	8	x<40 OR (x>25 AND x<60)
4	5	x>23 OR (x<0 OR x>90)
5	6	x>23 OR (x<0 OR x>90)
8	10	x<12 OR (x>15 AND x<20)
8	11	x>5 AND x<10
8	12	x<40 OR (x>5 AND x<18) OR (x>25 AND x<60)
10	9	x<12 OR (x>15 AND x<20)
11	13	x>5 AND x<10
12	14	(x>5 AND x<18) OR (x>25 AND x<60)
14	15	(x>5 AND x<18) OR (x>25 AND x<60)

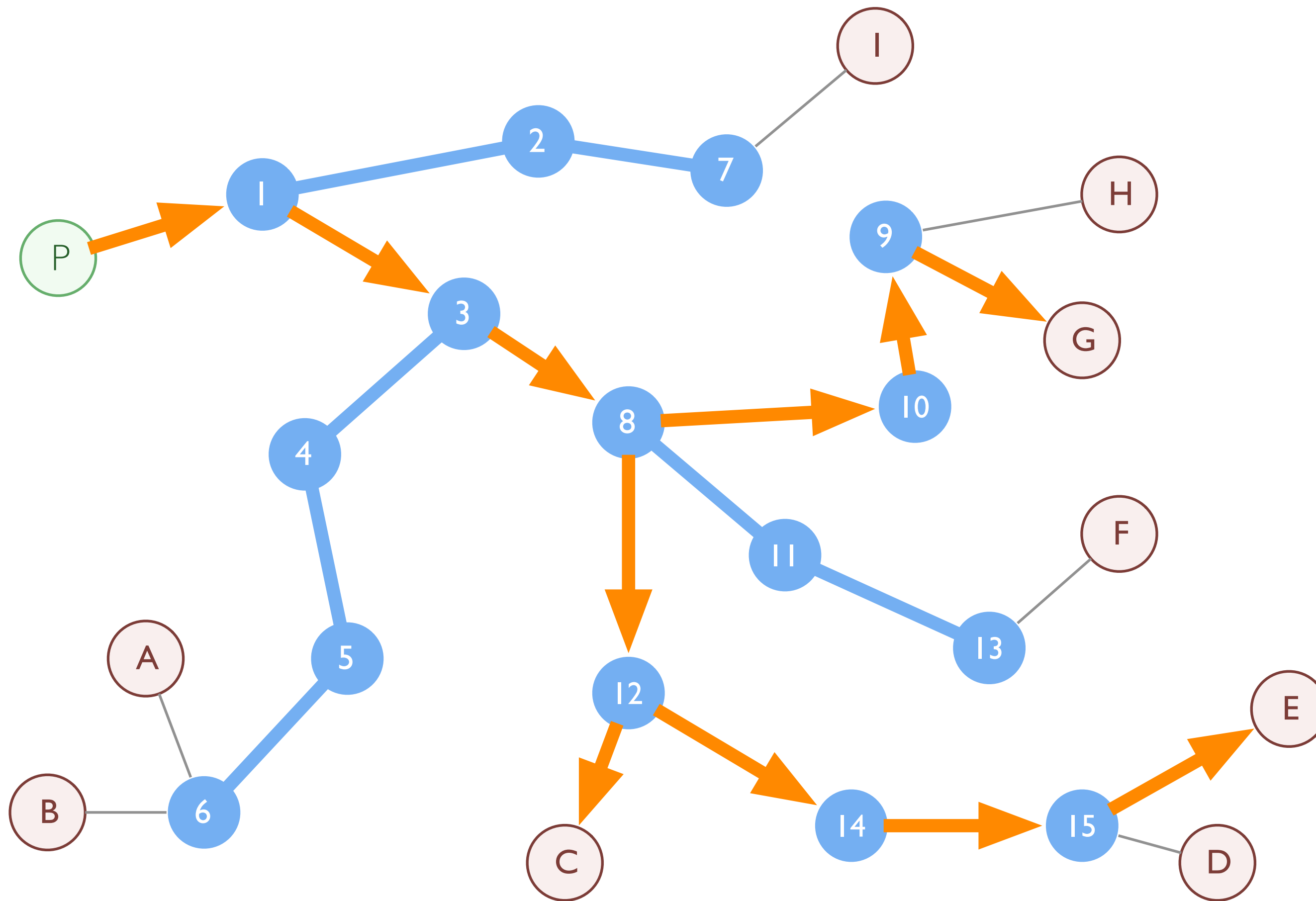
- We are referring to a run-time status where we can assume that, independently from the order used to issue subscriptions, the tables' content is perfect.

- Routing event x=16. Notified subscribers: C, E, G.
- The table reports which content-based addresses are satisfied by the event (in blue).

Broker	Interface	Content-based address
1	2	x>50
1	3	x>23 OR (x<0 OR x>90) OR x<40 OR (x>25 AND x<60)
2	7	x>50
3	4	x>23 OR (x<0 OR x>90)
3	8	x<40 OR (x>25 AND x<60)
4	5	x>23 OR (x<0 OR x>90)
5	6	x>23 OR (x<0 OR x>90)
8	10	x<12 OR (x>15 AND x<20)
8	11	x>5 AND x<10
8	12	x<40 OR (x>5 AND x<18) OR (x>25 AND x<60)
10	9	x<12 OR (x>15 AND x<20)
11	13	x>5 AND x<10
12	14	(x>5 AND x<18) OR (x>25 AND x<60)
14	15	(x>5 AND x<18) OR (x>25 AND x<60)

4I SIENA

- On the graph:



