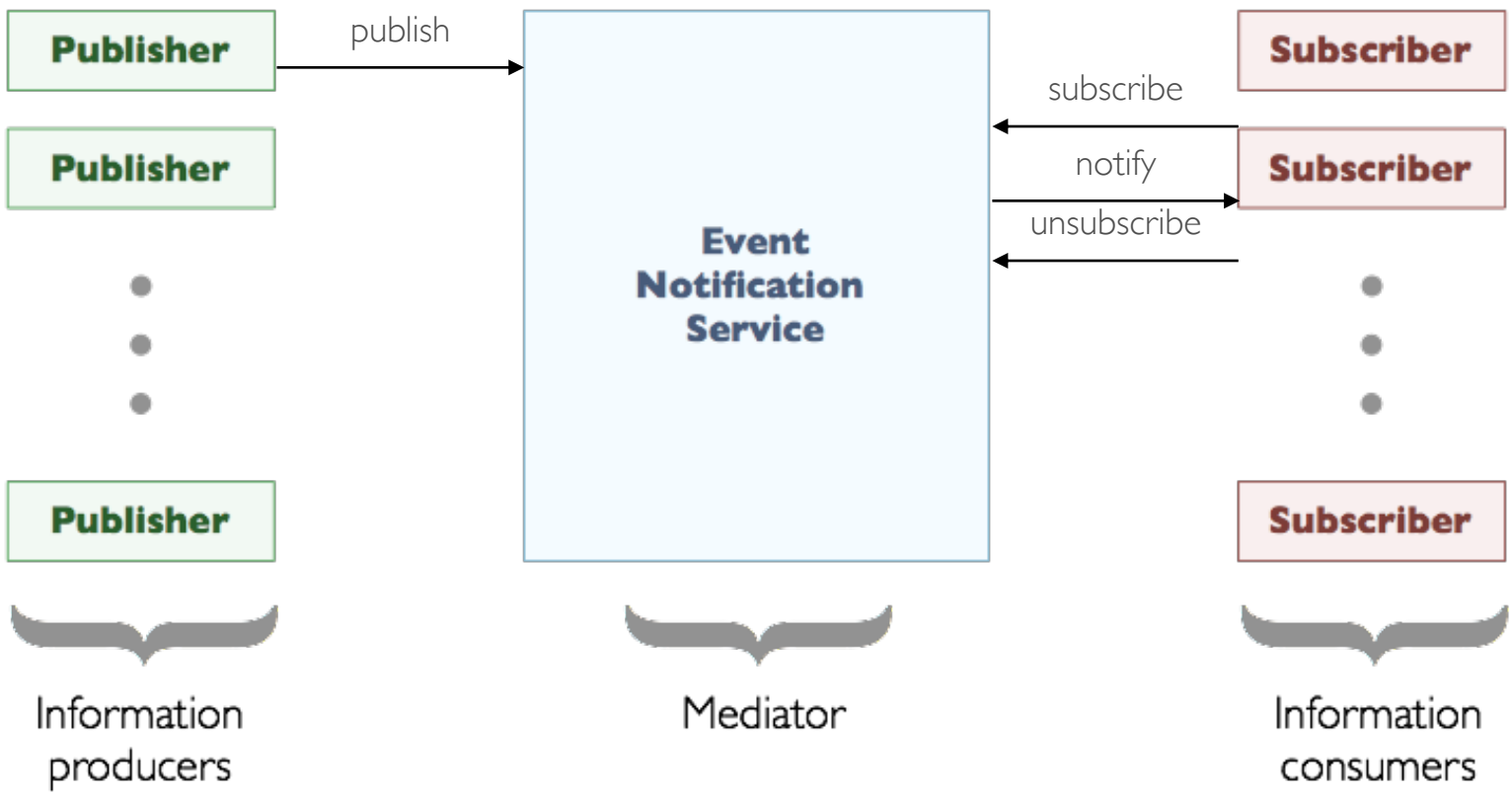
Università di Roma "La Sapienza"
Dipartimento di Informatica e Sistemistica

Information Dissemination in large scale systems: Publish/Subscribe

Schema

Credits: slides provided by Prof. Leonardo Querzoni and Prof. Roberto Baldoni

- 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.



- 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.

- 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.

- Example: 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

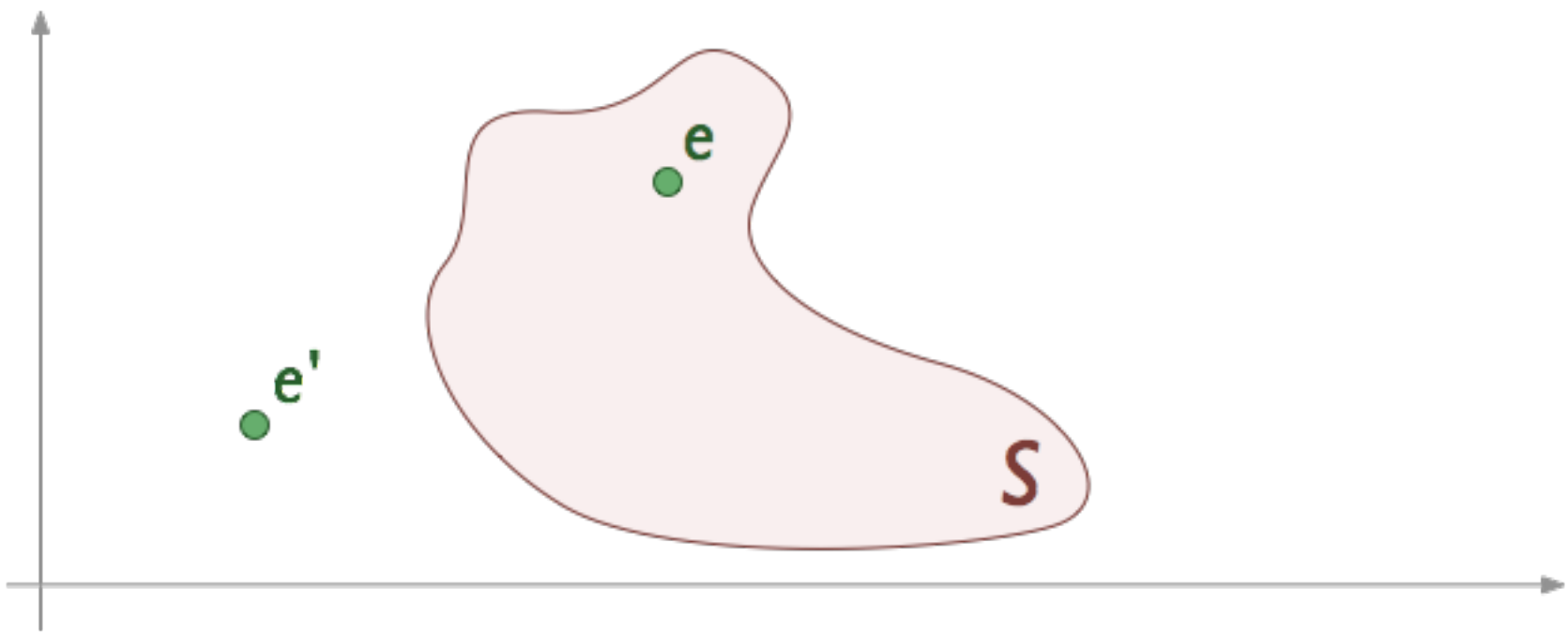
Event Schema

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.



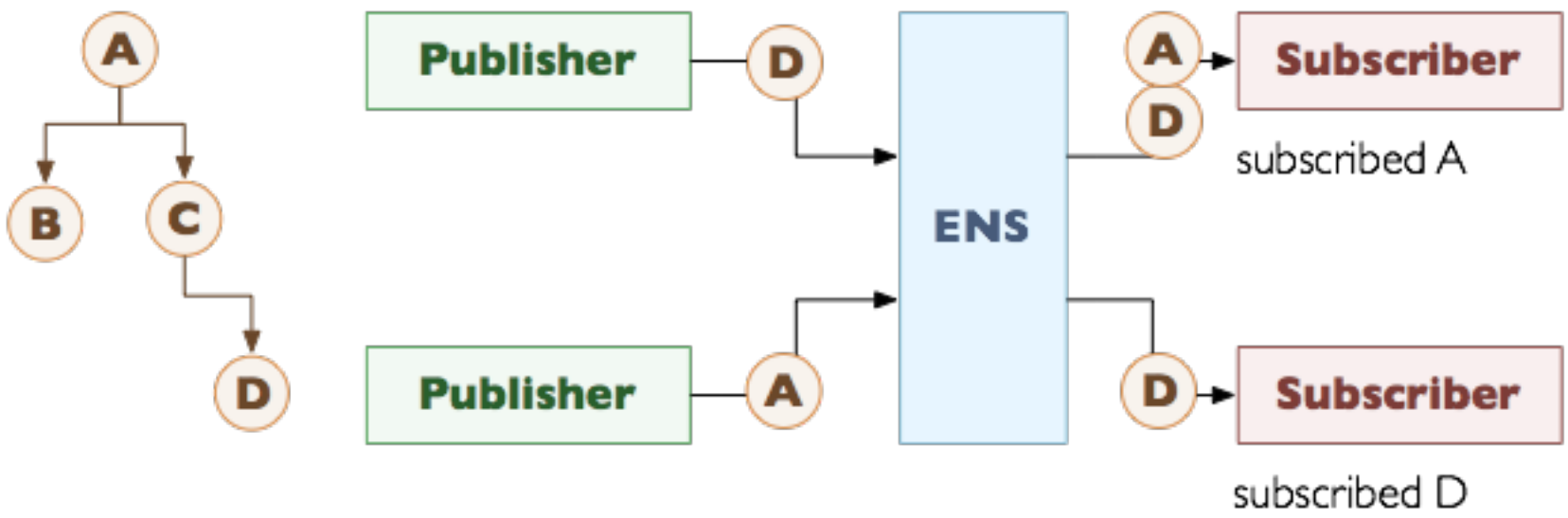
- Depending on the subscription model used we distinguish various flavors of publish/subscribe:
  - Topic-based
  - Hierarchy-based
  - Content-based
  - Type-based
  - Concept-based
  - XML-based
  - .....

- **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.

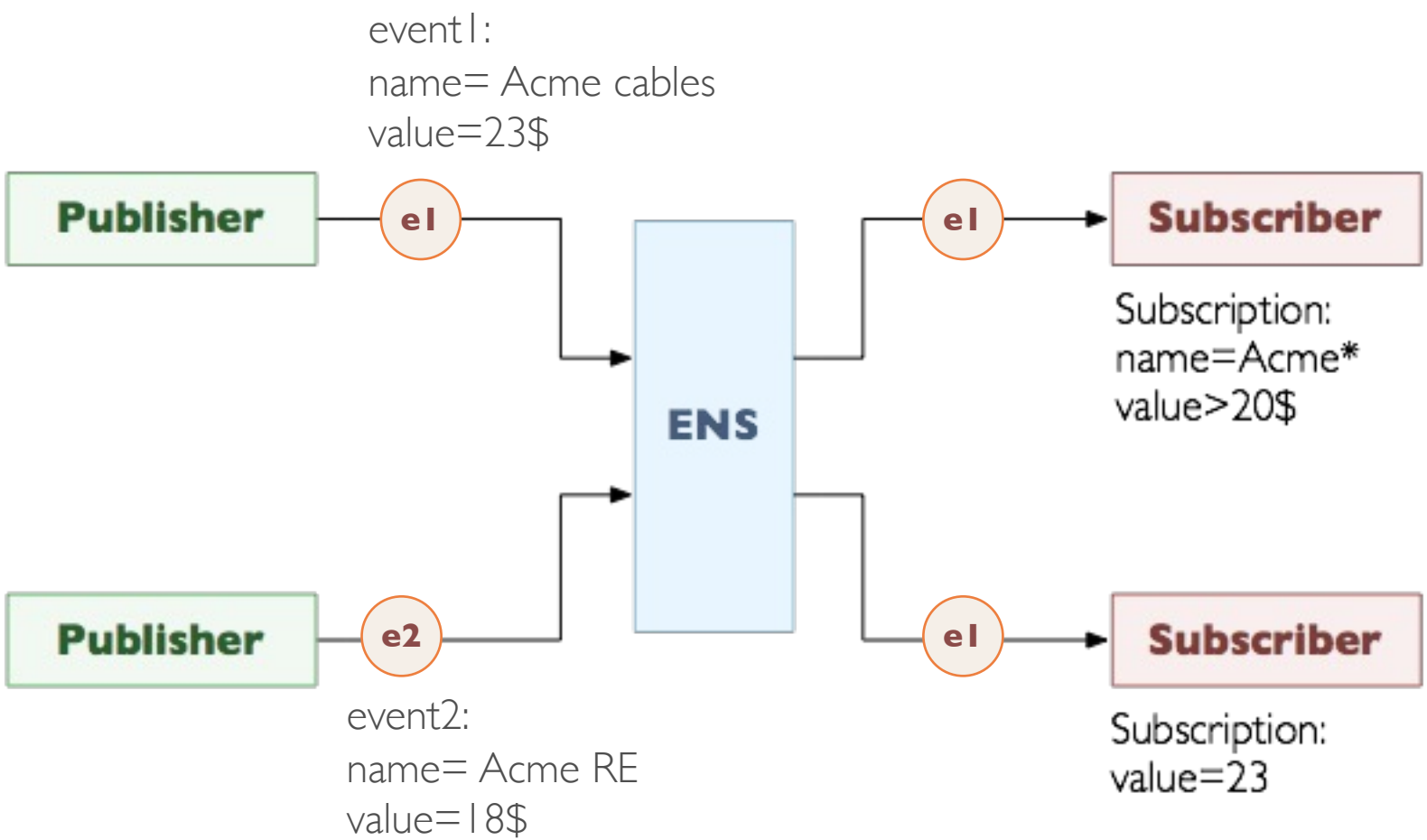
- A topic can be thus represented as a "virtual channel" connecting producers to consumers. For this reason the problem of data distribution in topic-based publish/subscribe systems is considered quite close to group communications.

- **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.



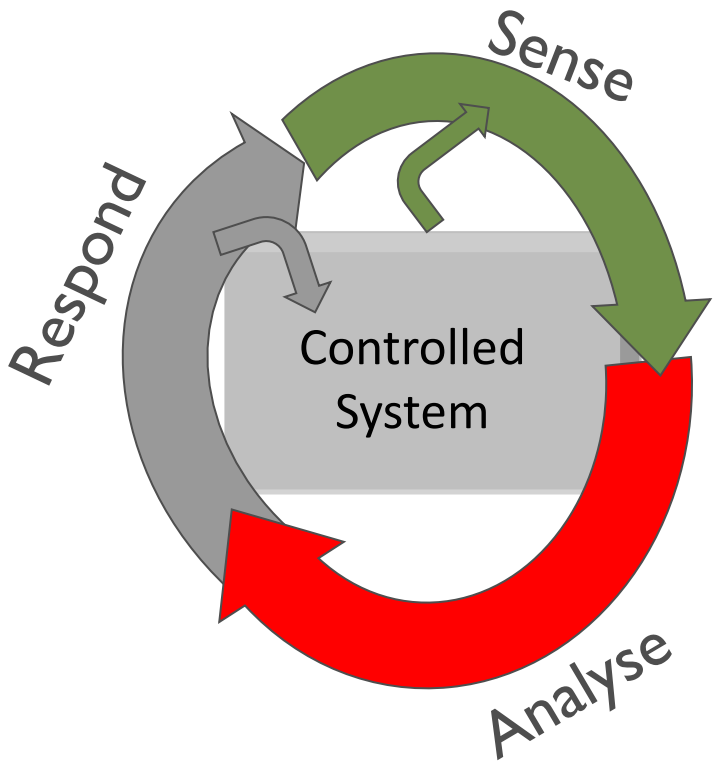
- **Content-based selection**: all the data published in the system is mostly structured. Each subscription can be expressed as a conjunction of constrains expressed on attributes. The Event Notification Service filters out useless events before notifying a subscriber.



