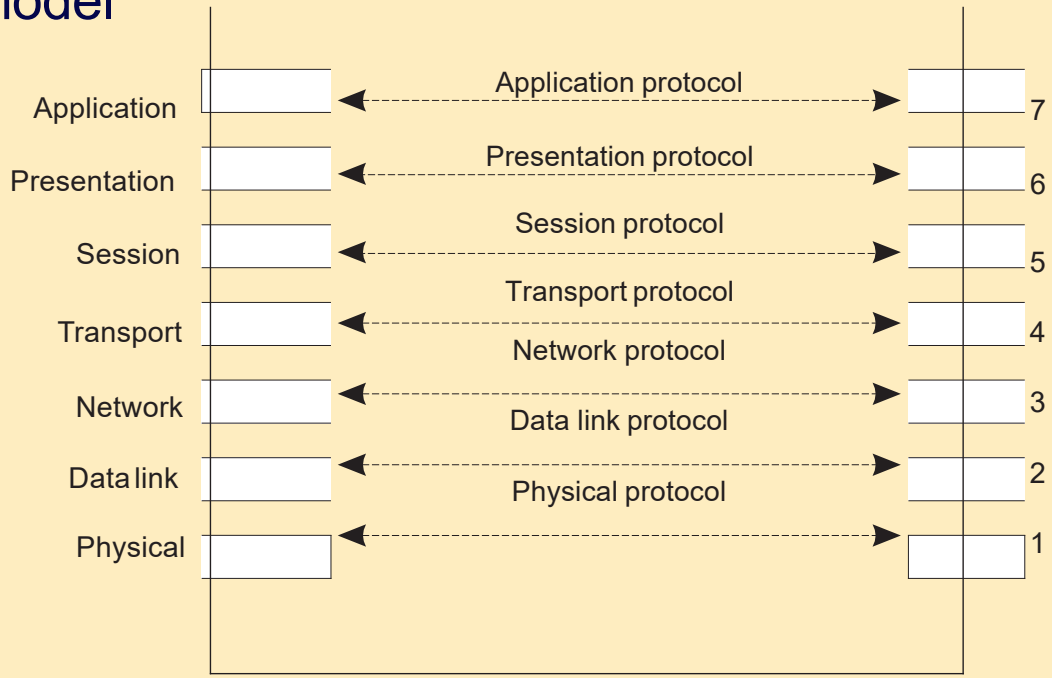
Distribution of computation
Interprocess communication

Communication for Distributed Systems

ISO-OSI layered model



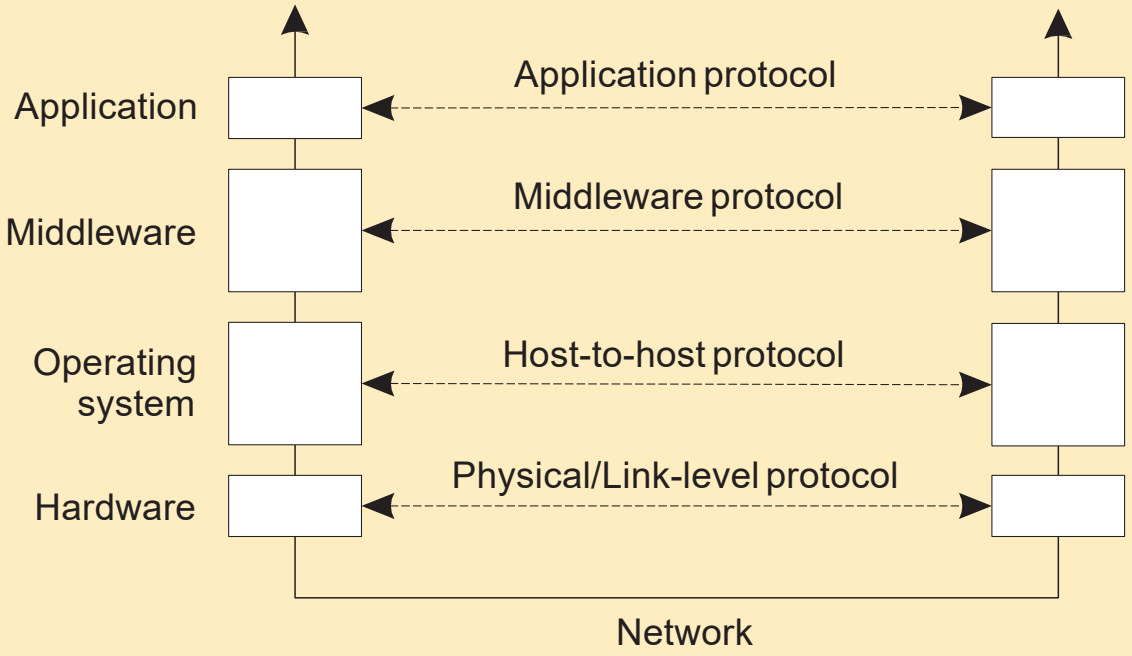
Drawbacks

- Focus on message-passing only
- Often unneeded or unwanted functionality
- Violates access transparency

Middleware layer

Middleware is invented to provide common services and protocols that can be used by many different applications

A rich set of communication protocols
(Un)marshaling of data, necessary for integrated systems
Naming protocols, to allow easy sharing of resources
Security protocols for secure communication
Scaling mechanisms, such as for replication and caching



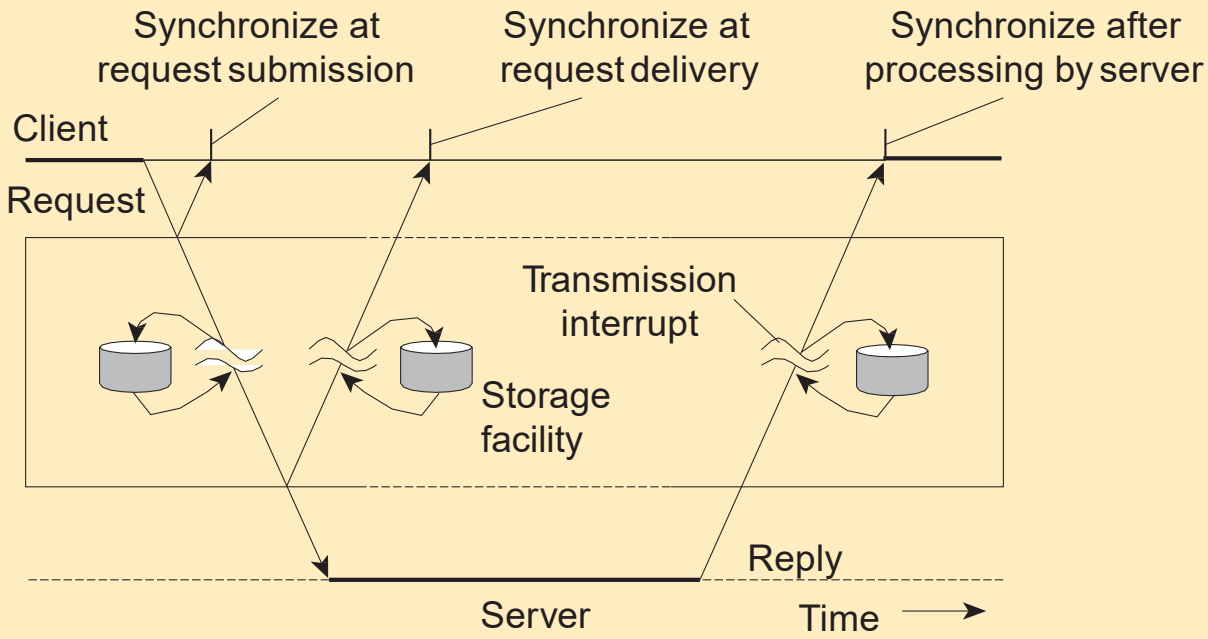
Types of communication

Transient versus persistent

Transient : comm. server discards message when it cannot be delivered at the next server, or at the receiver.

Persistent : a message is stored at a communication server as long as it takes to deliver it.

Places for synchronization

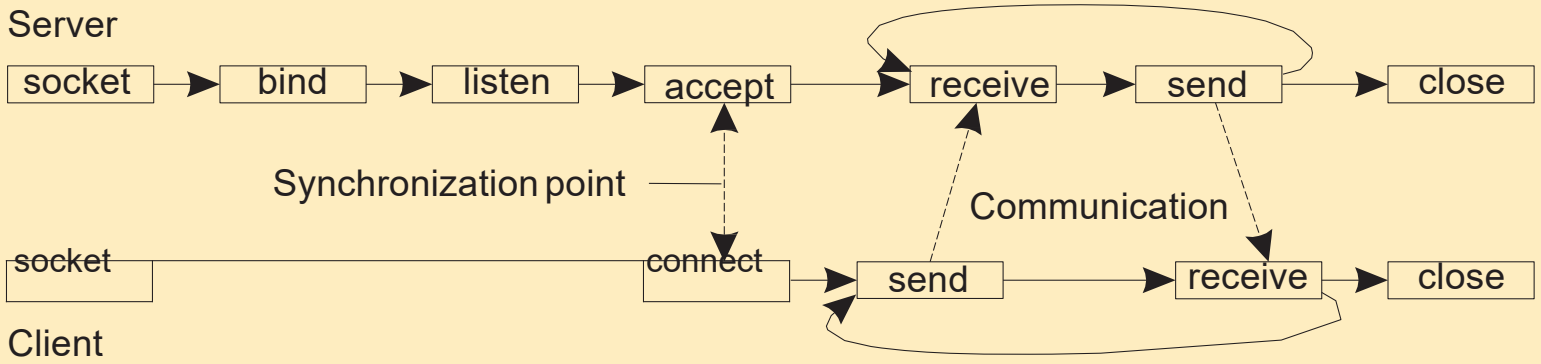


At request submission
At request delivery
After request processing

Transient messaging: sockets

Berkeley socket interface

Operation	Description
`socket`	Create a new communication end point
`Bind`	Attach a local address to a socket
`listen`	Tell operating system what the maximum number of pending connection requests should be
`accept`	Block caller until a connection request arrives
`connect`	Actively attempt to establish a connection
`send`	Send some data over the connection
`receive`	Receive some data over the connection
`close`	Release the connection



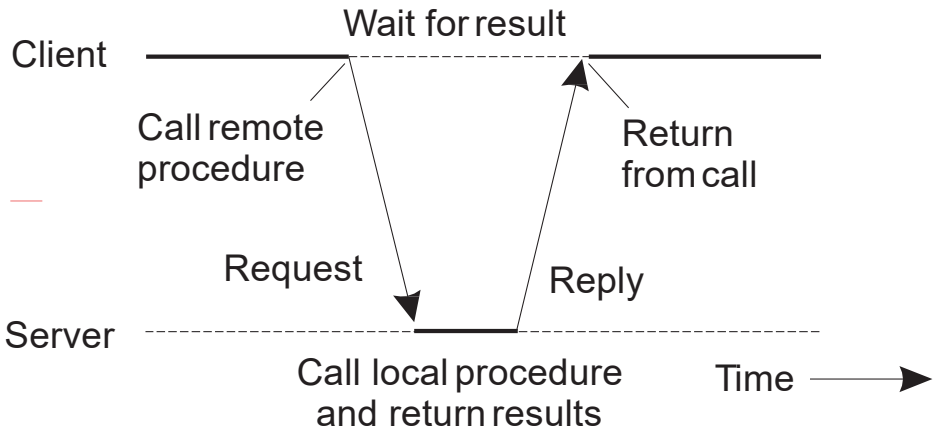
Basic RPC operation

Observations

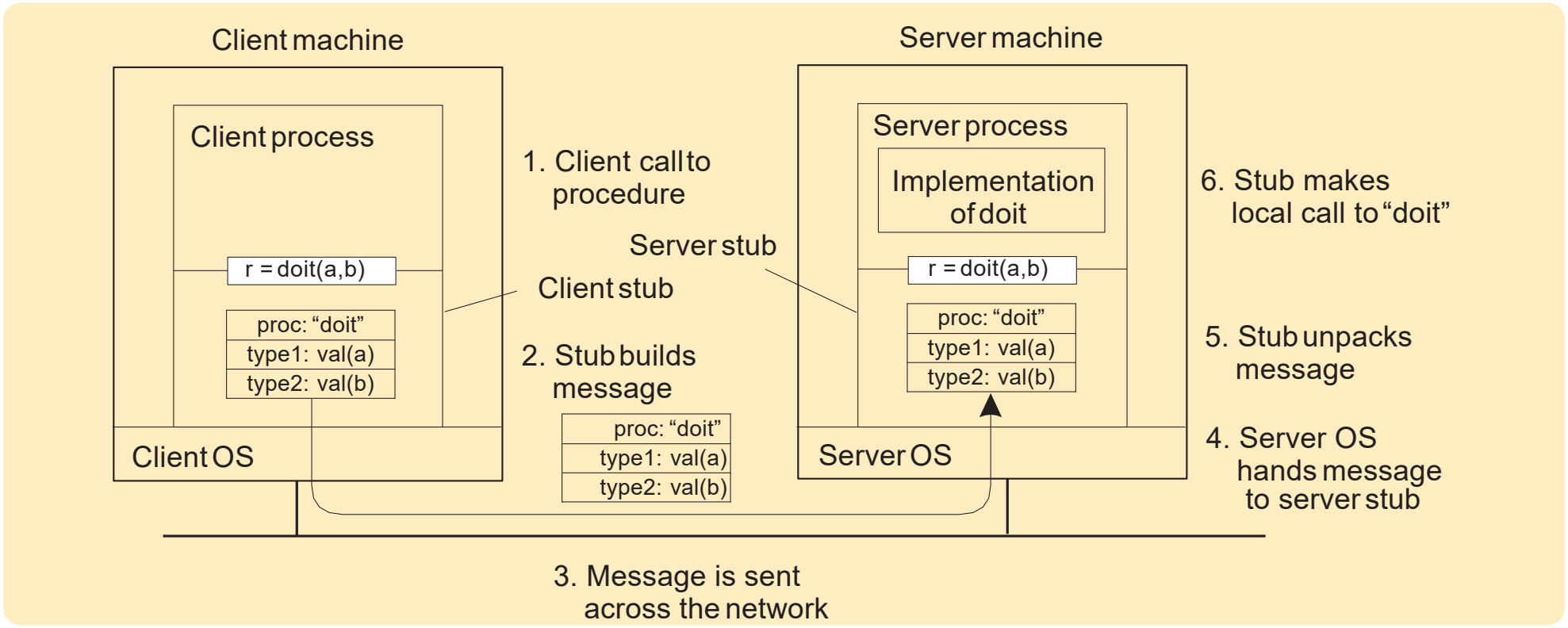
Application developers are familiar with simple procedure model
Well-engineered procedures operate in isolation (black box)
There is no fundamental reason not to execute procedures on separate machine

Conclusion

Communication between caller & callee can be hidden by using procedure-call mechanism.