SIENA

Miguel Castro, Peter Druschel, Anne-Marie Kermarrec and Antony Rowstron

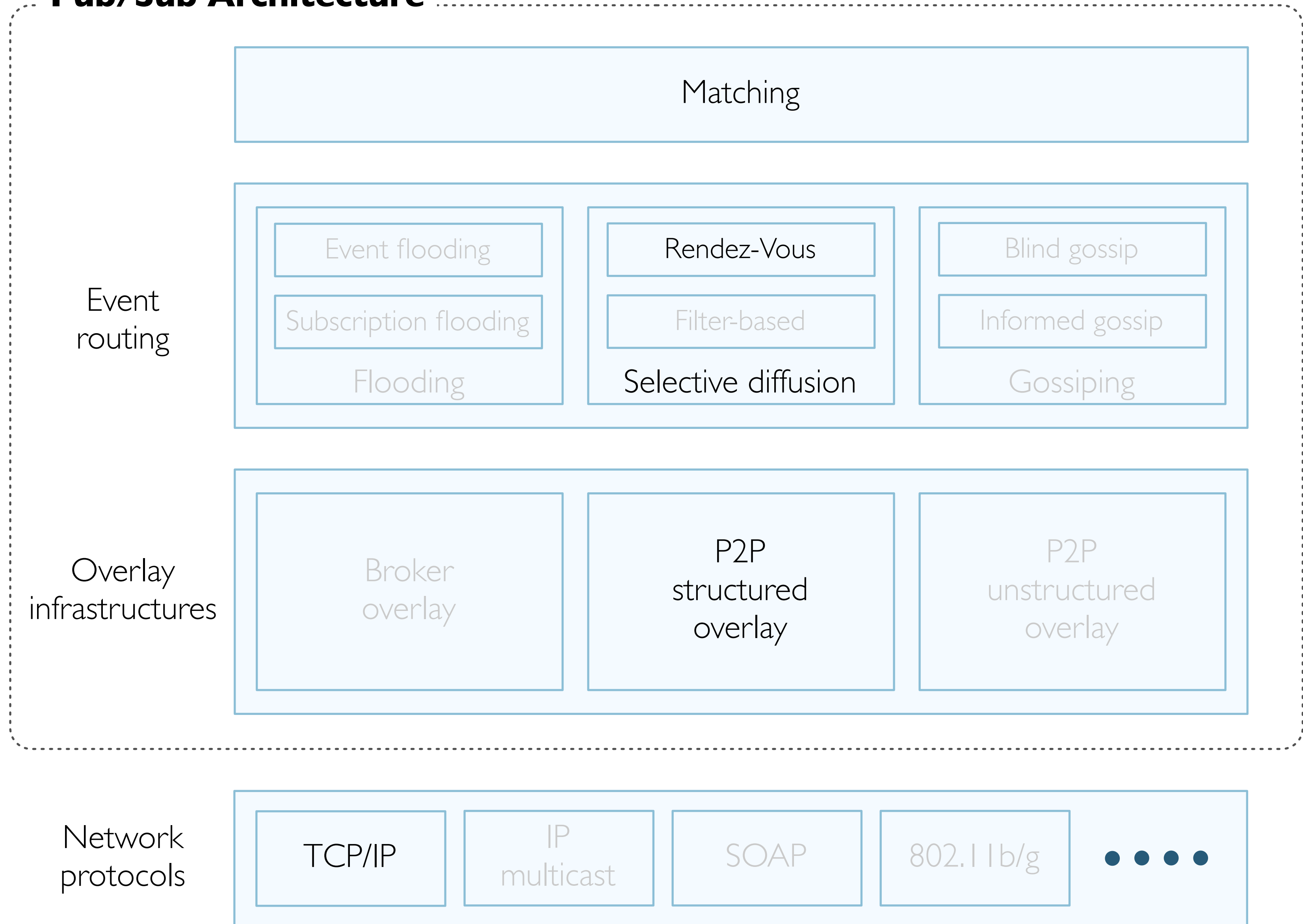
"SCRIBE: A large-scale and decentralized application-level multicast infrastructure"

Journal on Selected Areas in Communications, 2002.

Abstract:

"This paper presents Scribe, a scalable application-level multicast infrastructure. Scribe supports large numbers of groups, with a potentially large number of members per group. Scribe is built on top of Pastry, a generic peer-to-peer object location and routing substrate overlayed on the Internet, and leverages Pastry's reliability, self-organization, and locality properties. Pastry is used to create and manage groups and to build efficient multicast trees for the dissemination of messages to each group. Scribe provides best-effort reliability guarantees, but we outline how an application can extend Scribe to provide stronger reliability. Simulation results, based on a realistic network topology model, show that Scribe scales across a wide range of groups and group sizes. Also, it balances the load on the nodes while achieving acceptable delay and link stress when compared to IP multicast."

The specific architecture of this system:



44

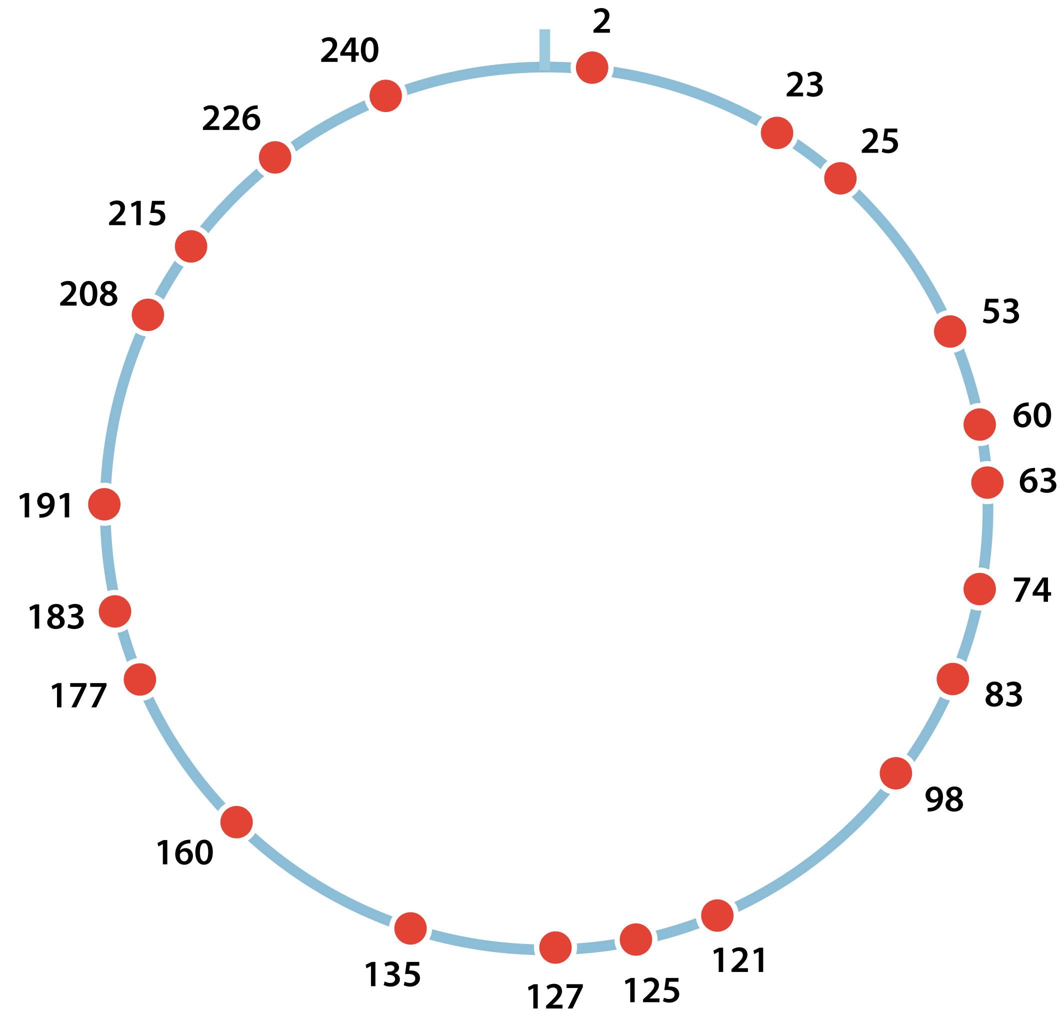
SCRIBE

- **Scribe** is a topic-based publish/subscribe system able to support a large number of groups with a potentially large number of publishers and subscribers.
- Each user in the system (publisher or subscriber) is also a broker. The event notification service is therefore constituted by all the users.
- Users can join and leave the system. The event notification service can therefore change at runtime.
- Scribe is built upon **Pastry**, a peer-to-peer location and routing service.
- Pastry is used to build and maintain the application-level topology that connects brokers in the event notification service.
- Pastry also provides applications with efficient primitives for object storage and location.

