Chapter -6

GROUP COMMUNICATION

Indirect communication

- Is communication between entities in a distributed system through an intermediary with no direct coupling between the sender and the receiver(s).
- The precise nature of the intermediary varies from approach to approach, as will be seen in the rest of this chapter.
- In addition, the precise nature of coupling varies significantly between systems, and again this will be brought out in the text that follows.
- Many indirect communication paradigms explicitly support one-to-many communication.
- Indirect communication is often used in distributed systems where change is anticipated – for example, in mobile environments where users may rapidly connect to and disconnect from the global network – and must be managed to provide more dependable services.
- Indirect communication is also exploited in key parts of the Google infrastructure.
- Indirect communication is also heavily used for event dissemination in distributed systems where the receivers may be unknown and liable to change – for example, in managing event feeds in financial systems

Two key properties stemming from the use of an intermediary:

- ***Space uncoupling,*** in which the sender does not know or need to know the identity of the receiver(s), and vice versa. Because of this space uncoupling, the system developer has many degrees of freedom in dealing with change: participants (senders or receivers) can be replaced, updated, replicated or migrated.

- ***Time uncoupling****,* in which the sender and receiver(s) can have independent lifetimes. In other words, the sender and receiver(s) do not need to exist at the same time to communicate. This has important benefits, for example, in more volatile environments where senders and receivers may come and go.

Figure 6.1 Space and time coupling in distributed systems

	Time-coupled	*Time-uncoupled*
Space coupling	*Properties*: Communication directed towards a given receiver or receivers; receiver(s) must exist at that moment in time *Examples*: Message passing, remote invocation (see Chapters 4 and 5)	*Properties*: Communication directed towards a given receiver or receivers; sender(s) and receiver(s) can have independent lifetimes *Examples*: See Exercise 6.3
Space uncoupling	*Properties*: Sender does not need to know the identity of the receiver(s); receiver(s) must exist at that moment in time *Examples*: IP multicast (see Chapter 4)	*Properties*: Sender does not need to know the identity of the receiver(s); sender(s) and receiver(s) can have independent lifetimes *Examples*: Most indirect communication paradigms covered in this chapter

The relationship with asynchronous communication

- Distinguishing between asynchronous communication and time uncoupling is important.

- In asynchronous communication, a sender sends a message and then continues (without blocking), and hence there is no need to meet in time with the receiver to communicate.

- Time uncoupling adds the extra dimension that the sender and receiver(s) can have independent existences; for example, the receiver may not exist at the time communication is initiated.

- Eugster *et al.* also recognize the important distinction between asynchronous communication (synchronization uncoupling) and time uncoupling.

- Many of the techniques examined in this chapter are time-uncoupled and asynchronous, but a few, such as the *MessageDispatcher* and *RpcDispatcher* operations in JGroups, offer a synchronous service over indirect communication.

6.2 Group communication

- *Group communication* offers a service whereby a message is sent to a group and then this message is delivered to all members of the group.

- In this action, the sender is not aware of the identities of the receivers. Group communication represents an abstraction over multicast communication and may be implemented over IP multicast or an equivalent overlay network, adding significant extra value in terms of managing group membership, detecting failures and providing reliability and ordering guarantees.

- With the added guarantees, group communication is to IP multicast what TCP is to the point-to-point service in IP.

- Group communication is an important building block for distributed systems, and particularly reliable distributed systems, with key areas of application including:

Group communication Scenarios

- The reliable dissemination of information to potentially large numbers of clients,including in the financial industry, where institutions require accurate and up-todate access to a wide variety of information sources.

- support for collaborative applications, where again events must be disseminated to multiple users to preserve a common user view – for example, in multiuser games.

- support for a range of fault-tolerance strategies, including the consistent update of replicated data or the implementation of highly available (replicated) servers.

- support for system monitoring and management, including for example load balancing strategies.

6.2.1 The programming model

- In group communication, the central concept is that of a *group* with associated *group membership*, whereby processes may *join* or *leave* the group.

- Processes can then send a message to this group and have it propagated to all members of the group with certain guarantees in terms of reliability and ordering.

- Thus, group communication implements *multicast* communication, in which a message is sent to all the members of the group by a single operation.

- Communication to *all* processes in the system, as opposed to a subgroup of them, is known as *broadcast*, whereas communication to a single process is known as *unicast*.

- The essential feature of group communication is that a process issues only one multicast operation to send a message to each of a group of processes (in Java this operation is *aGroup.send(aMessage)*) instead of issuing multiple send operations to individual processes.

Benefits of single multicast operation over multiple send operations

- Convenience for the programmer
- it enables the implementation to be efficient in its utilization of bandwidth.
- It can take steps to send the message no more than once over any communication link, by sending it over a distribution tree; and it can use network hardware support for multicast where this is available.
- The implementation can also minimize the total time taken to deliver the message to all destinations, as compared with transmitting it separately and serially.
- The use of a single multicast operation is also important in terms of delivery guarantees. If a process issues multiple independent send operations to individual processes, then there is no way for the implementation to provide guarantees that affect the group of processes as a whole. If the sender fails halfway through sending, then some members of the group may receive the message while others do not.
- In addition, the relative ordering of two messages delivered to any two group members is undefined.
- Group communication, however, has the potential to offer a range of guarantees in terms of reliability and ordering

Process groups

- *process groups* are groups where the communicating entities are processes. Such services are relatively low-level.

- Messages are delivered to processes and no further support for dispatching is provided.

- Messages are typically unstructured byte arrays with no support for marshalling of complex data types

- The level of service provided by process groups is therefore similar to that of sockets

Object groups

- *object groups* provide a higher-level approach to group computing. An object group is a collection of objects (normally instances of the same class) that process the same set of invocations concurrently, with each returning responses.
- Client objects need not be aware of the replication.

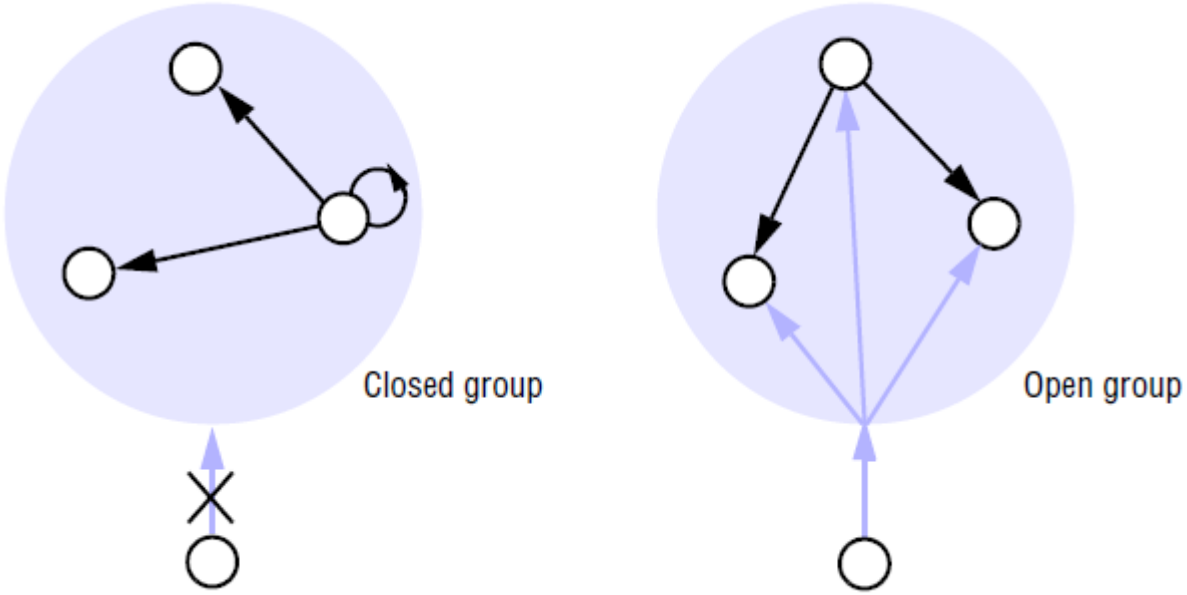
- They invoke operations on a single, local object, which acts as a proxy for the group. The proxy uses a group communication system to send the invocations to the members of the object group.

- Object parameters and results are marshalled as in RMI and the associated calls are dispatched automatically to the right destination objects/methods
- Electra is a CORBA-compliant system that supports object groups.

- Electra uses an extension of the standard CORBA Object Request Broker interface, with functions for creating and destroying object groups and managing their membership.
- Eternal and the Object Group Service also provide CORBA-compliant support for object groups.

- Despite the promise of object groups, however, process groups still dominate in terms of usage. For example, the popular Jgroups.

Closed and open groups



A group is said to be *closed* if only members of the group may multicast to it.
A process in a closed group delivers to itself any message that it multicasts to the group.

A group is *open* if processes outside the group may send to it.

(The categories ‘open’ and ‘closed’ also apply with analogous meanings to mailing lists.) Closed groups of processes are useful, for example, for cooperating servers to send messages to one another that only they should receive.

Open groups are useful, for example, for delivering events to groups of interested processes.

Overlapping and non-overlapping groups:

- In *overlapping groups*, entities (processes or objects) may be members of multiple groups, and non-overlapping groups imply that membership does not overlap (that is, any process belongs to at most one group).

- Note that in real-life systems, it is realistic to expect that group membership will overlap.

Synchronous and asynchronous systems: There is a requirement to consider group communication in both environments.

6.2.2 Implementation issues

We now turn to implementation issues for group communication services, discussing the properties of the underlying multicast service in terms of reliability and ordering and also the key role of group membership management in dynamic environments, where processes can join and leave or fail at any time.

- **Reliability and ordering in multicast**
- **Group membership management**

Reliability and ordering in multicast

- In group communication, all members of a group must receive copies of the messages sent to the group, generally with delivery guarantees.
- The guarantees include agreement on the set of messages that every process in the group should receive and on the delivery ordering across the group members.

- Group communication systems are extremely sophisticated. Even IP multicast, which provides minimal delivery guarantees, requires a major engineering effort.
- Reliability in one-to-one has two properties: integrity (the message received is the same as the one sent, and no messages are delivered twice) and validity (any outgoing message is eventually
- delivered).

- The interpretation for *reliable multicast* builds on these properties, with *integrity* defined the same way in terms of delivering the message correctly at most once, and *validity* guaranteeing that a message sent will eventually be delivered.
- To extend the semantics to cover delivery to multiple receivers, a third property is added – that of *agreement*, stating that if the message is delivered to one process, then it is delivered to all processes in the group.