Basic RPC operation



Client procedure calls client stub.
Stub builds message; calls local OS.
OS sends message to remote OS.
Remote OS gives message to stub.
Stub unpacks parameters; calls server.

Server does local call; returns result to stub.
Stub builds message; calls OS.
OS sends message to client's OS.
Client's OS gives message to stub.
Client stub unpacks result; returns to client.

RPC: Parameter passing

There's more than just wrapping parameters into a message

Client and server machines may have different data representations (think of byte ordering)

Wrapping a parameter means transforming a value into a sequence of bytes

Client and server have to agree on the same encoding:

- How are basic data values represented (integers, floats, chars)
- How are complex data values represented (arrays, unions)

Conclusion

Client and server need to properly interpret messages, transforming them into machine-dependent representations.

RPC: Parameter passing

Some assumptions

Copy in/copy out semantics: while procedure is executed, nothing can be assumed about parameter values.
All data that is to be operated on is passed by parameters. Excludes passing references to (global) data.

Conclusion

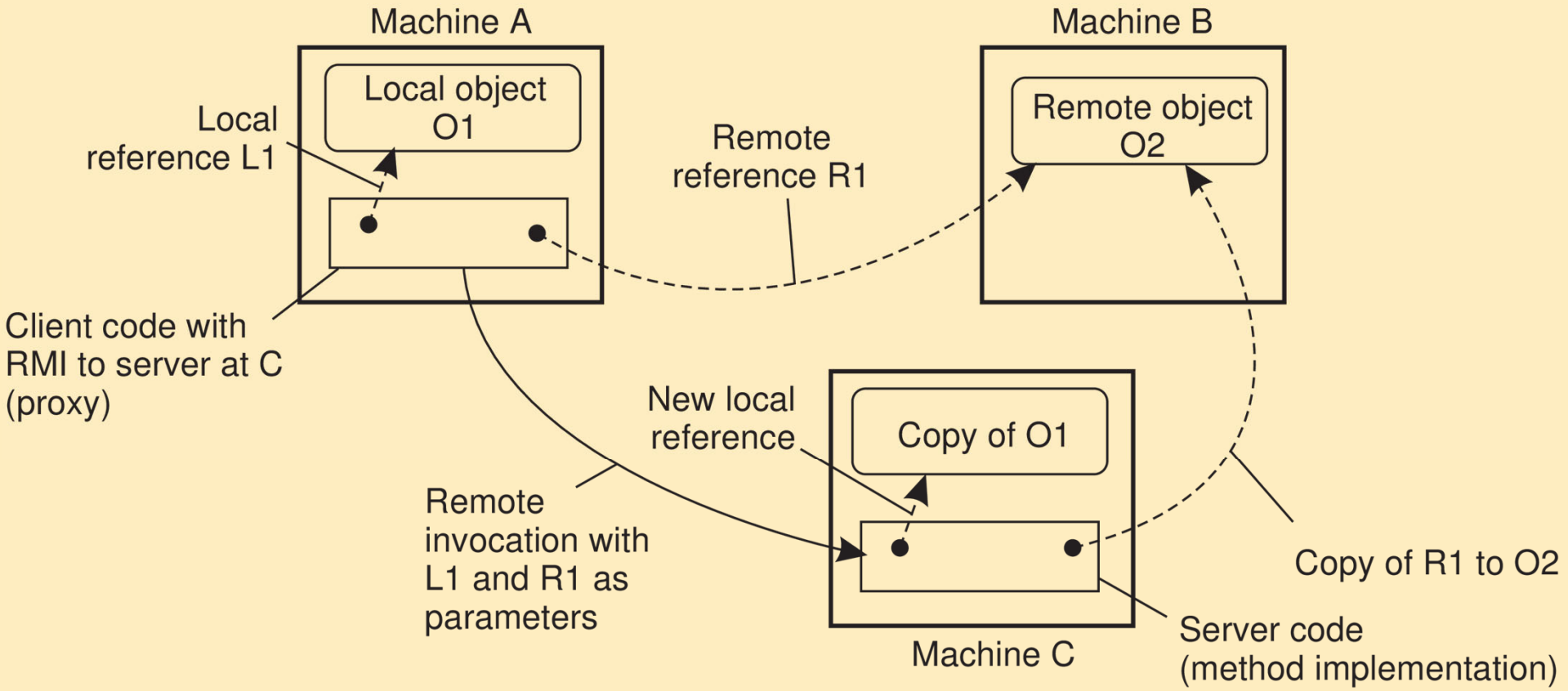
Full access transparency cannot be realized.

A remote reference mechanism enhances access transparency

Remote reference offers unified access to remote data
Remote references can be passed as parameter in RPCs
Note: stubs can sometimes be used as such references

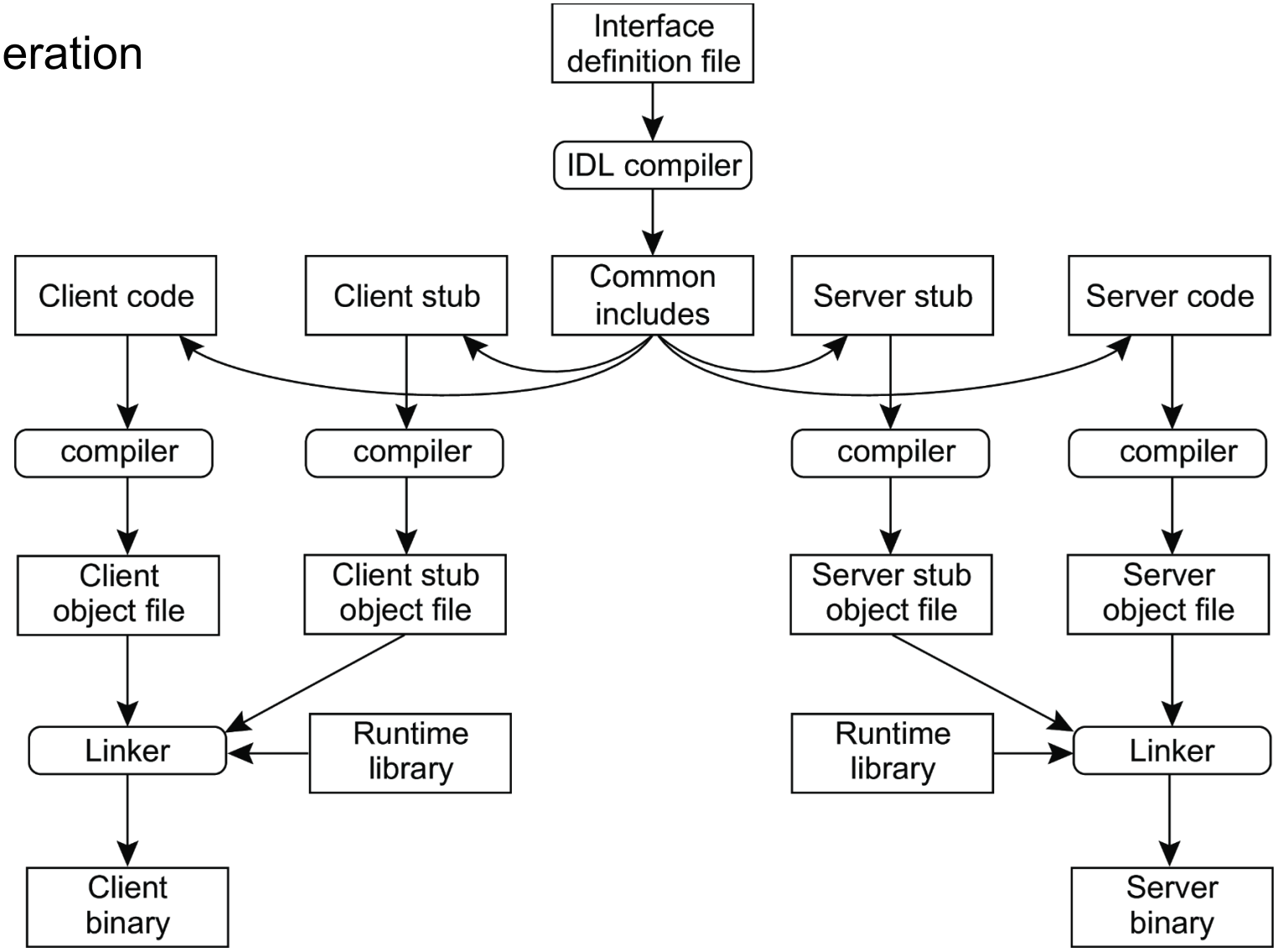
RPC: Parameter passing

Object based systems



RPC: Interface Definition Language

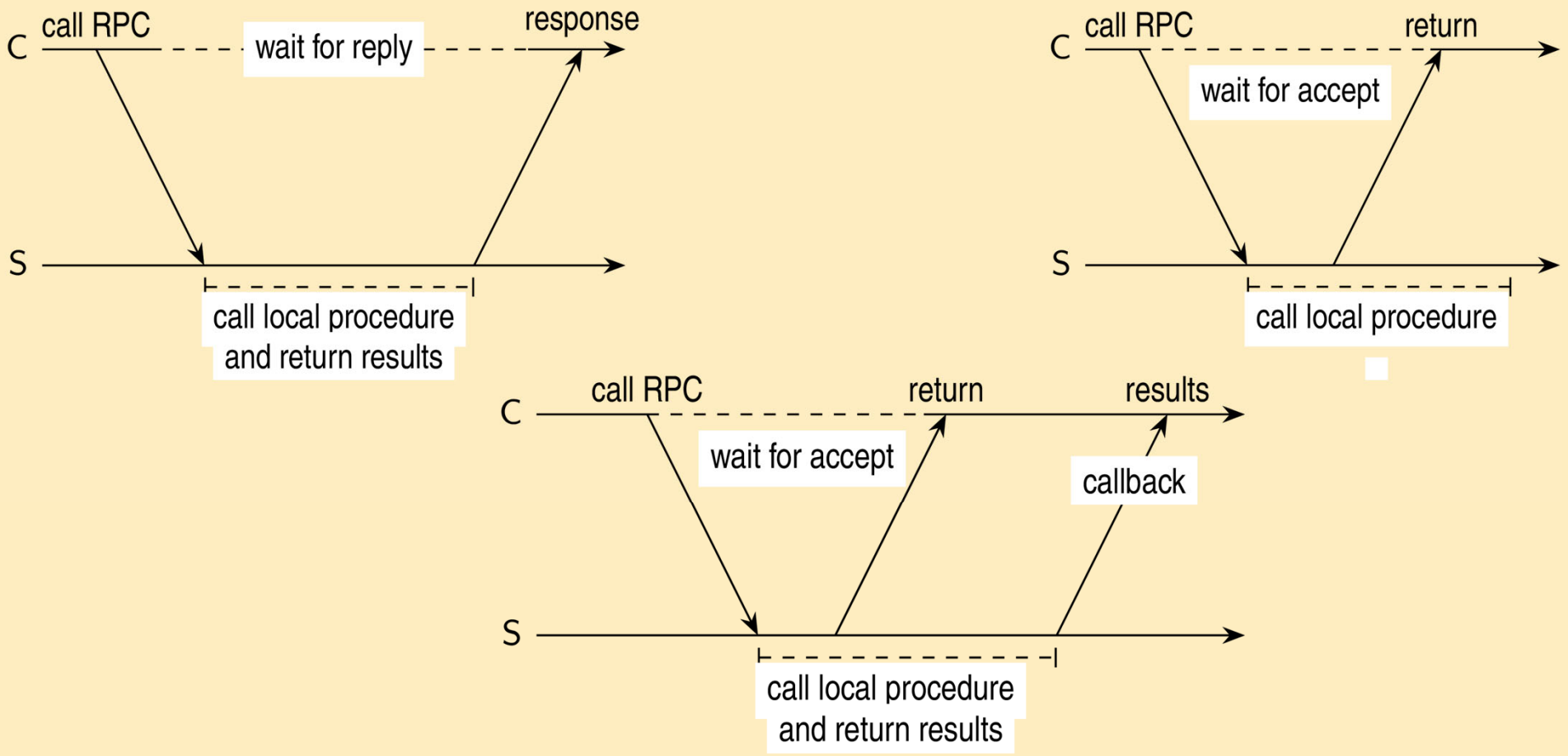
STUB generation



Asynchronous RPCs

Essence

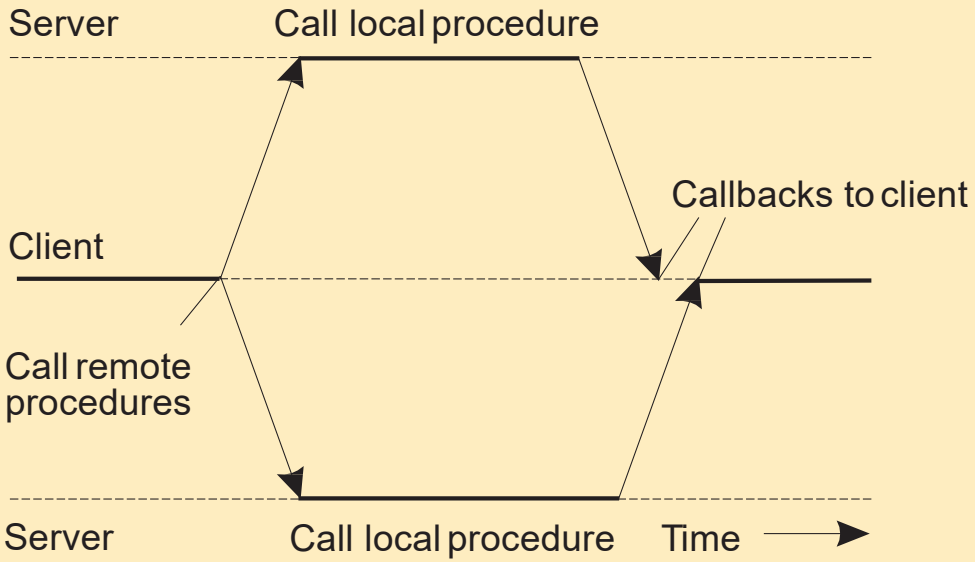
Try to get rid of the strict request-reply behavior, but let the client continue without waiting for an answer from the server.



Sending out multiple RPCs

Essence

Sending an RPC request to a group of servers.



Message-oriented middleware

Essence

Asynchronous persistent communication through support of middleware-level queues. Queues correspond to buffers at communication servers.