- Pastry implements a Distributed Hash Table:
  - Each object is associated with a *key*.
  - Each key is stored (together with the corresponding objects) in a node.
  - Each object can be efficiently located and retrieved knowing its key.
- Each node participating to Pastry is identified by 128-bit NodeID obtained applying a hash function *h* to its IP address.
- NodeId is in base 2^b, where *b* is a configuration parameter.
- The function *h* evenly distributes node identifiers in the circular key-space $[0, 2^{128}-1]$.
- Each object is stored on the node with the closest NodeID.
- Each node maintains three data structures:
  - Leaf set
  - Routing table
  - Neighborhood set

- **Leaf set**: contains the set of nodes with the L/2 numerically closest larger NodeIDs, and the L/2 nodes with numerically closest smaller NodeIDs, relative to the present node's NodeID.
- Example: node 60, L=6

LS_{60}
23
25
53
63
74
83



- **Routing table**: matrix of Log_2^b N rows and 2^b-1 columns. Entries in the *n*-th row match the first *n-1* digits of current NodeID. The n-th digit has one of the 2^b-1 possible values other than the n-th digit in current NodeID.
- Example: routing table at node 10233102, b=2

	Possible digit values			
	0	1	2	3
Row 1	-0-2212102	1	-2-2301203	-3-1203203
Row 2	0	1-1-301233	1-2-230203	1-3-021022
Row 3	10-0-31203	10-1-32102	2	10-3-23302
Row 4	102-0-0230	102-1-1302	102-2-2302	3
Row 5	1023-0-322	1023-1-000	1023-2-121	3
Row 6	10233-0-01	1	10233-2-32	
Row 7	0		102331-2-0	
Row 8			2	

48

SCRIBE

- **Neighborhood set**: list of the M closest nodes.
- Node distance is measured using a proximity metric (IP hops, latency, bandwidth, etc).
- Nodes in this list are used to update entries in the routing table.

- The main function provided by Pastry is *route(msg,key).*
- Routing is realized matching key prefixes with nodes stored in each routing table.
- In each routing step, the current node forwards the message to a node whose NodeID shares with the target key a prefix that is at least one digit longer than the prefix that the key shares with the current NodeID.
- If no such node is found in the routing table the message is forwarded to a node whose NodeID shares a prefix with the key as long as the current node, but numerically close to the key than the current NodeID.

- Scribe use the *key-node* mapping provided by Pastry to assign a rendez-vous node to each topic:
  - Each topic *t* (called Group in Scribe) is mapped to a key applying *h(t)*
  - *EN(e)=h(e), SN(s)=h(s)*
- Membership management:
  - Joining a group
  - Leaving a group
- Message diffusion

- When a node *n* wants to subscribe to *t* (joing group *t*):
    - it invokes *route(JOIN[t],h(t))*
    - the message is routed toward the rendez-vous node for *t*
    - each node *n'* along the route checks a local **groups list** to see if it is currently a forwarder for *t*
        - if so it accepts *n* as a child, and adds it to the local **children table**
        - otherwise it adds *t* to the groups list, add *n* to the children table and, finally, invokes *route(JOIN[t],h(t))*
- A node can unsubscribe *t* at any time:
    - if it has no children then it sends to its parent in the diffusion tree a LEAVE message
    - if it has still children for that group, it cannot leave the diffusion tree
- Routing is done in two steps:
    - the node that publish the event for topic *t* invokes *route(MCAST[e],h(t))*
    - when the message reaches the rendez-vous point it is diffused following links defined by children tables for that group.

- Example

Ring nodes: 2, 23, 25, 53, 60, 63, 74, 83, 98, 121, 125, 127, 135, 160, 177, 183, 191, 208, 215, 226, 240

Table near node 191:

h(t)	children	father
73	-	121

Table near node 74:

h(t)	children	father
73	83	83

Table near node 177:

h(t)	children	father
73	-	121

Table near node 83:

h(t)	children	father
73	121	74

Table near node 121:

h(t)	children	father
73	177,191	83

SCRIBE

R. Baldoni, R. Beraldi, V. Quema, L. Querzoni, S. Tucci Piergiovanni

"TERA: Topic-based Event Routing for Peer-to-Peer Architectures"

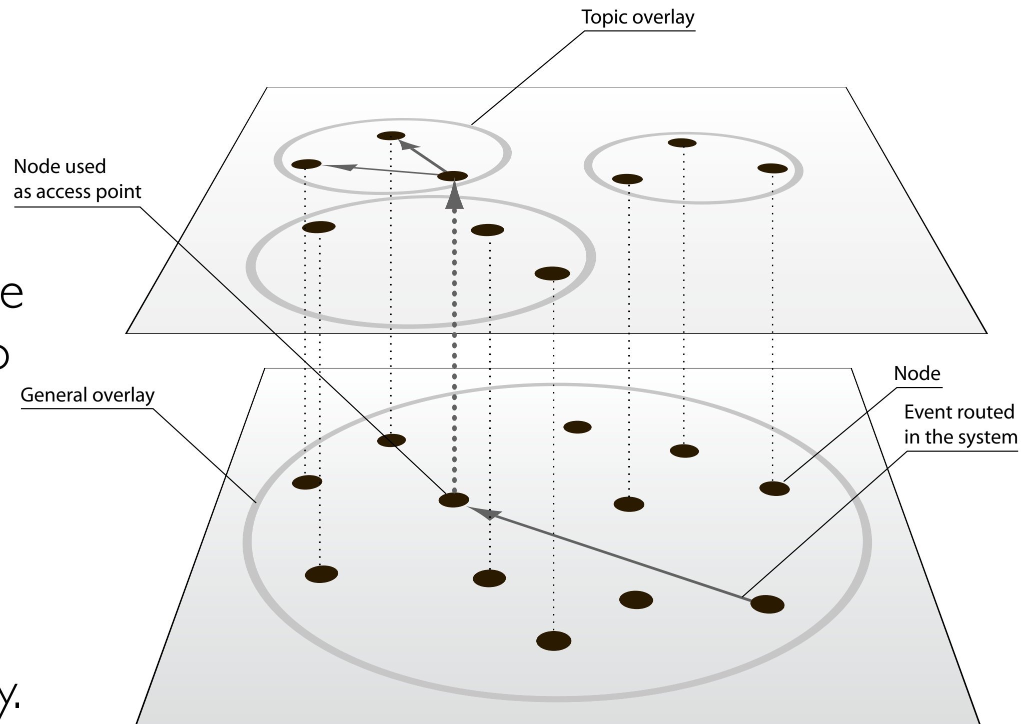
International Conference on Distributed Event-Based Systems (DEBS), 2007.

Abstract:

"The completely decoupled interaction model offered by the publish/subscribe communication paradigm perfectly suits the interoperability needs of todays large-scale, dynamic, peer-to-peer applications. Unmanaged inter-administrative environments, where these applications are expected to work, pose a series of problems (potentially wide number of partipants, low-reliability of nodes, absence of a centralized authority, etc.) that severely limit the scalability of existing approaches which were originally thought for supporting distributed applications built on the top of static and managed environments. In this paper we propose a novel architecture for implementing the topic-based publish/subscribe paradigm in large scale peer-to-peer systems. The proposed architecture is based on probabilistic mechanisms and peer-to-peer overlay management protocols. It achieves event diffusion by implementing traffic confinement (published events have a high probability to reach only interested subscribers), high scalability (with respect to several fundamental parameters like number of participants, subscriptions, topics and event publication rate) and fair load distribution (load distribution closely follows the distribution of subscription on nodes)."

