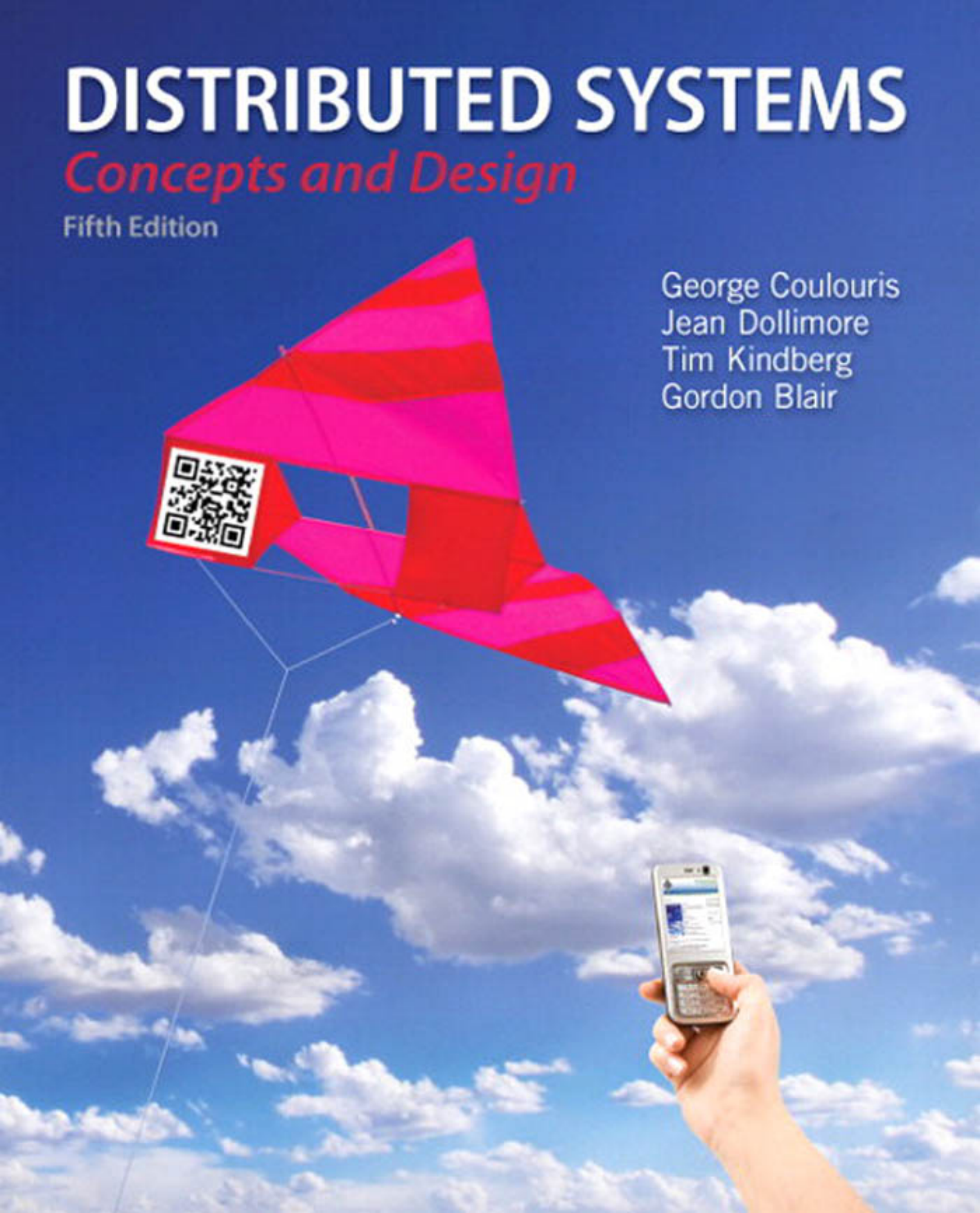
DISTRIBUTED SYSTEMS

Concepts and Design

Fifth Edition

George Coulouris
Jean Dollimore
Tim Kindberg
Gordon Blair



DISTRIBUTED SYSTEMS

Concepts and Design

Fifth Edition

6

INDIRECT COMMUNICATION

6.1 Introduction
6.2 Group communication
6.3 Publish-subscribe systems
6.4 Message queues
6.5 Shared memory approaches
6.6 Summary

This chapter completes our tour of communication paradigms by examining indirect communication; it builds on our studies of interprocess communication and remote invocation in Chapters 4 and 5, respectively. The essence of indirect communication is to communicate through an intermediary and hence have no direct coupling between the sender and the one or more receivers. The important concepts of space and time uncoupling are also introduced.