Operations

Operation	Description
`put`	Append a message to a specified queue
`get`	Block until the specified queue is nonempty, and remove the first message
`poll`	Check a specified queue for messages, and remove the first. Never block
`notify`	Install a handler to be called when a message is put into the specified queue

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.

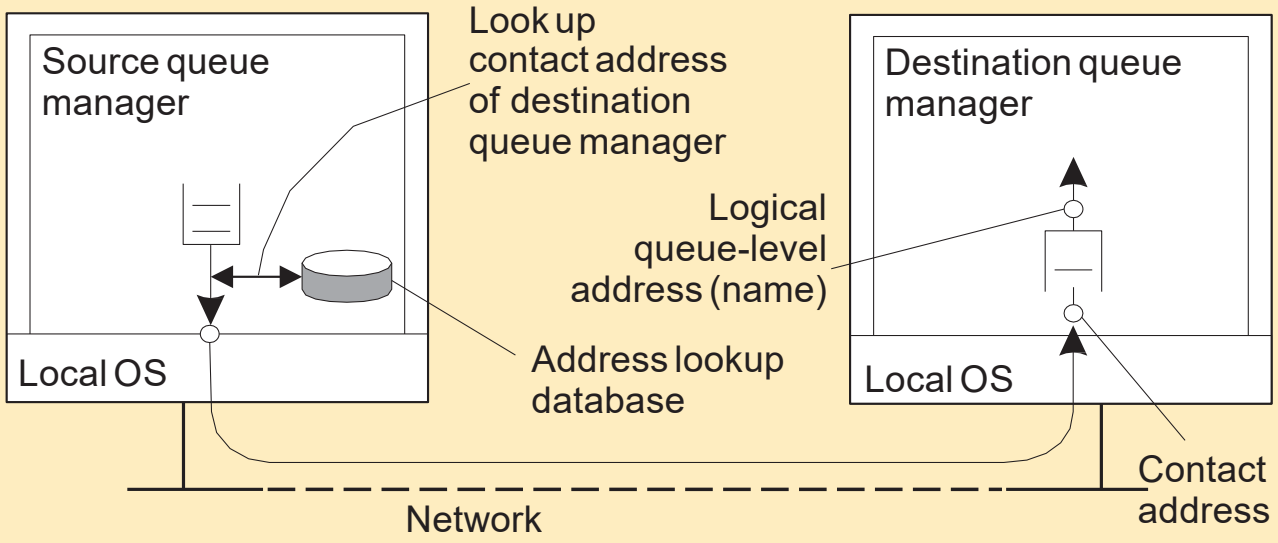
These characteristics make pub/sub perfectly suited for distributed applications relying on document-centric communication.

General model

Queue managers

Queues are managed by queue managers. An application can put messages only into a local queue. Getting a message is possible by extracting it from a local queue only ⇒ queue managers need to route messages.

Routing



Message broker

Observation

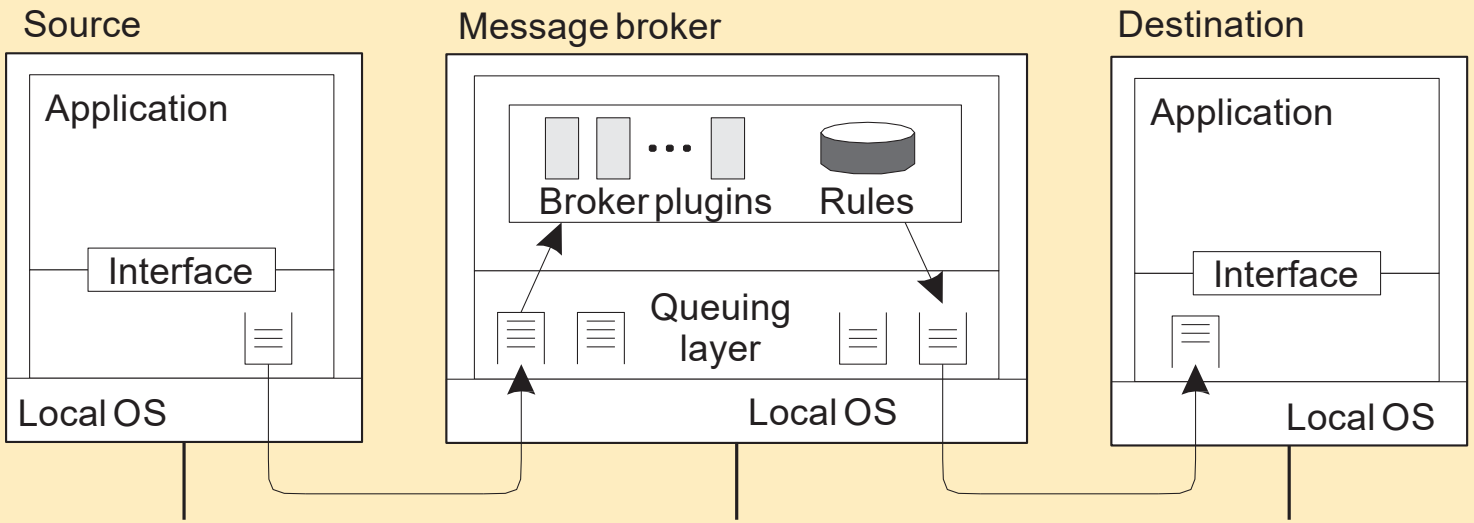
Message queuing systems assume a common messaging protocol:
all applications agree on message format (i.e., structure and data representation)

Broker handles application heterogeneity in an MQ system

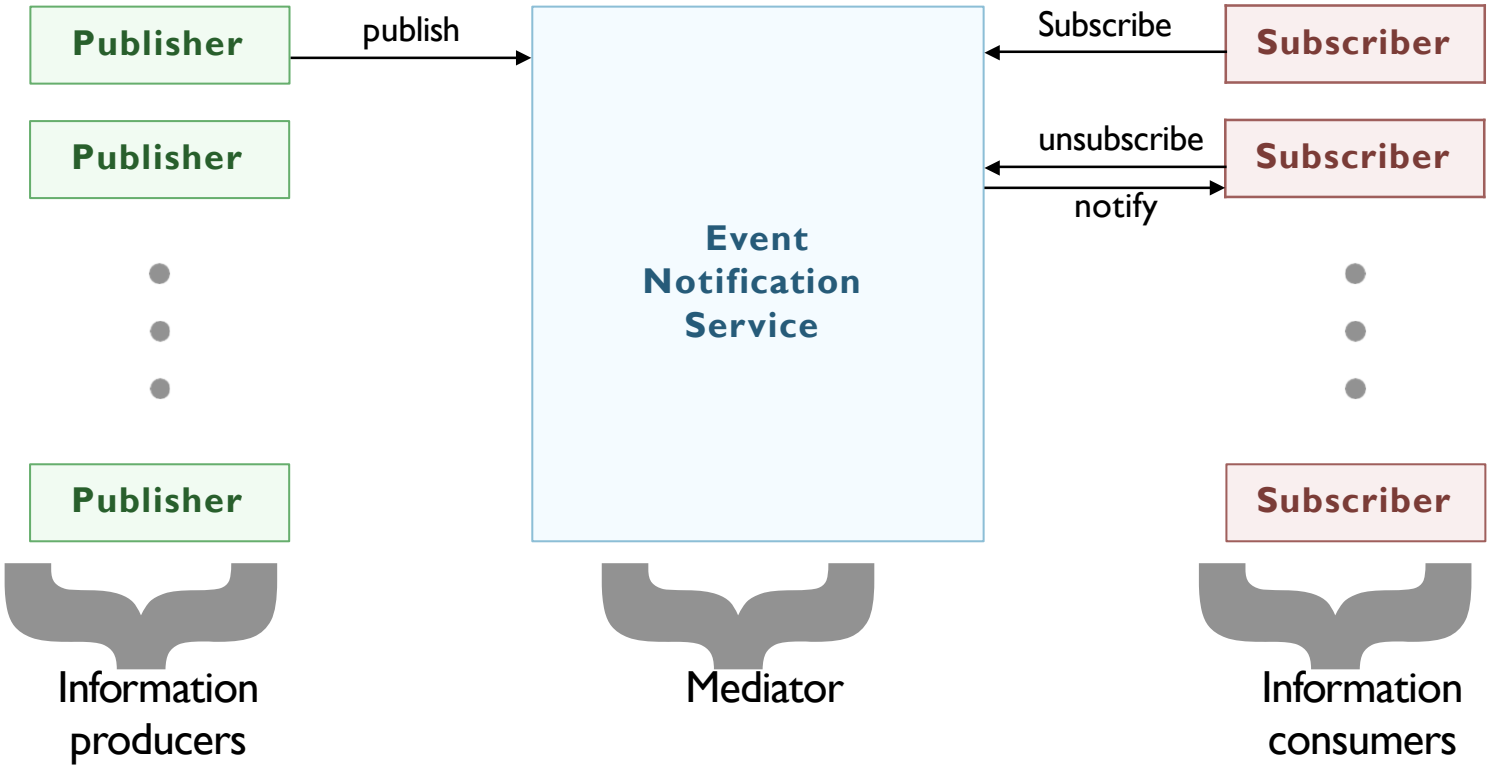
Transforms incoming messages to target format

Very often acts as an application gateway

May provide subject-based routing capabilities (i.e., publish-subscribe capabilities)



- 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.

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

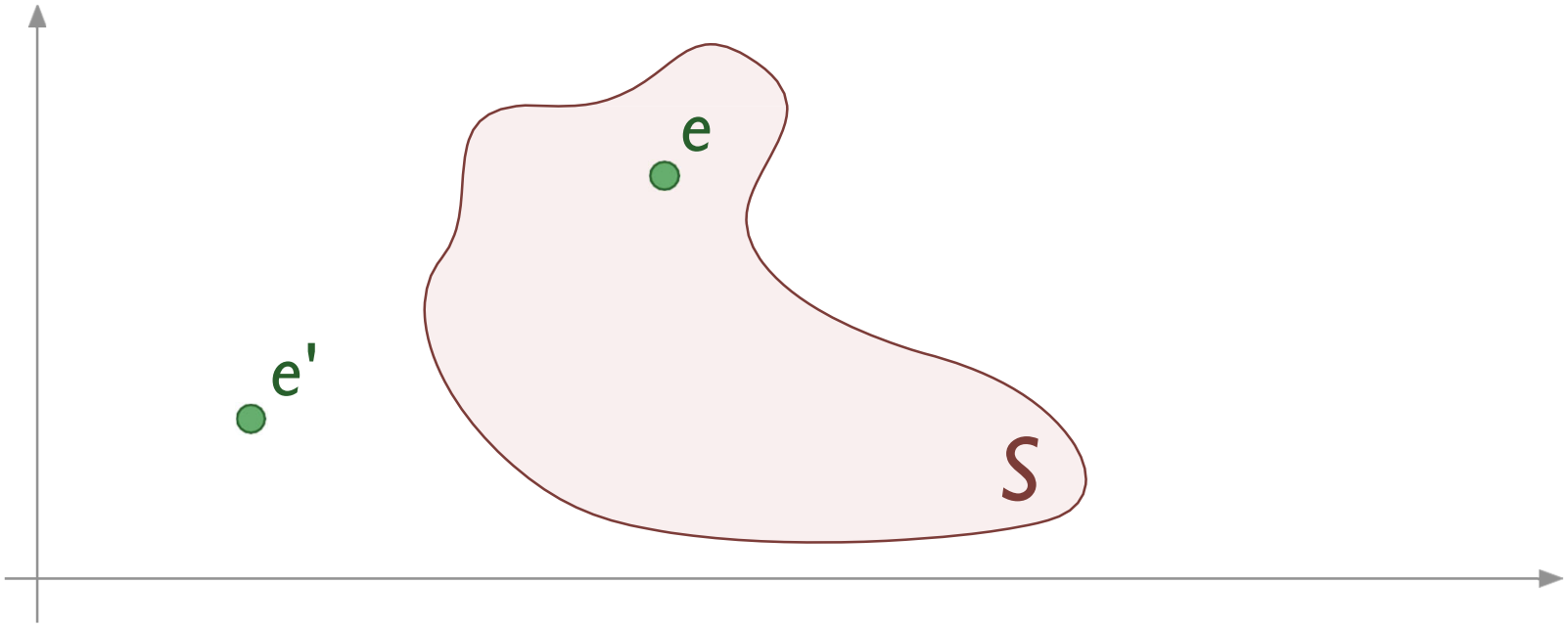
Event Schema

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.

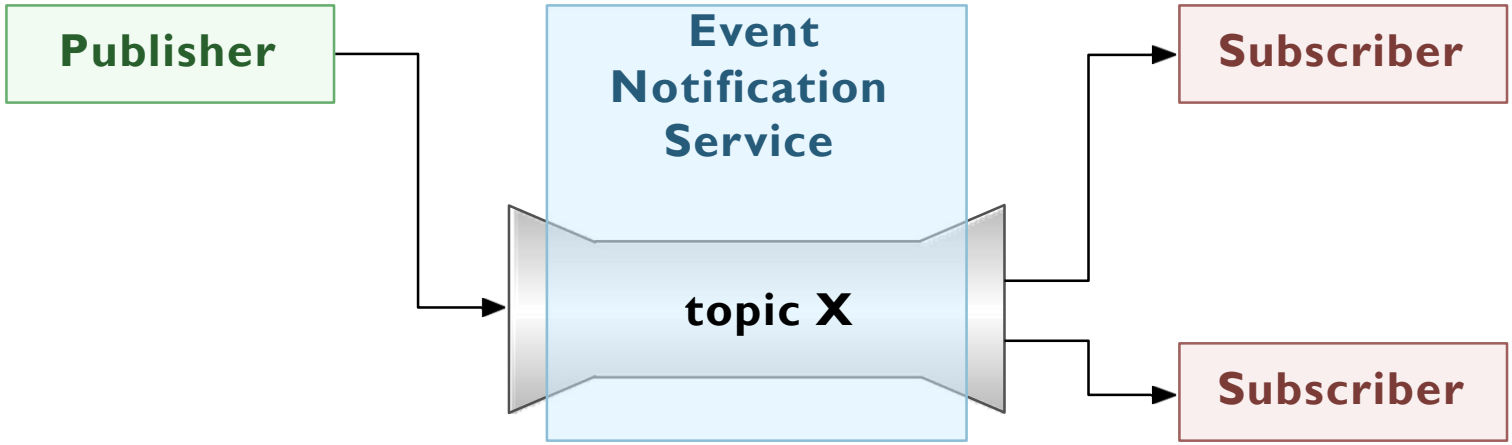
- From an abstract point of view the event schema defines an n-dimensional **event space** (where n is the number of attributes).
- In this space each event e represents a point.
- Each subscription s identifies a subspace.
- An event e matches the subscription s if, and only if, the corresponding point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