- A two-layer infrastructure:
  - All clients are connected by a single overlay network at the lower layer (general overlay).
  - Various overlay network instances at the upper layer connect clients subscribed to same topics (topic overlays).

- Event diffusion:
  - The event is routed in the general overlay toward one of the nodes subscribed to the target topic.
  - This node acts as an access point for the event that is then diffused in the correct topic overlay.



subscribe

Event M

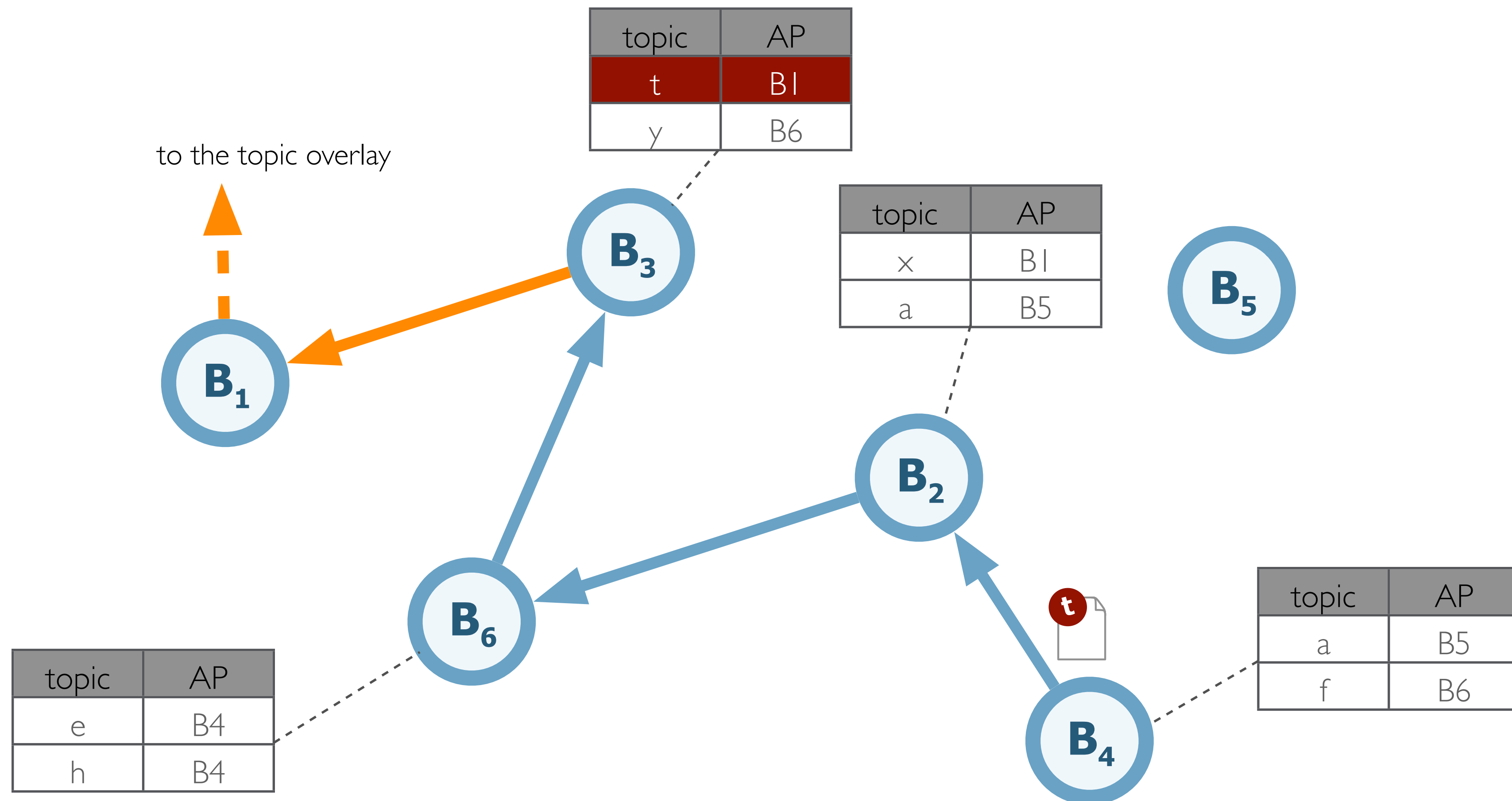
Access P

Partitio

Bro

TERA

- Event routing in the general overlay is realized through a random walk.
- The walk stops at the first broker that knows an access point for the target topic.