Where Publish subscribe is used

- Transport component of EDA architectures
- Data Distribution Services
- Transport component of context aware distributed applications

- **EDA characteristics:** three principal building blocks
  - **Sense:** The sensing block gathers data from within and outside the controlled system
  - **Analyse**: Data are analyzed
  - **Respond**: Analysis used to determine whether appropriate actions are to be timely undertaken in response to what has been sensed (respond block)



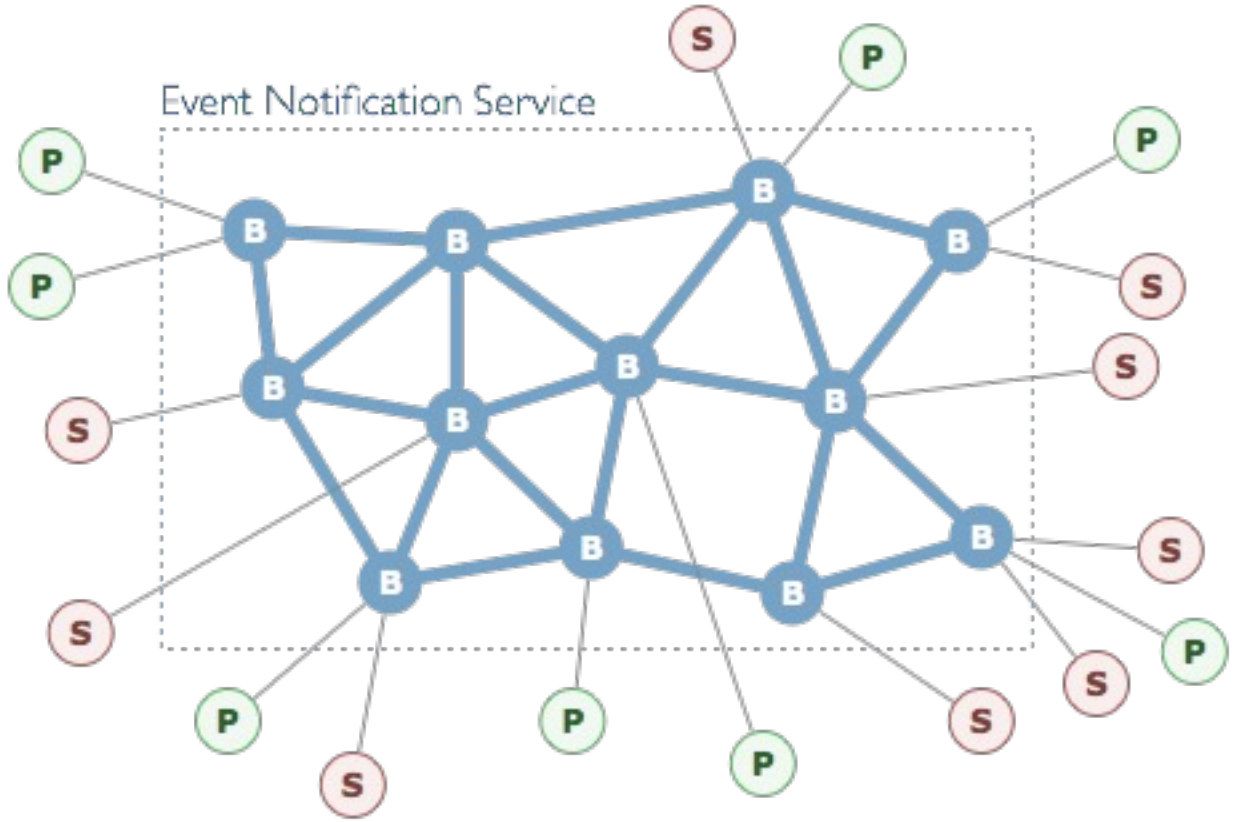
- A control loop is enabled
  - The sensing part obtains data that defines the "real" state
  - The analysis part correlates data in order to determine the current state
  - When reality deviates from that expected then the respond part acts on the system (e.g, sends alerts)

- **EDA Architecture**
  - Event Correlation and Processing
  - Event Distribution
    - Publish subscribe help in doing some pre-processing of the events as well as reducing the network load

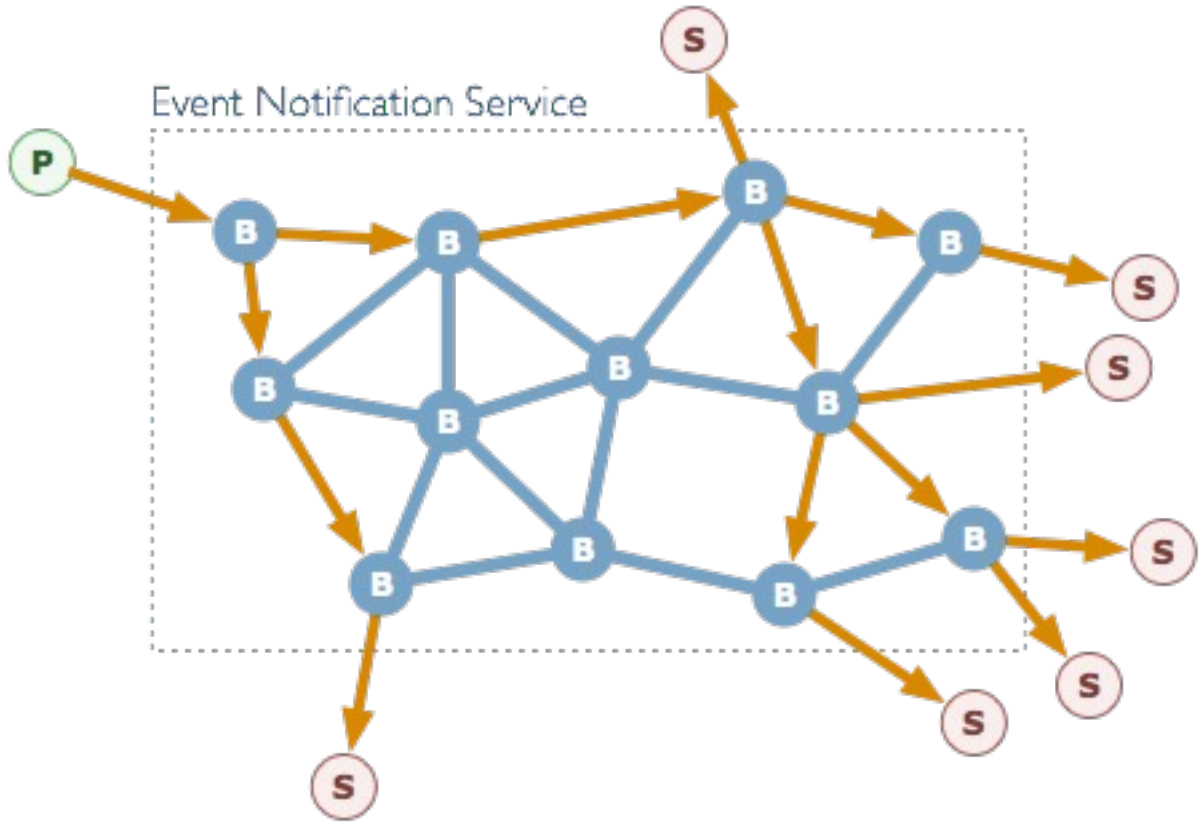
Complex Event Processing

Event Dissemination (Publish-Subscribe)

- 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.

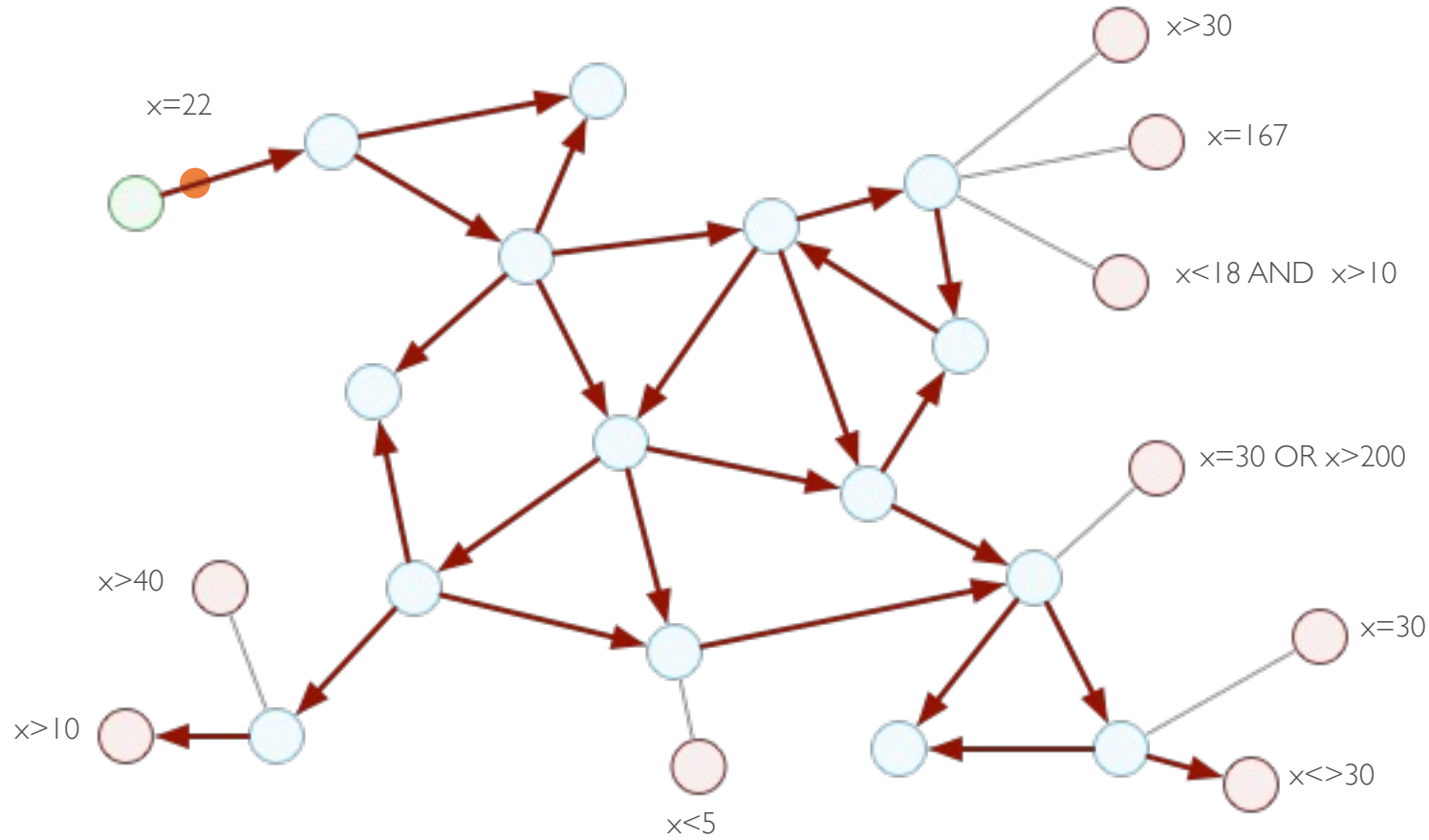


- Modern ENSs are implemented through a set of processes, called *event brokers,* forming an overlay network.
- Each client (publisher or subscriber) accesses the service through a broker that masks the system complexity.

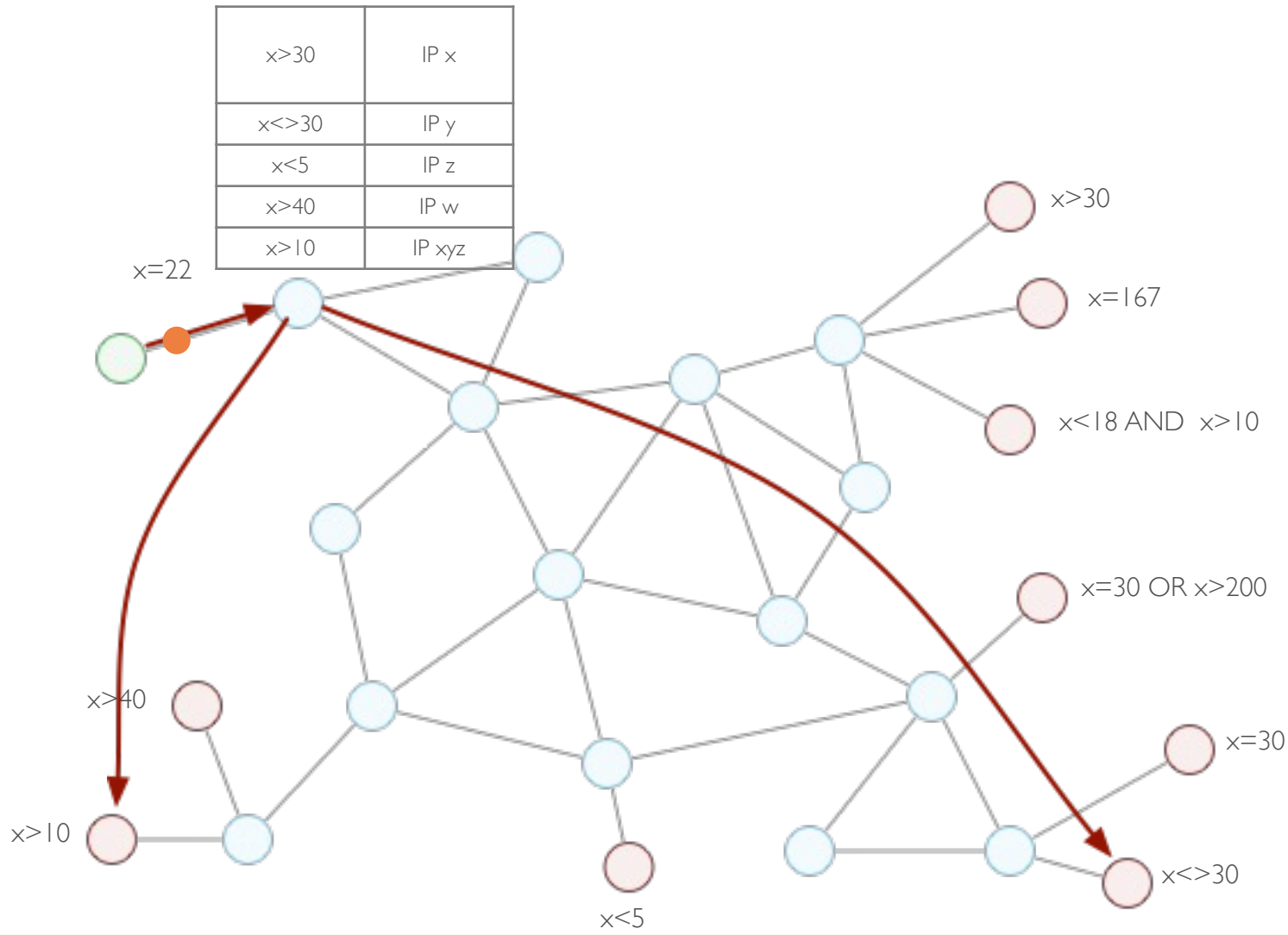


- An **event routing** mechanism routes each event inside the ENS from the broker where it is published to the broker(s) where it must be notified.