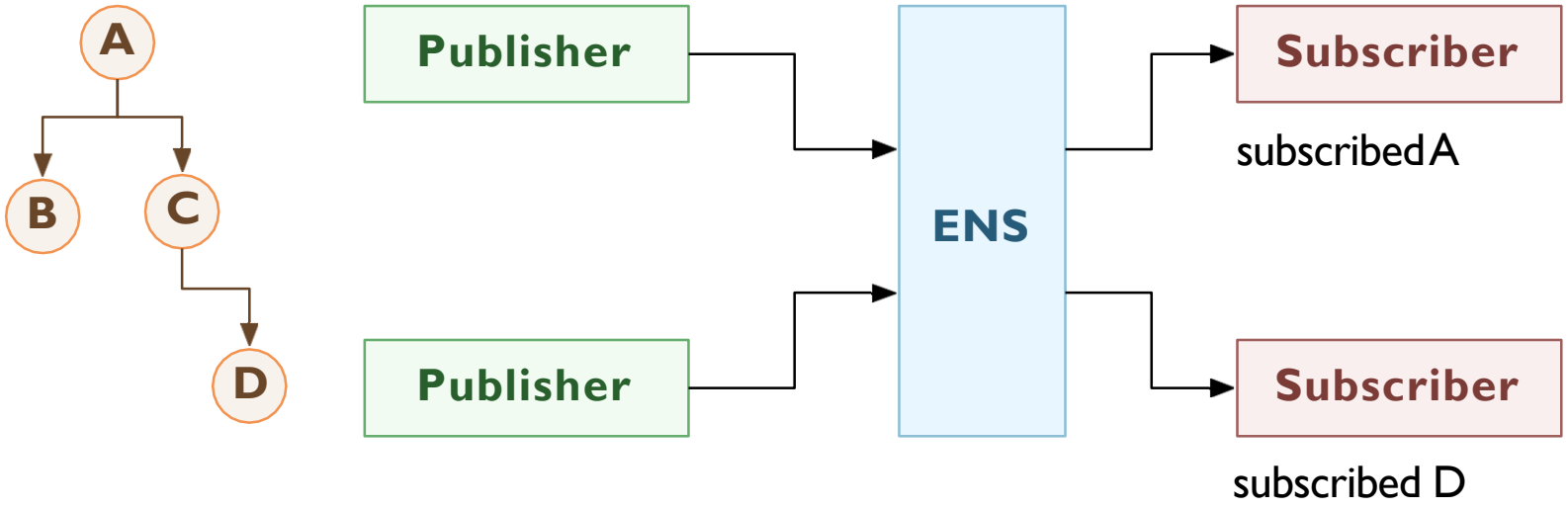
Topic-based selection: data published in the system is mostly unstructured, but each event is "tagged" with the identifier of a *topic* it is published in. Subscribers issue subscriptions containing the topics they are interested in.

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.

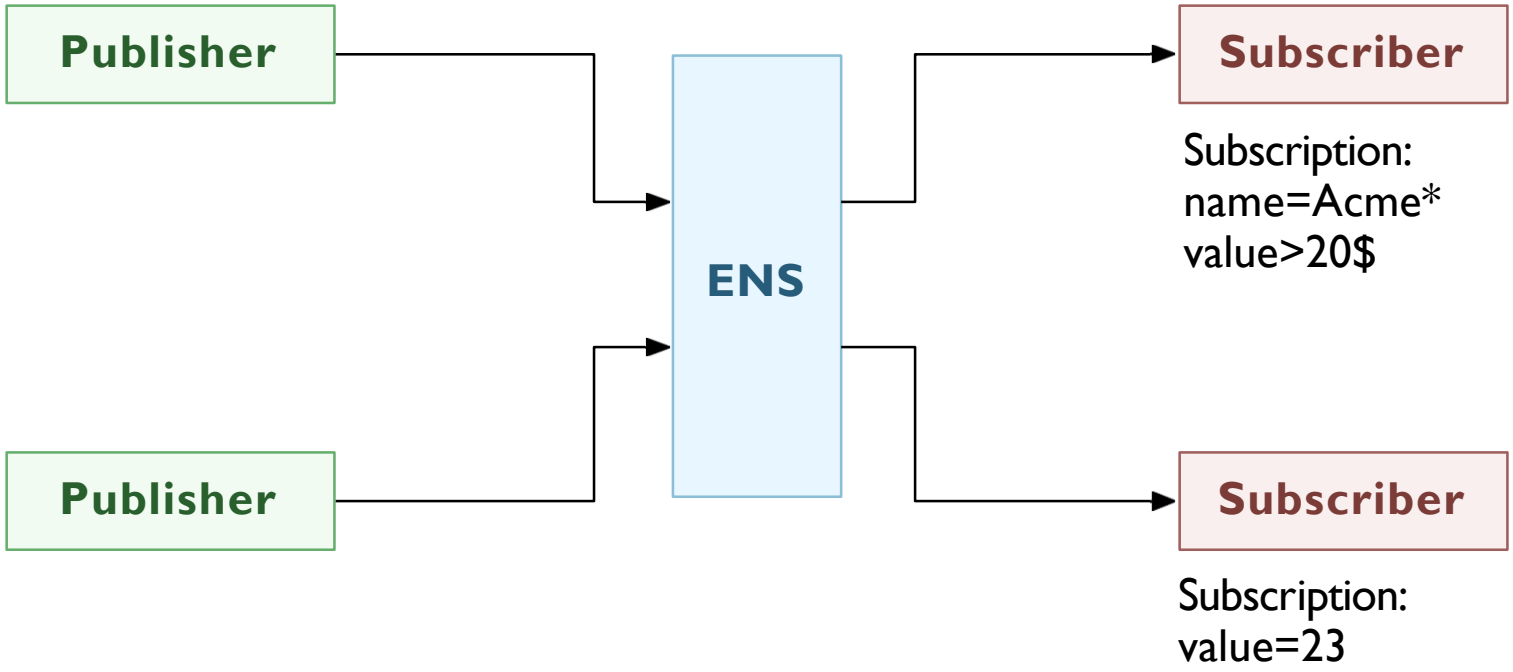


Hierarchy-based selection: even in this case each event is "tagged" with the *topic* it is published in, and Subscribers issue subscriptions containing the topics they are interested in.

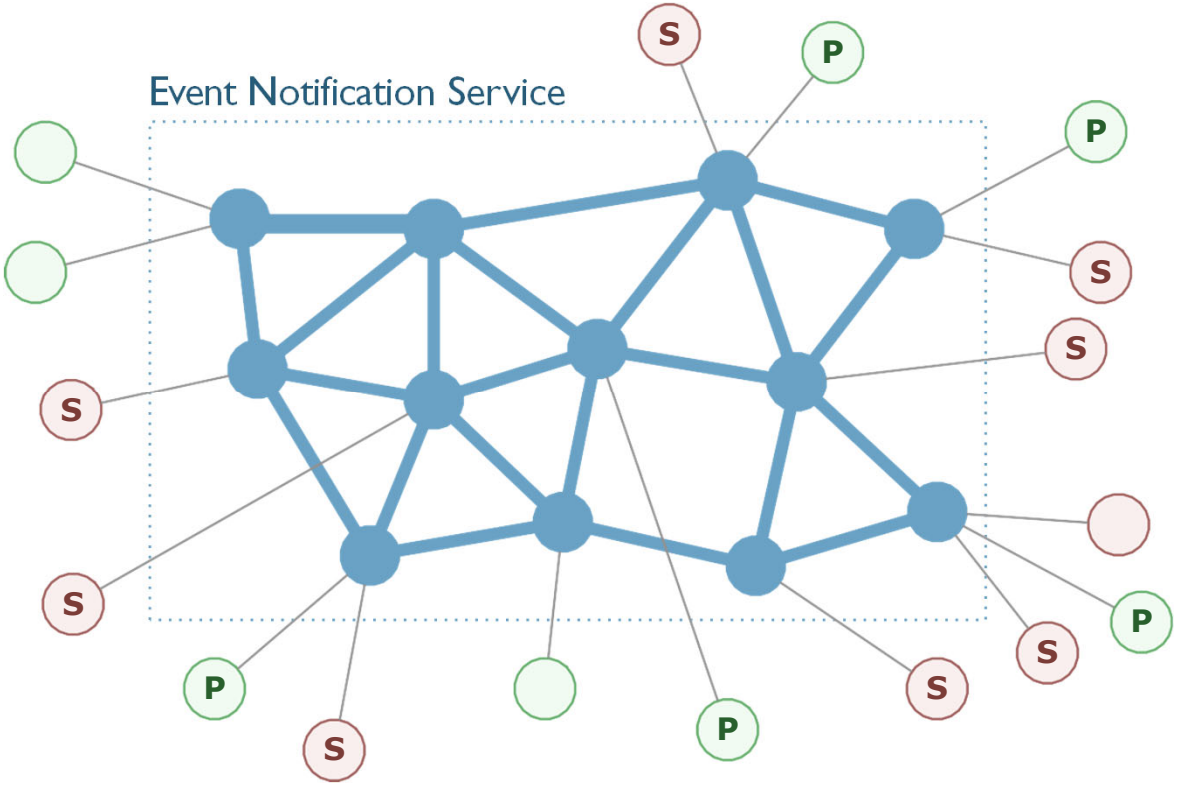
Contrarily to the previous model, here topics are organized in a hierarchical structure which express a notion of containment between topics. When a subscriber subscribe a topic, it will receive all the events published in that topic and in all the topics present in the corresponding sub-tree.



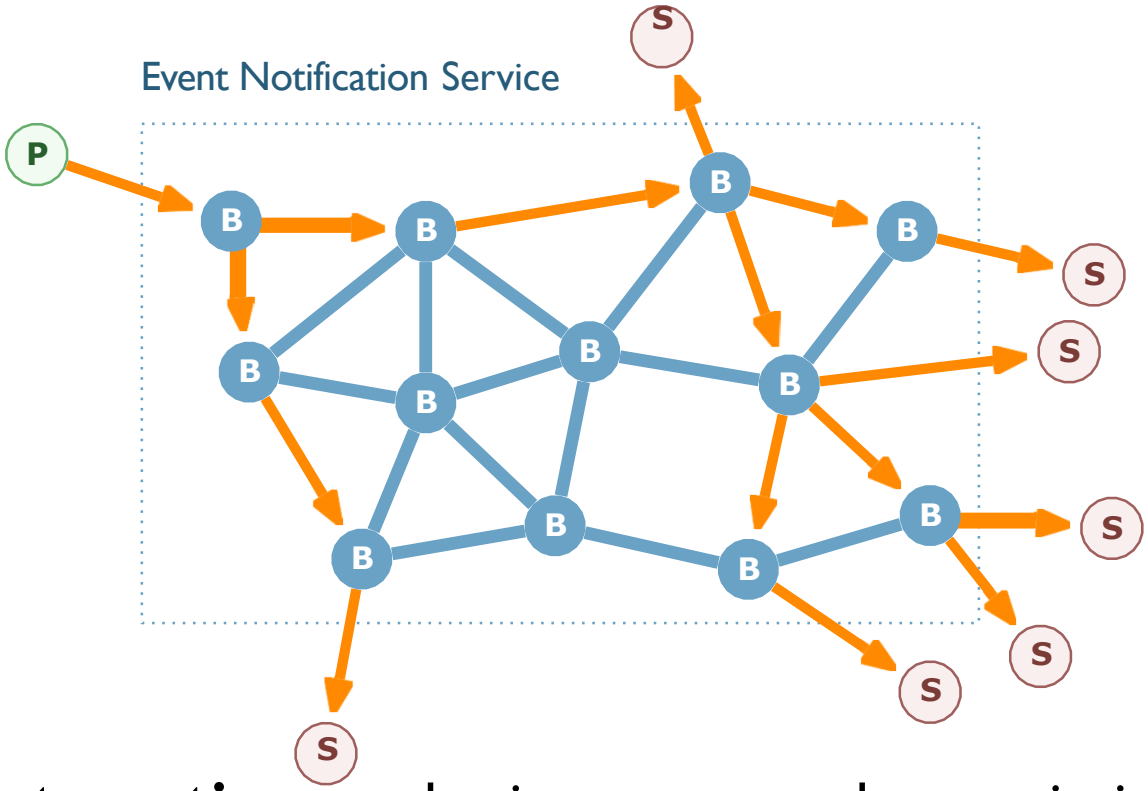
Content-based selection: all the data published in the system is mostly structured. Each subscription can be expressed as a conjunction of constrains expressed on attributes. The Event Notification Service filters out useless events before notifying a subscriber.



- 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.

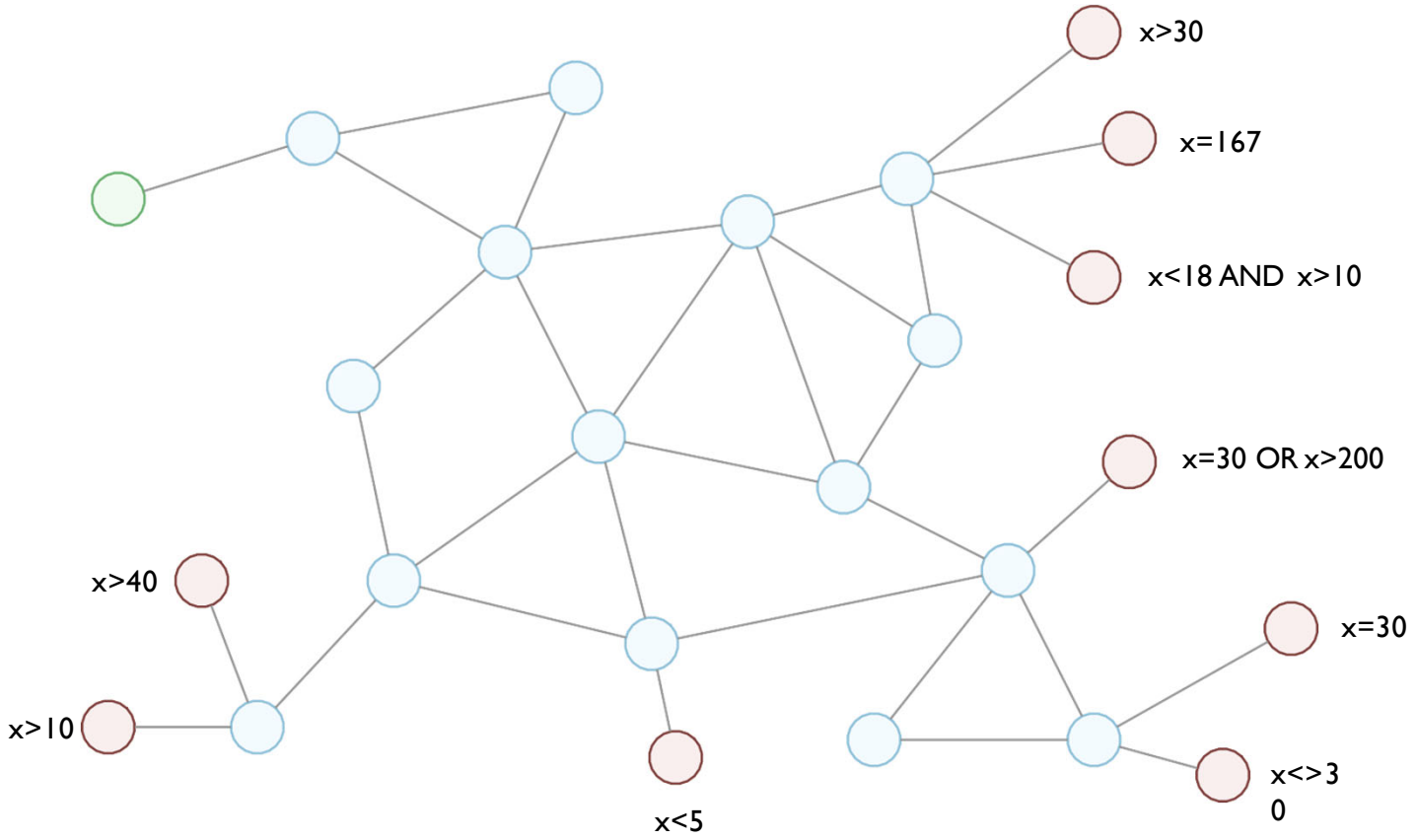


- An **event routing** mechanism routes each event inside the ENS from the broker where it is published to the broker(s) where it must be notified.