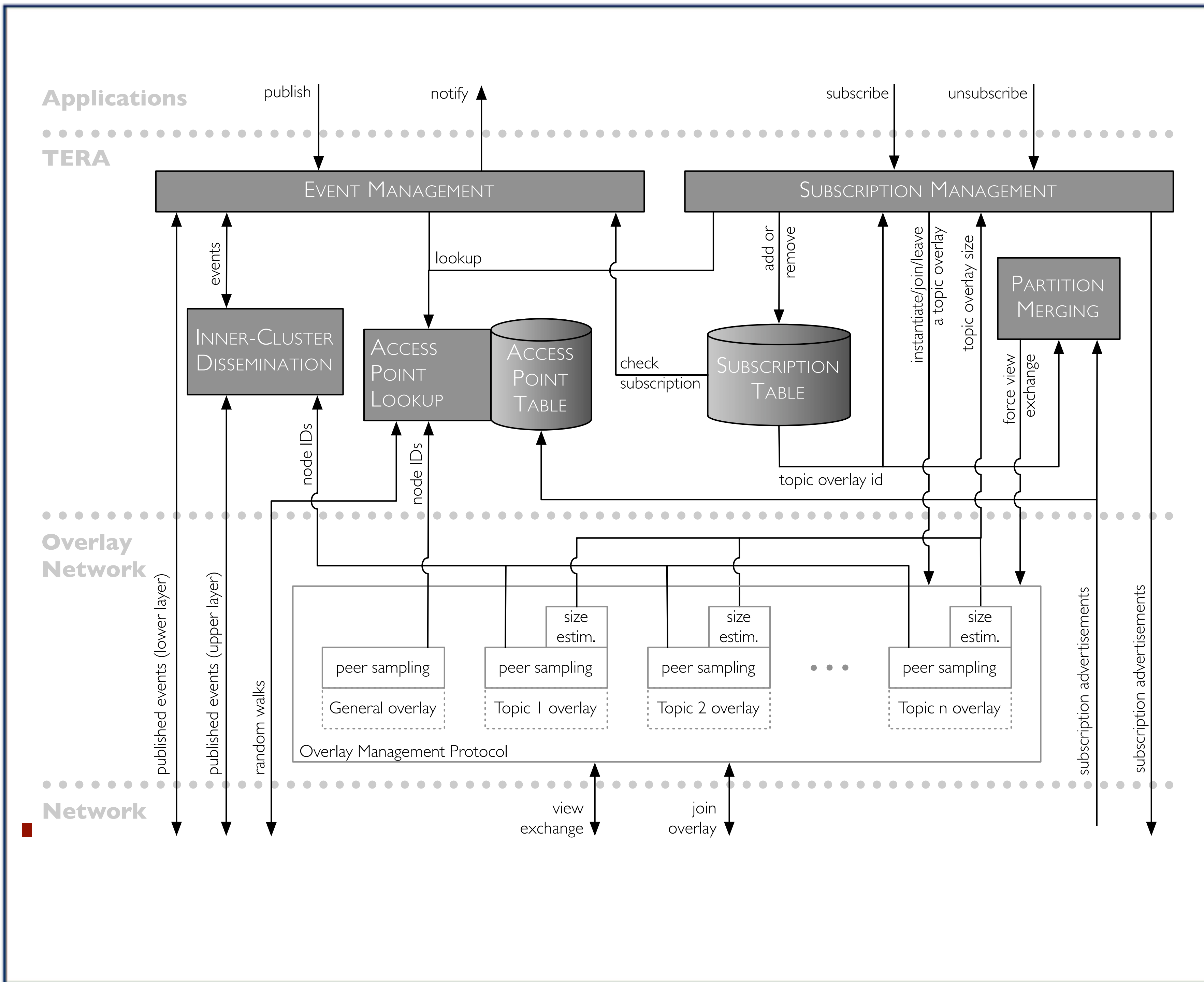


- Each node maintains locally an Access Point Table (APT)
- Each entry in the APT is a couple <topic, node address>
- An entry <*t*,*n*> represents the fact that *n* is an access point for topic *t*.
- The length of the APT is fixed.
- Goal:
  - each topic in the APT must be a uniform random sample among all the topics in the system;
  - the access point associated to a topic in an APT must be a uniform random sample among all the odes subscribed to that topic.

topic	AP
x	B1
a	B5

- Subscription advertisement:
  - each node periodically advertises its subscriptions to a set of nodes chosen uniformly at random among the population;
  - each advertisement is a set of couples <topic, popularity>
  - An advertisement <*t,p*> represents the fact that there are (approximately) *p* nodes subscribed to topic *t*.

- APT update. When a node receives and advertisement <*t,p*> from node *n*:
  - if the APT contains an entry for <*t,m*> it simply puts *m*=*n*
  - otherwise it puts a new entry <*t,n*> in the APT with probability *1/p*



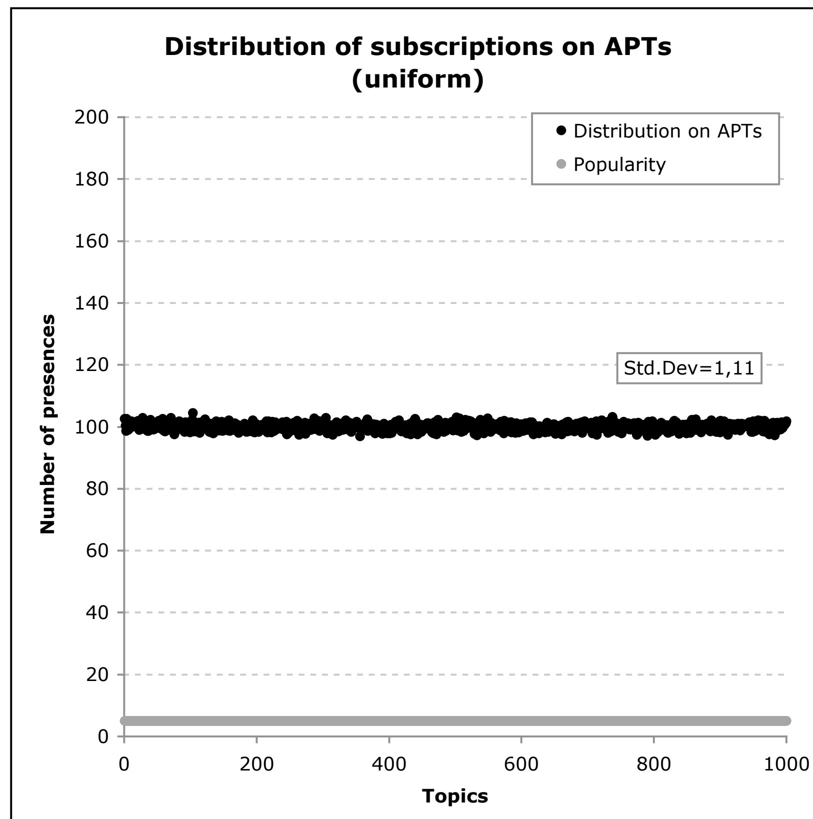
59

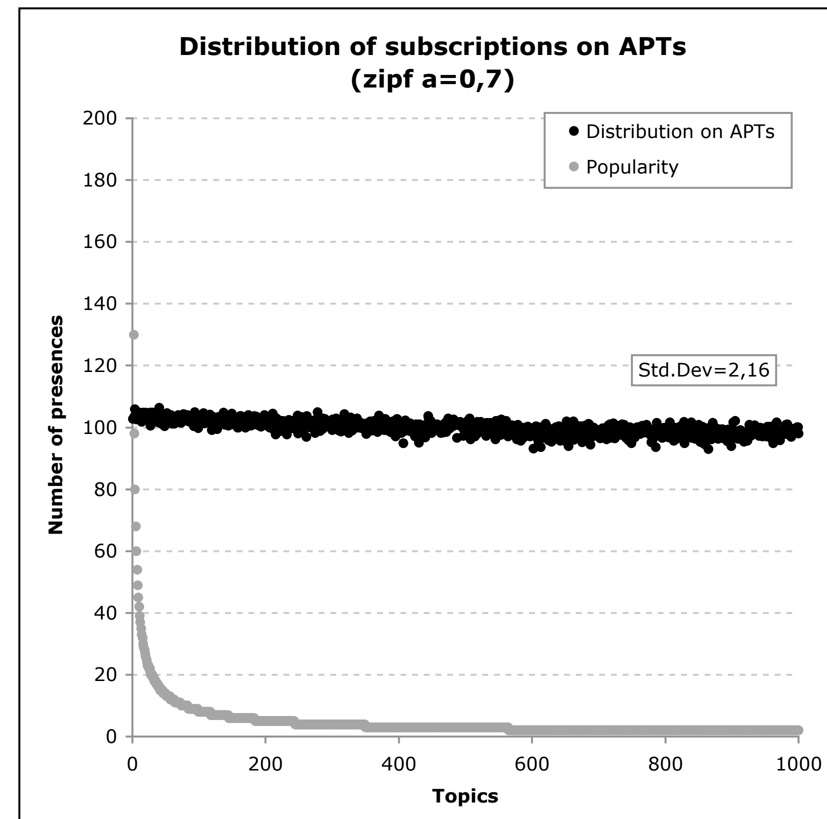
TERA

- We want every topic to appear \ APT, regardless of its popularity.