The chapter examines a range of indirect communication techniques:

- group communication, in which communication is via a group abstraction with the sender unaware of the identity of the recipients;
- publish-subscribe systems, a family of approaches that all share the common characteristic of disseminating events to multiple recipients through an intermediary;
- message queue systems, wherein messages are directed to the familiar abstraction of a queue with receivers extracting messages from such queues;
- shared memory–based approaches, including distributed shared memory and tuple space approaches, which present an abstraction of a global shared memory to programmers.

Case studies are used throughout the chapter to illustrate the main concepts introduced.

6.1 Introduction

This chapter concludes our examination of communication paradigms by examining indirect communication, building on the studies of interprocess communication and remote invocation in Chapters 4 and 5, respectively. Indirection is a fundamental concept in computer science, and its ubiquity and importance are captured nicely by the following quote, which emerged from the Titan Project at the University of Cambridge and is attributable to Roger Needham, Maurice Wilkes and David Wheeler:

> All problems in computer science can be solved by another level of indirection.

In terms of distributed systems, the concept of indirection is increasingly applied to communication paradigms.

Indirect communication is defined as 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. Note the optional plural associated with the receiver; this signifies that many indirect communication paradigms explicitly support one-to-many communication.

The techniques considered in Chapters 4 and 5 are all based on a direct coupling between a sender and a receiver, and this leads to a certain amount of rigidity in the system in terms of dealing with change. To illustrate this, consider a simple client-server interaction. Because of the direct coupling, it is more difficult to replace a server with an alternative one offering equivalent functionality. Similarly, if the server fails, this directly affects the client, which must explicitly deal with the failure. In contrast, indirect communication avoids this direct coupling and hence inherits interesting properties. The literature refers to 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.

For these reasons, 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 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, as featured in Chapter 1. Indirect communication is also exploited in key parts of the Google infrastructure, as discussed in the major case study in Chapter 21.

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 discussion above charts the advantages associated with indirect communication. The main disadvantage is that there will inevitably be a performance overhead introduced by the added level of indirection. Indeed, the quote above on indirection is often paired by the following quote, attributable to Jim Gray:

> There is no performance problem that cannot be solved by eliminating a level of indirection.

In addition, systems developed using indirect communication can be more difficult to manage precisely because of the lack of any direct (space or time) coupling.

A closer look at space and time uncoupling • It may be assumed that indirection implies *both* space and time uncoupling, but this is not always the case. The precise relationship is summarized in Figure 6.1.

From this table, it is clear that most of the techniques considered in this book are either coupled in both time and space or indeed uncoupled in both dimensions. The top-left box represents the communication paradigms featured in Chapters 4 and 5 where communication is direct with no space or time uncoupling. For example, message passing is both directed towards a particular entity and requires the receiver to be present at the time of the message send (but see Exercise 6.2 for an added dimension introduced by DNS name resolution). The range of remote invocation paradigms are also coupled in both space and time. The bottom-right box represents the main indirect communication paradigms that exhibit both properties. A small number of communication paradigms sit outside these two areas:

- IP multicast, as featured in Chapter 4, is space-uncoupled but time-coupled. It is space-uncoupled because messages are directed towards the multicast group, not any particular receiver. It is time-coupled, though, as all receivers must exist at the time of the message send to receive the multicast. Some implementations of group communication and indeed publish-subscribe systems, also fall into this category (see Section 6.6). This example illustrates the importance of *persistency* in the

communication channel to achieve time uncoupling – that is, the communication paradigm must store messages so that they can be delivered when the receiver(s) is ready to receive. IP multicast does not support this level of persistency.

- The case in which communication is space-coupled but time-uncoupled is more subtle. Space coupling implies that the sender knows the identity of a specific receiver or receivers, but time uncoupling implies that the receiver or receivers need not exist at the time of sending. Exercises 6.3 and 6.4 invite the reader to consider whether such a paradigm exists or could be constructed.