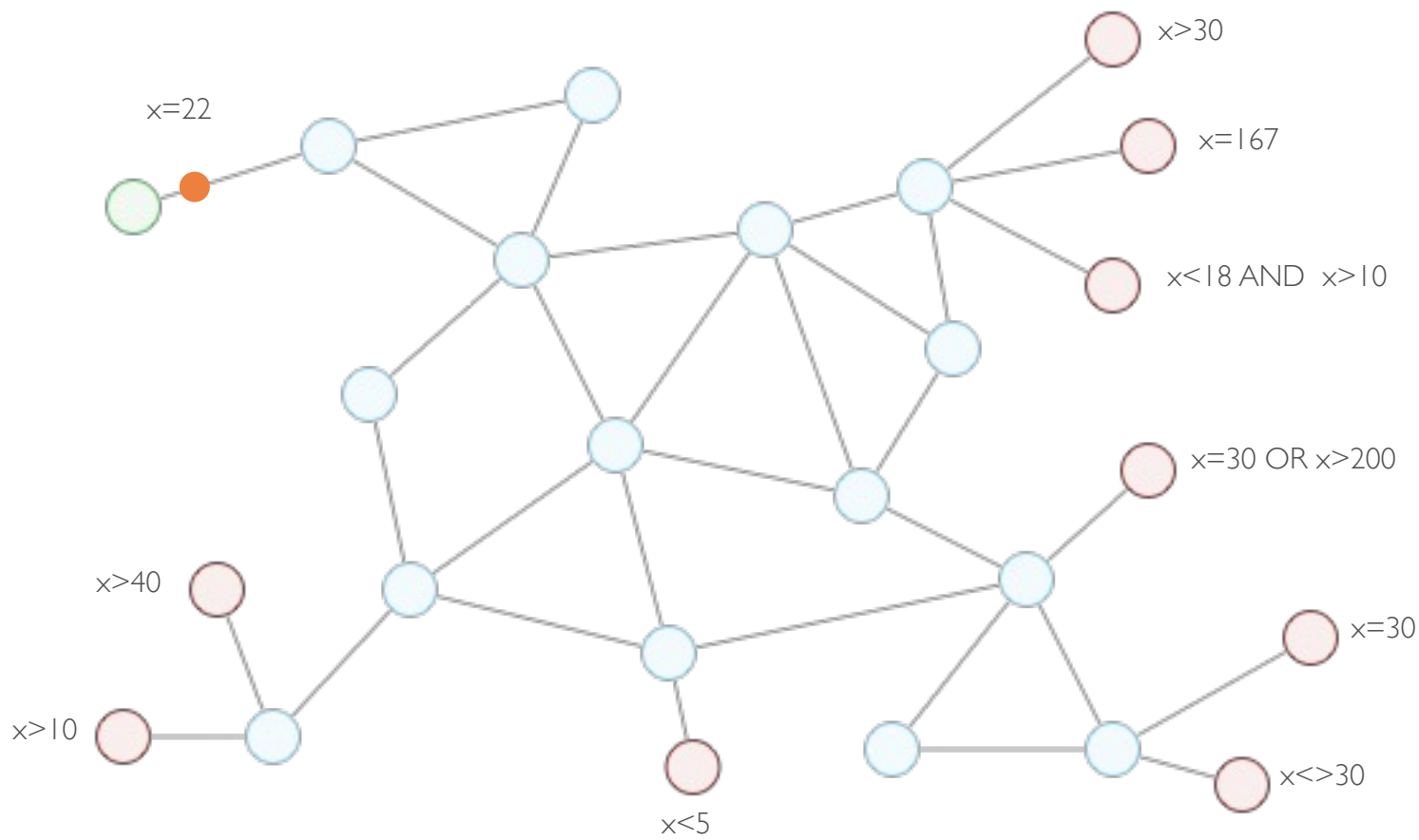
- **Event flooding**: each event is broadcast from the publisher in the whole system.
- The implementation is straightforward but very expensive.
- This solution has the highest message overhead with no memory overhead.



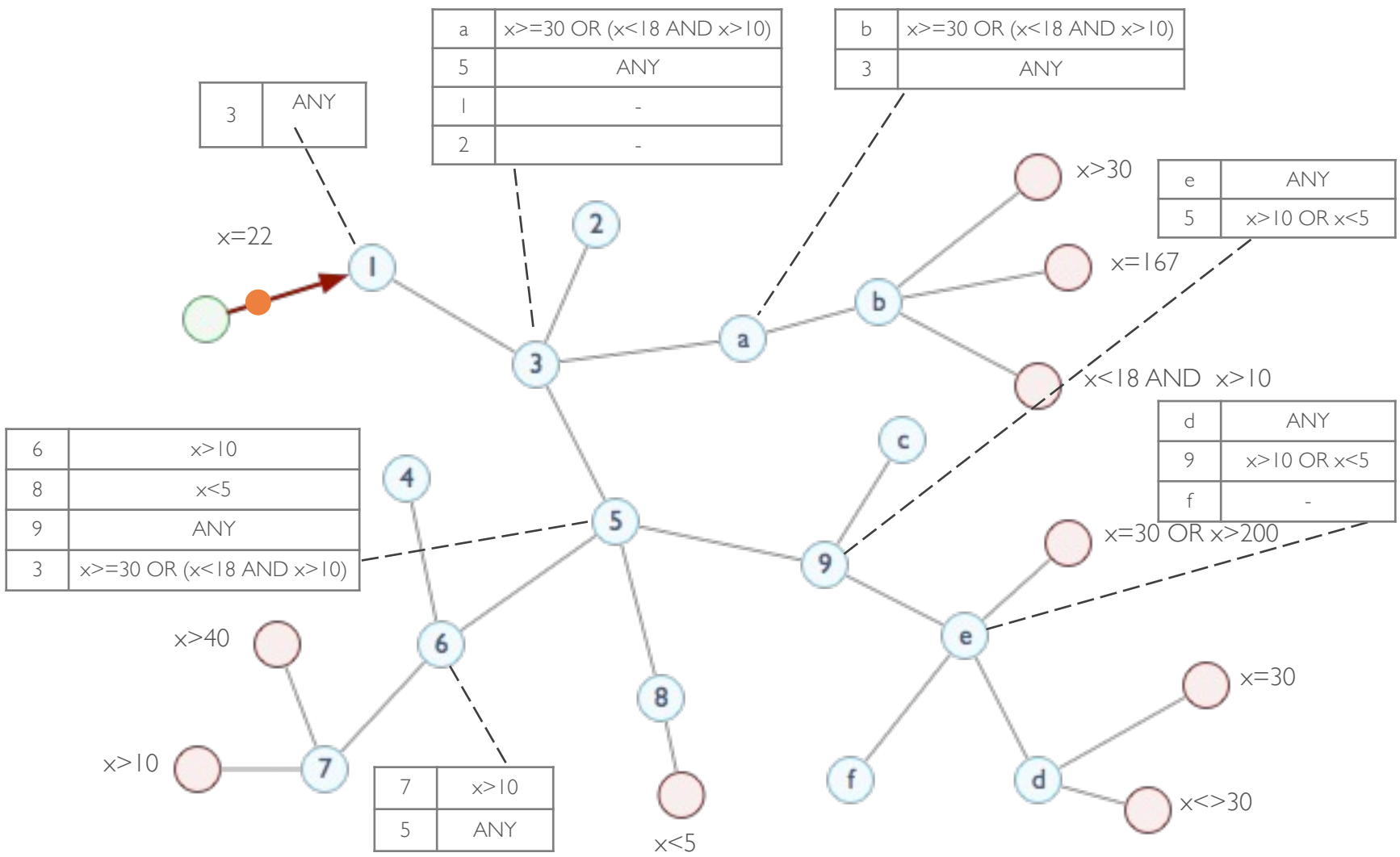
■**Subscription flooding**: each subscription is copied on every broker, in order to build locally complete subscription tables. These tables are then used to locally match events and directly notify interested subscribers. This approach suffers from a large memory overhead, but event diffusion is optimal.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



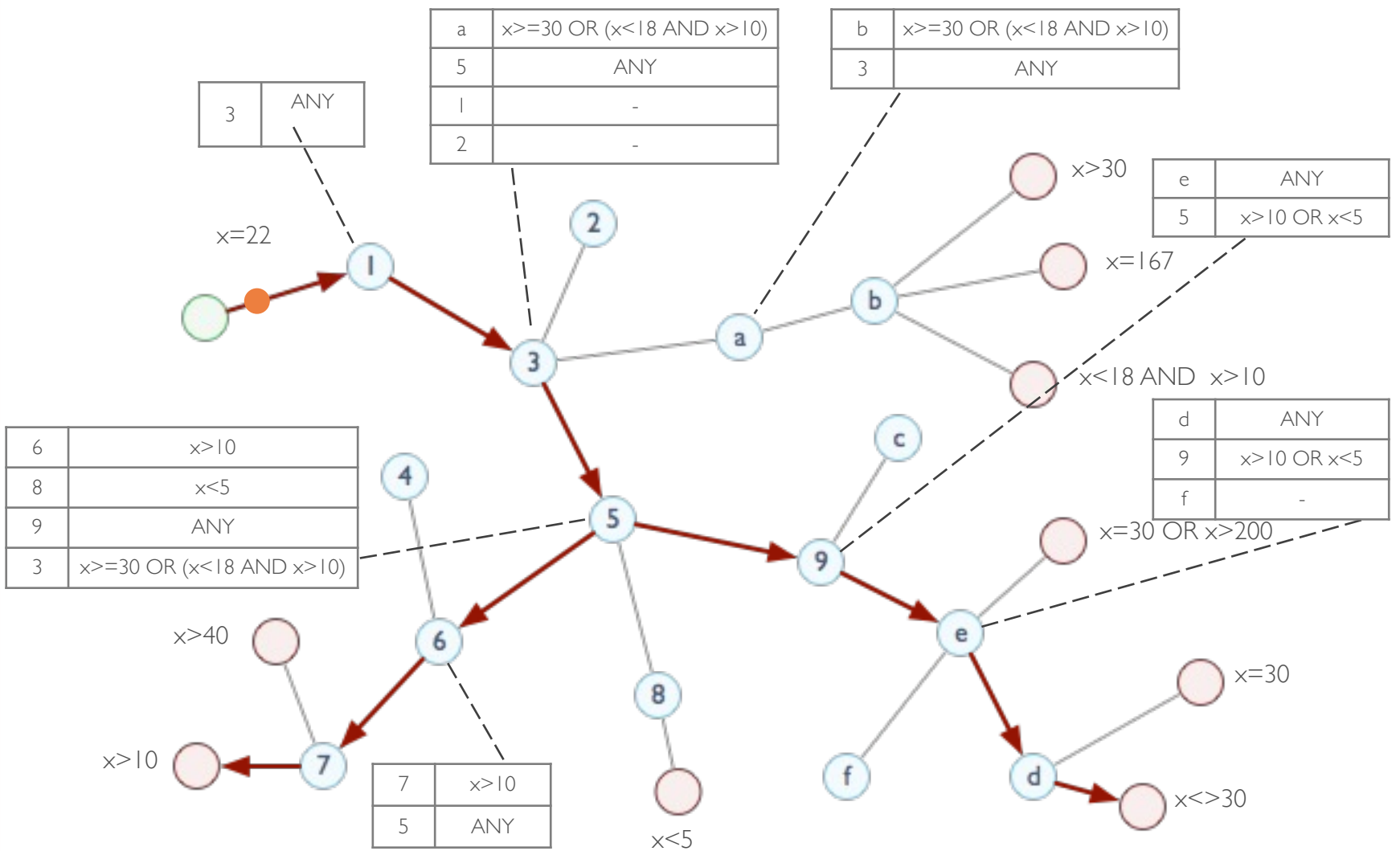
Filter-based routing: subscriptions are partially diffused in the system and used to build *routing tables. These tables, are then exploited during event diffusion to dynamically build a multicast tree that (hopefully) connects the publisher to all, and only, the interested subscribers.*

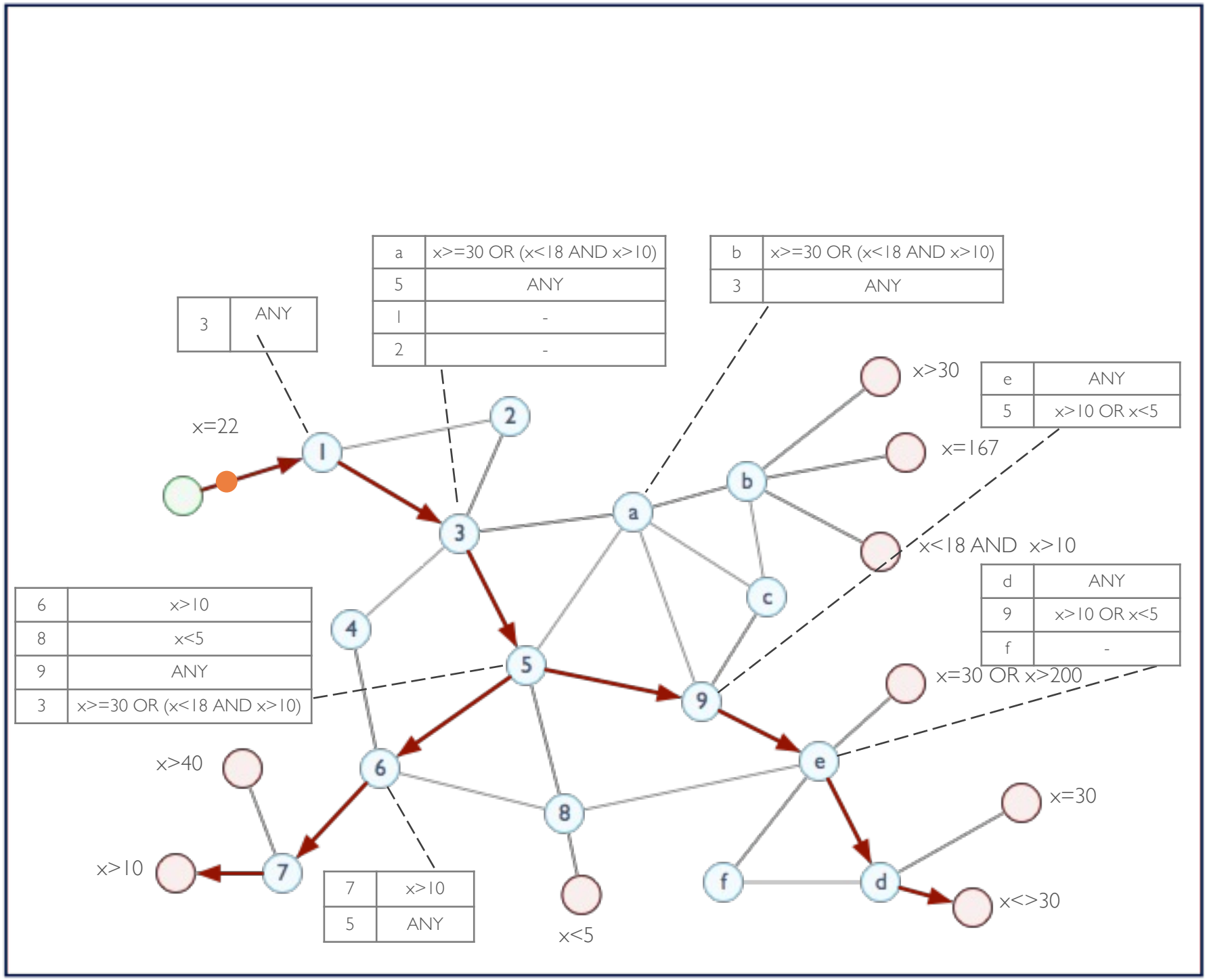


Filter-based routing: subscriptions are partially diffused in the system and used to build *routing tables. These tables, are then exploited during event diffusion to dynamically build a multicast tree that (hopefully) connects the publisher to all, and only, the interested subscribers.*