- As well as reliability guarantees, group communication demands extra guarantees in terms of the relative ordering of messages delivered to multiple destinations.

- Ordering is not guaranteed by underlying interprocess communication primitives. For example, if multicast is implemented by a series of one-to-one messages, they may be subject to arbitrary delays.

- Similar problems may occur if using IP multicast. To counter this, group communication services offer *ordered multicast*, with the option of one or more of the following properties (with hybrid solutions also possible):*FIFO ordering, Causal ordering, Total ordering*

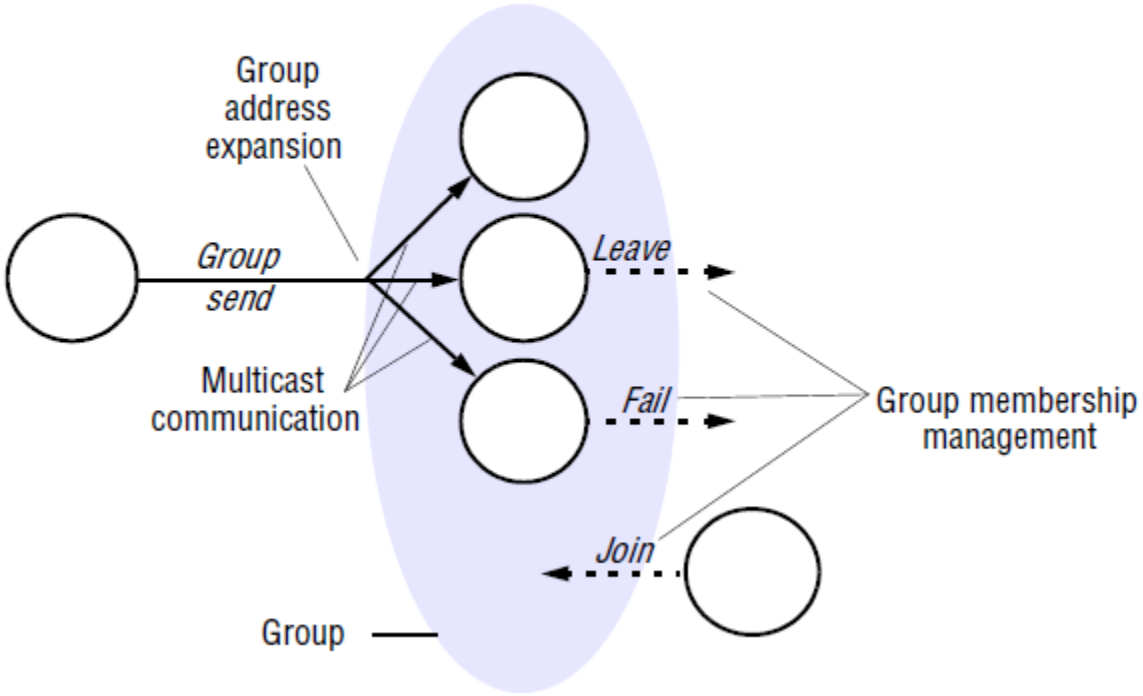
- ***FIFO ordering*:** First-in-first-out (FIFO) ordering (also referred to as source ordering) is concerned with preserving the order from the perspective of a sender process, in that if a process sends one message before another, it will be delivered in this order at all processes in the group.

- ***Causal ordering*:** Causal ordering takes into account causal relationships between messages, in that if a message *happens before* another message in the distributed system this so-called causal relationship will be preserved in the delivery of the associated messages at all processes .

- ***Total ordering*:** In total ordering, if a message is delivered before another message at one process, then the same order will be preserved at all processes.

Group membership management

Figure 6.3 The role of group membership management



This diagram illustrates the important role of group membership management in maintaining an accurate *view* of the current membership, given that entities may join, leave or indeed.

A group membership service has four main tasks:

- ***Providing an interface for group membership changes*:** The membership service provides operations to create and destroy process groups and to add or withdraw a process to or from a group. In most systems, a single process may belong to several groups at the same time.

- ***Failure detection*:** The service monitors the group members not only in case they should crash, but also in case they should become unreachable because of a communication failure. The detector marks processes as *Suspected* or *Unsuspected*. The service uses the failure detector to reach a decision about the group's membership: it excludes a process from membership if it is suspected to have failed or to have become unreachable.

- ***Notifying members of group membership changes*:** The service notifies the group's members when a process is added, or when a process is excluded (through failure or when the process is deliberately withdrawn from the group).

- ***Performing group address expansion*:** When a process multicasts a message, it supplies the group identifier rather than a list of processes in the group. The membership management service expands the identifier into the current group membership for delivery. The service can coordinate multicast delivery with membership changes by controlling address expansion. That is, it can decide consistently where to deliver any given message, even though the membership may be changing during delivery.

- **IP multicast** is a weak case of a group membership service, with some but not all of these properties. It does allow processes to join or leave groups dynamically and it performs address expansion, so that senders need only provide a single IP multicast address as the destination for a multicast message. But IP multicast does not itself provide group members with information about current membership, and multicast delivery is not coordinated with membership changes. Achieving these properties is complex and requires what is known as *view-synchronous group communication*.

- In general, the need to maintain group membership has a significant impact on the utility of group-based approaches. In particular, group communication is most effective in small-scale and static systems and does not operate as well in larger-scale environments or environments with a high degree of volatility.

- This can be traced to the need for a form of synchrony assumption.

- A more probabilistic approach to group membership design is to operate in more large-scale and dynamic environments, using an underlying gossip protocol.

6.2.3 Case study: the JGroups toolkit

- **JGroups - A Toolkit for Reliable Messaging** : http://www.jgroups.org/index.html

- JGroups is a toolkit for reliable messaging. It can be used to create clusters whose nodes can send messages to each other. The main features include
- ❑ Cluster creation and deletion. Cluster nodes can be spread across LANs or WANs Joining and leaving of clusters
- ❑ Membership detection and notification about joined/left/crashed cluster nodes
- ❑ Detection and removal of crashed nodes
- ❑ Sending and receiving of node-to-cluster messages (point-to-multipoint)
- ❑ Sending and receiving of node-to-node messages (point-to-point)

- The most powerful feature of JGroups is its flexible protocol stack, which allows developers to adapt it to exactly match their application requirements and network characteristics.

- The benefit of this is that you only pay for what you use. By mixing and matching protocols, various differing application requirements can be satisfied.

- JGroups comes with a large number of protocols (but anyone can write their own), for example
  - Transport protocols: UDP (IP Multicast) or TCP
  - Fragmentation of large messages
  - Reliable unicast and multicast message transmission. Lost messages are retransmitted
  - Failure detection: crashed nodes are excluded from the membership
  - Flow control to prevent slow receivers to get overrun by fast senders
  - Ordering protocols: FIFO, Total Order
  - Membership
  - Encryption
  - Compression

- JGroups allows developers to create reliable messaging (one-to-one or one-to-many) applications where *reliability* is a deployment issue, and does not have to be implemented by the application developer. This saves application developers significant amounts of time, and allows for the application to be deployed in different environments, without having to change code.

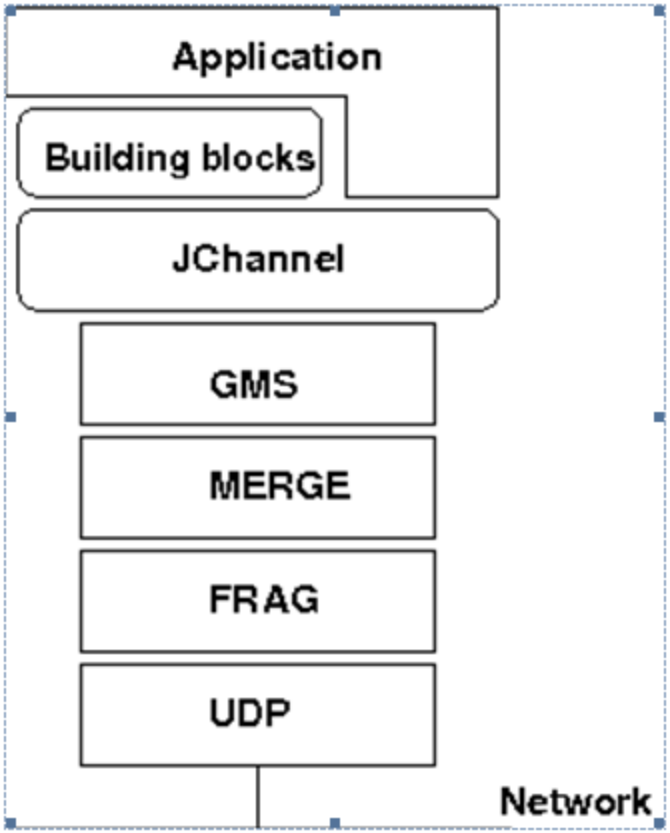
Architecture of JGroups

It consists of 3 parts:

(1) the Channel used by application programmers to build reliable group communication applications,

(2) the building blocks, which are layered on top of the channel and provide a higher abstraction level and

(3) the protocol stack, which implements the properties specified for a given channel.



Channel

- To join a group and send messages, a process has to create a *channel* and connect to it using the group name (all channels with the same name form a group). The channel is the handle to the group.

- While connected, a member may send and receive messages to/from all other group members. The client leaves a group by disconnecting from the channel. A channel can be reused.

- Each channel has a unique address. Channels always know who the other members are in the same group: a list of member addresses can be retrieved from any channel. This list is called a *view*.

Building Blocks

- JGroups offers building blocks that provide more sophisticated APIs on top of a Channel. Building blocks either create and use channels internally, or require an existing channel to be specified when creating a building block.

- Applications communicate directly with the building block, rather than the channel. Building blocks are intended to save the application programmer from having to write tedious and recurring code.

The Protocol Stack

- The protocol stack containins a number of protocol layers in a bidirectional list. All messages sent and received over the channel have to pass through all protocols. Every layer may modify, reorder, pass or drop a message, or add a header to a message.

- A fragmentation layer might break up a message into several smaller messages, adding a header with an id to each fragment, and re-assemble the fragments on the receiver's side.

- This shows a protocol that consists of five layers

- The layer referred to as UDP is the most common transport layer in JGroups. Note that, despite the name, this is not entirely quivalent to the UDP protocol; rather, the layer utilizes IP multicast for sending to all members in a group and UDP datagrams specifically for point-to-point communication. This layer therefore assumes that IP multicast is available.