Returning to our definition, we treat *all* paradigms that involve an intermediary as indirect and recognize that the precise level of coupling will vary from system to system. We revisit the properties of different indirect communication paradigms in Section 6.6, once we have had a chance to study the precise characteristics of each approach.

The relationship with asynchronous communication • Note that, to fully understand this area, it is important to distinguish between asynchronous communication (as defined in Chapter 4) and time uncoupling. 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 [2003].

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, discussed in Section 6.2.3, offer a synchronous service over indirect communication.

The rest of the chapter examines specific examples of indirect communication starting with group communication in Section 6.2. Section 6.3 then examines the fundamentals of publish-subscribe systems, with Section 6.4 examining the contrasting approach offered by message queues. Following this, Section 6.5 considers approaches based on shared memory abstractions, specifically distributed shared memory and tuple space–based approaches.

6.2 Group communication

Group communication provides our first example of an indirect communication paradigm. *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:

- the reliable dissemination of information to potentially large numbers of clients, including in the financial industry, where institutions require accurate and up-to-date 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, as discussed in Chapter 1;
- support for a range of fault-tolerance strategies, including the consistent update of replicated data (as discussed in detail in Chapter 18) or the implementation of highly available (replicated) servers;
- support for system monitoring and management, including for example load balancing strategies.

We look at group communication in more detail below, examining the programming model offered and the associated implementation issues. We examine the JGroups toolkit as a case study of a group communication service.

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.

The use of a single multicast operation instead of multiple send operations amounts to much more than a 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.

To see these advantages, compare the bandwidth utilization and the total transmission time taken when sending the same message from a computer in London to two computers on the same Ethernet in Palo Alto, (a) by two separate UDP sends, and (b) by a single IP multicast operation. In the former case, two copies of the message are sent independently, and the second is delayed by the first. In the latter case, a set of multicast-aware routers forward a single copy of the message from London to a router on the destination LAN in California. That router then uses hardware multicast (provided by the Ethernet) to deliver the message to both destinations at once, instead of sending it twice.

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, as discussed in Section 6.2.2 below.

Group communication has been the subject of many research projects, including the V-system [Cheriton and Zwaenepoel 1985], Chorus [Rozier *et al.* 1988], Amoeba [Kaashoek *et al.* 1989; Kaashoek and Tanenbaum 1991], Trans/Total [Melliar-Smith *et al.* 1990], Delta-4 [Powell 1991], Isis [Birman 1993], Horus [van Renesse *et al.* 1996], Totem [Moser *et al.* 1996] and Transis [Dolev and Malki 1996] – and we shall cite other notable work in the course of this chapter and indeed throughout the book (particularly in Chapters 15 and 18).

Process groups and object groups • Most work on group services focuses on the concept of *process groups*, that is, groups where the communicating entities are processes. Such services are relatively low-level in that:

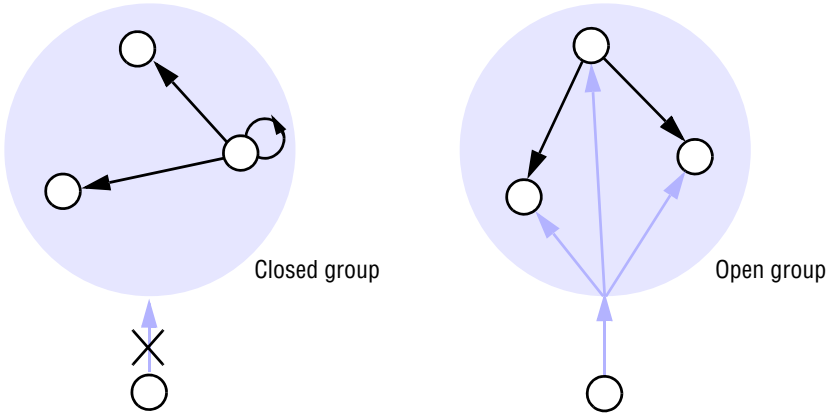
- Messages are delivered to processes and no further support for dispatching is provided.
- Messages are typically unstructured byte arrays with no support for marshalling of complex data types (as provided, for example, in RPC or RMI – see Chapter 5).