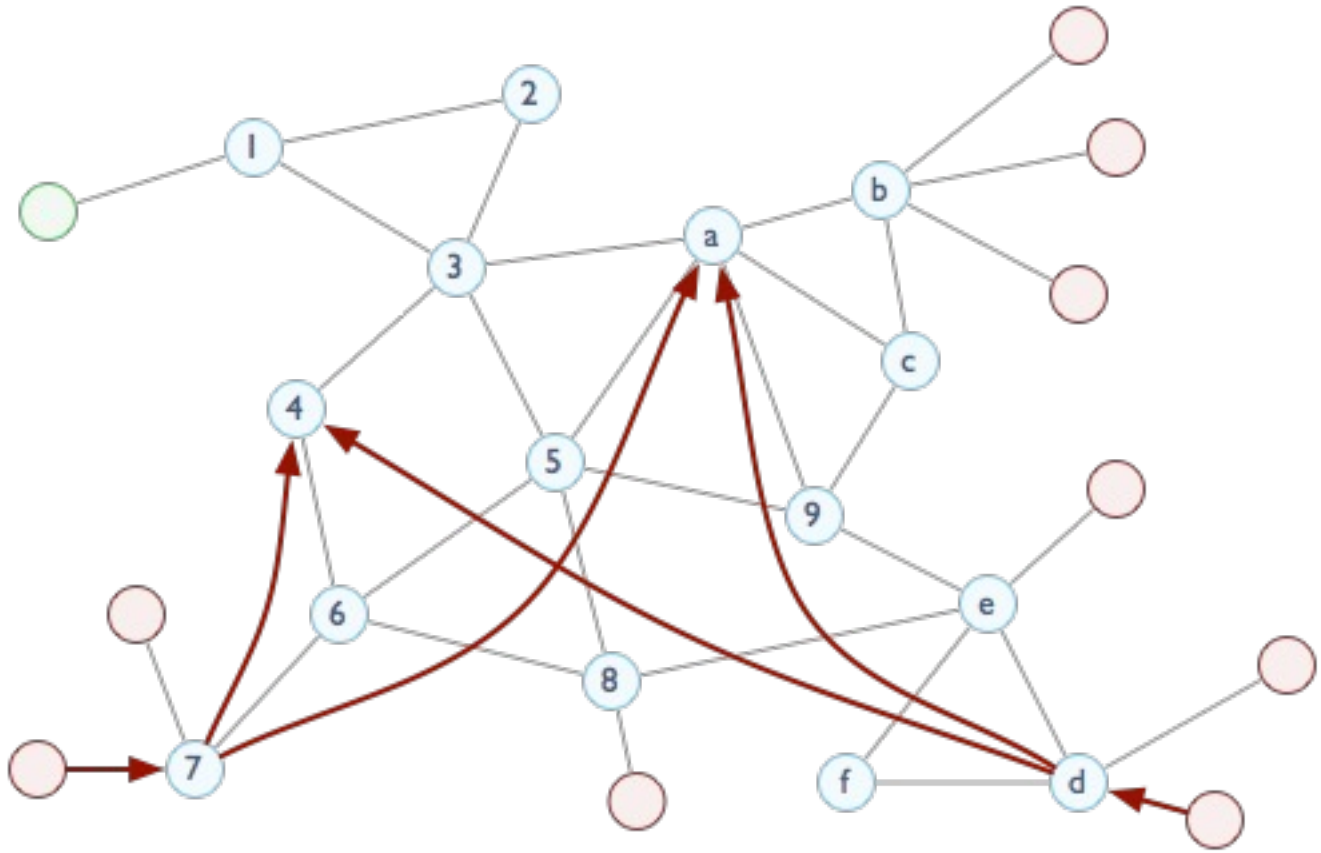
Filter-based routing: subscriptions are partially diffused in the system and used to build *routing tables. These tables, are then exploited during event diffusion to dynamically build a multicast tree that (hopefully) connects the publisher to all, and only, the interested subscribers.*





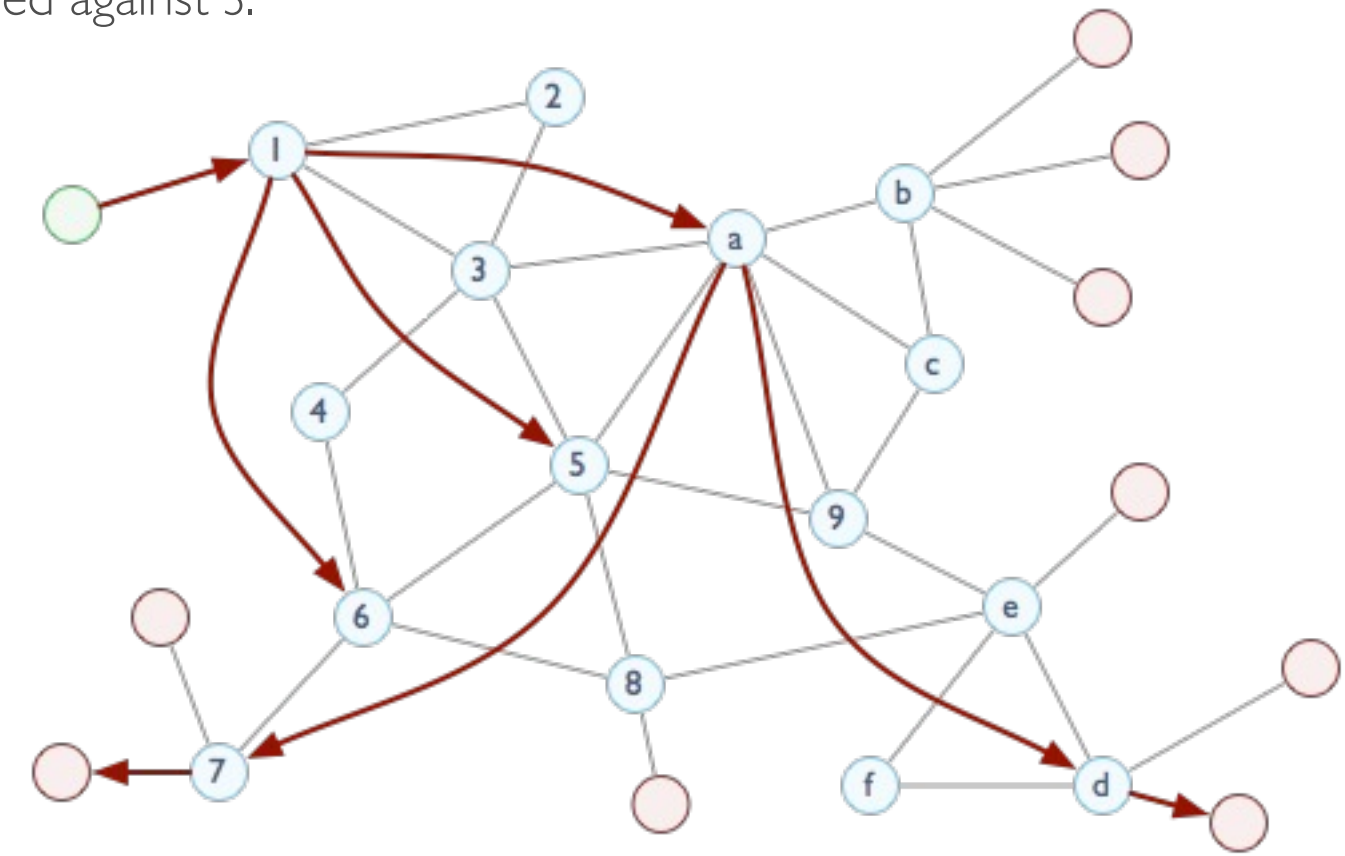
- **Rendez-Vous routing**: it is based on two functions, namely *SN* and *EN*, used to associate respectively subscriptions and events to brokers in the system.
- Given a subscription *s*, *SN(s)* returns a set of nodes which are responsible for storing *s* and forwarding received events matching *s* 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*.



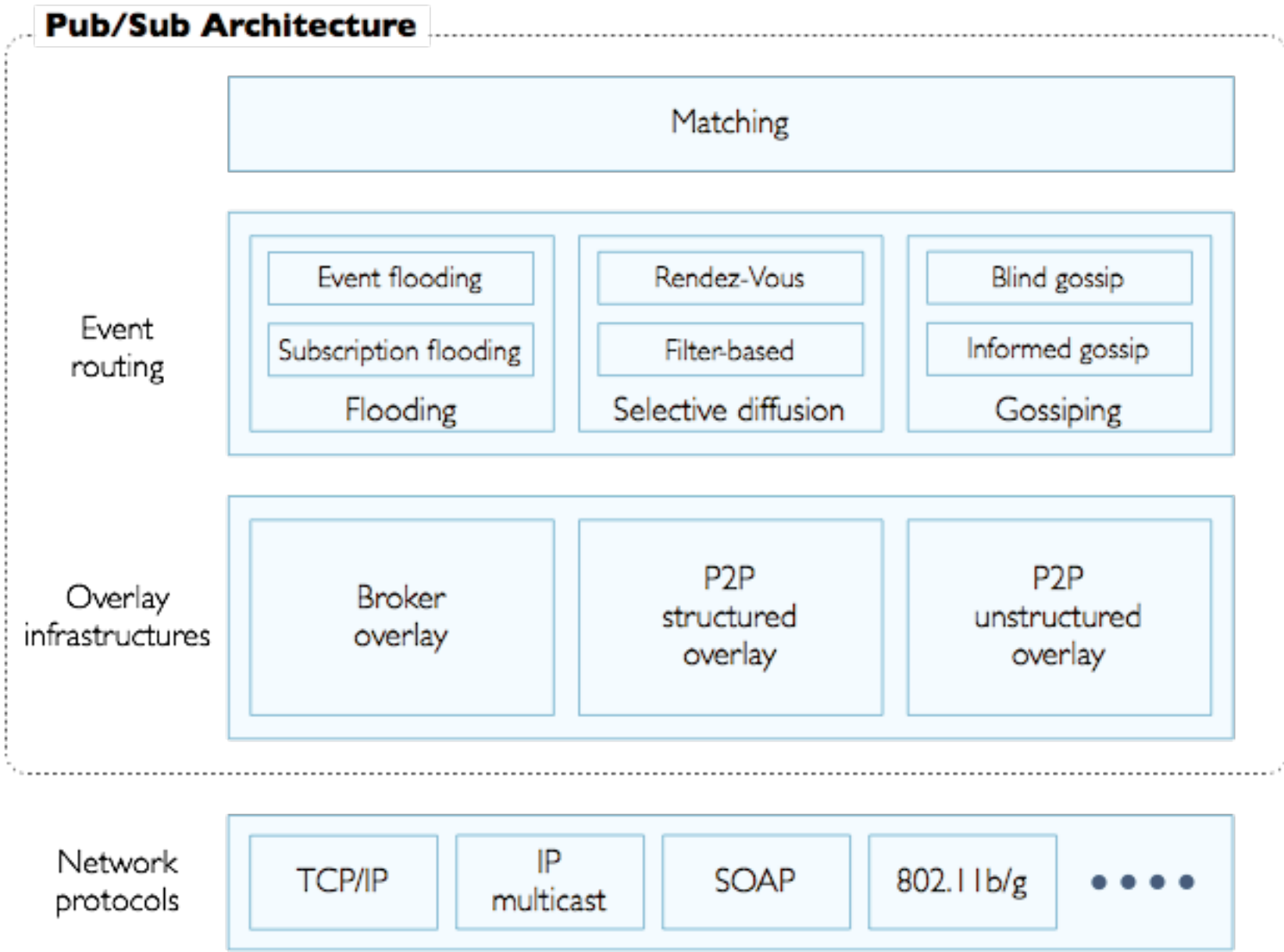
- SN(S) = {4,a}

- **Rendez-Vous routing**: example.
- Phase 1I: 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*.

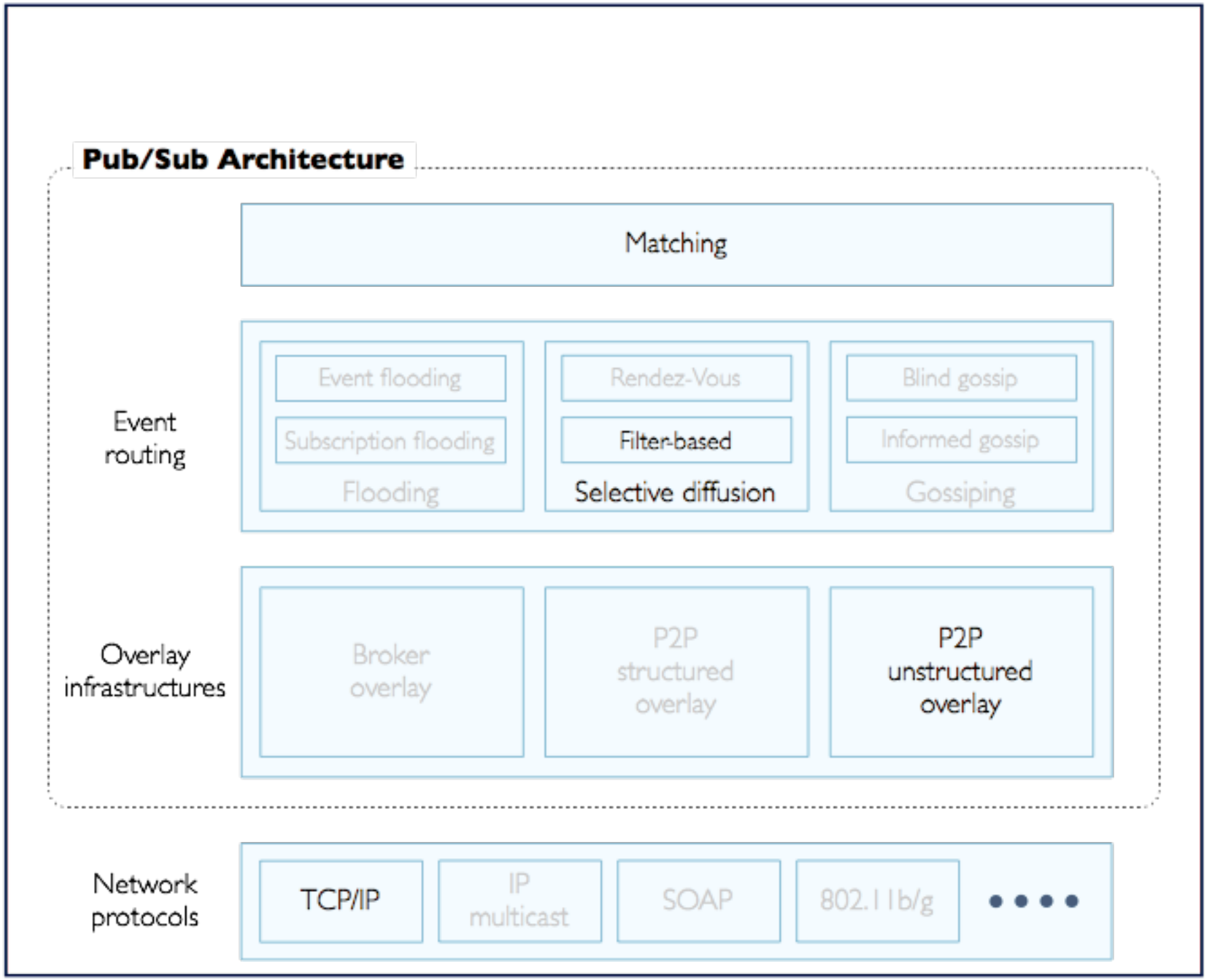
- A generic architecture of a publish/subscribe system:



From "Distributed Event Routing in Publish/Subscribe Communication Systems: a survey" R.Baldoni, L. Querzoni, S. Takoma, A. Virgillito midlab tech.rep. 2007, to appear (springer)