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

The implementation is straightforward but very expensive.

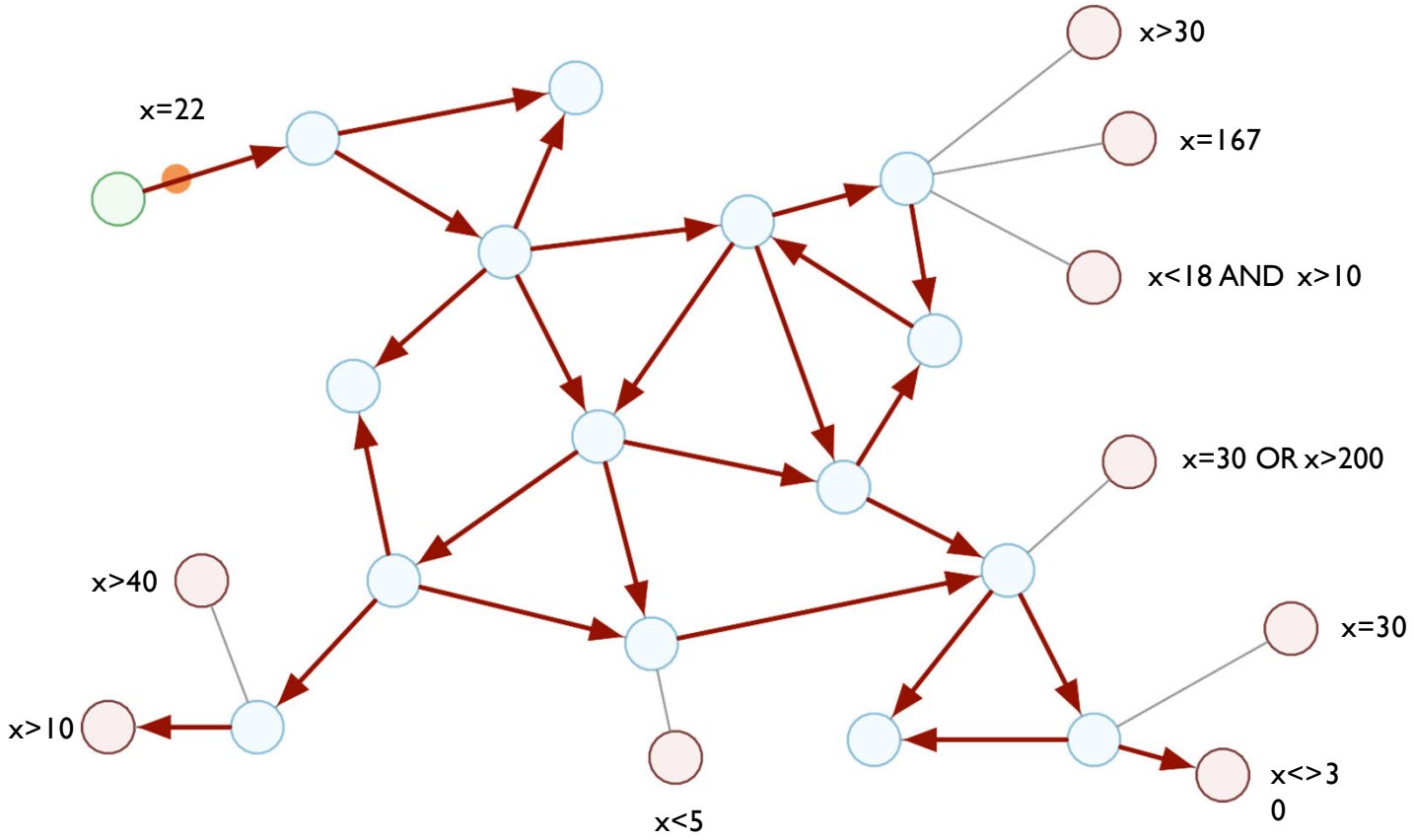
This solution has the highest message overhead with no memory overhead.



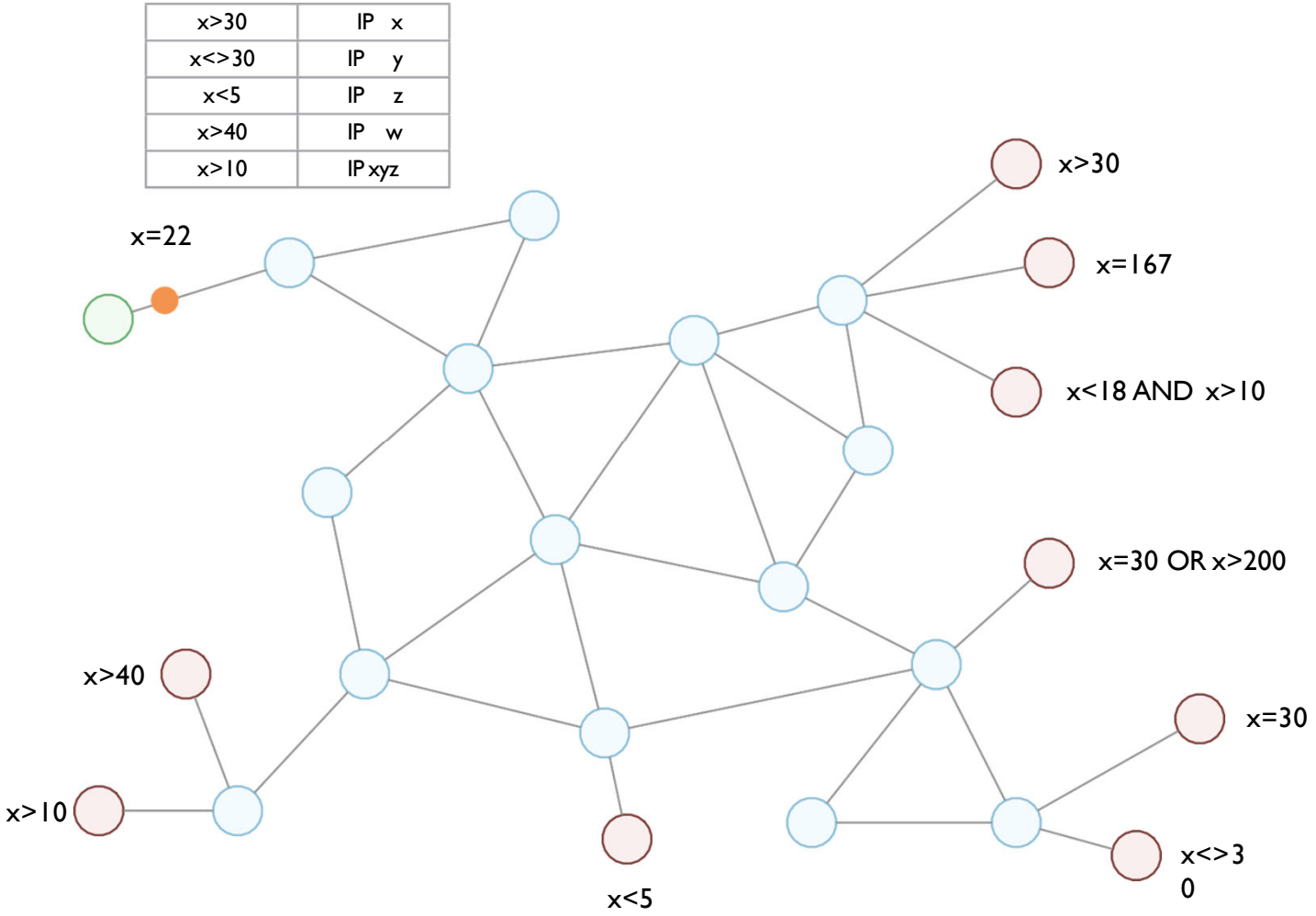
Event flooding: each event is broadcast from the publisher in the whole system.

The implementation is straightforward but very expensive.

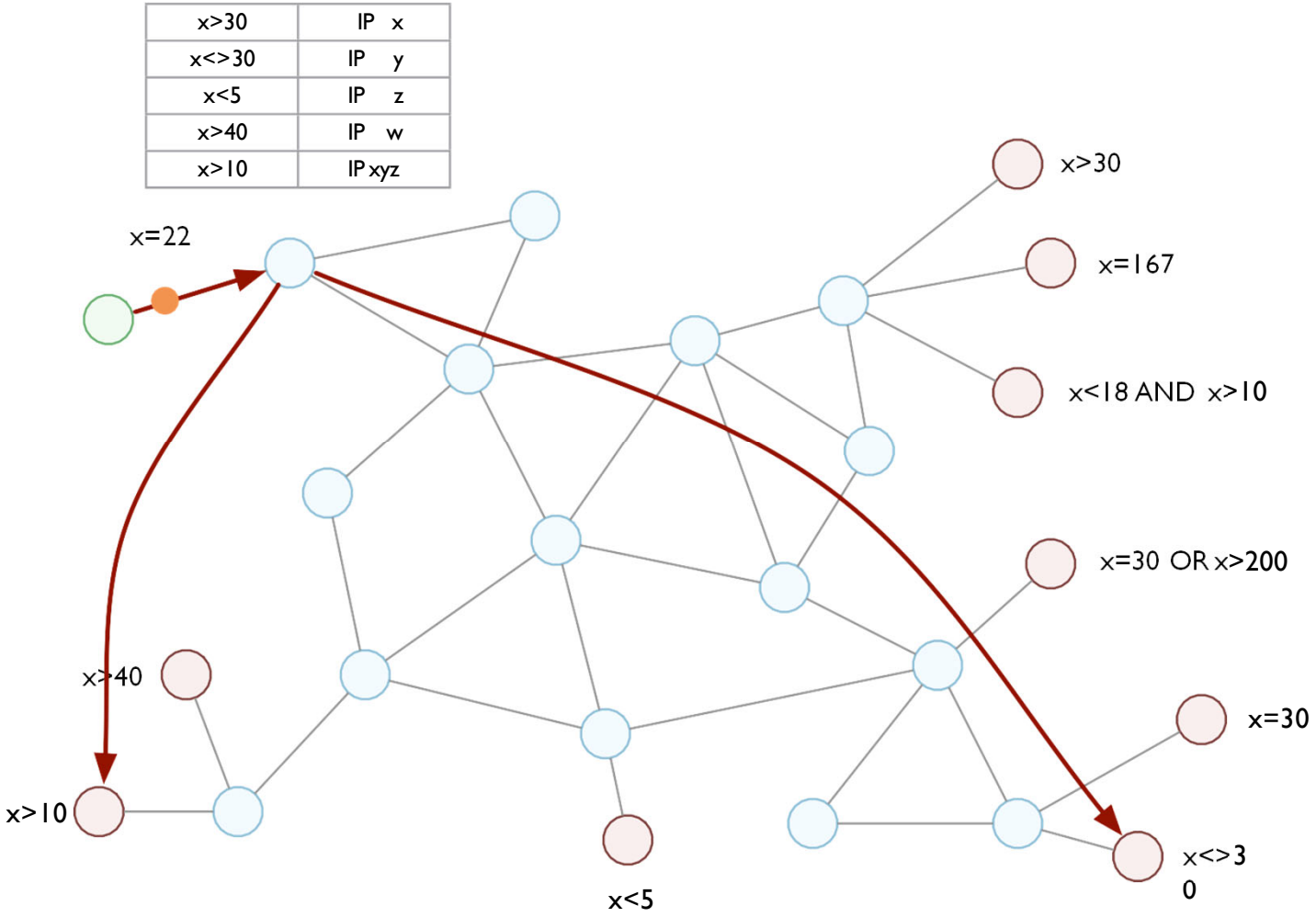
This solution has the highest message overhead with no memory overhead.