The level of service provided by process groups is therefore similar to that of sockets, as discussed in Chapter 4. In contrast, *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 [Maffeis 1995] is a CORBA-compliant system that supports object groups. An Electra group can be interfaced to any CORBA-compliant application. Electra was originally built on top of the Horus group communication system, which it uses to manage the membership of the group and to multicast invocations. In 'transparent mode', the local proxy returns the first available response to a client object. In 'non-transparent mode', the client object can access all the responses returned by the group members. 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 [Moser *et al.* 1998] and the Object Group Service [Guerraoui *et al.* 1998] 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 toolkit, discussed in Section 6.2.3, is a classical process group approach.

Figure 6.2 Open and closed groups



Other key distinctions • A wide range of group communication services has been developed, and they vary in the assumptions they make:

Closed and open groups: A group is said to be *closed* if only members of the group may multicast to it (Figure 6.2). 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.

Such distinctions can have a significant impact on the underlying multicast algorithms. For example, some algorithms assume that groups are closed. The same effect as openness can be achieved with a closed group by picking a member of the group and sending it a message (one-to-one) for it to multicast to its group. Rodrigues *et al.* [1998] discuss multicast to open groups. Issues related to open and closed groups arise in Chapter 15, when algorithms for reliability and ordering are considered. That chapter also considers the impact of overlapping groups and whether the system is synchronous or asynchronous on such protocols.

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 • 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.

So far, we have discussed reliability and ordering in rather general terms. We now look in more detail at what such properties mean.

Reliability in one-to-one communication was defined in Section 2.4.2 in terms of 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):

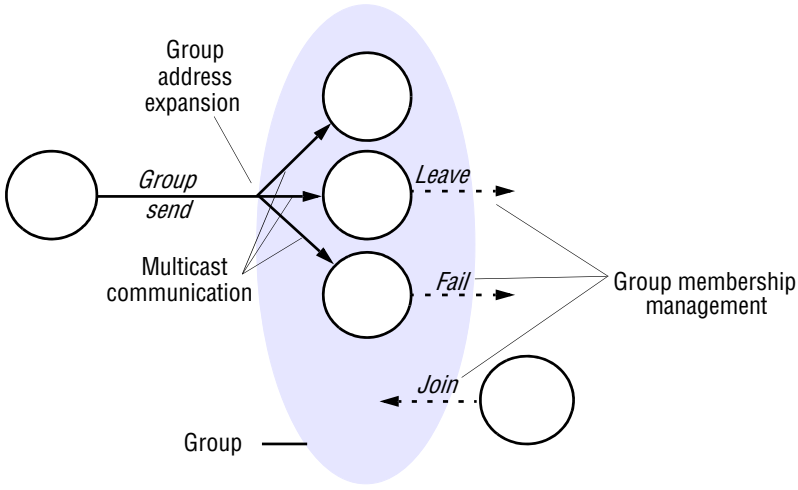
FIFO ordering: First-in-first-out (FIFO) ordering (also referred to as source ordering) is concerned with preserving the order from the perspective of a sender process, in that if a process sends one message before another, it will be delivered in this order at all processes in the group.

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 (see Chapter 14 for a detailed discussion of the meaning of 'happens before').

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.

Reliability and ordering are examples of coordination and agreement in distributed systems, and hence further consideration of this is deferred to Chapter 15, which focuses exclusively on this topic. In particular, Chapter 15 provides more complete definitions

Figure 6.3 The role of group membership management



of integrity, validity, agreement and the various ordering properties and also examines in detail algorithms to realize reliable and ordered multicast.

Group membership management • The key elements of group communication management are summarized in Figure 6.3, which shows an open group. 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 fail. In more detail, 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 (overlapping groups, as defined above). This is true of IP multicast, for example.

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.

Note that 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*. Further consideration of this important issue is deferred to Chapter 18, which discusses the maintenance of group views and how to realize view-synchronous group communication in the context of supporting replication in distributed systems.

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. Ganesh et al [2003] present a more probabilistic approach to group membership designed to operate in more large-scale and dynamic environments, using an underlying gossip protocol (see Section 10.5.3). Researchers have also developed group membership protocols specifically for ad hoc networks and mobile environments [Prakash and Baldoni 1998; Roman *et al*. 2001; Liu *et al*. 2005].