- FRAG implements message packetization and is configurable in terms of the maximum message size (8,192 bytes by default).

- MERGE is a protocol that deals with unexpected network partitioning and the subsequent merging of subgroups after the partition. A series of alternative merge layers are actually available, ranging from the simple to ones that deal with, for example, state transfer

- GMS implements a group membership protocol to maintain consistent views of membership across the group.

- CAUSAL implements causal ordering.

- A wide range of other protocol layers are available, including protocols for FIFO and total ordering, for membership discovery and failure detection, for encryption of messages and for implementing flow-control strategies .

- Note that because all layers implement the same interface, they can be combined in any order, although many of the resultant protocol stacks would not make sense. All members of a group must share the same protocol stack.

6.3 Publish-subscribe systems

- *Also referred as distributed event-based systems.*

- A publish-subscribe system is a system where *publishers* publish structured events to an event service and *subscribers* express interest in particular events through *subscriptions* which can be arbitrary patterns over the structured events.

- For example, a subscriber could express an interest in all events related to this textbook, such as the availability of a new edition or updates to the related web site.

- The task of the publishsubscribe system is to match subscriptions against published events and ensure the correct delivery of *event notifications*.
- A given event will be delivered to potentially many subscribers, and hence publish-subscribe is fundamentally a one-to-many communications paradigm.

Applications of publish-subscribe systems

Publish-subscribe systems are used in a wide variety of application domains, particularly those related to the large-scale dissemination of events. Examples include:

- financial information systems;
- other areas with live feeds of real-time data (including RSS feeds);
- support for cooperative working, where a number of participants need to be informed of events of shared interest;
- support for ubiquitous computing, including the management of events emanating from the ubiquitous infrastructure (for example, location events);
- a broad set of monitoring applications, including network monitoring in the Internet.

Publish-subscribe is also a key component of Google's infrastructure, including for example the dissemination of events related to advertisements, such as 'ad clicks', tointerested parties.

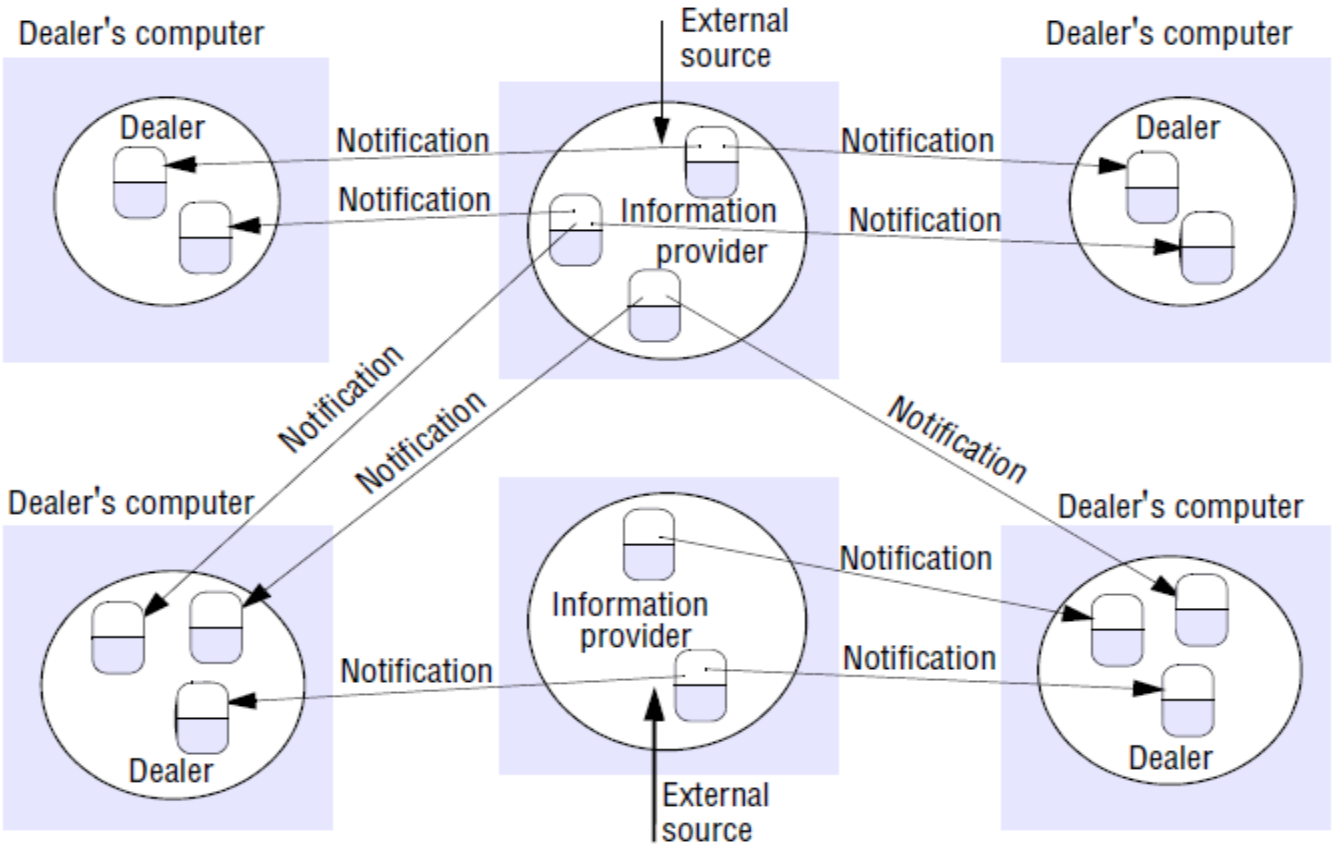
Dealing Room System

- Consider a simple dealing room system whose task is to allow dealers using computers to see the latest information about the market prices of the stocks they deal in. The market price for a single named stock is represented by an associated object.
- The information arrives in the dealing room from several different external sources in the form of updates to some or all of the objects representing the stocks and is collected by processes we call *information providers*.
- Dealers are typically interested only in their own specialist stocks. A dealing room system could be implemented by processes with two different tasks:

An information provider process continuously receives new trading information from a single external source. Each of the updates is regarded as an event. The information provider publishes such events to the publish-subscribe system for delivery to all of the dealers who have expressed an interest in the corresponding stock. There will be a separate information provider process for each external source.

- A dealer process creates a subscription representing each named stock that the user asks to have displayed. Each subscription expresses an interest in events related to a given stock at the relevant information provider. It then receives all the information sent to it in notifications and displays it to the user

Figure 6.7 Dealing room system



Characteristics of publish-subscribe systems

- ***Heterogeneity*** *:* When event notifications are used as a means of communication, components in a distributed system that were not designed to interoperate can be made to work together. All that is required is that event-generating objects publish the types of events they offer, and that other objects subscribe to patterns of events and provide an interface for receiving and dealing with the resultant notifications.

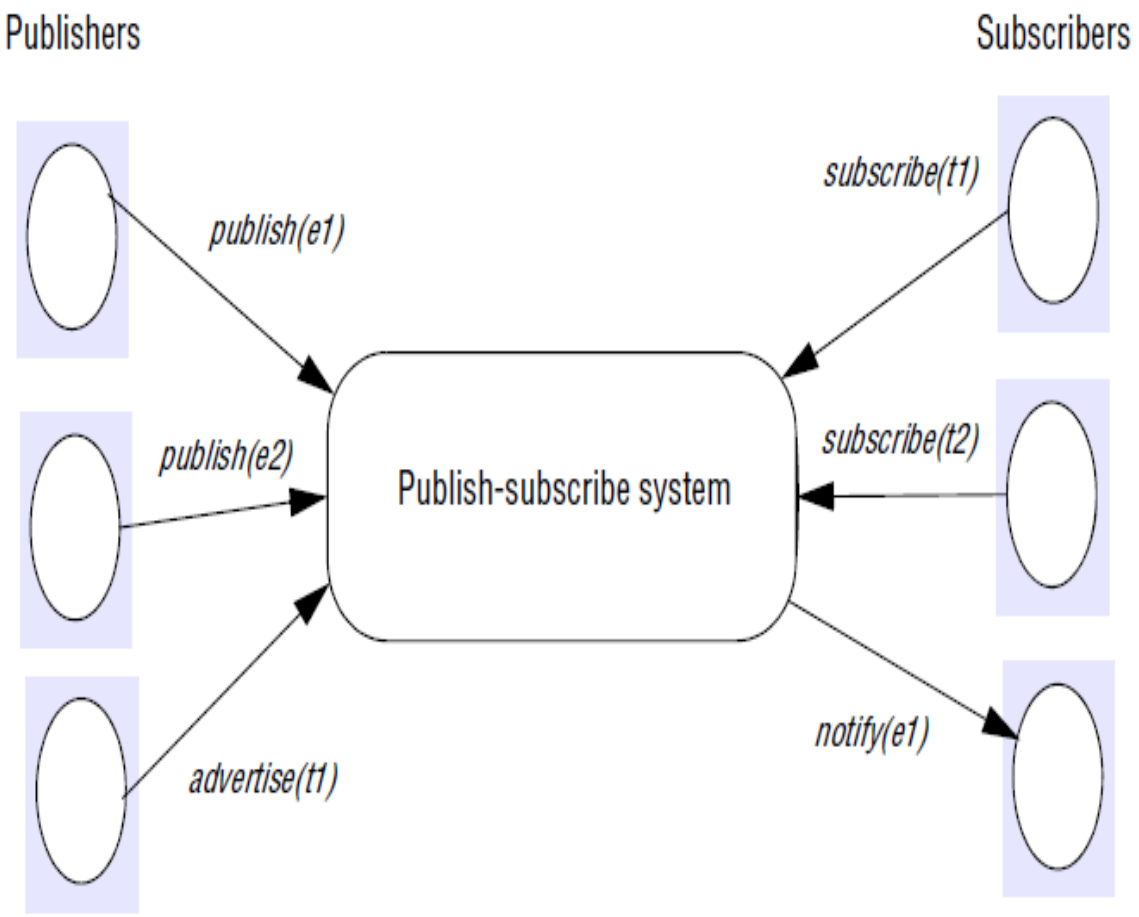
- ***Asynchronicity-*** Notifications are sent synchronously by event-generating publishers to all the subscribers that have expressed an interest in them to prevent publishers needing to synchronize with subscribers – publishers and subscribers need to be decoupled. A variety of different *delivery guarantees* can be provided for notifications

Mushroom [Kindberg *et al.* 1996]

- It is an object-based publishsubscribe system designed to support collaborative work, in which the user interface displays objects representing users and information objects such as documents and notepads within shared workspaces called *network places*.
- The state of each place is replicated at the computers of users currently in that place. Events are used to describe changes to objects and to a user's focus of interest.
- For example, an event could specify that a particular user has entered or left a place or has performed a particular action on an object. Each replica of any object to which particular types of events are relevant expresses an interest in them through a subscription and receives notifications when they occur.
- But subscribers to events are decoupled from objects experiencing events, because different users are active at different times
- In the Mushroom system mentioned above, notifications about changes in object state are delivered reliably to a server, whose responsibility it is to maintain up-to-date copies of objects.