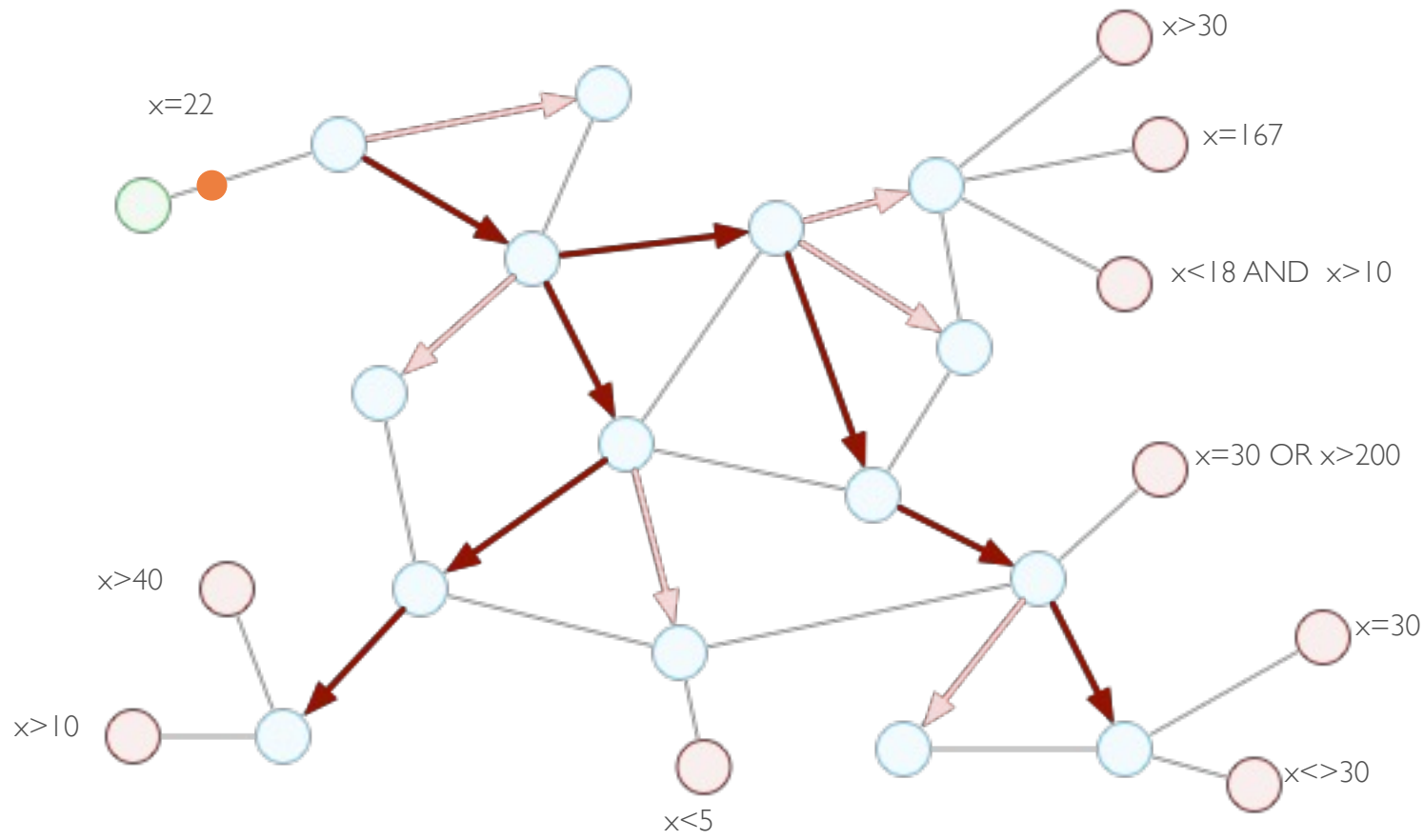
Siena

Antonio Carzaniga, Matthew J. Rutherford, Alexander J. Wolf "A Routing Scheme for Content-Based Networking" INFOCOM 2004



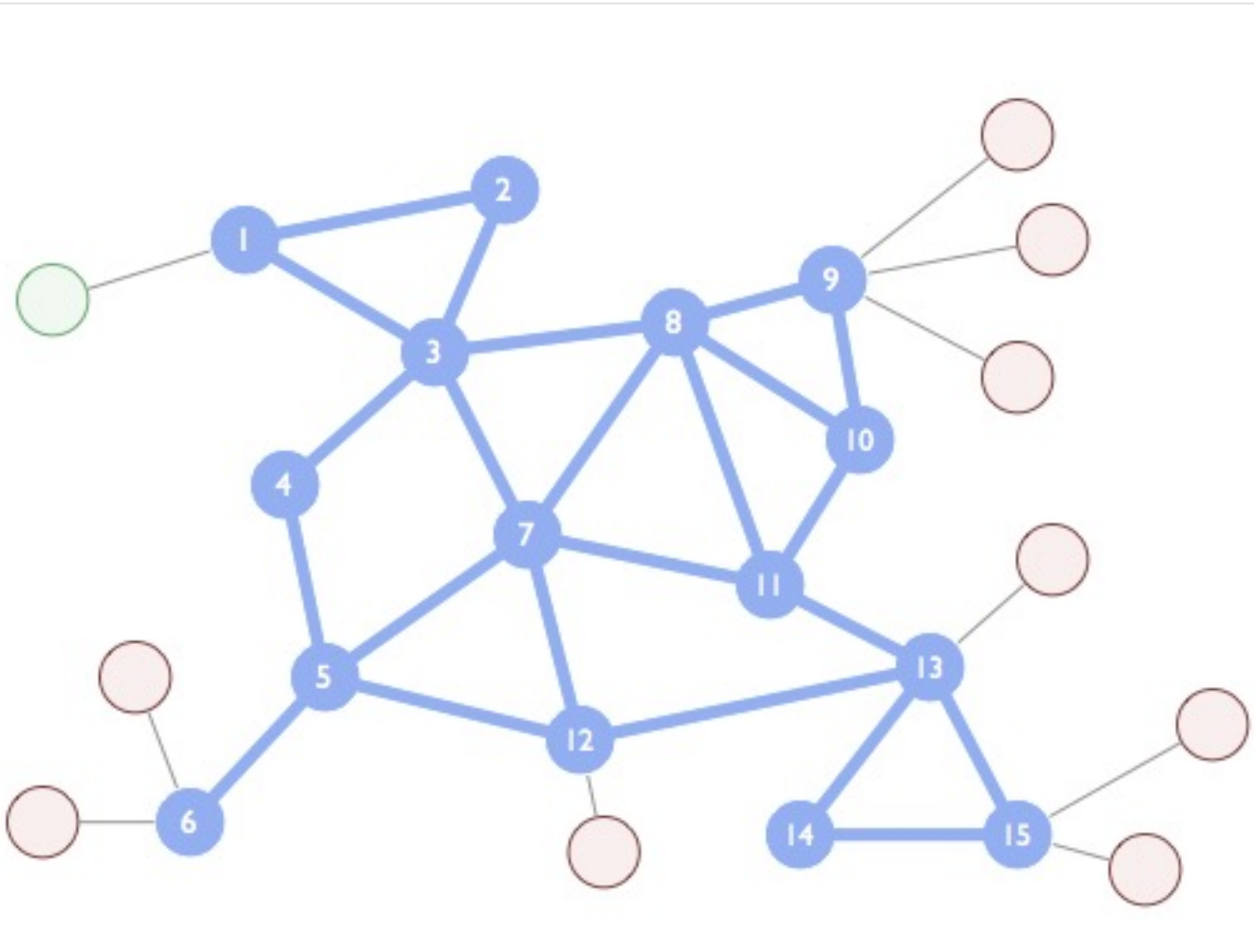
- Each node has a service interface consisting of two operations:
  - send_message(m)
  - set_predicate(p)
- A predicate is a disjunction of conjunctions of constraints of individual attributes.
- A content-based network can be seen as a dynamically-configurable broadcast network, where each message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

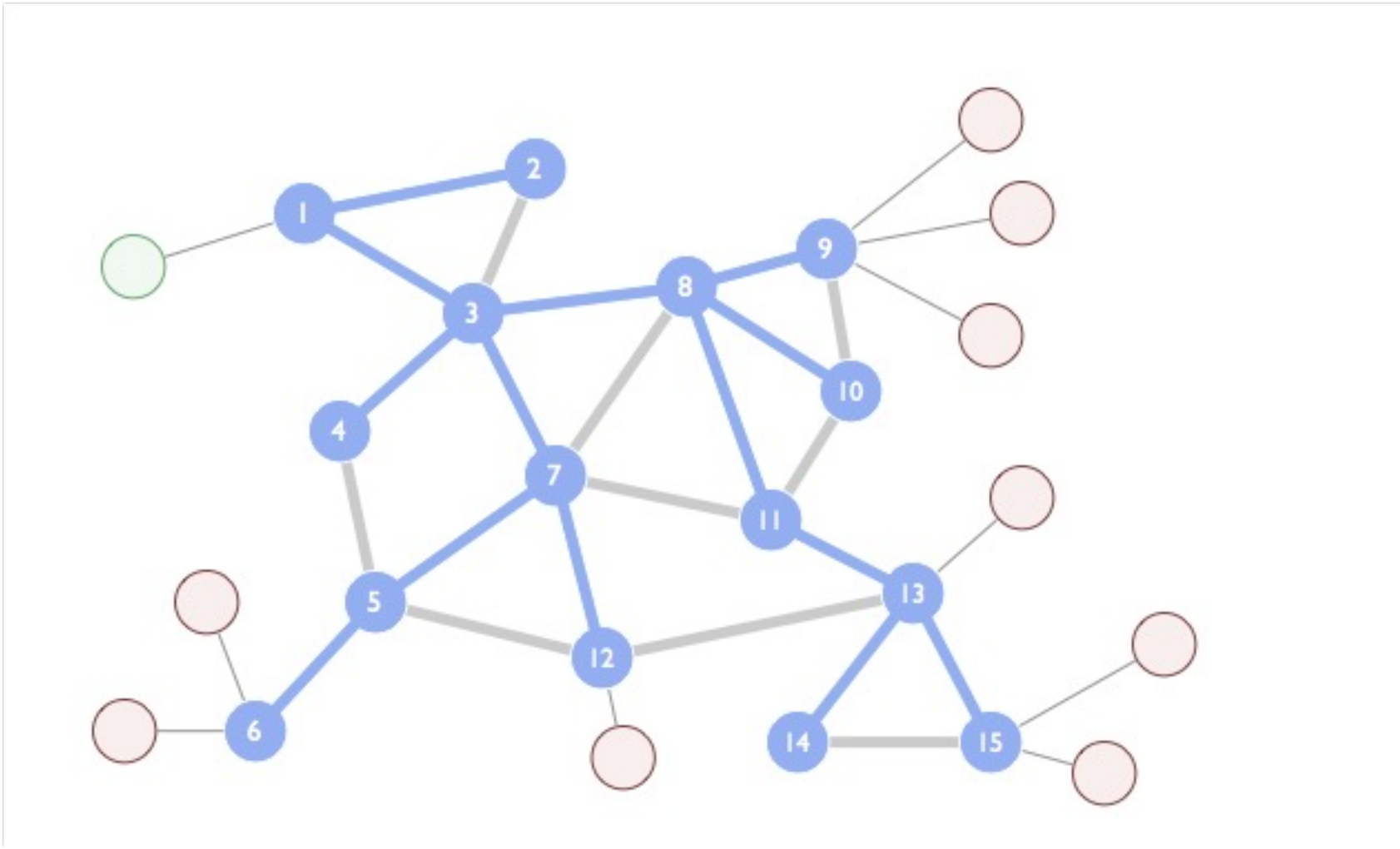


- The broadcast layer:
  - A *broadcast function* $B : N \times I \rightarrow I^*$ is available at each router. Given a source node s and an input interface i, it returns a set of output interfaces.
  - The broadcast function defines a broadcast tree routed at each source node.
  - The broadcast function 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.

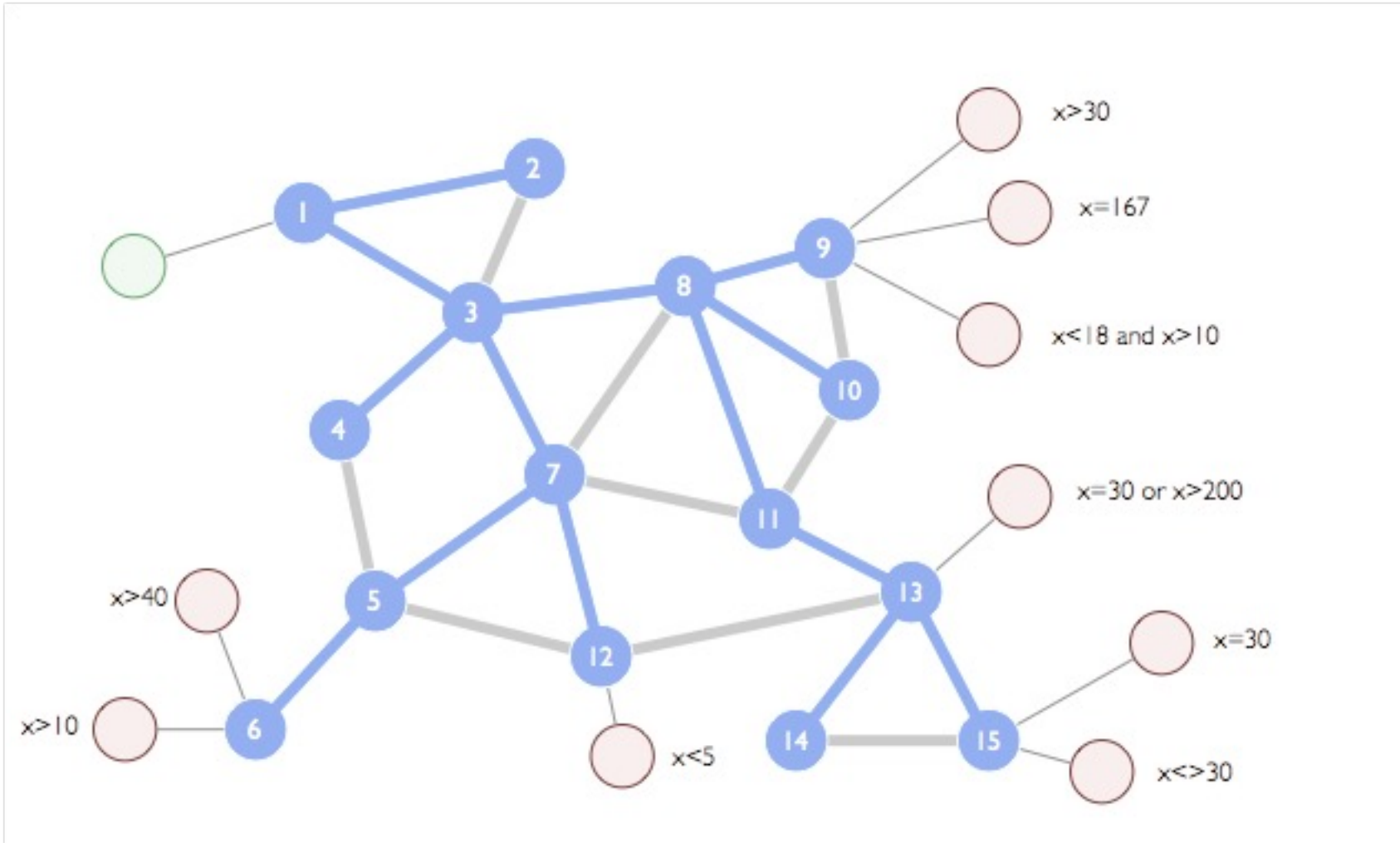
- Example:



- Example:

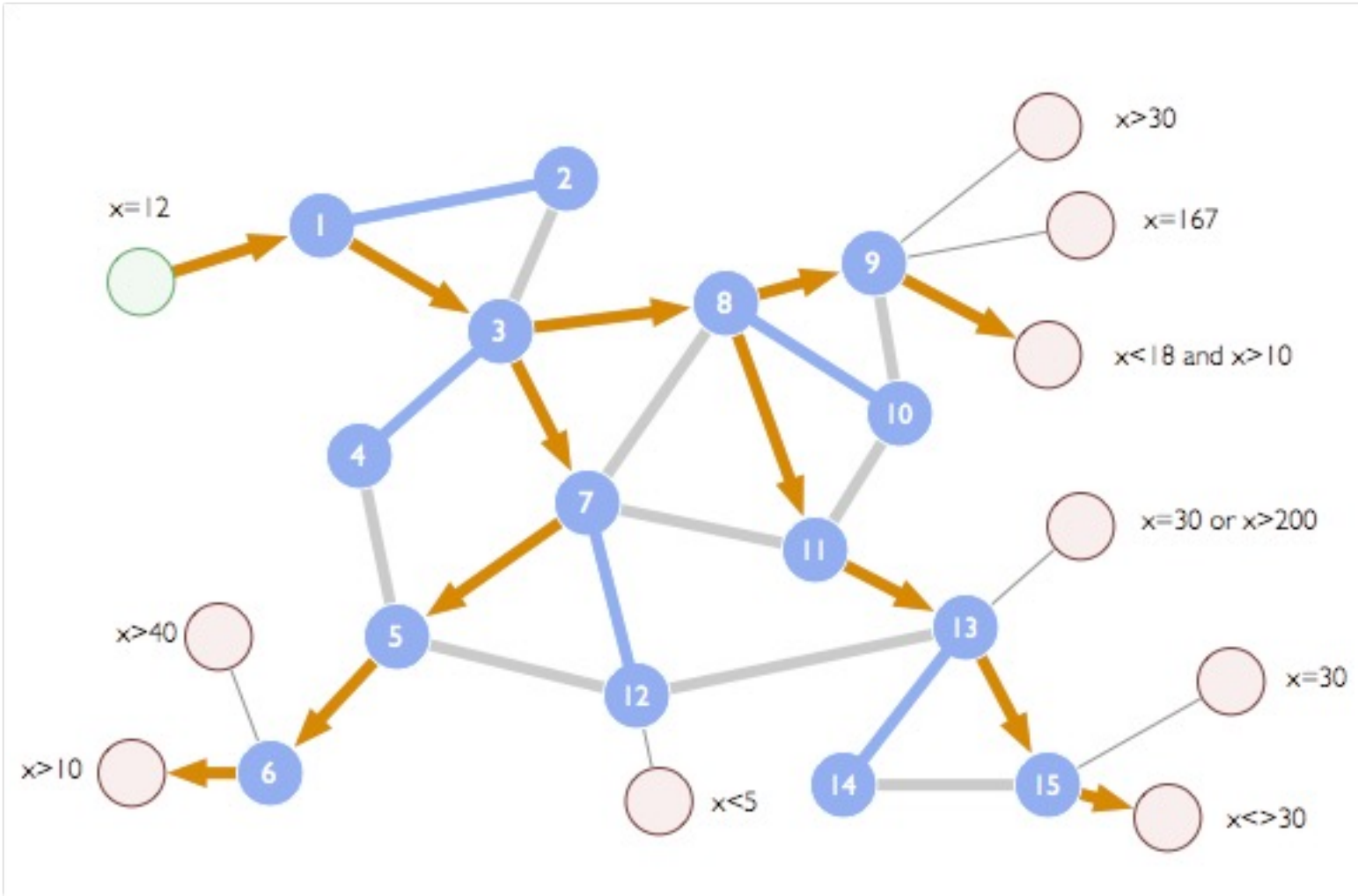


- The content-based layer:
  - Maintains forwarding state in the form of a *content-based forwarding table*. The table, for each node, associates a content-based address to each interface.