6.2.3 Case study: the JGroups toolkit

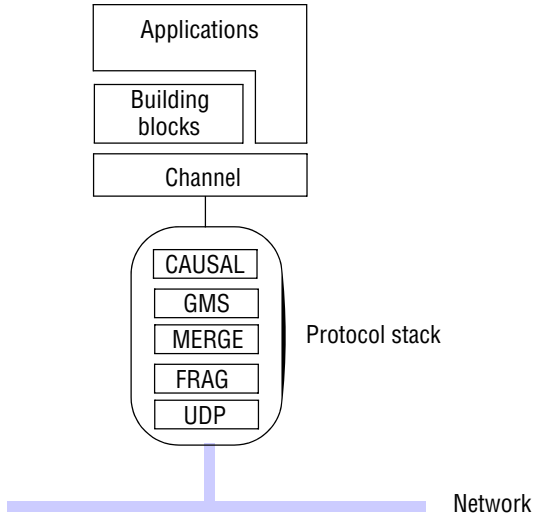
JGroups is a toolkit for reliable group communication written in Java. The toolkit is a part of the lineage of group communication tools that have emerged from Cornell University, building on the fundamental concepts developed in ISIS [Birman 1993], Horus [van Renesse *et al.* 1996] and Ensemble [van Renesse *et al.* 1998]. The toolkit is now maintained and developed by the JGroups open source community [www.jgroups.org], which is part of the JBoss middleware community, as discussed in Chapter 8 [www.jboss.org].

JGroups supports process groups in which processes are able to join or leave a group, send a message to all members of the group or indeed to a single member, and receive messages from the group. The toolkit supports a variety of reliability and ordering guarantees, which are discussed in more detail below, and also offers a group membership service.

The architecture of JGroups is shown in Figure 6.4, which shows the main components of the JGroups implementation:

- *Channels* represent the most primitive interface for application developers, offering the core functions of joining, leaving, sending and receiving.
- *Building blocks* offer higher-level abstractions, building on the underlying service offered by channels.

Figure 6.4 The architecture of JGroups



- The *protocol stack* provides the underlying communication protocol, constructed as a stack of composable protocol layers.

We look at each in turn below.

Channels • A process interacts with a group through a *channel* object, which acts as a handle onto a group. When created, it is disconnected, but a subsequent *connect* operation binds that handle to a particular named group; if the named group does not exist, it is implicitly created at the time of the first connect. To leave the group, the process executes the corresponding *disconnect* operation. A *close* operation is also provided to render the channel unusable. Note that a channel can only be connected to one group at a time; if a process wants to connect to two or more groups, it must create multiple channels. When connected, a process can send or receive via a channel. Messages are sent by reliable multicast, with the precise semantics defined by the protocol stack deployed (as discussed further below).

A range of other operations are defined on channels, most notably to return management information associated with the channel. For example, *getView* returns the current view defined in terms of the current member list, while *getState* returns the historical application state associated with the group (this can be used, for example, by a new group member to catch up with previous events).

Note that the term *channel* should not be confused with *channel-based publish-subscribe*, as introduced in Section 6.3.1. A channel in JGroups is synonymous with an instance of a group as defined in Section 6.2.1.

Figure 6.5 Java class *FireAlarmJG*

```
import org.jgroups.JChannel;

public class FireAlarmJG {
    public void raise() {
        try {
            JChannel channel = new JChannel();
            channel.connect("AlarmChannel");
            Message msg = new Message(null, null, "Fire!");
            channel.send(msg);
        }
        catch(Exception e) {
        }
    }
}
```

We illustrate the use of channels further by a simple example, a service whereby an intelligent fire alarm can send a "Fire!" multicast message to any registered receivers. The code for the fire alarm is as shown in Figure 6.5.

When an alarm is raised, the first step is to create a new instance of *JChannel* (the class representing channels in JGroups) and then connect to a group called *AlarmChannel*. If this is the first connect, then the group will be created at this stage (unlikely in this example, or the alarm is not going to be very effective). The constructor for a message takes three parameters: the destination, the source and the payload. In this case, the destination is *null,* which specifies that the message is to be sent to all members (if an address is specified, it is sent to that address only). The source is also *null*; this need not be provided in JGroups as it will be included automatically. The payload is an unstructured byte array that is delivered to all members of the group through the *send* method. The code to create a new instance of the *FireAlarmJG* class and then raise an alarm would be:

```
FireAlarmJG alarm = new FireAlarmJG();
alarm.raise();
```

The corresponding code for the receiver end has a similar structure and is shown in Figure 6.6. In this case, however, after connecting a *receive* method is called. This method takes one parameter, a timeout. If it is set to zero, as in this case, the receive message will block until a message is received. Note that in JGroups incoming messages are buffered and *receive* returns the top element in the buffer. If no messages are present, then *receive* blocks awaiting the next message. Strictly speaking, *receive* can return a range of object types – for example, notification of a change in membership or of a suspected failure of a group member (hence the cast to *Message* above).