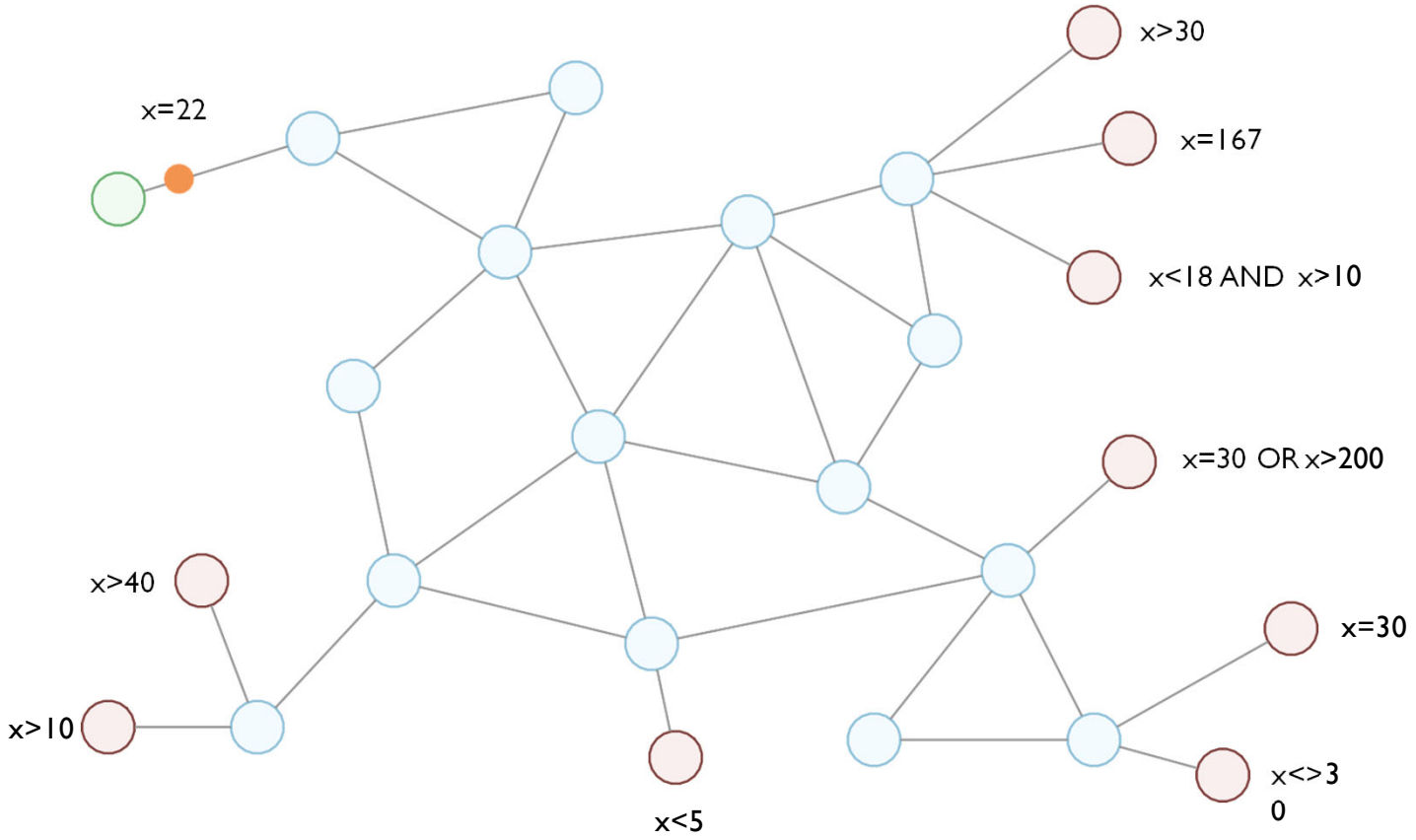
Subscription flooding: each subscription is copied on every broker, in order to build locally complete subscription tables. These tables are then used to locally match events and directly notify interested subscribers. This approach suffers from a large memory overhead, but event diffusion is optimal.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



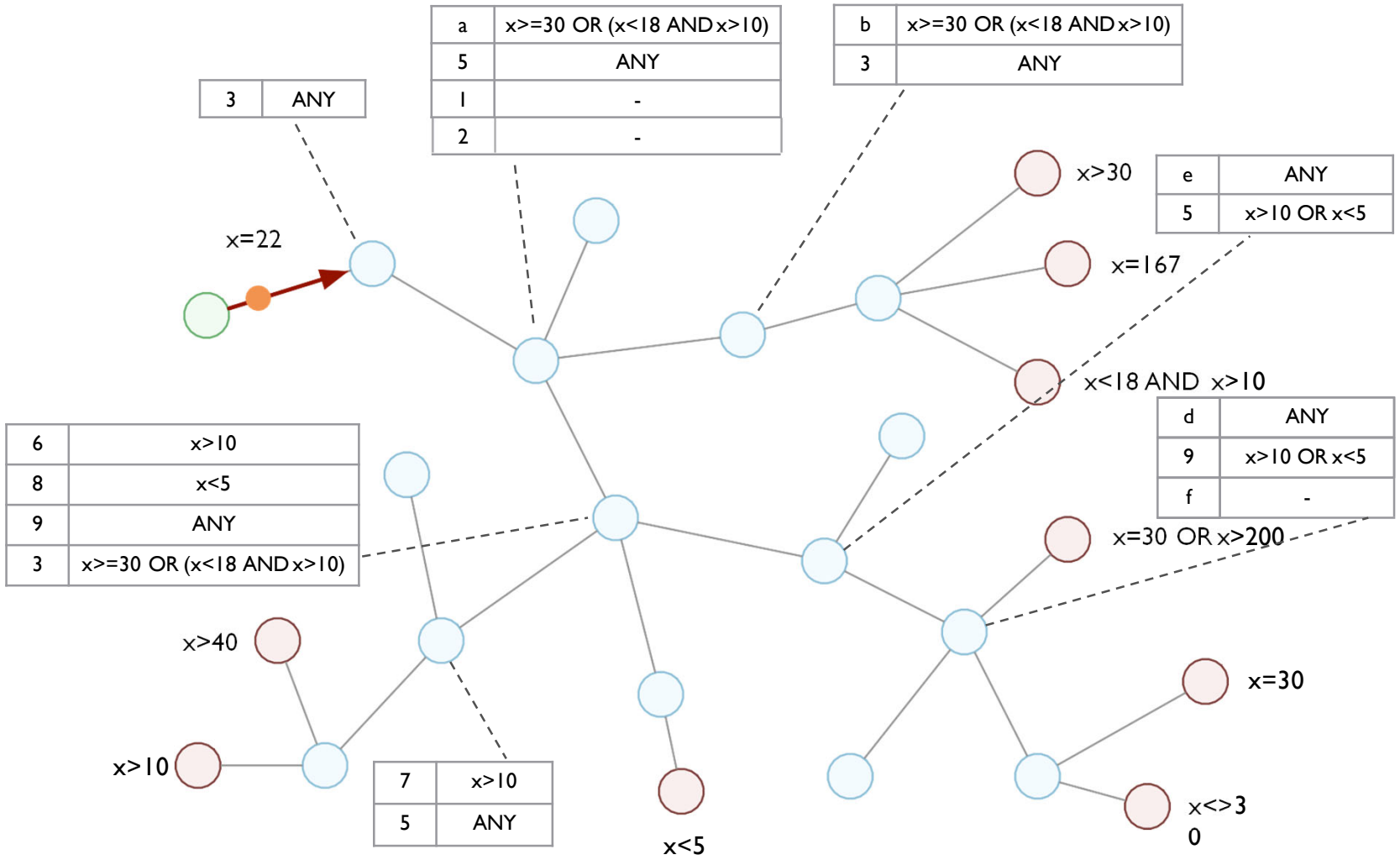
Subscription flooding: each subscription is copied on every broker, in order to build locally complete subscription tables. These tables are then used to locally match events and directly notify interested subscribers. This approach suffers from a large memory overhead, but event diffusion is optimal.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



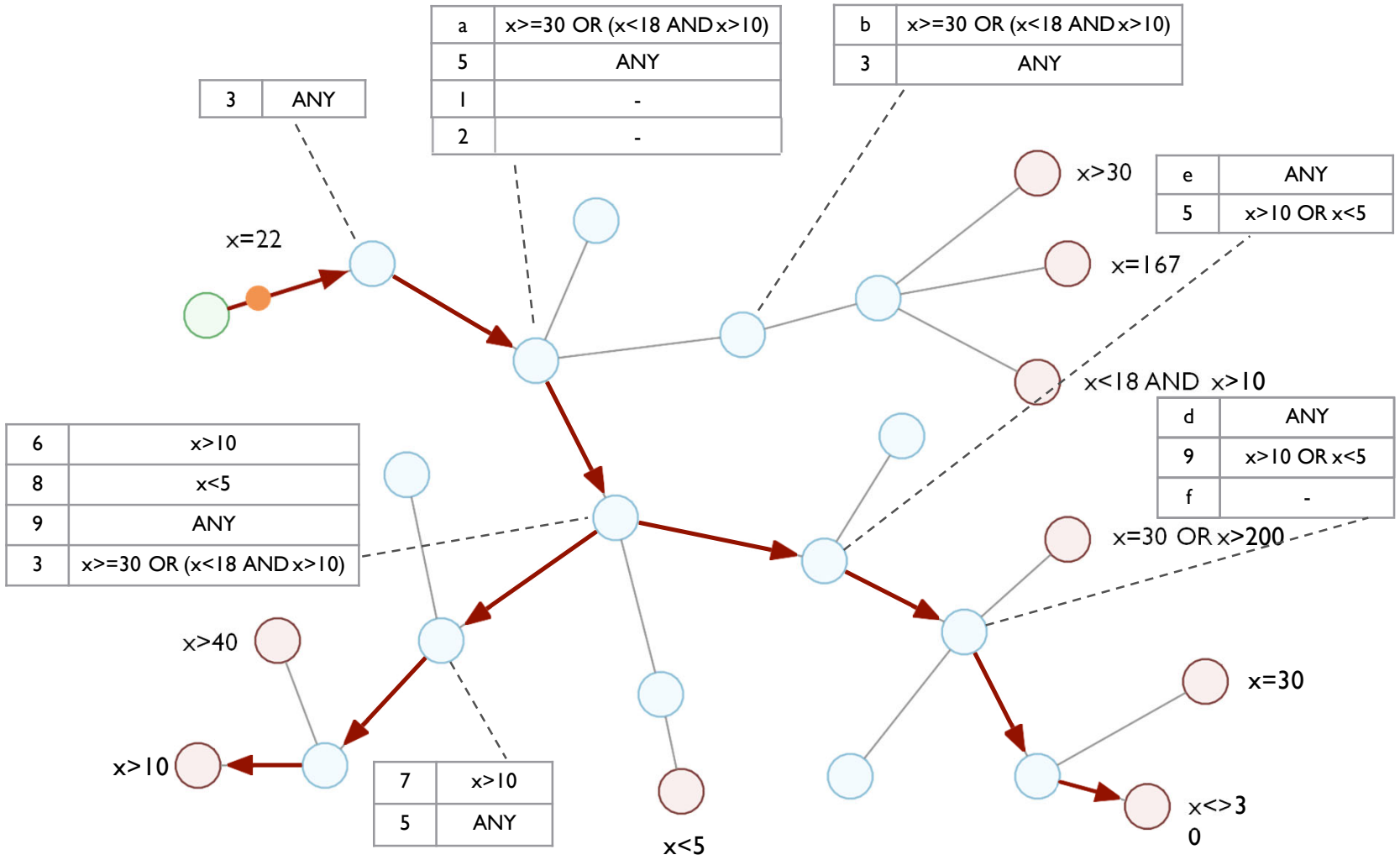
Filter-based routing: subscriptions are partially diffused in the system and used to build *routing tables*. These tables, are then exploited during event diffusion to dynamically build a multicast tree that (hopefully) connects the publisher to all, and only, the interested subscribers.



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.

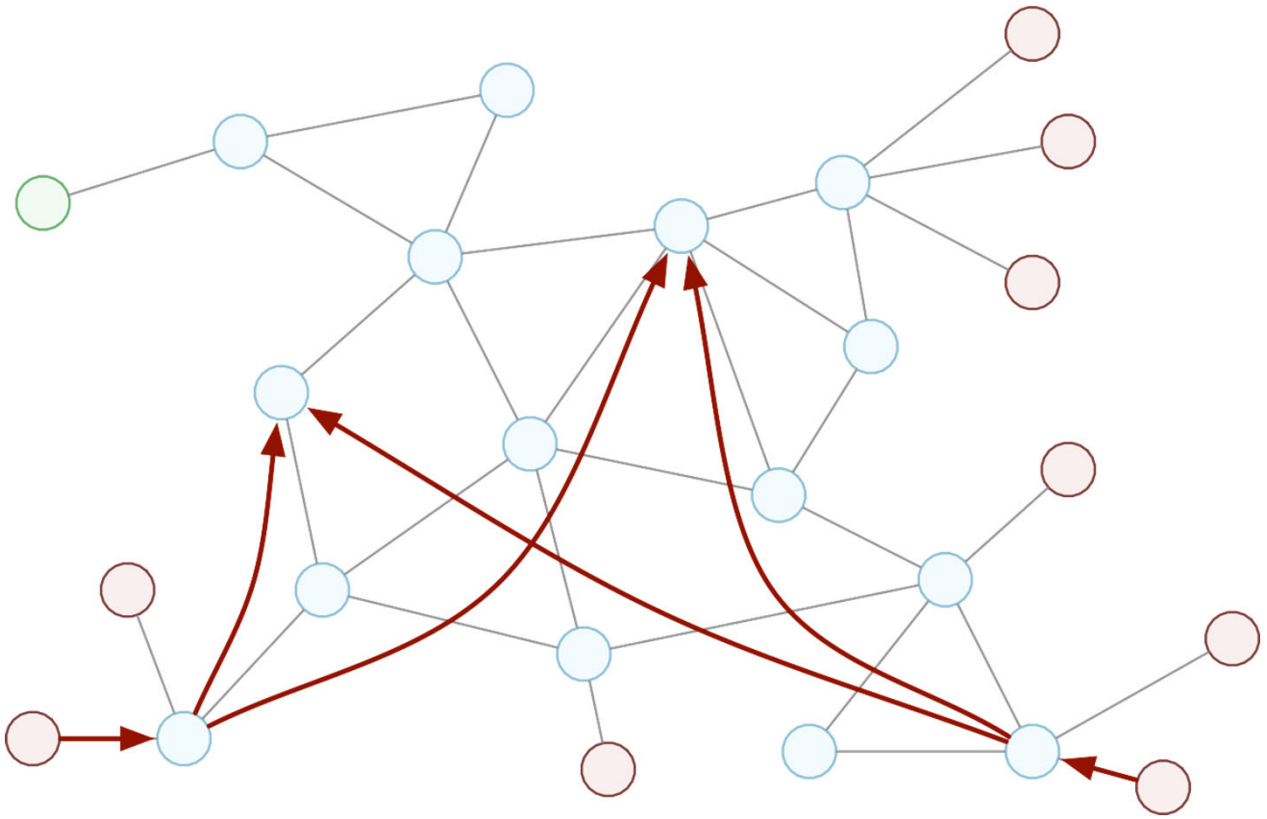
Given an event *e*, *EN(e)* returns a set of nodes which must receive *e* to match it against the subscriptions they store.

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.