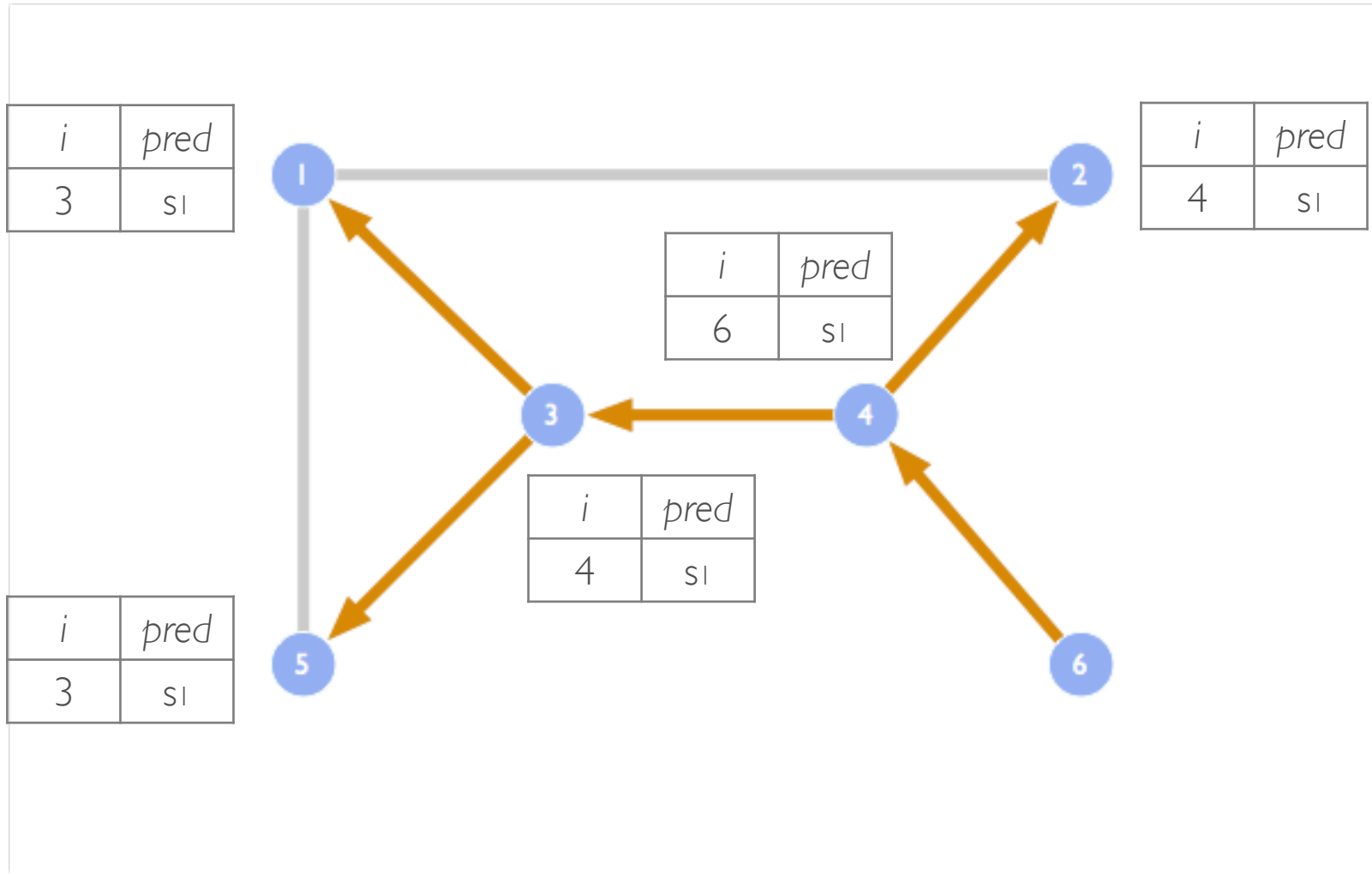
- The message forwarding mechanism:
  - The content-based forwarding table is used by a forwarding function F_c that, given a message *m*, selects the subset of interfaces associated with predicates matching *m*.
  - The result of F_c is then combined with the broadcast function *B*, computed for the original source of *m*.
  - A message is therefore forwarded along the set of interfaces returned by the following formula:
    - $(B(source(m), incoming\_if(m)) \cup \{l_0\}) \cap F_c(m)$

- Example:

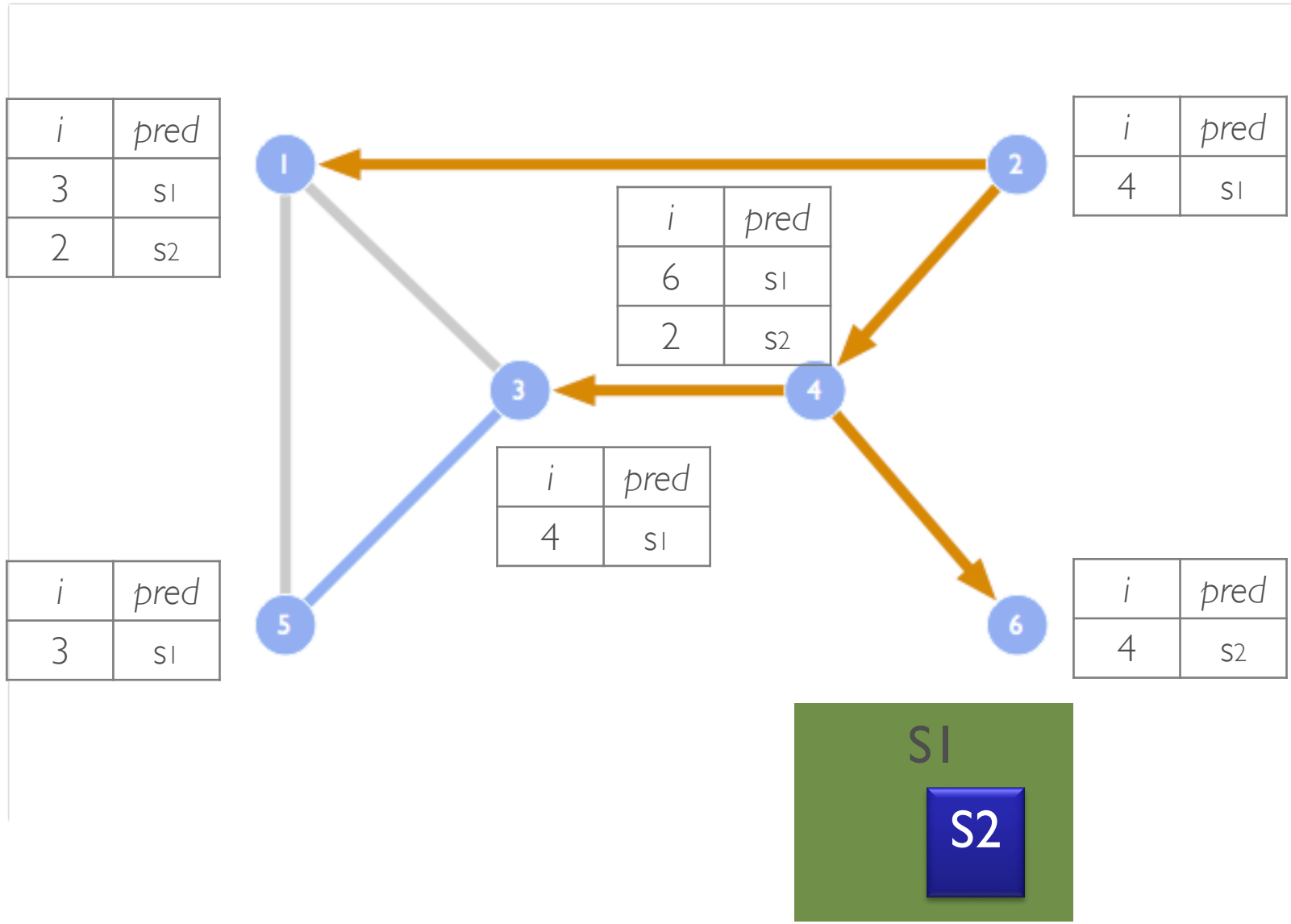


- Forwarding tables maintenance:
  - *Push* mechanism based on *receiver advertisements.*
  - *Pull* mechanism based on *sender requests* and *update replies.*
- *Receiver advertisements:*
  - are issued by nodes periodically and/or when the node changes its local content-based address p_0.
  - *Content-based 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 $p_i \leftarrow p_i \vee p_{RA}$.

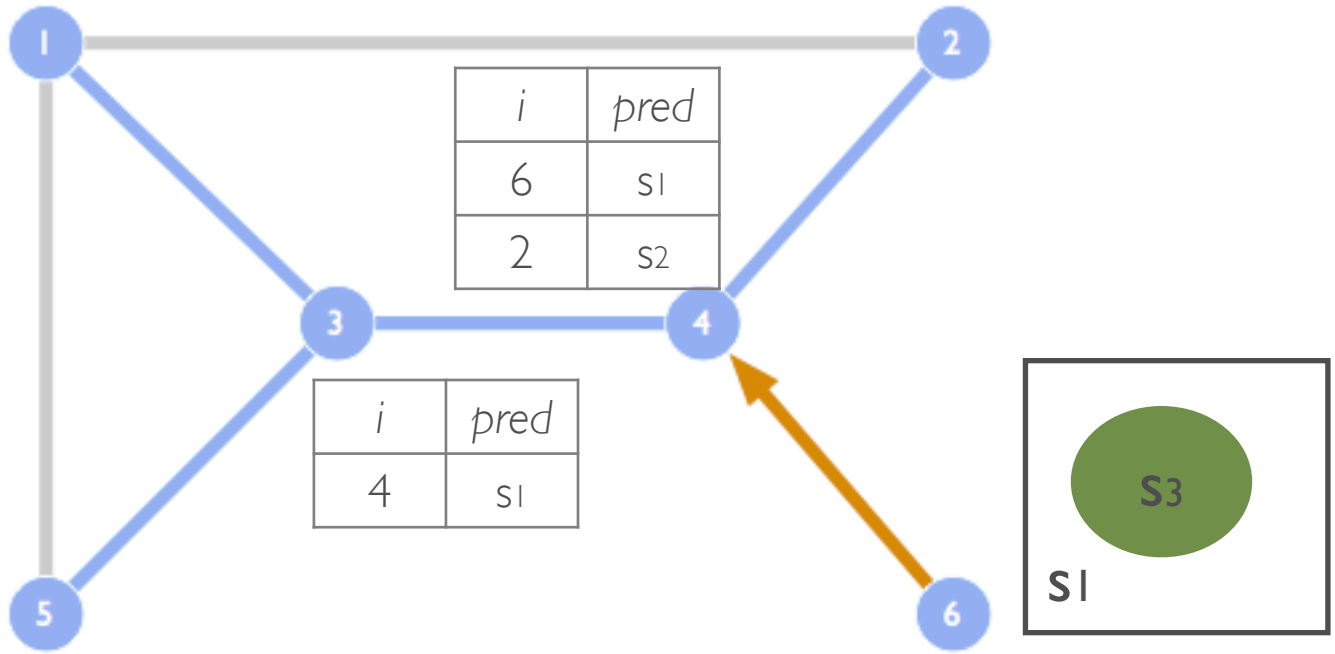
- Example: Broker 6 issues subscription *s1*



- Example: Broker 2 issues subscription $s2 \prec s1$

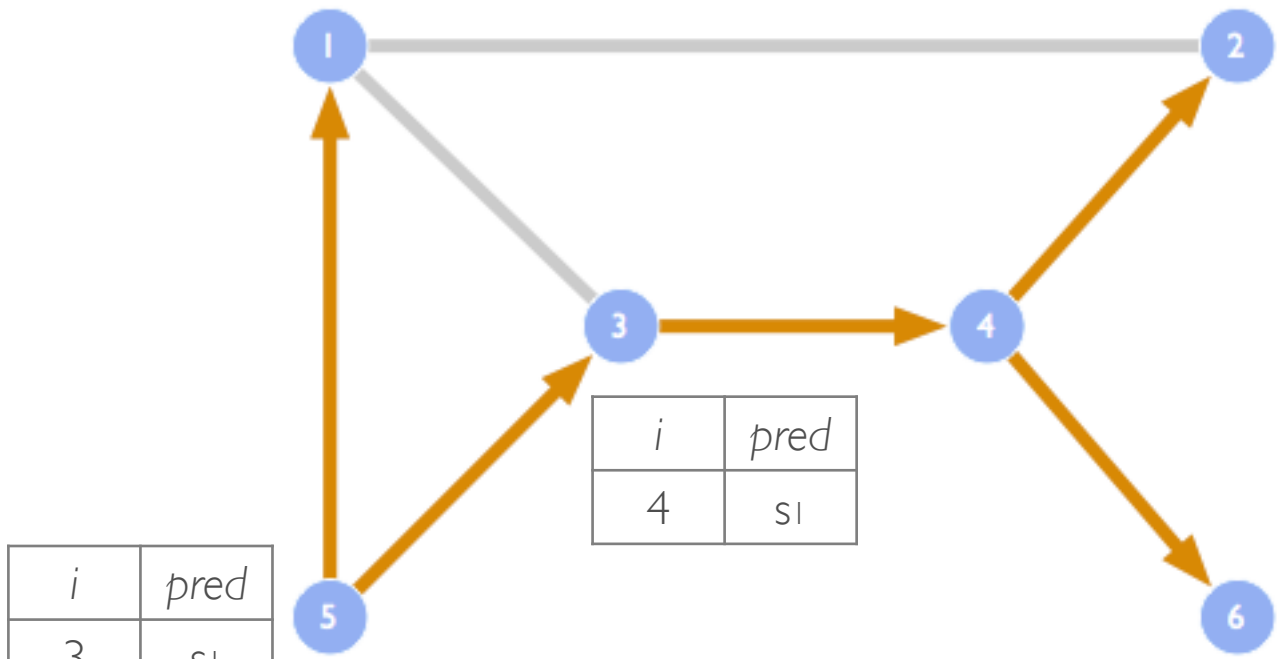


- Notice that, because of the ingress filtering rule, the RA protocol can only widen the selection of the content-based addresses stored in routing tables. In the long run, this may cause an "inflation" of those content-based addresses.
- Example: Broker 6 substitute its predicate with $s3 \prec s1$



- *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 $p_i \leftarrow p_{UR}$.