A given receiver must include the following code to await an alarm:

```
FireAlarmConsumerJG alarmCall = new FireAlarmConsumerJG();
String msg = alarmCall.await();
System.out.println("Alarm received: " + msg);
```

Figure 6.6 Java class *FireAlarmConsumerJG*

```
import org.jgroups.JChannel;

public class FireAlarmConsumerJG {
    public String await() {
        try {
            JChannel channel = new JChannel();
            channel.connect("AlarmChannel");
            Message msg = (Message) channel.receive(0);
            return (String) msg.GetObject();
        }
        catch(Exception e) {
            return null;
        }
    }
}
```

Building blocks • Building blocks are higher-level abstractions on top of the channel class discussed above. Channels are similar in level to sockets. Building blocks are analogous to the more advanced communication paradigms discussed in Chapter 5, offering support for commonly occurring patterns of communication (but in this case targeted at multicast communication). Examples of building blocks in JGroups are:

- *MessageDispatcher* is the most intuitive of the building blocks offered in JGroups. In group communication, it is often useful for a sender to send a message to a group and then wait for some or all of the replies. *MessageDispatcher* supports this by providing a *castMessage* method that sends a message to a group and blocks until a specified number of replies are received (for example, until a specified number *n*, a majority, or all messages are received).
- *RpcDispatcher* takes a specific method (together with optional parameters and results) and then invokes this method on all objects associated with a group. As with *MessageDispatcher*, the caller can block awaiting some or all of the replies.
- *NotificationBus* is an implementation of a distributed event bus, in which an event is any serializable Java object. This class is often used to implement consistency in replicated caches.

The protocol stack • JGroups follows the architectures offered by Horus and Ensemble by constructing protocol stacks out of protocol layers (initially referred to as micro-protocols in the literature [van Renesse *et al.* 1996, 1998]). In this approach, a protocol is a bidirectional stack of protocol layers with each layer implementing the following two methods:

```
public Object up (Event evt);
public Object down (Event evt);
```

Protocol processing therefore happens by passing events up and down the stack. In JGroups, events may be incoming or outgoing messages or management events, for example related to view changes. Each layer can carry out arbitrary processing on the

message, including modifying its contents, adding a header or indeed dropping or re-ordering the message.

To illustrate the concept further, let us examine the protocol stack shown in Figure 6.4. 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 equivalent 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. If it is not, the layer can be configured to send a series of unicast messages to members, relying on another layer for membership discovery (in particular, a layer known as PING). For larger-scale systems operating over wide area networks, a TCP layer may be preferred (using the TCP protocol to send unicast messages and again relying on PING for membership discovery).
- 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 (see Chapter 18 for further details of algorithms for group membership management).
- CAUSAL implements causal ordering, introduced in Section 6.2.2 (and discussed further in Chapter 15).

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 (see the JGroups web site for details [www.jgroups.org]). 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

We now turn our attention to the area of *publish-subscribe systems* [Baldoni and Virgillito 2005], sometimes also referred to as *distributed event-based systems* [Muhl *et al*. 2006]. These are the most widely used of all the indirect communication techniques discussed in this chapter. Chapter 1 has already highlighted that many classes of system are fundamentally concerned with the communication and processing of events (for example financial trading systems). More specifically, whereas many systems naturally map onto a request-reply or a remote invocation pattern of interaction as discussed in Chapter 5, many do not and are more naturally modelled by the more decoupled and reactive style of programming offered by events.

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 publish-subscribe 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.