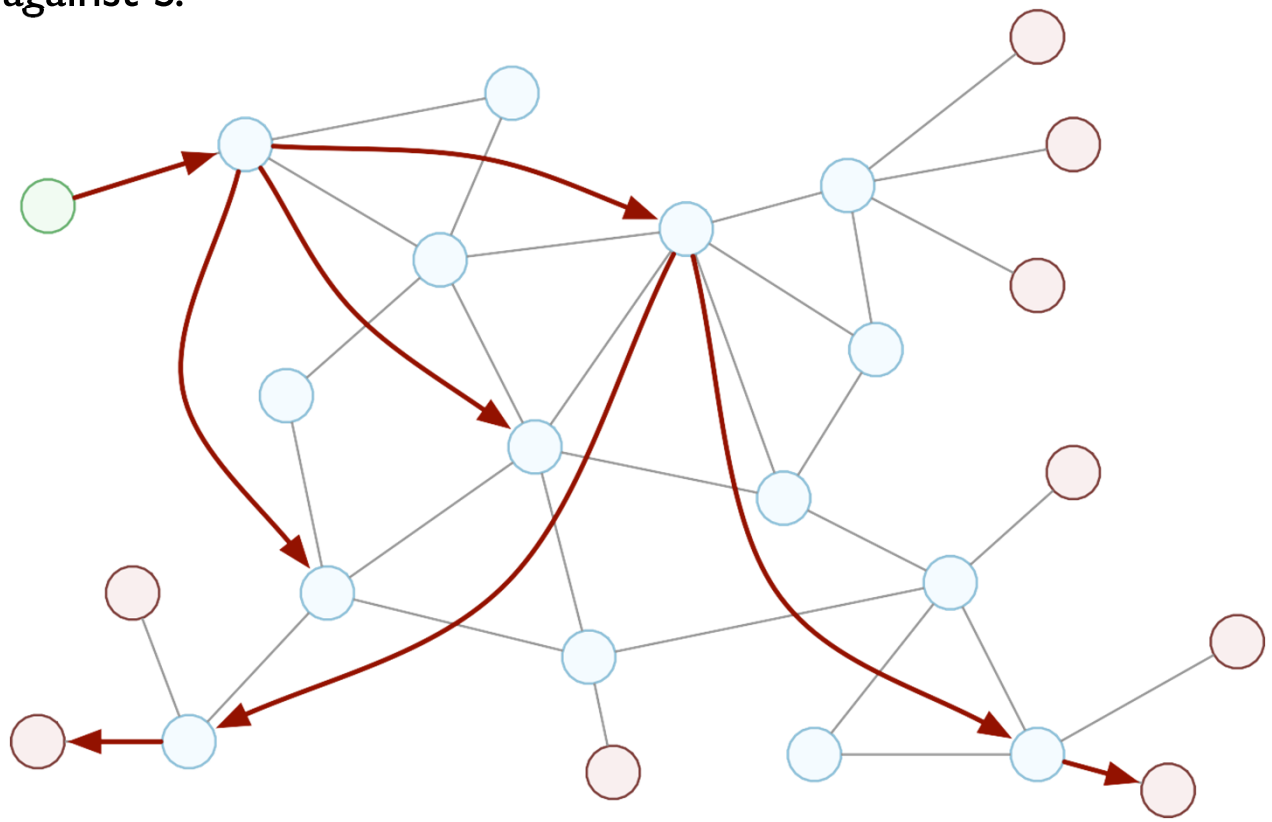
Phase 1: two nodes issue the same subscription *S*.



SN(S) = {4,a}

Rendez-Vous routing: example.

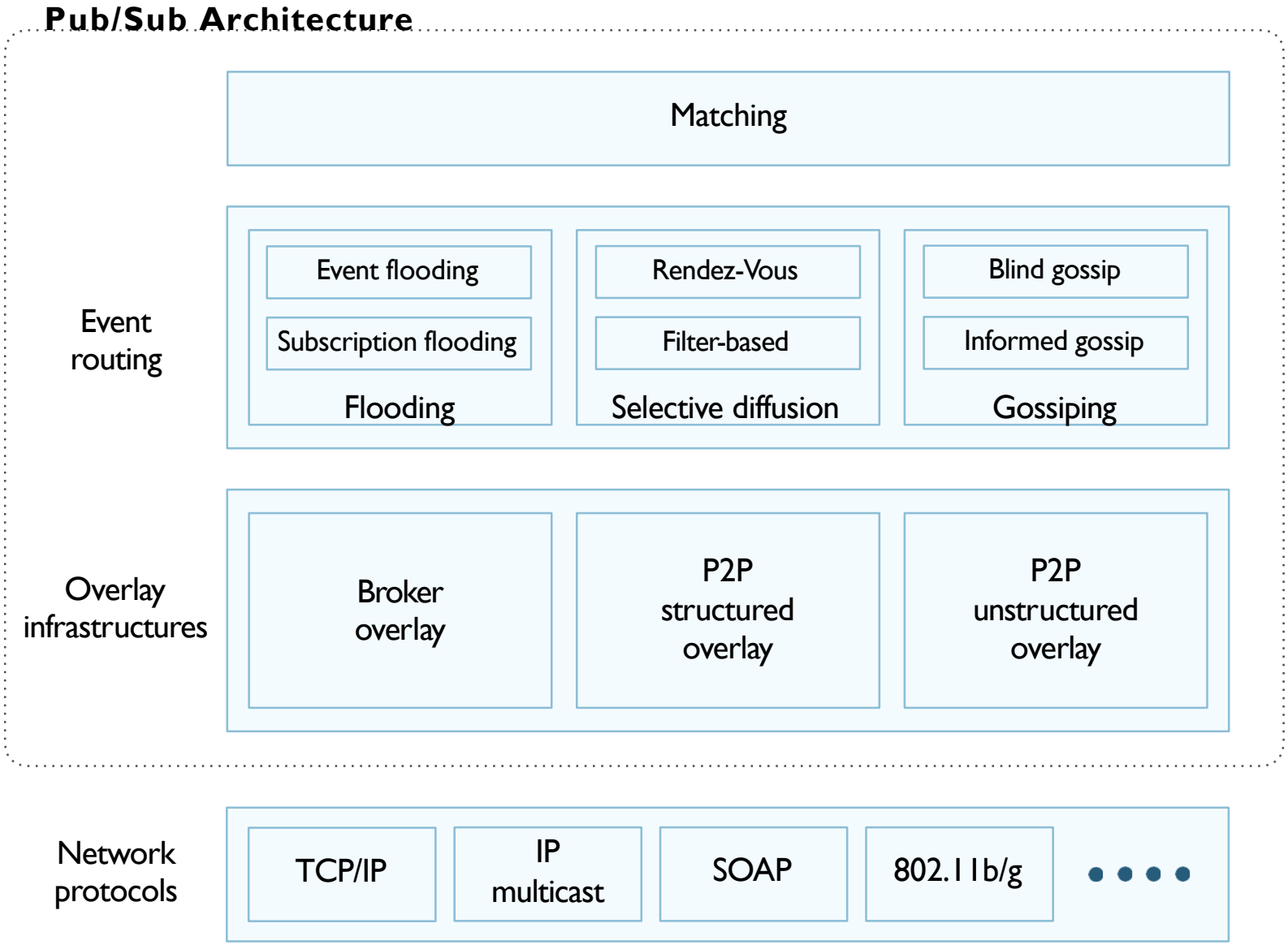
Phase 1I: an event e matching *S* is routed toward the rendez-vous node where it is matched against *S*.



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:

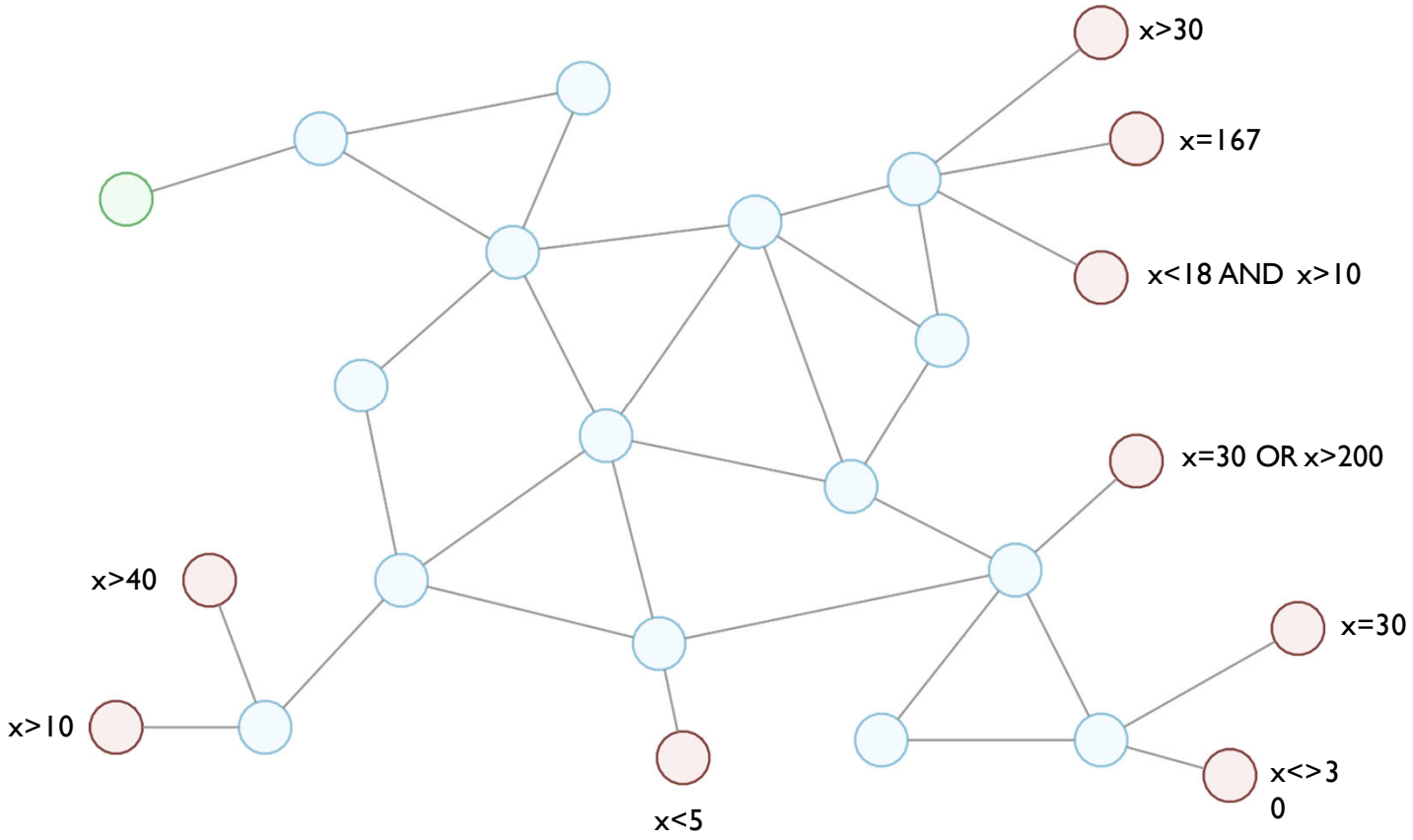


Combined Broadcast and Content-Based (CBCB) routing scheme.

Content-based layer: "prunes" broadcast forwarding paths

Broadcast layer: diffuses messages in the network

Overlay point-to-point network: manages connections

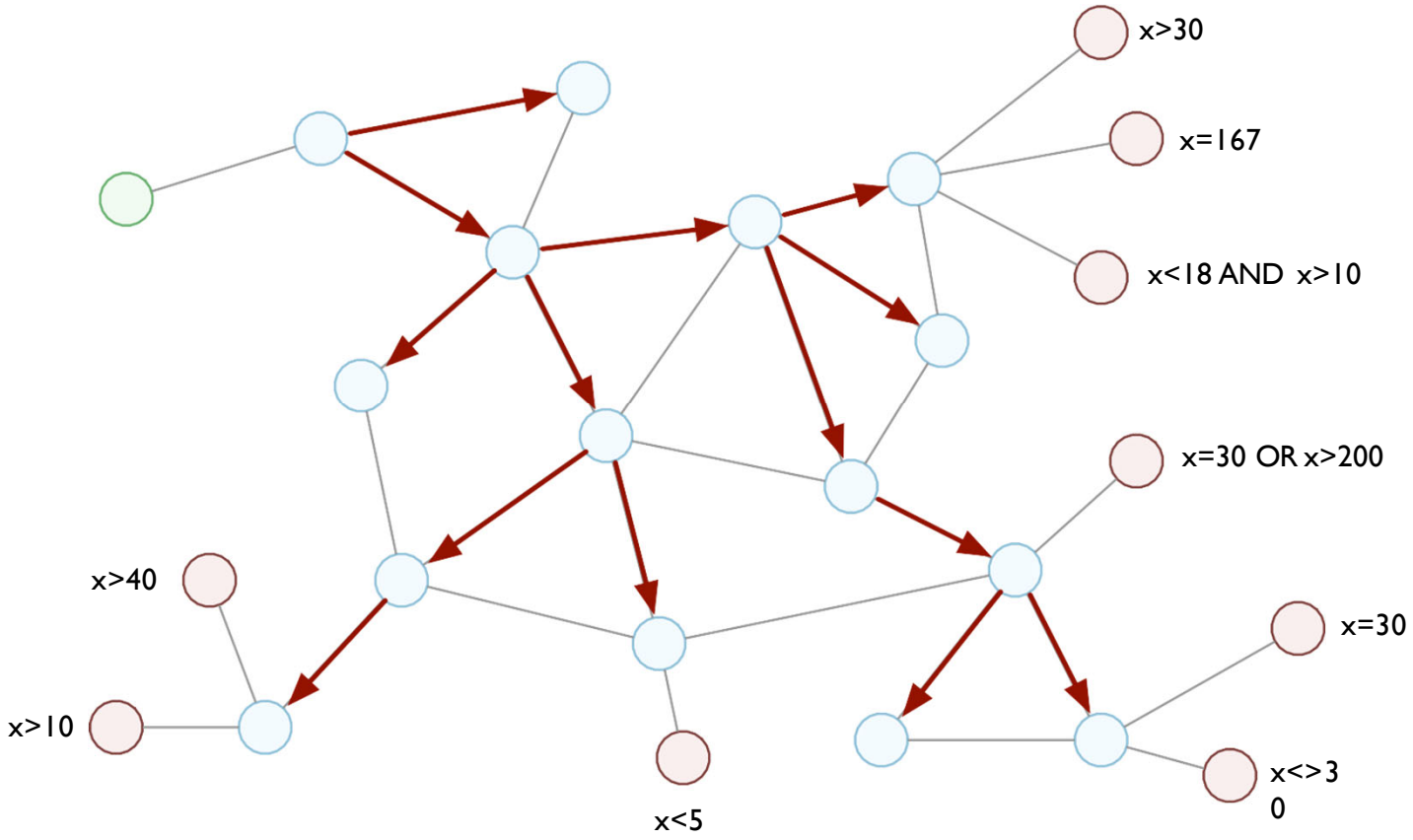


Combined Broadcast and Content-Based (CBCB) routing scheme.