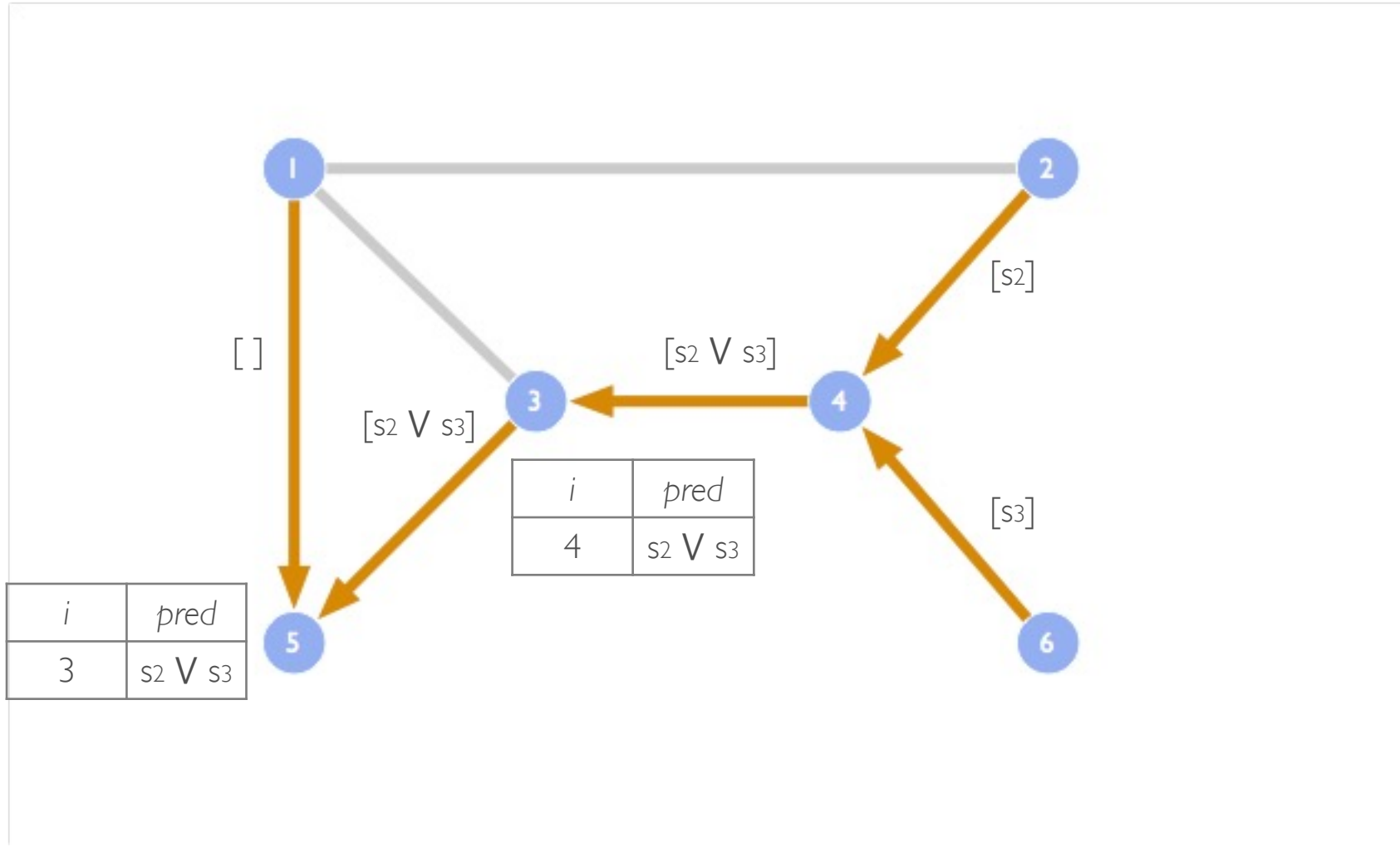
- Example: Broker 5 sends a Sender Request (SR) to refresh its forwarding table.



i	*pred*
4	sI

i	*pred*
3	sI

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

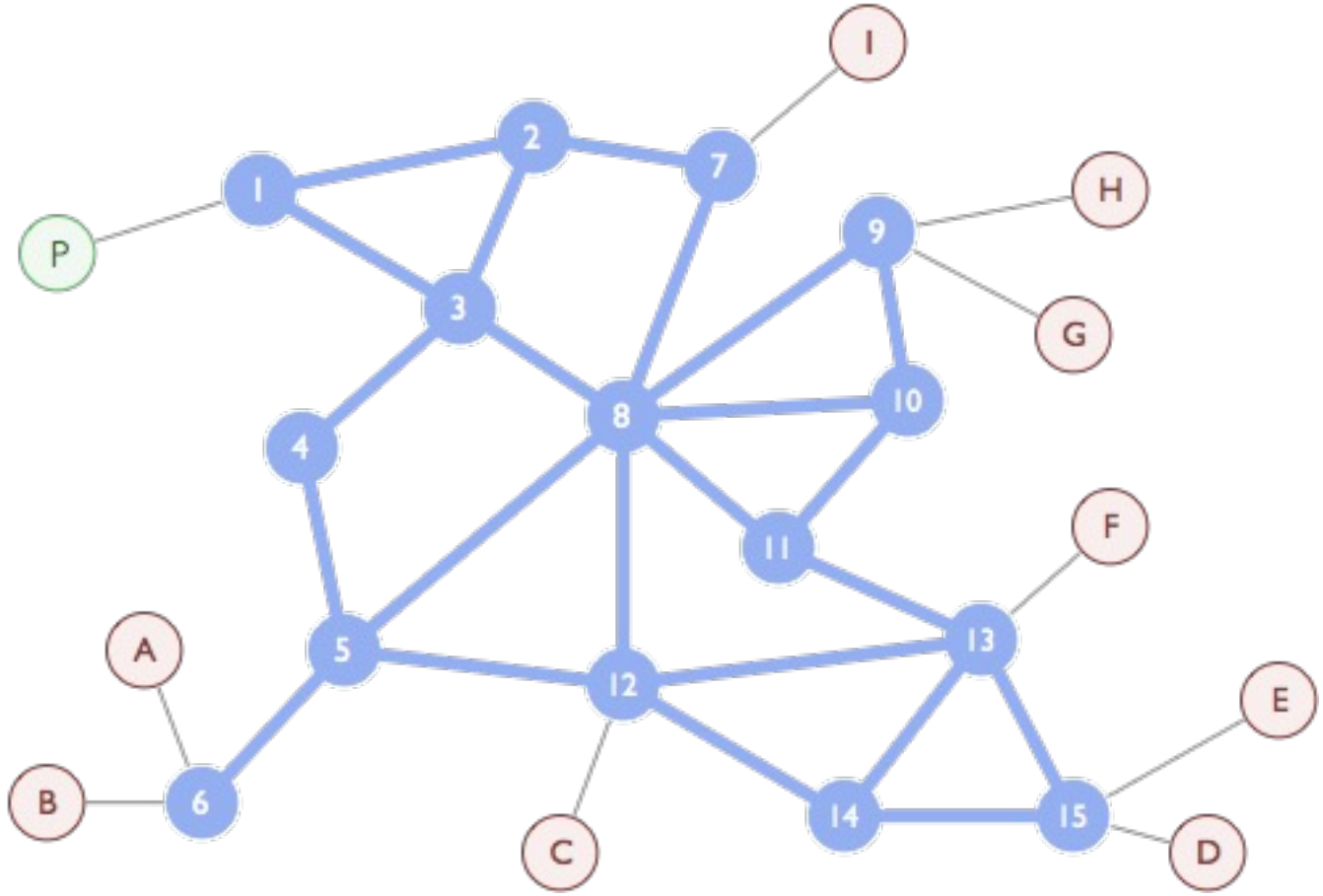


i	*pred*
4	s2 V s3

i	*pred*
3	s2 V s3

Exercise on Siena…..

- Exercise: consider the following system:



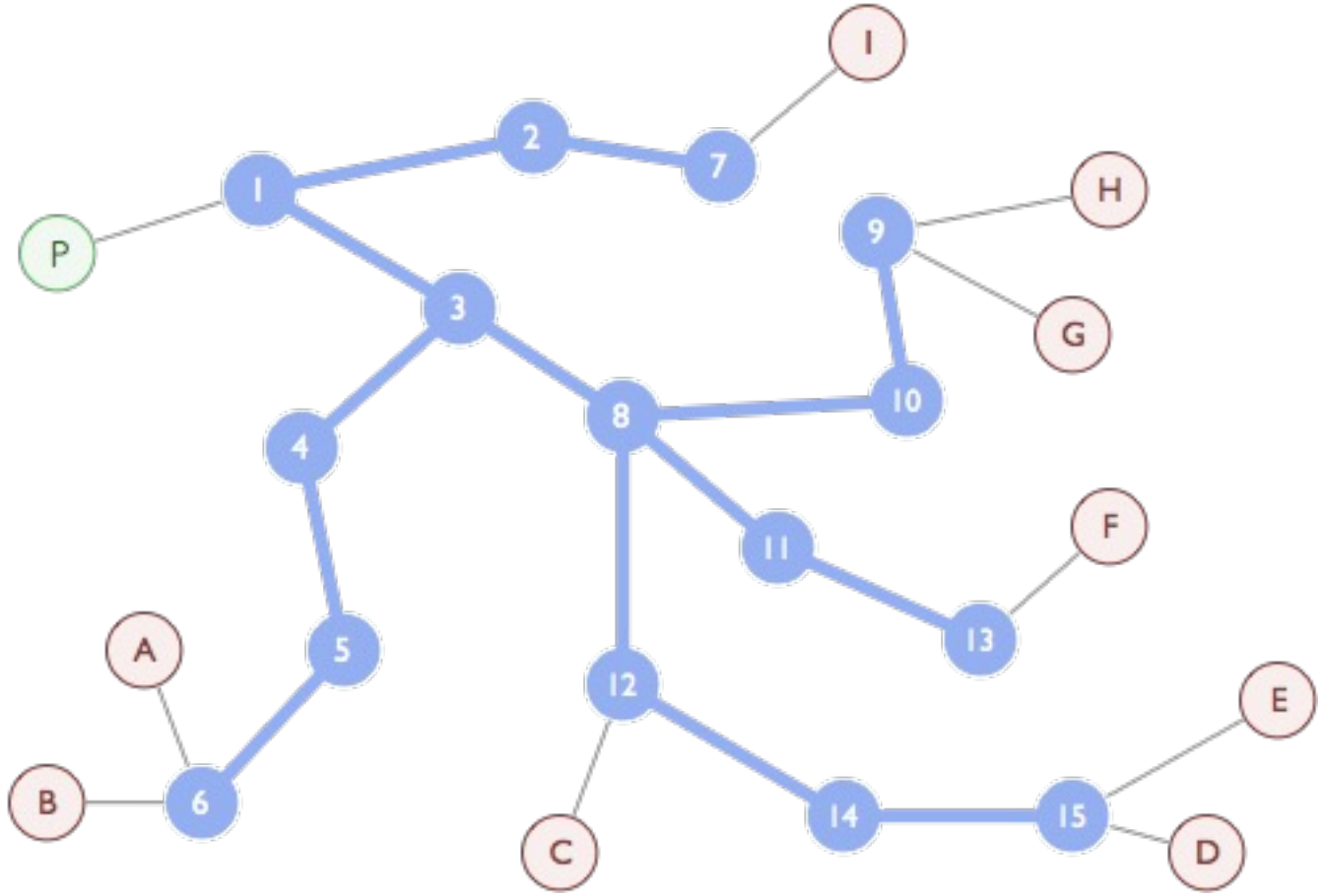
- The event space is represented by a single numerical attribute x which can assume real values. Subscriptions can be expressed using the operators <=>.

- 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.

- 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).

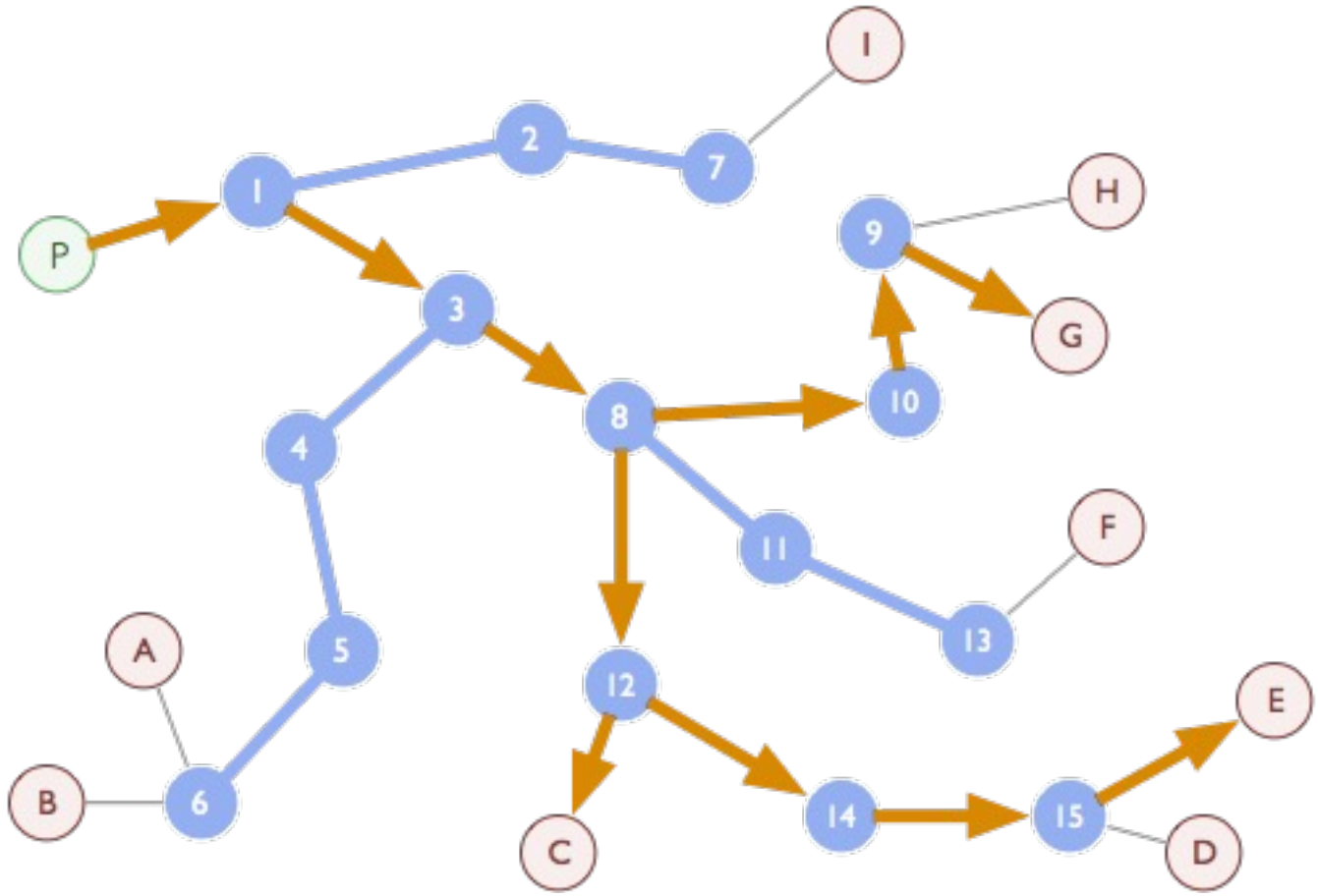
Broker	Interface	Content-based address
1	2	x>50
1	3	x>23 OR (x<0 OR x>90) OR x<40 OR (x>25 AND x<60)
2	7	x>50
3	4	x>23 OR (x<0 OR x>90)
3	8	x<40 OR (x>25 AND x<60)
4	5	x>23 OR (x<0 OR x>90)
5	6	x>23 OR (x<0 OR x>90)
8	10	x<12 OR (x>15 AND x<20)
8	11	x>5 AND x<10
8	12	x<40 OR (x>5 AND x<18) OR (x>25 AND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

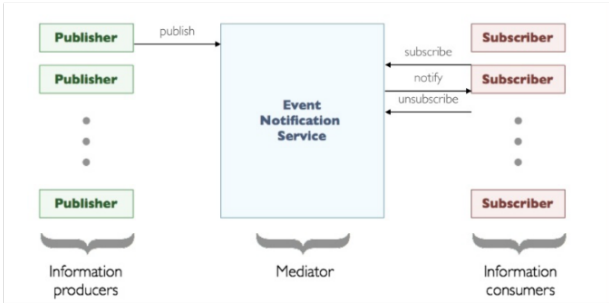
- We are referring to a run-time status where we can assume that, independently from the order used to issue subscriptions, the tables' content is perfect.