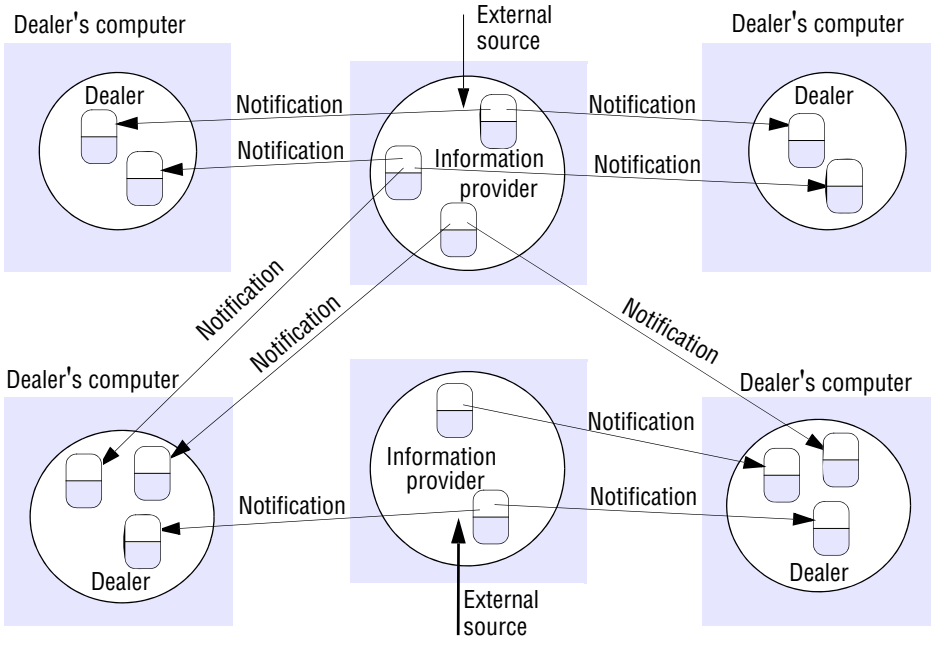
Publish-subscribe is also a key component of Google's infrastructure, including for example the dissemination of events related to advertisements, such as 'ad clicks', to interested parties (see Chapter 21).

To illustrate the concept further, we consider a simple dealing room system as an example of the broader class of financial information systems.

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. The communication of notifications is illustrated in Figure 6.7.

Figure 6.7 Dealing room system



Characteristics of publish-subscribe systems • Publish-subscribe systems have two main characteristics:

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. For example, Bates *et al.* [1996] describe how publish-subscribe systems can be used to connect heterogeneous components in the Internet. They describe a system in which applications can be made aware of users' locations and activities, such as using computers, printers or electronically tagged books. They envisage its future use in the context of a home network supporting commands such as: 'if the children come home, turn on the central heating'.

Asynchronicity: Notifications are sent asynchronously 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. Mushroom [Kindberg *et al.* 1996] is an object-based publish-subscribe 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 addition, a variety of different *delivery guarantees* can be provided for notifications – the one that is chosen should depend on the application requirements. For example, if IP multicast is used to send notifications to a group of receivers, the failure model will relate to the one described for IP multicast in Section 4.4.1 and will not guarantee that any particular recipient will receive a particular notification message. This is adequate for some applications – for example, to deliver the latest state of a player in an Internet game – because the next update is likely to get through.

However, other applications have stronger requirements. Consider the dealing room application: to be fair to the dealers interested in a particular stock, we require that all the dealers for the same stock receive the same information. This implies that a reliable multicast protocol should be used.

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. However, notifications may also be sent to object replicas in users' computers by means of unreliable multicast; in the case that the latter lose notifications, they can retrieve an object's state from the server. When the application requires it, notifications may be ordered and sent reliably to object replicas.

Some applications have real-time requirements.These include events in a nuclear power station or a hospital patient monitor. It is possible to design multicast protocols that provide real-time guarantees as well as reliability and ordering in a system that satisfies the properties of a synchronous distributed system.