6.3.1 The programming model of Publish- Subscribe Systems

Figure 6.8 The publish-subscribe paradigm



is based on a small set of Operations :

subscribe(f)
unsubscribe(f)
notify(e)
advertise(f)
unadvertise(f)

- Publishers disseminate an event *e* through a ***publish(e)*** operation and subscribers express an interest in a set of events through subscriptions.
- In particular, they achieve this through a ***subscribe(f)*** operation where ***f*** refers to a filter – that is, a pattern defined over the set of all possible events.
- The expressiveness of filters (and hence of subscriptions) is determined by the **subscription model.**
- Subscribers can later revoke this interest through a corresponding ***unsubscribe(f)*** operation.
- When events arrive at a subscriber, the events are delivered using a ***notify(e)*** operation.
- Some systems complement the above set of operations by introducing the concept of advertisements. With advertisements, publishers have the option of declaring the nature of future events through an ***advertise(f)*** operation.
- The advertisements are defined in terms of the types of events of interest (these happen to take the same form as filters).
- Hence, subscribers declare their interests in terms of subscriptions and publishers optionally declare the styles of events they will generate through advertisements.
- Advertisements can be revoked through a call of ***unadvertise(f)*.**

Subscription (filter) model of PS Systems

The expressiveness of publish-subscribe systems is determined by the subscription (filter) model, with a number of schemes defined and considered here in increasing order of sophistication:

- *Channel-based*
- *Topic-based (also referred to as subject-based)*:
- *Content-based*
- *Type-based*

***Channel-based*:**

- In this approach, publishers publish events to named channels and subscribers then subscribe to one of these named channels to receive all events sent to that channel.
- This is a rather primitive scheme and the only one that defines a physical channel.

***Topic-based (also referred to as subject-based)*:**

- In this approach, we make the assumption that each notification is expressed in terms of a number of fields, with one field denoting the topic.
- Subscriptions are then defined in terms of the topic of interest. This approach is equivalent to channel-based approaches, with the difference that topics are implicitly defined in the case of channels but explicitly declared as one of the fields in topic-based approaches.
- The expressiveness of topic based approaches can also be enhanced by introducing hierarchical organization of topics.

***Content-based*:**

- Content-based approaches are a generalization of topic-based approaches allowing the expression of subscriptions over a range of fields in an event notification. More specifically, a content-based filter is a query defined in terms of compositions of constraints over the values of event attributes.
- The sophistication of the associated query languages varies from system to system, but in general this approach is significantly more expressive than channel- or topic-based approaches, but with significant new implementation challenges

***Type-based*:**

- These approaches are intrinsically linked with object-based approaches where objects have a specified type. In type-based approaches, subscriptions are defined in terms of types of events and matching is defined in terms of types or subtypes of the given filter. This approach can express a range of filters, from coarsegrained filtering based on overall type names to more fine-grained queries defining attributes and methods of a given object.
- Such fine-grained filters are similar in expressiveness to content-based approaches. The advantages of type-based approaches are that they can be integrated elegantly into programming languages and they can check the type correctness of subscriptions, eliminating some kinds of subscription errors.

6.3.2 Implementation issues of PS DS

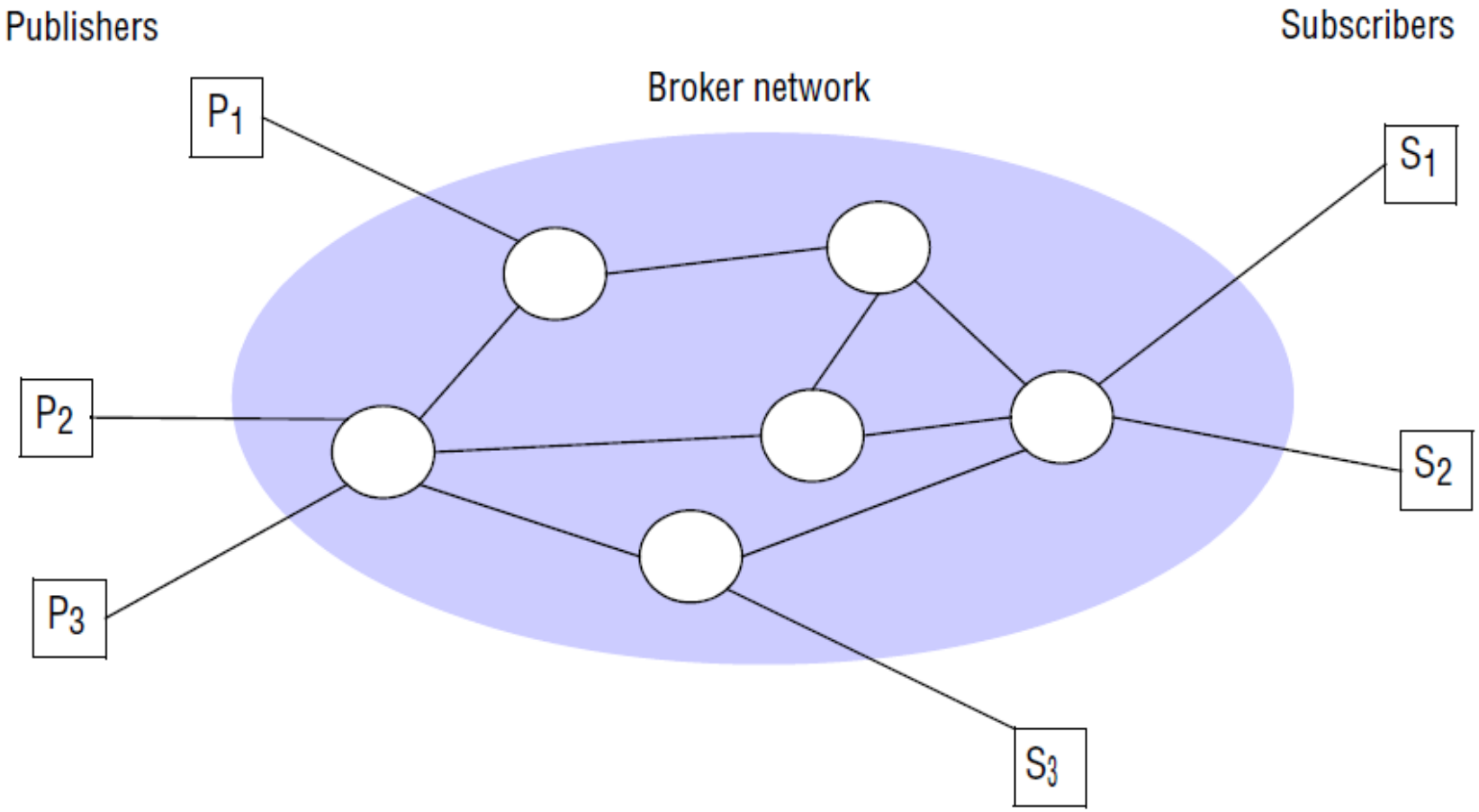
- The task of a publish-subscribe system is to ensure that events are delivered efficiently to all subscribers that have filters defined that match the event. Added to this, there may be additiona requirements in terms of security, scalability, failure handling, concurrency and quality of service.
- This makes the implementation of publish-subscribe systems rather complex, to implement publish-subscribe systems (particularly distributed implementations of content-based approaches).

- **Centralized versus distributed implementations**

A number of architectures for the implementation of publish-subscribe systems have been identified. The simplest approach is to centralize the implementation in a single node with a server on that node acting as an event broker.

Publishers then publish events (and optionally send advertisements) to this broker, and subscribers send subscriptions to the broker and receive notifications in return.

Figure 6.9 A network of brokers



Interaction with the broker is then through a series of point-to-point messages; this can be implemented using message passing or remote invocation.