Nu

Topics





Distribution of subscriptions on APTs (zipf a=0,7)

Distribution on APTs

Popularity

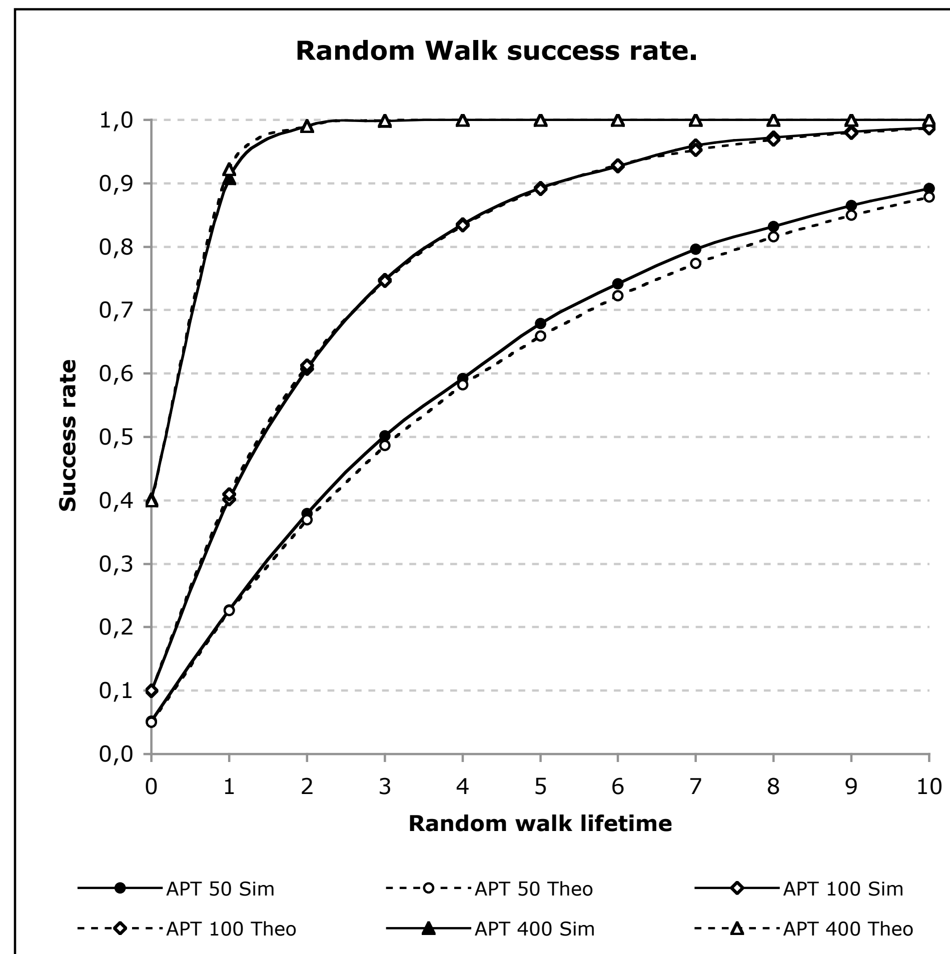
Distribution of subscriptions on APTs (zipf a=2,0)

Distribution on APTs

Popularity

TERA

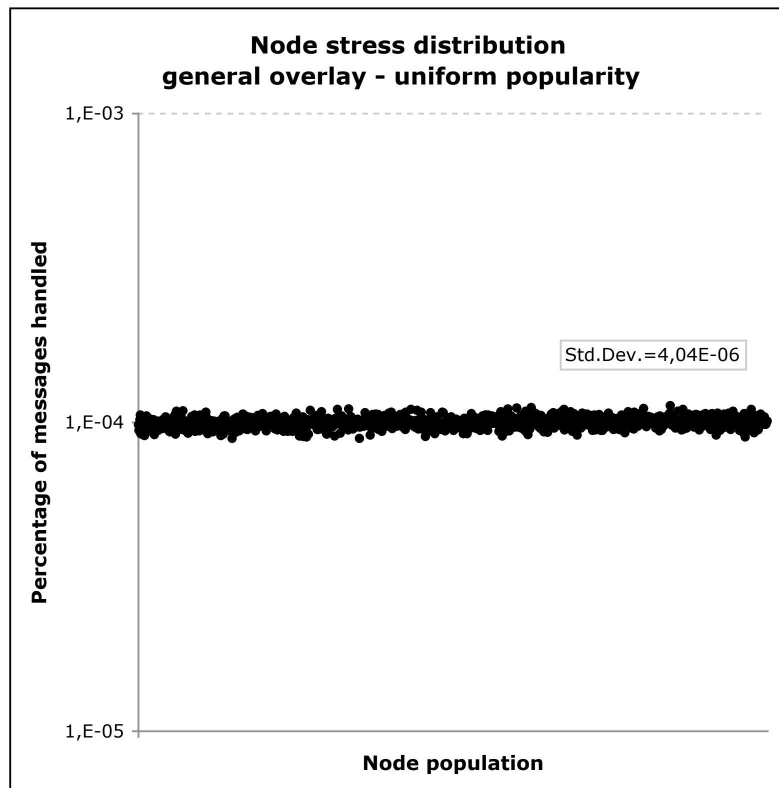
- Which is the probability for an event to be correctly routed in the general overlay toward an access point ?
- Depends on:
  - uniform randomness of topics contained in access point tables;
  - access point table size;
  - random walk lifetime.