- 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)

- On the graph:





20 Publish/Subscribe Systems

In the **publish/subscribe communication paradigm** we have:

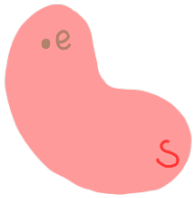
- **Publisher**: who produce data in form of **event**s;
- **Subscribers**: declare interests on *published data* with *subscriptions*;
- Each **subscription** is a *filter* on the set of published *event*s; → constraints expressed on the event schema
- An **Event Notification Service** (**ENS**) notifies to each *subscriber* every published *event* that matches at least one of its *subscriptions*;

This paradigm is used in various cases, we can have a **many to many communication** in which various *producers* and *consumers* can *communicate* all at the same time, each *information* can be delivered at the same time to various *consumers*, all the *interacting parties* don't know each other, the *information delivery* is mediated through a *third party*, so we don't need *synchronization* between the *interacting parties*. ; SPACE, TIME, SYNCHRONIZATION Decoupling

Events represent information structured following an **event schema**, that is *fixed a-priori* and known to all the *participants*, and it defines a set of *fields* or *attributes*, each constituted by a *name* and a *type*. So given an *event schema*, an *event* is a collection of *values*, one for each *attribute* defined in the *schema*. So we can see it as a *table* in which we have *name*, *type of value* and *allowed values* for each *attribute* of the *event*.

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 and in this case we say that e **matches** s. **Subscriptions** can take various forms, for example it can be a *conjunction* of *constraints* each expressed on a *single attribute* in which each *constraint* can be an $>=<$ *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** and each *subscription* s identifies a **subspace**, and an *event* e **matches** the *subscription* s iff the corresponding *point* is in the *portion* of the *space* of s.



20.1 Types of Publish/Subscribe

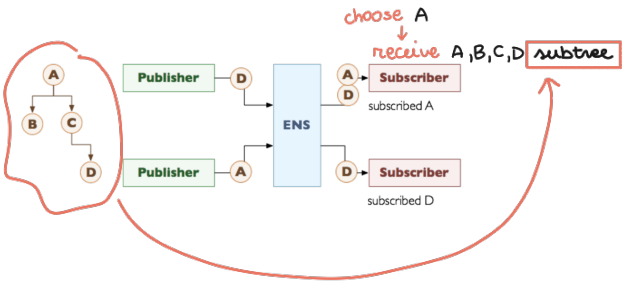
Depending on the **subscription model** used we distinguish various flavors of **publish/subscribe**:

20.1.1 Topic-based selection

In which the *data* published in the *system* is *unstructured* but each *event* is **tagged** with the *identifier* of a **topic** it is published in, so the *subscribers* issue a *subscription* that contains the *topics* they are interested in. So a *topic* can be thus represented as a *virtual channel* connecting *producers* to *consumers*.

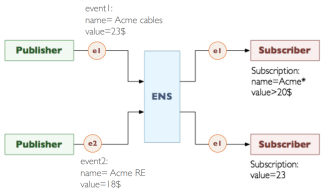
20.1.2 Hierarchy-based selection

In which each *event* is *tagged* with the *topic* it is published in like before, but 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*.



20.1.3 Content-based selection

In which all the *data* published in the *system* are **mostly structured**, so each *subscription* can be expressed as a **conjunction of constrains** expressed on **attributes**, and the **ENS** will filter out the *useless events* before notifying a *subscriber*.

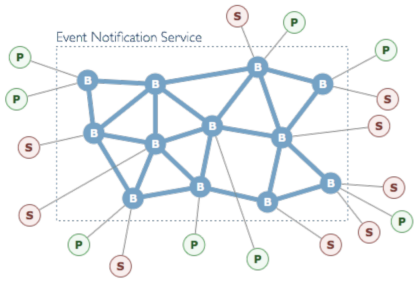


20.2 Event Notification Service

The **Event Notification Service** can be *implemented* in two ways:

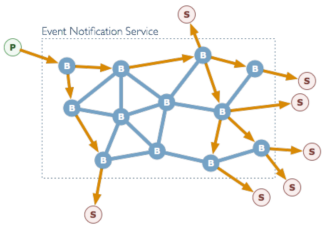
- **Centralized service**: the *ENS* is implemented on a *single server*.
- **Distributed service**: the *ENS* is constituted by a set of *nodes*, called **event brokers**, which *cooperate* to implement the *service*;

The **distributed service** is usually preferred for *large setting* where **scalability** is a *fundamental issue*.



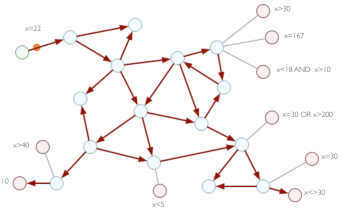
20.3 Event routing mechanism

These **event brokers** forms an *overlay network*, so each *Client* (*publisher* or *subscriber*) accesses the *service* through a *broker* that masks the *system complexity*. We need to introduce an **event routing** mechanism that *routes* each *event* inside the *ENS* from the *broker* where it is *published* to the *brokers* where it must be *notified*.



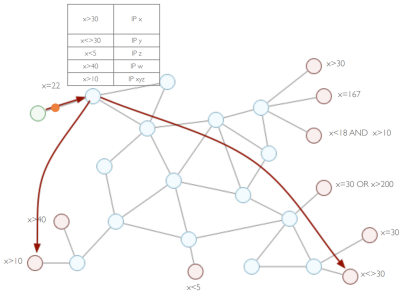
20.3.1 Event Flooding

In which each *event* is **broadcasted** from the *publisher* in whole the *system*, this implementation is *straightforward* but very **expensive**, so we have *highest message overhead* and no *memory overhead*.



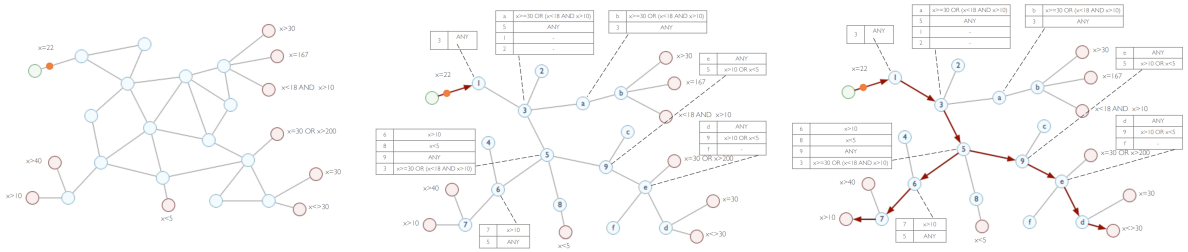
20.3.2 Subscription Flooding

In which each each *subscription* is copied on every *broker*, in order to have in each *broker* a **complete subscription table**, that we will use to *locally match events* and directly notify interested *subscribers*. This approach suffers from a *large memory overhead*, but *event diffusion* is optimal. The problem is that we cannot use this in *applications* where *subscriptions* change frequently.



20.3.3 Filter-based routing

In which *subscriptions* are **partially diffused** in the *system* and used to build **routing tables**, that are then exploited during *event diffusion* to dynamically build a **multi-cast tree** that connects the *publisher* to all the interested *subscribers*. Each *broker* will have the *subscriptions* of the **closest brokers**, so during the *event diffusion* we will go back finding through the *routing tables* all the interested *subscribers*.



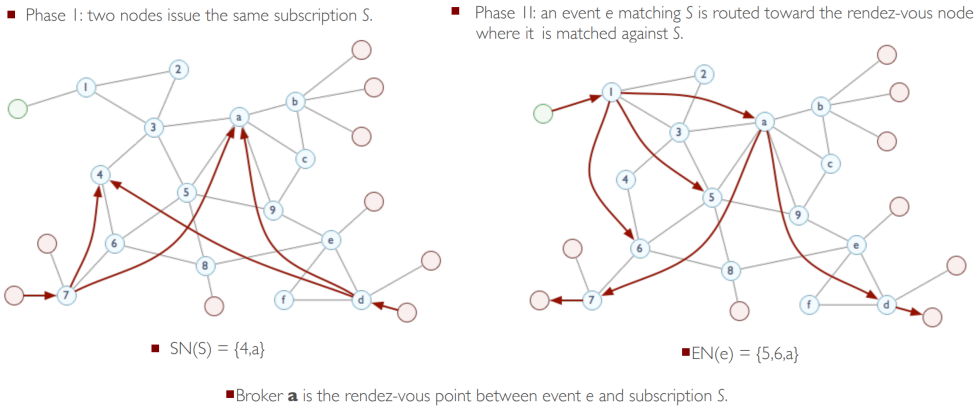
20.3.4 Rendez-Vous routing

The **Rendez-Vous routing** is based on two functions: SN and EN used to associate *subscriptions* and *events* to *brokers* in the *system*. Given a *subscription* s then the function $SN(s)$ 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 the function $EN(e)$ returns a *set of nodes* which must receive e to match it against the *subscriptions* they store. The goal of this routing is to

obtain the intersection (not empty) between the output of the functions SN and EN. It is divided in two phases:

- **Phase 1**:
  - Given a **subscription** s issued by one or more *subscriber*, the *brokers* that are directly linked to these *subscribers* will ask to the *network* with the function $SN(s)$, in order to get the *set of nodes* that will store the *subscription* s;
- **Phase 2**:
  - When an *event* e that matches s is asked by a *publisher*, all the *nodes* linked to that *publisher* will ask for the *function* $EN(e)$, that will return the *set of nodes* that needs to receive that *message* in order to respect all the *subscriptions* that they contains.

At the end we will check the *nodes* in the **intersection** between the two *sets of nodes* obtained in the two phases, and this *node* will be the **rendez-vous point** between *event* e and subscription s.



20.4 Publish/Subscribe Architecture

In the **publish/subscribe architecture** we have that each *node* has a *service interface* consisting of two **operations**: *send_message*(m) and *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*.

ridotto

CBCB routing scheme

We will have a fusion between **Content-Based routing scheme** and **Broadcast** cause if a *node* need to send an *event* in the *network* he will *broadcast* it to all its *neighbor nodes* but if a *node* is receiving a *message* he will *rebroadcast* it only if all the *conditions* of the *subscriptions* inside him are respected. So the *broadcast* will take care of the **flooding** of the *message* in the *network*, instead the *content based* will create the **sub-tree** used to send the *events* to the correct *subscribers*. So we have two layers, a **Content-based layer** and a **Broadcast Layer**.

The **broadcast layer** is composed by a **broadcast function** available at each *router*, which given

↓ diffuses msgs in the net

source s
input i ↦ output i

a *source node* s and an *input interface* i will return the set of *output interfaces*. So this *function* defines a **broadcast tree** routed at each *source node* and it satisfy the *all-pairs path symmetry property*.

reduces broadcast forwarding paths

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*. So after the *broadcast layer* has initialized a *tree path symmetric* for each *source node*, then the *content-based* will be used in order to use the *forwarding mechanism* to route the *events* from the *source* to the correct *subscriber*.

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. So the *message* is forwarded along the set of *interfaces* returned by: $(B(source(m), incoming_i f(m)) \cup l_0) \cap F_c(m)$, this means we will get the *intersection* between the *nodes* of the *tree* returned by the *broadcast function* and the set of *nodes* returned by the *forwarding function*. The **forwarding tables** maintenance is based on two *mechanism*: **push** that is based on *receiver advertisements*, and **pull** that is based on *sender requests* and *update replies*.

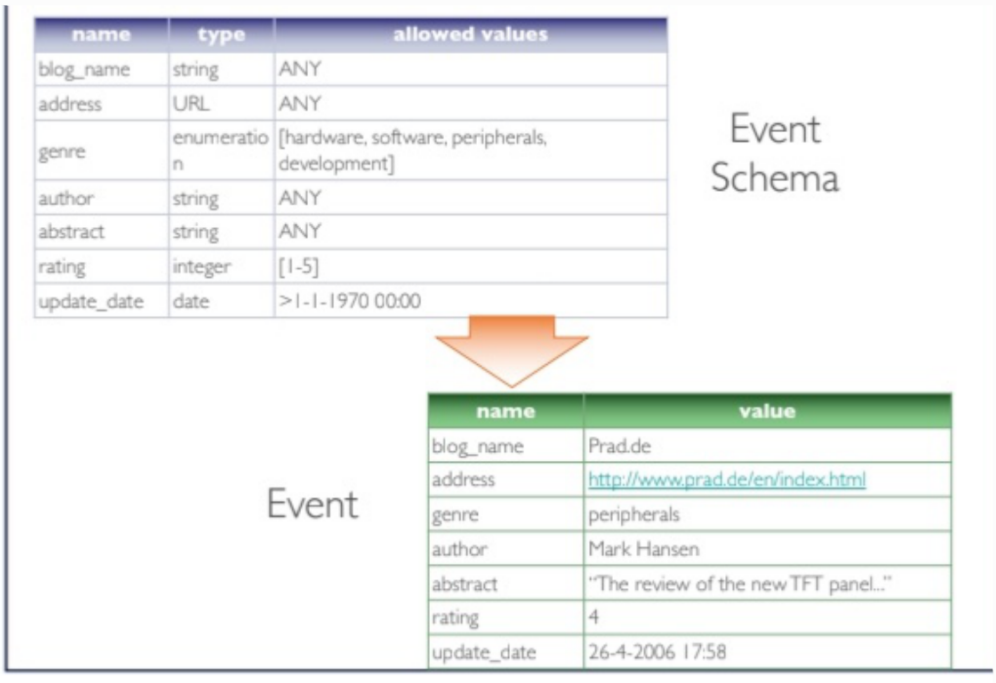
The **Receiver advertisements** are issued by *nodes* periodically and/or when the *node* changes its **local content-based address** p_0. When a *node* **changes** its **advertisements table**, will send to the *nodes* of the *broadcast tree* a *message* of type RA like (n, P) where n is the *identifier* of the *interface* from which comes the *change request* and P is the new *predicate* of the *interface*. When a *node* receives an RA message will do two things:

- First will check if in its own *table* there already exists a *predicate* p_i that contains P of the *interface* n, and in this case the *request* RA is **dropped**;
- Instead if p_i doesn't cover P then the *router* will add the *predicate* to the old predicate and will *rebroadcast* the *message* RA to all the *nodes* near him in the *tree path* of the *broadcast function*.

With the **RA protocol** we can have some *inflation* problems, in which a *node* drops the RA *request* since the *predicate* it's already in its own *table*, but in this way he will not *rebroadcast* the *request*, so these *nodes* cannot update their table.

In the **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*, and the result of an SR come back to the *issuer* 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. In the RA *protocol* all the *tables* were updated through a *mono-directional communication* now we have a *two-way communication*. We start from a *node* n that want to *update* the *table* with a message SR, so will ask to all the *nodes* in the *network* their table through the *broadcast function*, so this *request* is *rebroadcasted* to all the *nodes* until it reaches the *leafs* of the *tree* created by the *broadcast function*. These *leafs* will replays with a *message* UR and they will send back their *tables*, so all the *nodes* will *update* their *table* and they will *rebroadcast* the UR *message* until they arrive to the *original node* n.

EVENT



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