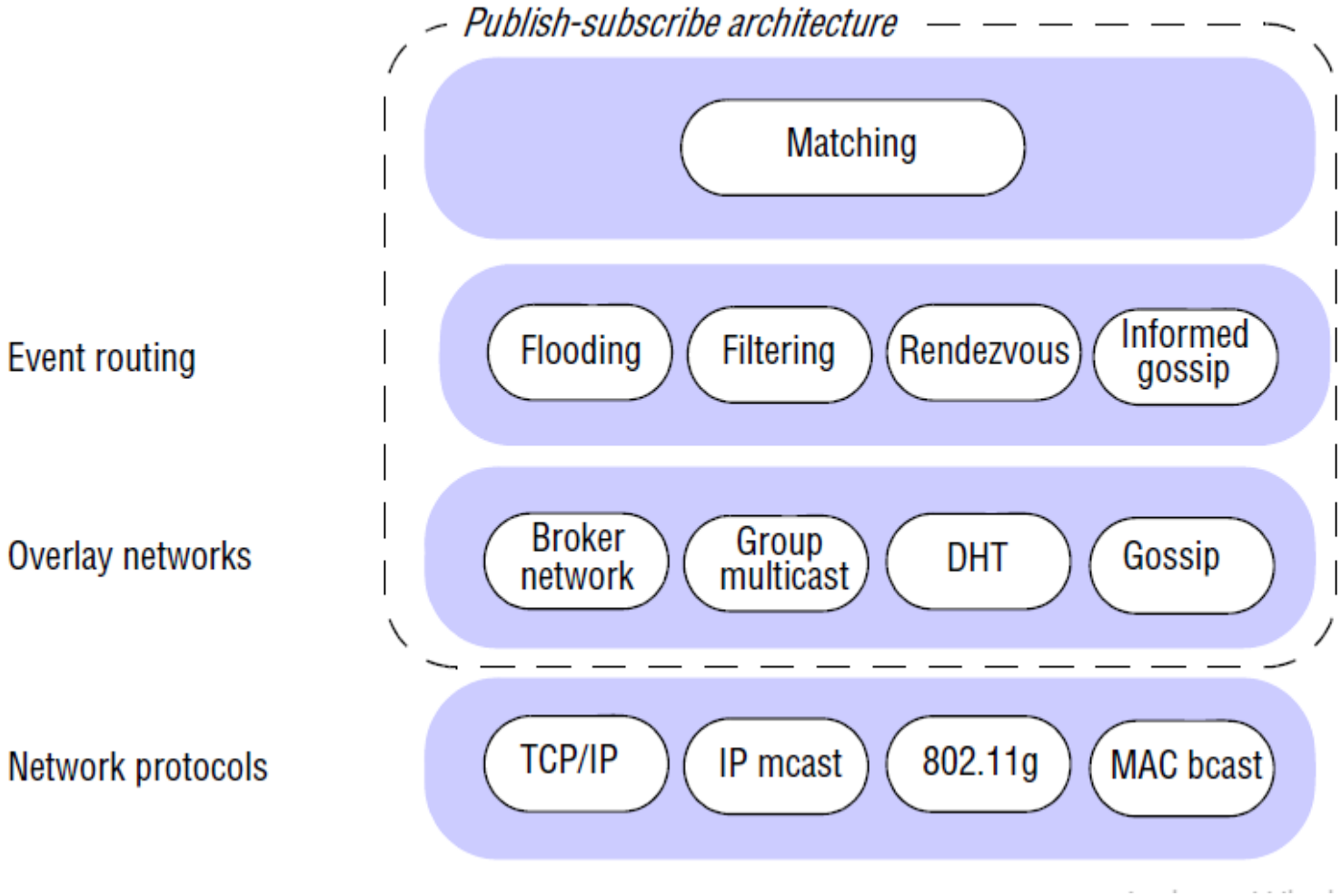
- This approach is straightforward to implement, but the design lacks resilience and scalability, since the centralized broker represents a single point for potential system failure and a performance bottleneck. Consequently, distributed implementations of publish-subscribe systems are also available. In such schemes, the centralized broker is replaced by a *network of brokers* that cooperate to offer the desired functionality .
- A fully *peer-to-peer* implementation of a publish-subscribe system is a very popular implementation strategy for recent systems. In this approach, there is no distinction between publishers, subscribers and brokers; all nodes act as brokers, cooperatively implementing the required event routing functionality.

- **Overall systems architecture**

The implementation of centralized schemes is relatively straightforward, with the central service maintaining a repository of subscriptions and matching event notifications with this set of subscriptions. Similarly, the implementations of channel-based or topic-based schemes are relatively straightforward

The range of architectural choices for such approaches is captured in Figure 6.10.

Figure 6.10 The architecture of publish-subscribe systems



In the bottom layer, publish-subscribe systems make use of a range of interprocess communication services, such as TCP/IP, IP multicast (where available) or more specialized services, as offered for example by wireless networks.

The heart of the architecture is provided by the event routing layer supported by a network overlay infrastructure.

- Event routing performs the task of ensuring that event notifications are routed as efficiently as possible to appropriate subscribers, whereas the overlay infrastructure supports this by setting up appropriate networks of brokers or peer-to-peer structures.

- For content-based approaches, this problem is referred to as ***content-based routing*** (CBR), with the goal being to exploit content information to efficiently route events to their required destination. The top layer implements matching – that is, ensuring that events match a given subscription. While this can be implemented as a discrete layer, often matching is pushed down into the event routing mechanisms, as will become apparent shortly.

- General principles behind content-based routing:

- ❑ *Flooding*:
- ❑ *Filtering*:
- ❑ *Advertisements*:
- ❑ *Rendezvous*:

Flooding:

- The simplest approach is based on *flooding*, that is, sending an event notification to all nodes in the network and then carrying out the appropriate matching at the subsciber end.

- As an alternative, flooding can be used to send subscriptions back to all possible publishers, with the matching carried out at the publishing end and matched events sent directly to the relevant subscribers using point-to-point communication.

- Flooding can be implemented using an underlying broadcast or multicast facility. Alternatively, brokers can be arranged in an acyclic graph in which each forwards incoming event notifications to all its neighbours.

Filtering:

- One principle that underpins many approaches is to apply *filtering* in the network of brokers. This is referred to as *filtering-based routing*. Brokers forward notifications through the network only where there is a path to a valid subscriber.

- This is achieved by propagating subscription information through the network towards potential publishers and then storing associated state at each broker. More specifically, each node must maintain a *neighbours list* containing a list of all connected neighbours in the network of brokers, a subscription list containing a list of all directly connected subscribers serviced by this node, and a routing table.

- Crucially, this routing table maintains a list of neighbours and valid subscriptions for that pathway. This approach also requires an implementation of matching on each node in the network of brokers: in particular, a *match* function takes a given event notification and a list of nodes together with associated subscriptions and returns a set of nodes where the notification matches the subscription.

Advertisements:

- The pure filtering-based approach described above can generate a lot of traffic due to propagation of subscriptions, with subscriptions essentially using a flooding approach back towards all possible publishers.
- In systems with *advertisements* this burden can be reduced by propagating the advertisements towards subscribers in a similar (actually, symmetrical) way to the propagation of subscriptions.

Rendezvous:

- Another approach to control the propagation of subscriptions (and to achieve a natural load balancing) is the *rendezvous* approach. To understand this approach, it is necessary to view the set of all possible events as an event space and to partition responsibility for this event space between the set of brokers in the network.

- In particular, this approach defines rendezvous nodes, which are broker nodes responsible for a given subset of the event space. To achieve this, a given *rendezvous-based routing* algorithm must define two functions. First, *SN(s)* takes a given subscription, *s*, and returns one or more rendezvous nodes that take responsibility for that subscription. Each such rendezvous node maintains a subscription list as in the filtering approach above, and forwards all matching events to the set of subscribing nodes. Second, when an event *e* is published, the function *EN(e)* also returns one or more rendezvous nodes, this time responsible for matching *e* against subscriptions in the system.

Figure 6.13 Example publish-subscribe systems

System (and further reading)	*Subscription model*	*Distribution model*	*Event routing*
CORBA Event Service (Chapter 8)	Channel-based	Centralized	-
TIB Rendezvouz [Oki *et al.* 1993]	Topic-based	Distributed	Ffiltering
Scribe [Castro *et al.* 2002b]	Topic-based	Peer-to-peer (DHT)	Rendezvous
TERA [Baldoni *et al.* 2007]	Topic-based	Peer-to-peer	Informed gossip
Siena [Carzaniga *et al.* 2001]	Content-based	Distributed	Filtering
Gryphon [www.research.ibm.com]	Content-based	Distributed	Filtering
Hermes [Pietzuch and Bacon 2002]	Topic- and content-based	Distributed	Rendezvous and filtering
MEDYM [Cao and Singh 2005]	Content-based	Distributed	Flooding
Meghdoot [Gupta *et al.* 2004]	Content-based	Peer-to-peer	Rendezvous
Structure-less CBR [Baldoni *et al.* 2005]	Content-based	Peer-to-peer	Informed gossip

One interesting interpretation of rendezvous-based routing is to map the event space onto a ***distributed hash table*** **(DHT)** which is a style of network overlay that distributes a hash table over a set of nodes in a peer-to-peer network. The key observation for rendezvous-based routing is that the hash function can be used to map both events and subscriptions onto a corresponding rendezvous node for the management of such subscriptions

6.4 Message queues

- Whereas groups and publish subscribe provide a one-to-many style of communication, message queues provide a ***point-to-point* service** using the concept of a message queue as an indirection, thus achieving the desired properties of space and time uncoupling.

- They are point-to-point in that the sender places the message into a queue, and it is then removed by a single process.

- Message queues are also referred to as Message-Oriented Middleware. This is a major class of commercial middleware with key implementations including IBM's WebSphere MQ, Microsoft's MSMQ and Oracle's Streams Advanced Queuing (AQ).

- The main use of such products is to achieve *Enterprise Application Integration* (EAI) – that is, integration between applications within a given enterprise – a goal that is achieved by the inherent loose coupling of message queues.

- They are also extensively used as the basis for *commercial transaction processing systems* because of their intrinsic support for transactions

6.4.1 The programming model

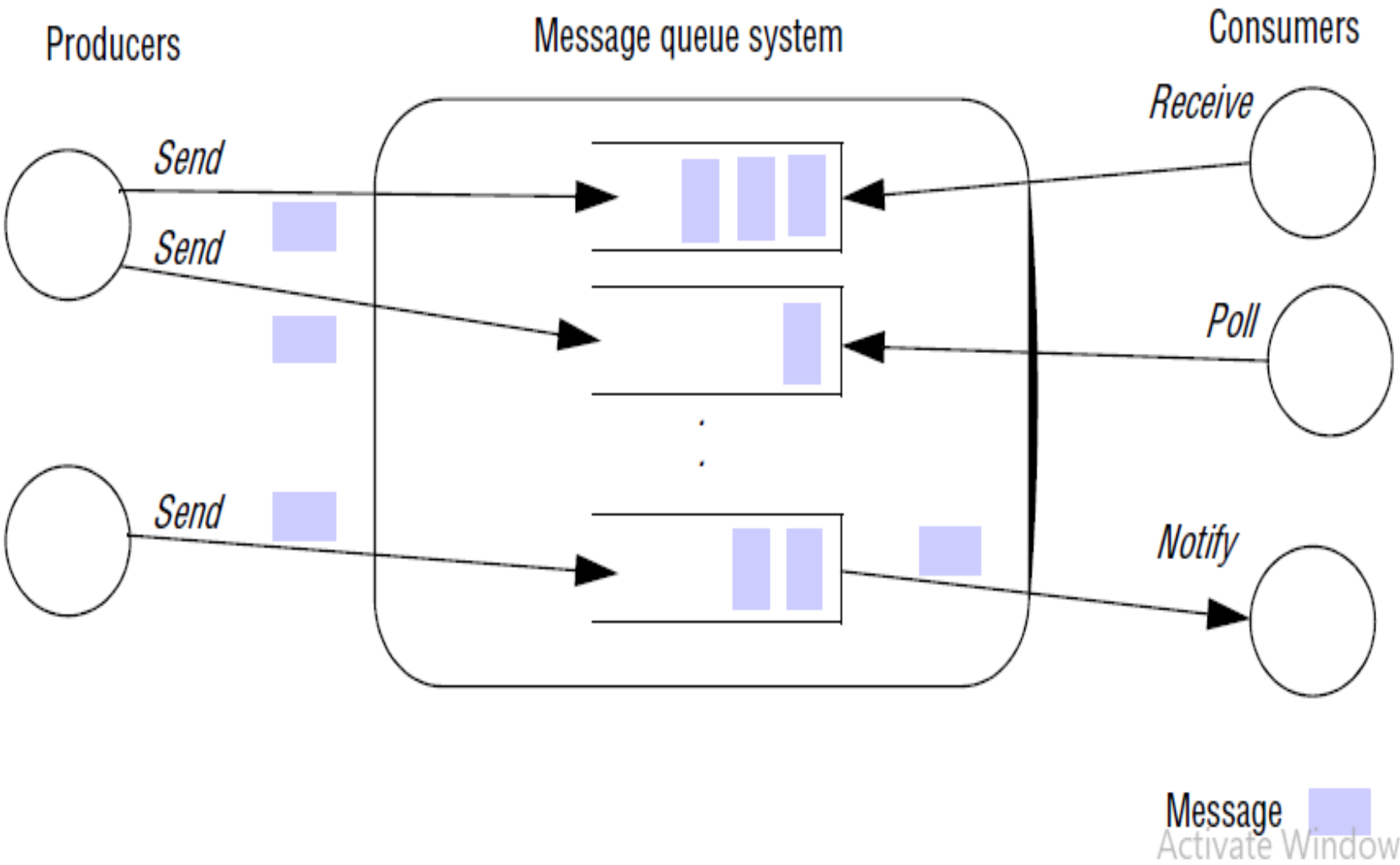
In particular, producer processes can *send* messages to a specific queue and other (consumer) processes can then receive messages from this queue. Three styles of receive are generally supported:

- a ***blocking receive*****,** which will block until an appropriate message is available;
- a ***non-blocking receive*** (a polling operation), which will check the status of the queue and return a message if available, or a not available indication otherwise;
- a ***notify*** **operation**, which will issue an event notification when when a message is available in the associated queue.

A number of processes can send messages to the same queue, and likewise a number of receivers can remove messages from a queue. The queuing policy is normally ***first-in-first-out*** **(FIFO),** but most message queue implementations also support the concept of priority, with higher-priority messages delivered first.

Consumer processes can also *select* messages from the queue based on properties of a message. In more detail, a message consists of a *destination* (that is, a unique identifier designating the destination queue), ***metadat****a* associated with the message, including fields such as the priority of the message and the delivery mode, and also the *body* of the message.

Figure 6.14 The message queue paradigm



- The associatedcontent is serialized using any of the standard approaches that is, marshalled data types, object serialization or XML structured messages.

- Message sizes are configurable and can be very large – for example, on the order of a 100 Mbytes Given the fact that message bodies are opaque, message selection is normally expressed through predicates defined over the metadata.

- In Oracle AQ, messages are rows in a database table, and queues are database tables that can be queried using the full power of a database query language.

- One crucial property of message queue systems is that messages are ***persistent*** – that is, message queues will store the messages indefinitely (until they are consumed) and will also commit the messages to disk to enable *reliable delivery.*

- Message queue systems therefore guarantee that messages will be delivered (and delivered once) but cannot say anything about the timing of the delivery.

Message passing systems can also support additional functionality:

- Most commercially available systems provide support for the sending or receiving of a message to be contained within a ***transaction***. The goal is to ensure that all the steps in the transaction are completed, or the transaction has no effect at all (the 'all or nothing' property). This relies on interfacing with an external transaction service, provided by the middleware environment.

- A number of systems also support **message transformation**, whereby an arbitrary transformation can be performed on an arriving message. The most common application of this concept is to transform messages between formats to deal with heterogeneity in underlying data representations (big-endian to little-endian).

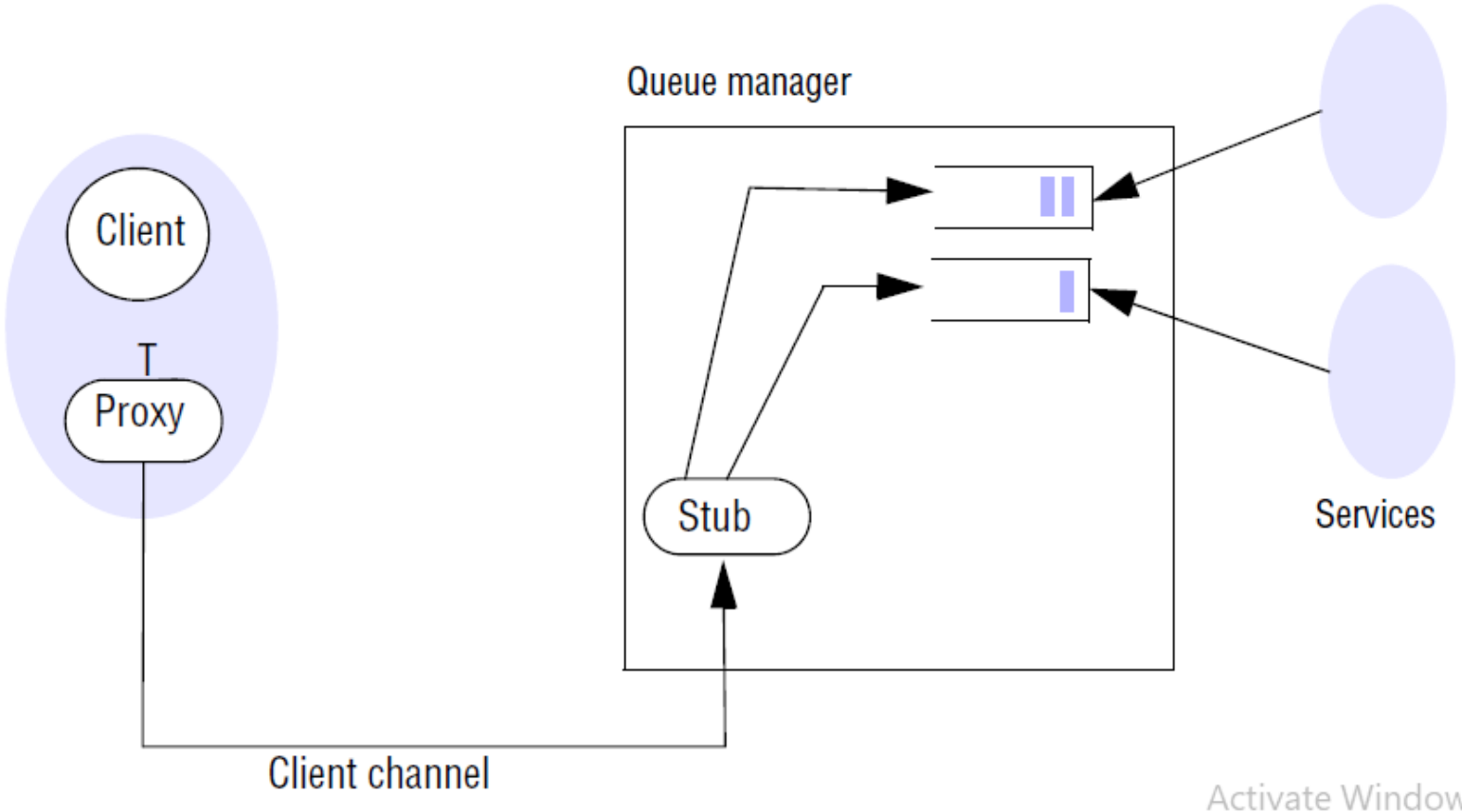
- Some message queue implementations also provide support for ***security***. For example, WebSphere MQ provides support for the confidential transmission of data using the Secure Sockets Layer (SSL) together with support for authentication and access control

6.4.2 Implementation issues

- The key implementation issue for message queuing systems is the choice between centralized and distributed implementations of the concept.
- Some implementations are centralized, with one or more message queues managed by a queue manager located at a given node.
- The advantage of this scheme is simplicity, but such managers can become rather heavyweight components and have the potential to become a bottleneck or a single point of failure. As a result, more distributed implementations have been proposed.

- **Case study: WebSphere MQ** • WebSphere MQ is middleware developed by IBM based on the concept of message queues, offering an indirection between senders and receivers of messages [www.redbooks.ibm.com].

Figure 6.15 A simple networked topology in WebSphere MQ



In this configuration, a client application is sending messages to a remote queue manager and multiple services (on the same machine as the server) are then consuming the incoming messages.

This is a very simple use of WebSphere MQ, and in practice it is more common for queue managers to be linked together into a federated structure, mirroring the approach often adopted in publish-subscribe systems (with networks of brokers).

- MQ introduces the concept of a *message channel* as a unidirectional connection between two queue managers that is used to forward messages asynchronously from one queue to another. Note the terminology here: a message channel is a connection between two queue managers, whereas a client channel is a connection between a client application and a queue manager. A message channel is managed by a ***message channel agent*** (MCA) at each end.

- The two agents are responsible for establishing and maintaining the channel, including an initial negotiation to agree on the properties of the channel (including security properties). Routing tables are also included in each queue manager, and together with channels this allows arbitrary topologies to be created.

- This ability to create customized topologies is crucial to WebSphere MQ, allowing users to determine the right topology for their application domain.

- A wide range of topologies can be created, including trees, meshes or a bus-based configuration.

The hub-and-spoke approach:

- In the hub-and-spoke topology, one queue manager is designated as the hub. The hub hosts a range of services.
- Client applications do not connect directly to this hub but rather connect through queue managers designated as spokes.
-
- Spokes relay messages to the message queue of the hub for processing by the various services. Spokes are placed strategically around the network to support different clients.
- The hub is placed somewhere appropriate in the network, on a node with sufficient resources to deal with the volume of traffic.
- Most applications and services are located on the hub, although it is also possible to have some more local services on spokes.
- This topology is heavily used with WebSphere MQ. Clearly, the drawback of this architecture is that the hub can be a potential bottleneck and a single point of failure.