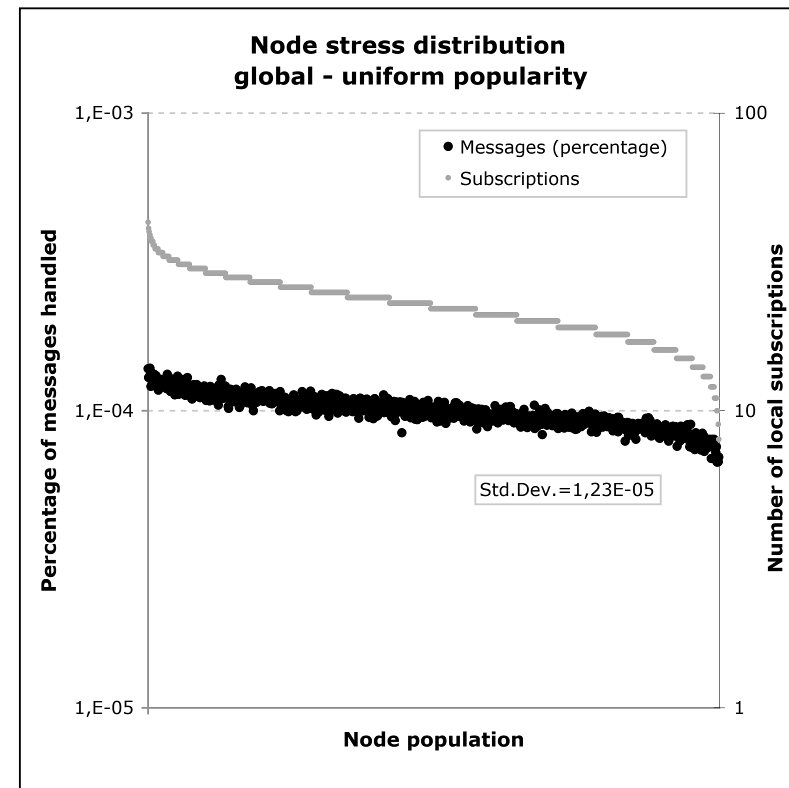


6I

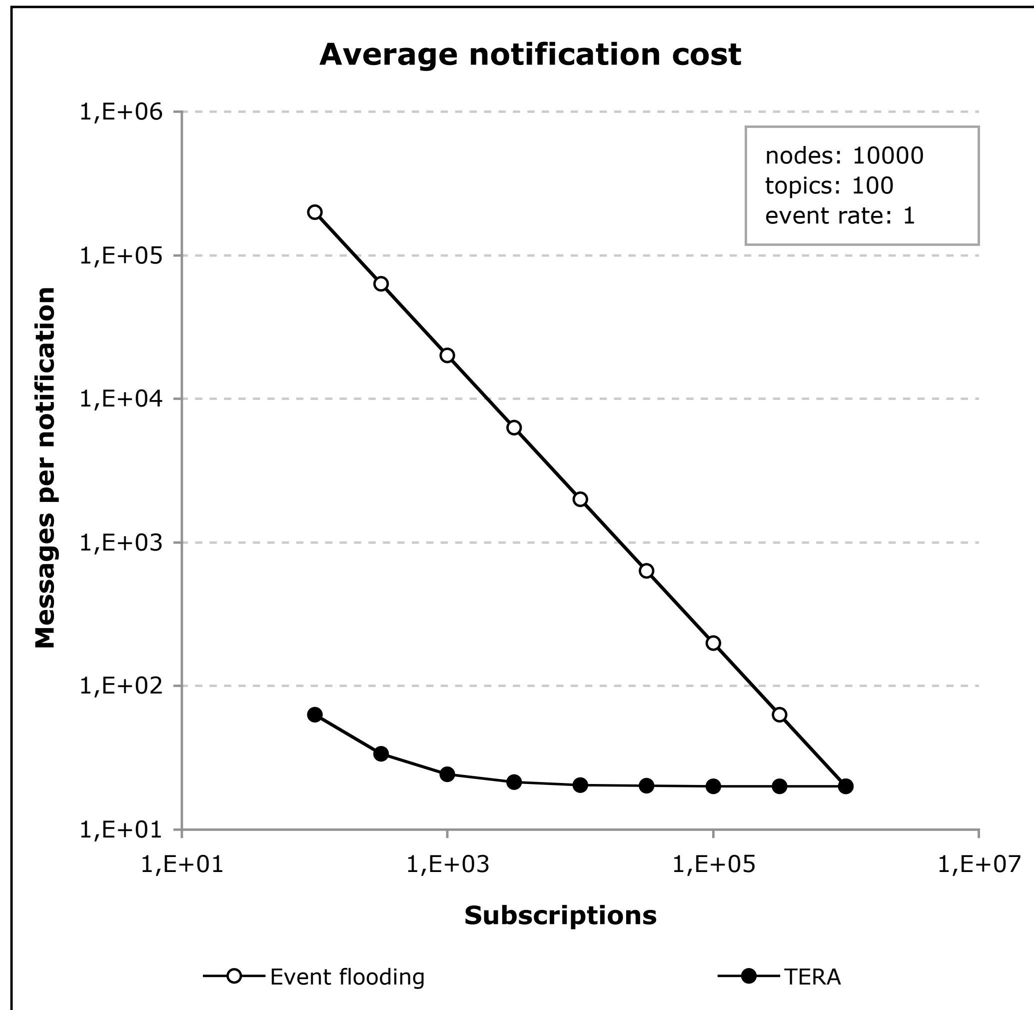
TERA

- Load imposed on nodes is fairly distributed:
  - no hot spots or single points of failure;
  - Nodes that subscribe to more topics suffer more load.

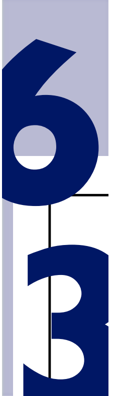




- Experiments show how the system scales with respect to:
  - Number of subscriptions.
  - Number of topics.
  - Event publication rate.
  - Number of nodes.



- (reference figure is given by a simple event flooding approach)