Content-based layer: "prunes" broadcast forwarding paths

Broadcast layer: diffuses messages in the network

Overlay point-to-point network: manages connections

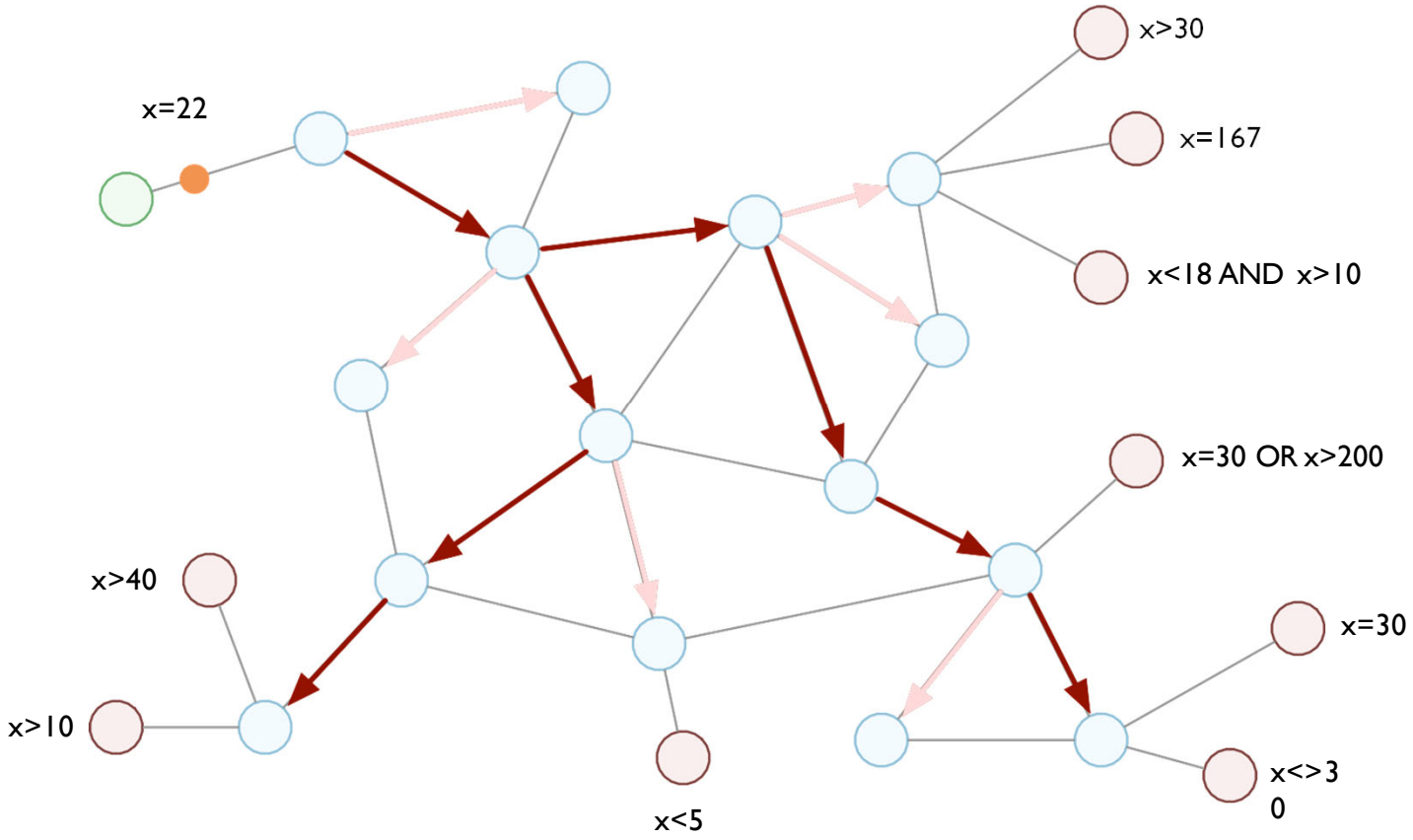


Combined Broadcast and Content-Based (CBCB) routing scheme.

Content-based layer: "prunes" broadcast forwarding paths

Broadcast layer: diffuses messages in the network

Overlay point-to-point network: manages connections



Application-level multicasting

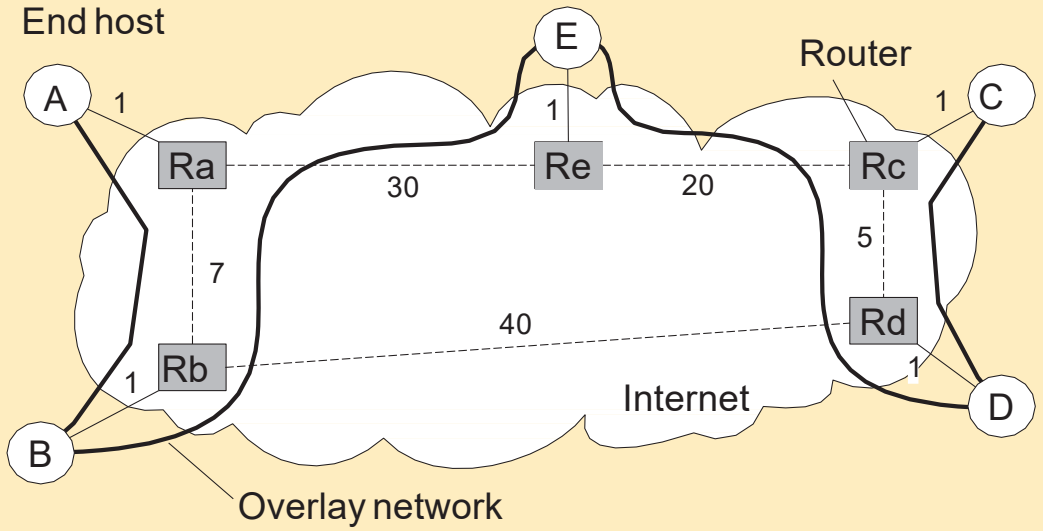
Essence

Organize nodes of a distributed system into an overlay network and use that network to disseminate data:

- Oftentimes a tree, leading to unique paths
- Alternatively, also mesh networks, requiring a form of routing
- Flooding
- Epidemic/Gossip: Anti-entropy
  - Rumor spreading

ALM: Some costs

Different metrics



Link stress: How often does an ALM message cross the same physical link?
Example: message from *A* to *D* needs to cross *(Ra, Rb)* twice.
Stretch: Ratio in delay between ALM-level path and network-level path.
Example: messages *B* to *C* follow path of length 73 at ALM, but 47 at network level ⇒ stretch = 73/47.

Flooding

Essence

P simply sends a message *m* to each of its neighbors. Each neighbor will forward that message, except to *P*, and only if it had not seen *m* before.

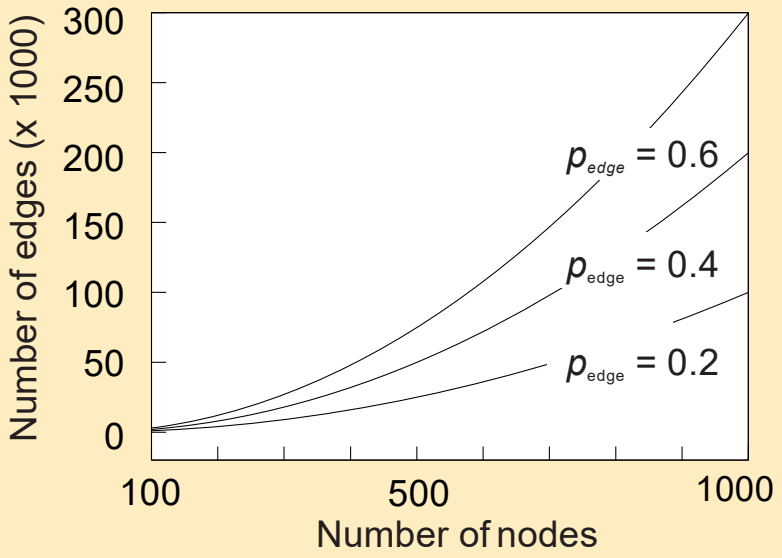
N nodes fully connected :

½ N (N-1) messages

N node connected with p_{edge}

½ p_{edge} N (N-1) messages

The size of a random overlay as function of the number of nodes



Variation

Let *Q* forward a message with a certain probability p_{flood} , possibly even dependent on its own number of neighbors (i.e., node degree) or the degree of its neighbors.

Epidemic (gossip) protocols

Assume there are no write–write conflicts

Update operations are performed at a single node

A node passes updated state to only a few neighbors

Update propagation is lazy, i.e., not immediate

Eventually, each update should reach every node

Two forms of epidemics

Anti-entropy: Each node regularly chooses another node at random, and exchanges state differences, leading to identical states at both afterwards

Rumor spreading: A node which has just been updated (i.e., has been contaminated), tells a number of other nodes about its update (contaminating them as well).

Anti-entropy

Principle operations

A node *P* selects another node *Q* from the system at random.

Pull: *P* only pulls in new updates from *Q*

Push: *P* only pushes its own updates to *Q*

Push-pull: *P* and *Q* send updates to each other

Observation

For push-pull it takes $O(log(N))$ rounds to disseminate updates to all *N* nodes (round = when every node has taken the initiative to start an exchange).

Anti-entropy: analysis

Basics

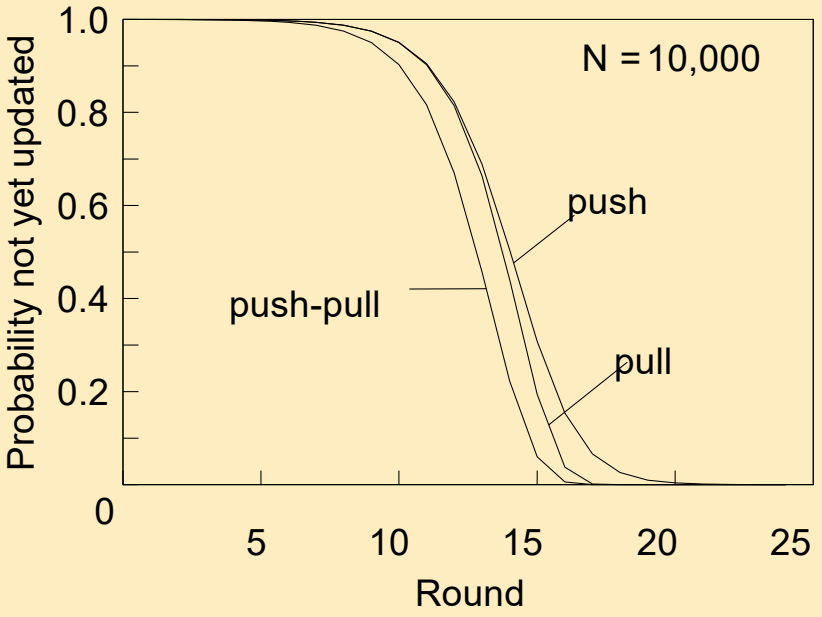
Consider a single source, propagating its update. Let p_i be the probability that a node has not received the update after the i^{th} round.

Analysis: staying ignorant

With pull, $p_{i+1} = (p_i)^2$: the node was not updated during the i^{th} round and should contact another ignorant node during the next round.

With push, $p_{i+1} = p_i\left(1 - \frac{1}{N}\right)^{N(1-p_i)} \approx p_i e^{-1}$ (for small p_i and large N): the node was ignorant during the i^{th} round and no updated node chooses to contact it during the next round.

With push-pull: $(p_i)^2 \cdot (p_i e^{-1})$



Rumor spreading

Basic model

A node N having an update to report, contacts other nodes. If a node is contacted to which the update has already propagated, N stops contacting other nodes with probability p_{stop}.

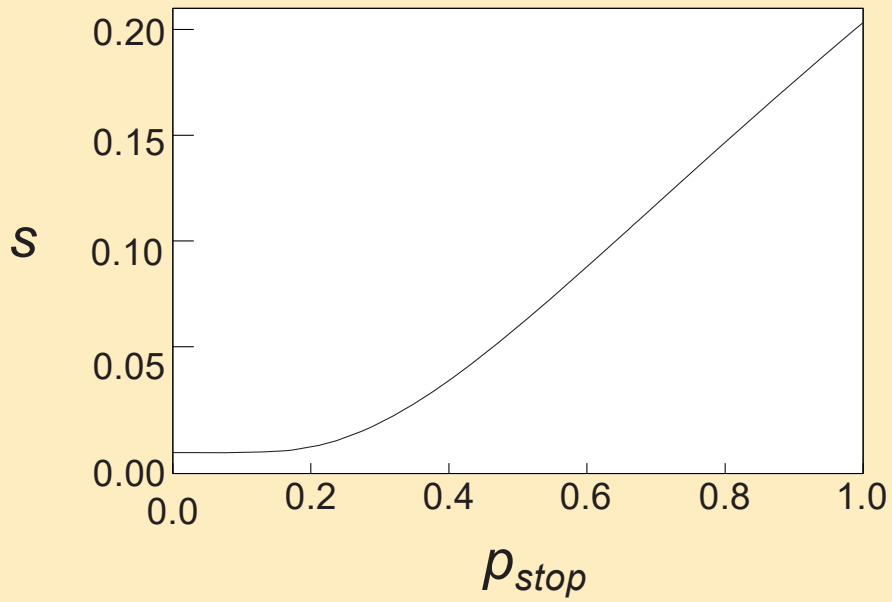
Observation

If s is the fraction of ignorant nodes (i.e., which are unaware of the update), it can be shown that with many nodes

$$s = e^{-(1/p_{stop}+1)(1-s)}$$

Rumor spreading

The effect of stopping



Consider 10,000 nodes		
$1/p_{stop}$	s	N_s
1	0.203188	2032
2	0.059520	595
3	0.019827	198
4	0.006977	70
5	0.002516	25
6	0.000918	9
7	0.000336	3

Note

If we really have to ensure that all nodes are eventually updated, rumor spreading alone is not enough

Deleting values

Fundamental problem

We cannot remove an old value from a server and expect the removal to propagate. Instead, mere removal will be undone in due time using epidemic algorithms

Solution

Removal has to be registered as a special update by inserting a death certificate

Deleting values

When to remove a death certificate (it is not allowed to stay for ever)

Run a global algorithm to detect whether the removal is known everywhere, and then collect the death certificates (looks like garbage collection)

Assume death certificates propagate in finite time, and associate a maximum lifetime for a certificate (can be done at risk of not reaching all servers)

Note

It is necessary that a removal actually reaches all servers.