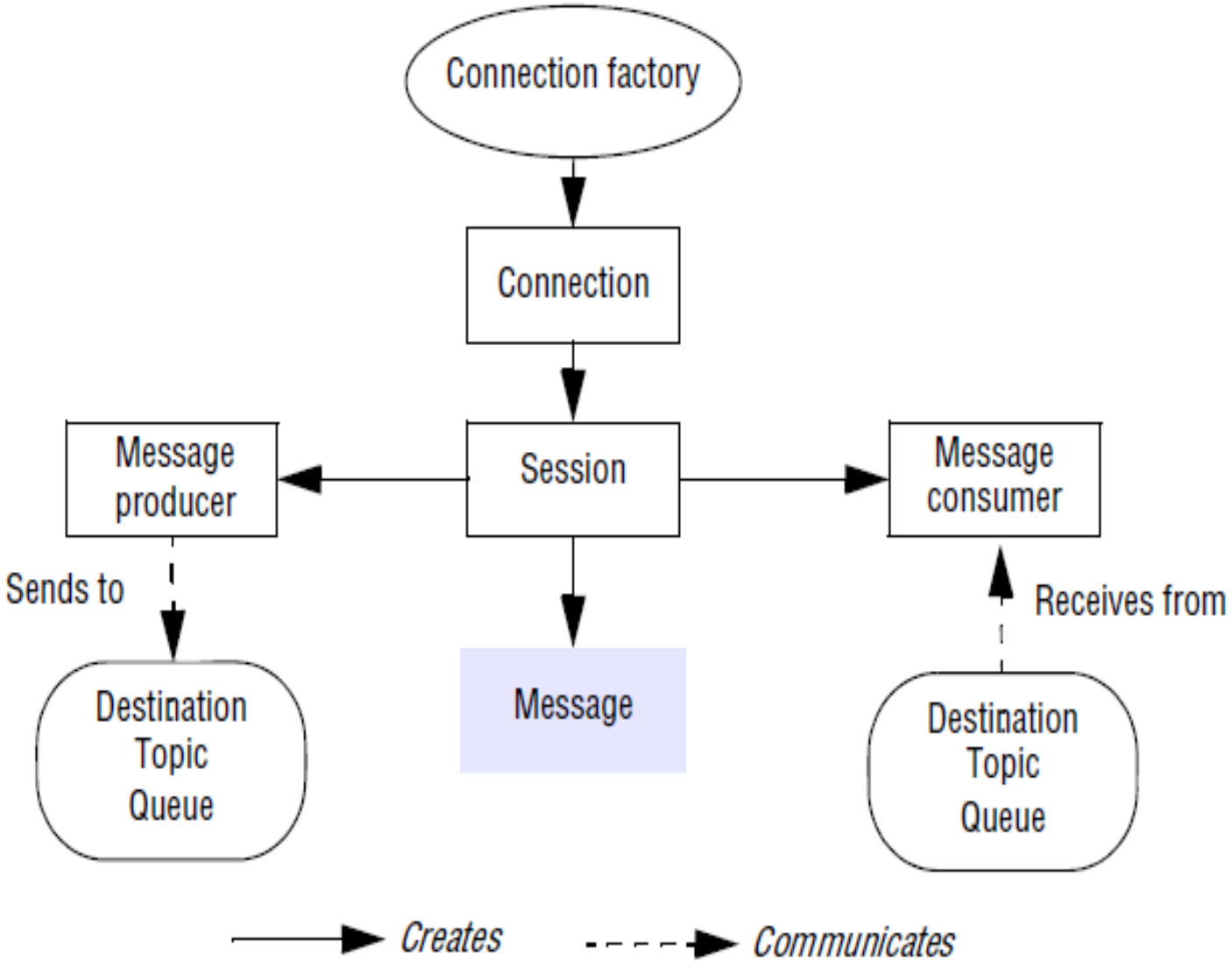
6.4.3 Case study: The Java Messaging Service (JMS)

The *Java Messaging Service* (JMS) java.sun.com XI] is a specification of a standardized way for distributed Java programs to communicate indirectly.

JMS distinguishes between the following key roles:

- A **JMS client** is a Java program or component that produces or consumes messages, a JMS producer is a program that creates and produces messages and a JMS consumer is a program that receives and consumes messages.
- A **JMS provider** is any of the multiple systems that implement the JMS specification.
- A **JMS message** is an object that is used to communicate information between JMS clients (from producers to consumers).
- A **JMS destination** is an object supporting indirect communication in JMS. It is either a JMS topic or a JMS queue.

Figure 6.16 The programming model offered by JMS



Programming with JMS

Java class FireAlarmJMS

```
import javax.jms.*;
import javax.naming.*;
public class FireAlarmJMS {
public void raise() {
try { 1
Context ctx = new InitialContext(); 2
TopicConnectionFactory topicFactory = 3
(TopicConnectionFactory)ctx.lookup("TopicConnectionFactory"); 4
Topic topic = (Topic)ctx.lookup("Alarms"); 5
TopicConnection topicConn = 6
topicConnectionFactory.createTopicConnection(); 7
TopicSession topicSess = topicConn.createTopicSession(false, 8
Session.AUTO_ACKNOWLEDGE); 9
TopicPublisher topicPub = topicSess.createPublisher(topic); 10
TextMessage msg = topicSess.createTextMessage(); 11
msg.setText("Fire!"); 12
topicPub.publish(message); 13
} catch (Exception e) { 14
} 15
}
```

Both publish-subscribe and message queues can be supported by a single middleware solution (in this case JMS), offering the programmer the choice of one-to-many or point-to-point variants of indirect communication, respectively.

6.5 Shared memory approaches

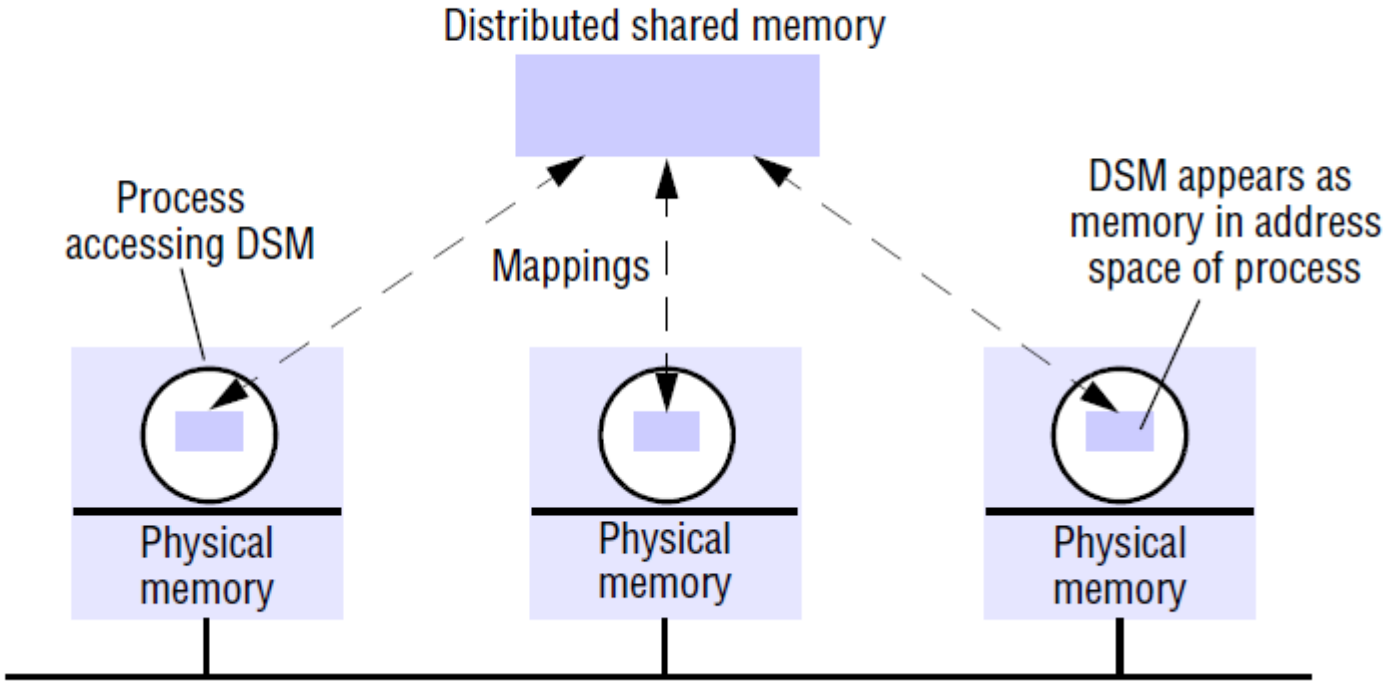
- **Distributed shared memory** operates at the level of reading and writing bytes, whereas **Tuple spaces** offer a higher-level perspective in the form of semistructured data.

- In addition, whereas distributed shared memory is accessed by address, tuple spaces are *associative*, offering a form of content-addressable memory

6.5.1 Distributed shared memory

- Distributed shared memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory.
- Processes access DSM by reads and updates to what appears to be ordinary memory within their address space. However, an underlying runtime system ensures transparently that processes executing at different computers observe the updates made by one another.
- It is as though the processes access a single shared memory, but in fact the physical memory is distributed.
- The main point of DSM is that it spares the programmer the concerns of message passing when writing applications that might otherwise have to use it.
- DSM is primarily a tool for parallel applications or for any distributed application or group of applications in which individual shared data items can be accessed directly.
- DSM is in general less appropriate in client-server systems, where clients normally view server-held resources as abstract data and access them by request (for reasons of modularity and protection).

Figure 6.19 The distributed shared memory abstraction



Message passing cannot be avoided altogether in a distributed system: in the absence of physically shared memory, the DSM runtime support has to send updates in messages between computers.

DSM systems manage replicated data: each computer has a local copy of recently accessed data items stored in DSM, for speed of access. The problems of implementing DSM are related to the replication issues

- At the hardware architectural level, developments include both caching strategies and fast processor-memory interconnections, aimed at maximizing the number of processors that can be sustained while achieving fast memory access latency and throughput.
- **Where processes are connected to memory modules over a common bus, the practical limit is on the order of 10 processors before performance degrades drastically due to bus contention.**
- Processors sharing memory are commonly constructed in groups of four, sharing a memory module over a bus on a single circuit board. Multiprocessors with up to 64 processors in total are constructed from such boards in a ***Non-Uniform Memory Access*** **(NUMA) architecture**.
- **In a NUMA architecture, processors see a single address space containing all the memory of all the boards. But the access latency for on-board memory is less than that for a memory module on a different board – hence the name of this architecture.**
- In *distributed-memory multiprocessors* and clusters of off-the-shelf computing components, the processors do not share memory but are connected by a very high speed network. These systems, like general-purpose distributed systems, can scale to much greater numbers of processors than a sharedmemory multiprocessor's 64 or so.
- **A central question that has been pursued by the DSM and multiprocessor research communities is whether the investment in knowledge of shared memory algorithms and the associated software can be directly transferred to a more scalable distributed memory architecture.**

Message passing versus DSM

As a communication mechanism, DSM is comparable with message passing rather than with request-reply-based communication, since its application to parallel processing, in particular, entails the use of asynchronous communication. The DSM and message-passing approaches to programming can be contrasted as follows:

***Service offered*:**

- Under the message-passing model, variables have to be marshalled from one process, transmitted and unmarshalled into other variables at the receiving process. By contrast, with shared memory the processes involved share variables directly, so no marshalling is necessary – even of pointers to shared variables – and thus no separate communication operations are necessary.