- Experiments show how the system scales with

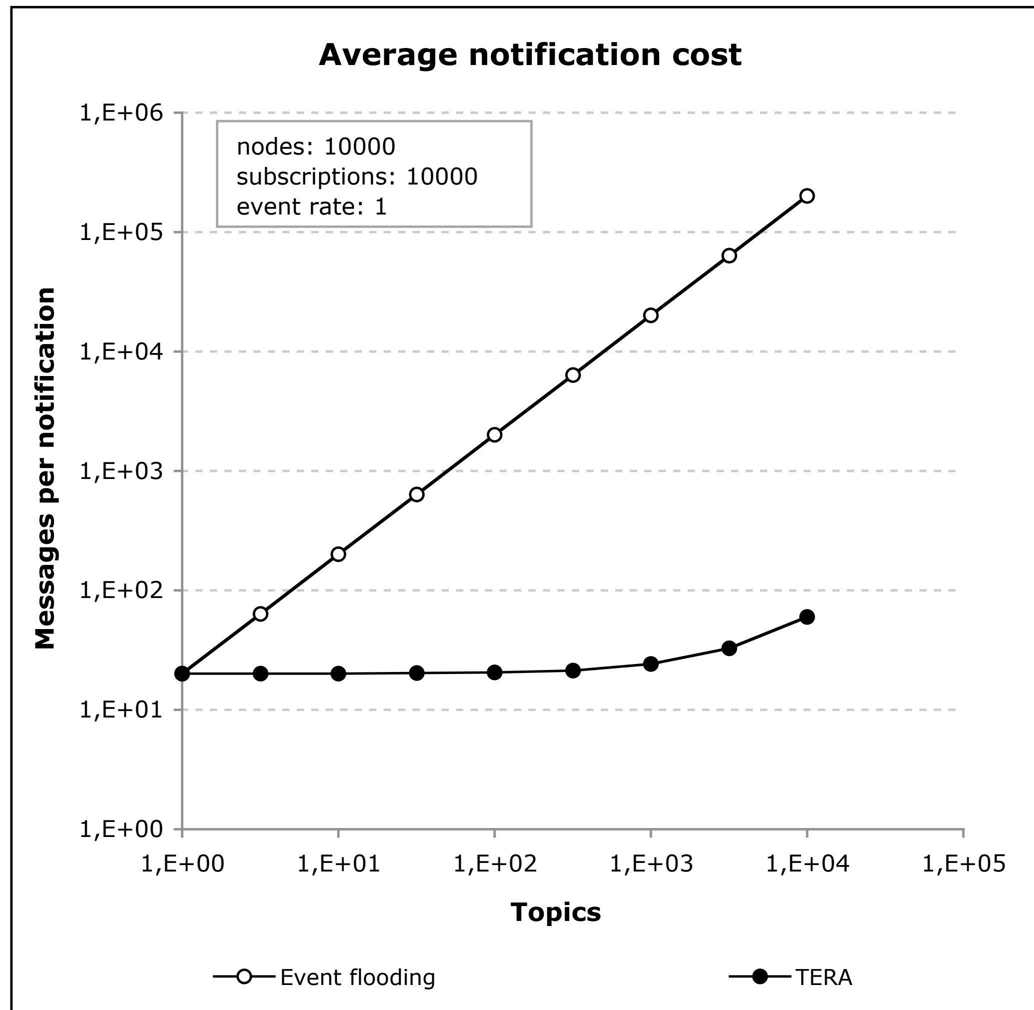
verage notification cost

nodes: 10000
topics: 100
event rate: 1

1,E+03 1,E+05 1,E+07

Subscriptions

looding TERA



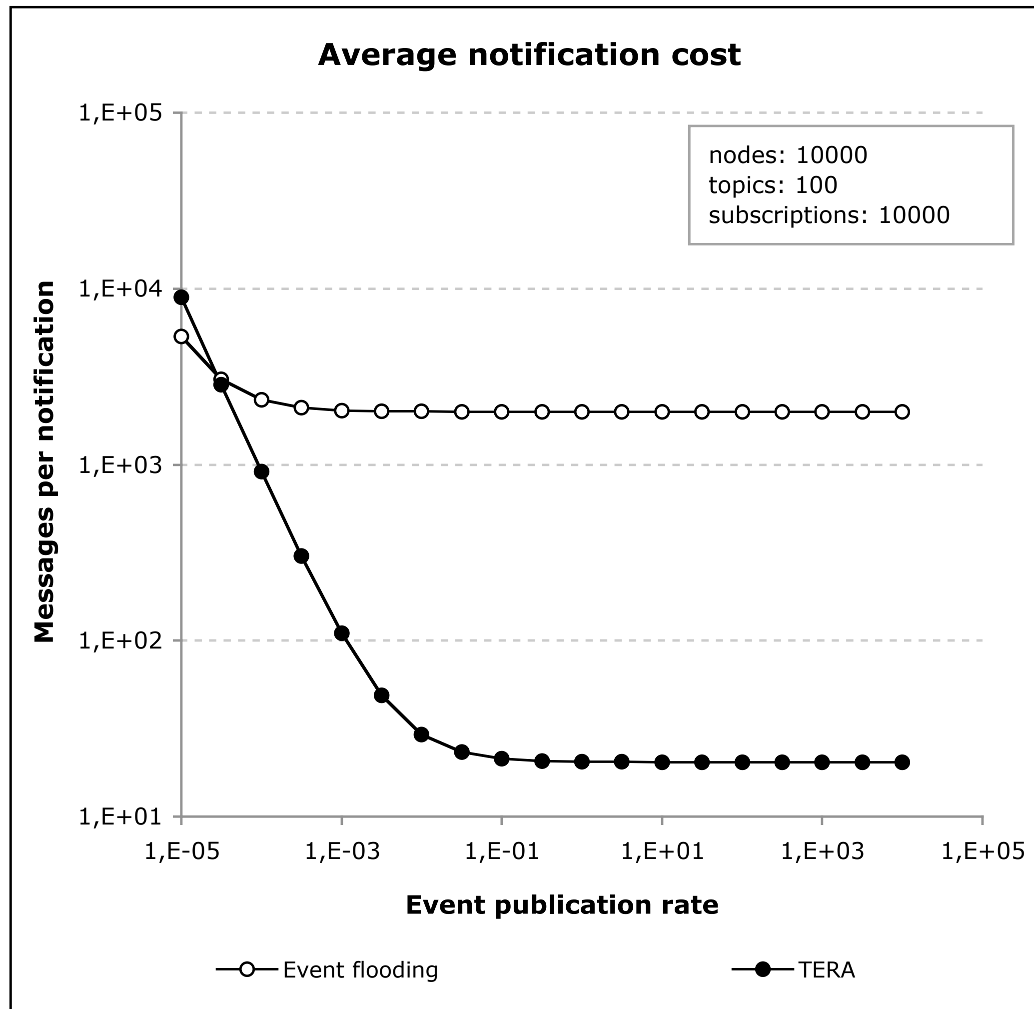
64

Messages per notification

1,E+04 1,E+03 1,E+02 1,E+01

TERA

Experiments show how
the system scales with



TERA 85

- Experiments show how the system scales with

Average notification cost

nodes: 10000
topics: 100
subscriptions: 10000

03 | 1,E-01 | 1,E+01 | 1,E+03 | 1,E+05

Event publication rate

flooding | TERA

Average notification cost

subscriptions: 10000
topics: 100
event rate: 1

Messages per notification

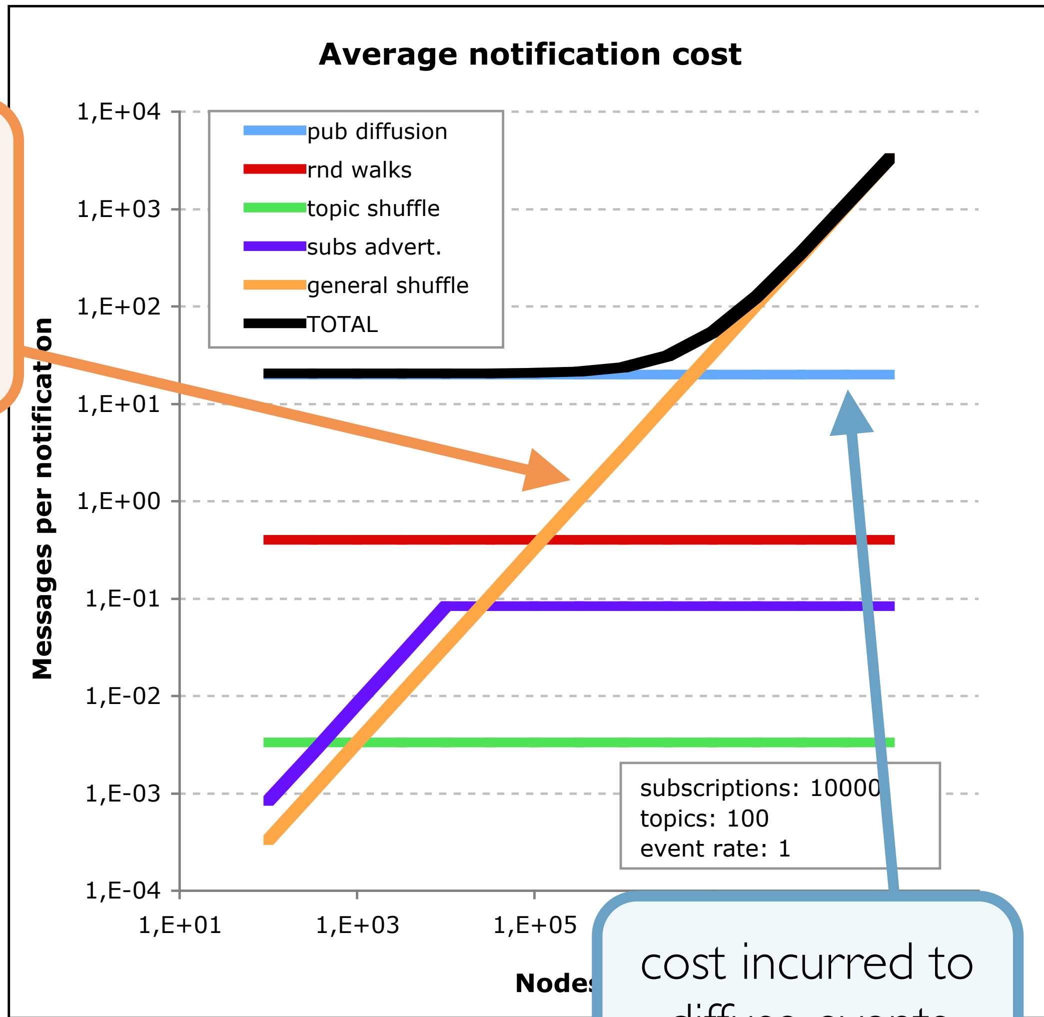
1,E+09 | 1,E+08 | 1,E+07 | 1,E+06 | 1,E+05 | 1,E+04 | 1,E+03 | 1,E+02 | 1,E+01

1,E+01 | 1,E+03 | 1,E+05 | 1,E+07 | 1,E+09

Nodes

Event flooding | TERA

- Experiments show how the system scales with respect to:
  - Number
  - Number
  - Event pu
  - Number
- (reference figure is given by a simple event flooding approach)



TERA