- Most implementations allow variables stored in DSM to be named and accessed similarly to ordinary unshared variables. In the case of DSM, synchronization is via normal constructs for shared-memory programming such as locks and semaphores (although these require different implementations in the distributed memory environment)

- In favour of message passing, on the other hand, is that it allows processes to communicate while being protected from one another by having private address spaces, whereas processes sharing DSM can, for example, cause one another to fail by erroneously altering data.

- Furthermore, when message passing is used between heterogeneous computers, marshalling takes care of differences in data representation; Synchronization between processes is achieved in the message model through message passing primitives themselves, using techniques such as the lock server implementation.

- Finally, since DSM can be made persistent, processes communicating via DSM may execute with non-overlapping lifetimes. A process can leave data in an agreed memory location for the other to examine when it runs. By contrast, processes communicating via message passing must execute at the same time.

***Efficiency*:**

Experiments show that certain parallel programs developed for DSM can be made to perform about as well as functionally equivalent programs written for message-passing platforms on the same hardware at least in the case of relatively small numbers of computers (10 or so).

- However, this result cannot be generalized. The performance of a program based on DSM depends upon many factors, as we shall discuss below – particularly the pattern of data sharing (such as whether an item is updated by several processes).

- There is a difference in the visibility of costs associated with the two types of programming. In message passing, all remote data accesses are explicit and therefore the programmer is always aware of whether a particular operation is in-process or involves the expense of communication. Using DSM, however, any particular read or update may or may not involve communication by the underlying runtime support.

- Whether it does or not depends upon such factors as whether the data have been accessed before and the sharing pattern between processes at different computers.

6.5.2 Tuple space communication

- In this approach, processes communicate indirectly by placing tuples in a tuple space, from which other processes can read or remove them.
- Tuples do not have an address but are accessed by pattern matching on content (content-addressable memory).
- Tuple space communication has also been influential in the field of ubiquitous computing.
- Examples - Linda programming model, JavaSpaces from Sun, IBM’s Tspaces.

Tuple Space programming model

- In the tuple space programming model, processes communicate through a tuple space – a shared collection of tuples.

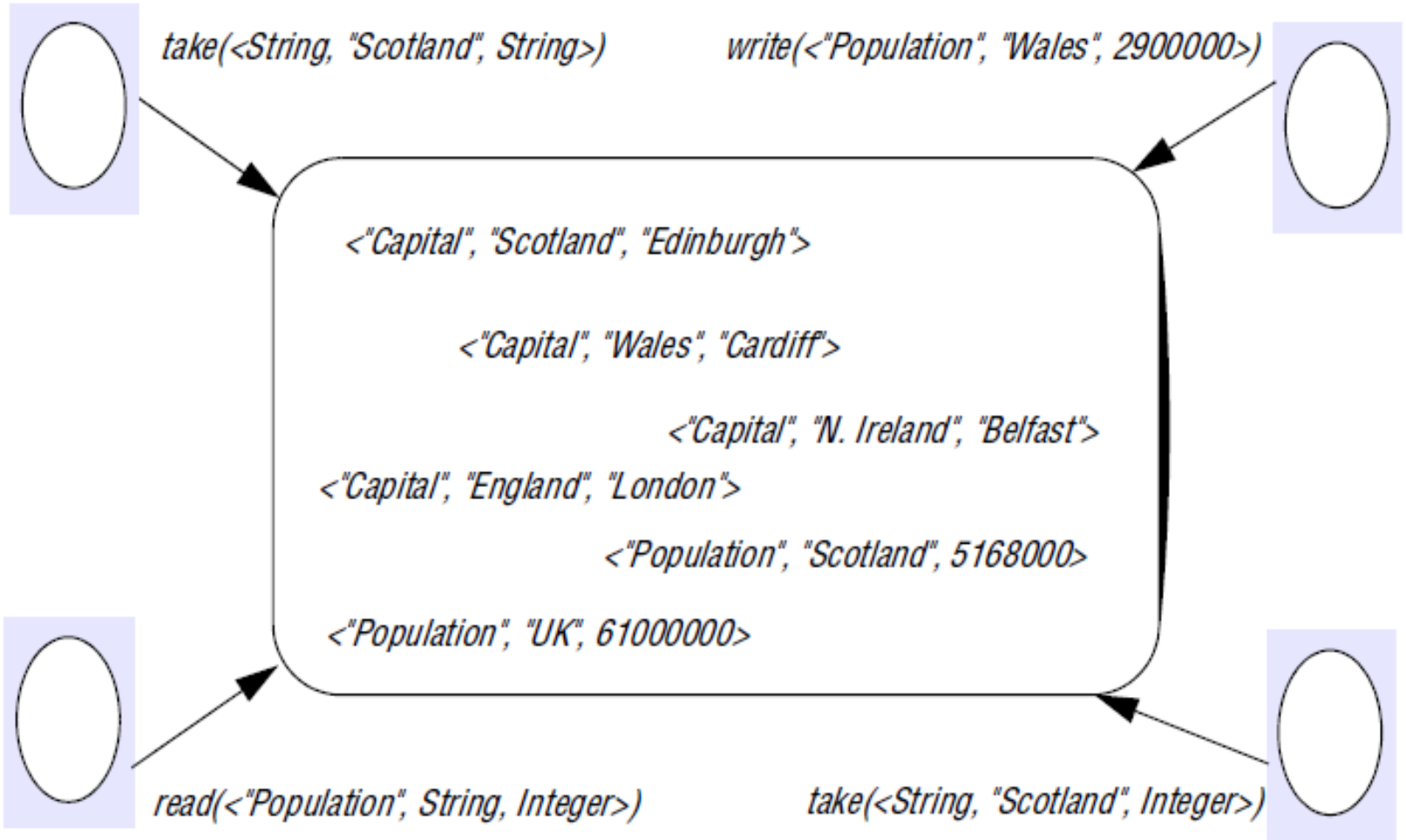
- Tuples in turn consist of a sequence of one or more typed data fields such as *<"fred", 1958>*, *<"sid", 1964>* and *<4, 9.8, "Yes">*. Any combination of types of tuples may exist in the same tuple space.

- Processes share data by accessing the same tuple space: they place tuples in tuple space using the *write* operation and read or extract them from tuple space using the *read* or *take* operation.

- The *write* operation adds a tuple without affecting existing tuples in the space. The *read* operation returns the value of one tuple without affecting the contents of the tuple space. The *take* operation also returns a tuple, but in this case it also removes the tuple from the tuple space.

- When reading or removing a tuple from tuple space, a process provides a tuple specification and the tuple space returns any tuple that matches that specification – as mentioned above, this is a type of associative addressing.

Figure 6.20 The tuple space abstraction



- In the tuple space paradigm, no direct access to tuples in tuple space is allowed and processes have to replace tuples in the tuple space instead of modifying them. Thus, tuples are *immutable*.

- Suppose, for example, that a set of processes maintains a shared counter in tuple space. The current count (say, 64) is in the tuple *<"counter", 64>*.

- A process must execute code of the following form in order to increment the counter in a tuple space *myTS*:

<s, count> := myTS.take(<"counter", integer>);

myTS.write(<"counter", count+1>);

Properties associated with tuple spaces:

- *Space uncoupling*: A tuple placed in tuple space may originate from any number of sender processes and may be delivered to any one of a number of potential recipients. This property is also referred to as *distributed naming* in Linda.

- *Time uncoupling*: A tuple placed in tuple space will remain in that tuple space until removed (potentially indefinitely), and hence the sender and receiver do not need to overlap in time.

- Together, these features provide an approach that is fully distributed in space and time and also provide for a form of *distributed sharing* of shared variables via the tuple space.

Figure 6.21 Replication and the tuple space operations [Xu and Liskov 1989]

write

1. The requesting site multicasts the *write* request to all members of the view;
2. On receiving this request, members insert the tuple into their replica and acknowledge this action;
3. Step 1 is repeated until all acknowledgements are received.

read

1. The requesting site multicasts the *read* request to all members of the view;
2. On receiving this request, a member returns a matching tuple to the requestor;
3. The requestor returns the first matching tuple received as the result of the operation (ignoring others);
4. Step 1 is repeated until at least one response is received.

take

Phase 1: Selecting the tuple to be removed

1. The requesting site multicasts the *take* request to all members of the view;
2. On receiving this request, each replica acquires a lock on the associated tuple set and, if the lock cannot be acquired, the *take* request is rejected;
3. All accepting members reply with the set of all matching tuples;
4. Step 1 is repeated until all sites have accepted the request and responded with their set of tuples and the intersection is non-null;
5. A particular tuple is selected as the result of the operation (selected randomly from the intersection of all the replies);
6. If only a minority accept the request, this minority are asked to release their locks and phase 1 repeats.

Phase 2: Removing the selected tuple

1. The requesting site multicasts a *remove* request to all members of the view citing the tuple to be removed;
2. On receiving this request, members remove the tuple from their replica, send an acknowledgement and release the lock;
3. Step 1 is repeated until all acknowledgements are received.

Figure 6.23 The JavaSpaces API

Operation	*Effect*
Lease write(Entry e, Transaction txn, long lease)	Places an entry into a particular JavaSpace
Entry read(Entry tmpl, Transaction txn, long timeout)	Returns a copy of an entry matching a specified template
Entry readIfExists(Entry tmpl, Transaction txn, long timeout)	As above, but not blocking
Entry take(Entry tmpl, Transaction txn, long timeout)	Retrieves (and removes) an entry matching a specified template
Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)	As above, but not blocking
EventRegistration notify(Entry tmpl, Transaction txn, RemoteEventListener listen, long lease, MarshalledObject handback)	Notifies a process if a tuple matching a specified template is written to a JavaSpace

The goals of the JavaSpaces technology are:

- to offer a platform that simplifies the design of distributed applications and services;
- to be simple and minimal in terms of the number and size of associated classes and
- to have a small footprint to allow the code to run on resource-limited devices (such as smart phones);
- to enable replicated implementations of the specification (although in practice most implementations are centralized).