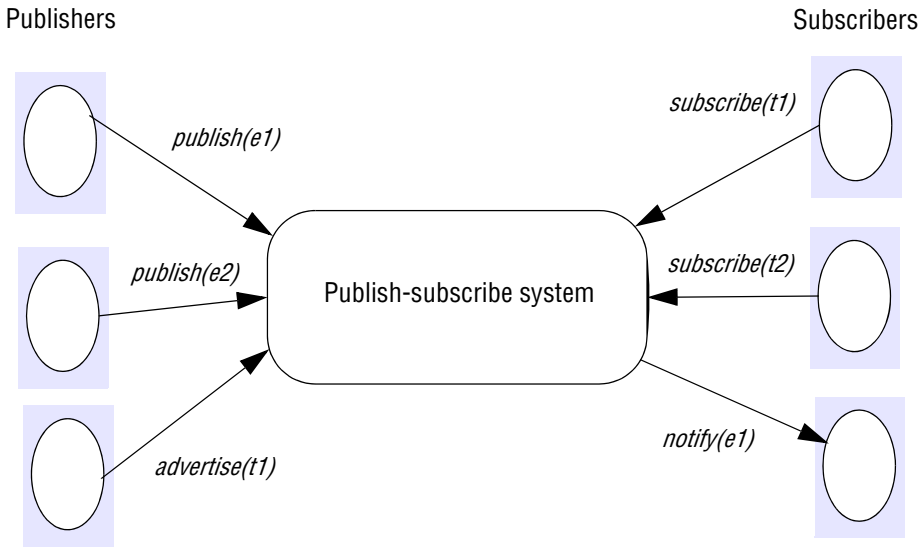
We discuss publish-subscribe systems in more detail in the following sections, considering the programming model they offer before examining some of the key implementation challenges, particularly related to large-scale dissemination of events in the Internet.

6.3.1 The programming model

The programming model in publish-subscribe systems is based on a small set of operations, captured in Figure 6.8. 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; which we discuss in more detail below. 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).

Figure 6.8 The publish-subscribe paradigm



In other words, 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)*.

As mentioned above, 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: 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; all other schemes employ some form of filtering over the content of events as we shall see below. Although simple, this scheme has been used successfully in the CORBA Event Service (see Chapter 8).

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. For example, let us consider a publish-subscribe system for this book. Subscriptions could be defined in terms of *indirect_communication* or *indirect_communication/publish-subscribe*. Subscribers expressing interest in the former will receive all events related to this chapter, whereas with the latter subscribers can instead express an interest in the more specific topic of publish-subscribe.

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. For example, a subscriber could express interest in events that relate to the topic of publish-subscribe systems, where the system in question is the 'CORBA Event Service' and where the author is 'Tim Kindberg' or 'Gordon Blair'. 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 (discussed below).

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 coarse-grained 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.

As well as these classical categories, a number of commercial systems are based on subscribing directly to *objects of interest*. Such systems are similar to type-based approaches in that they are intrinsically linked to object-based approaches, although they differ by focusing on changes of state of the objects of interest rather than predicates associated with the types of objects. They allow one object to react to a change occurring in another object. Notifications of events are asynchronous and determined by their receivers. In particular, in interactive applications, the actions that the user performs on objects – for example, by manipulating a button with the mouse or entering text in a text box via the keyboard – are seen as events that cause changes in the objects that maintain the state of the application. The objects that are responsible for displaying a view of the current state are notified whenever the state changes.

Rosenblum and Wolf [1997] describe a general architecture for this style of publish-subscribe system. The main component in their architecture is an event service that maintains a database of event notifications and of interests of subscribers. The event service is notified of events that occur at objects of intetest. Subscribers inform the event service about the types of events they are interested in. When an event occurs at an object of interest a message containing the notification is sent directly to the subscribers of that type of event.

The Jini distributed event specification described by Arnold *et al.* [1999] is one leading example of this approach, and a case study on Jini, together with further background information on this style of approach, can be found on the companion web site for the book [www.cdk5.net/rmi]. Note, however, that Jini is a relatively primitive example of a distributed event-based system that allows direct connectivity between producers and consumers of events (hence compromising time and space uncoupling).

A number of more experimental approaches are also being investigated. For example, some researchers are considering the added expressiveness of *context* [Frey

and Roman 2007, Meier and Cahill 2010]. Context and context-awareness are major concepts in mobile and ubiquitous computing. Context is defined in Chapter 19 as an aspect of the physical circumstances of relevance to the system behaviour. One intuitive example of context is location, and such systems have the potential for users to subscribe to events associated with a given location – for example, any emergency messages associated with the building where a user is located. Cilia *et al*. [2004] have also introduced *concept-based* subscription models whereby filters are expressed in terms of the semantics as well as the syntax of events. More specifically, data items have an associated semantic context that captures the meaning of those items, allowing for interpretation and possible translation into different data formats, thus addressing heterogeneity.

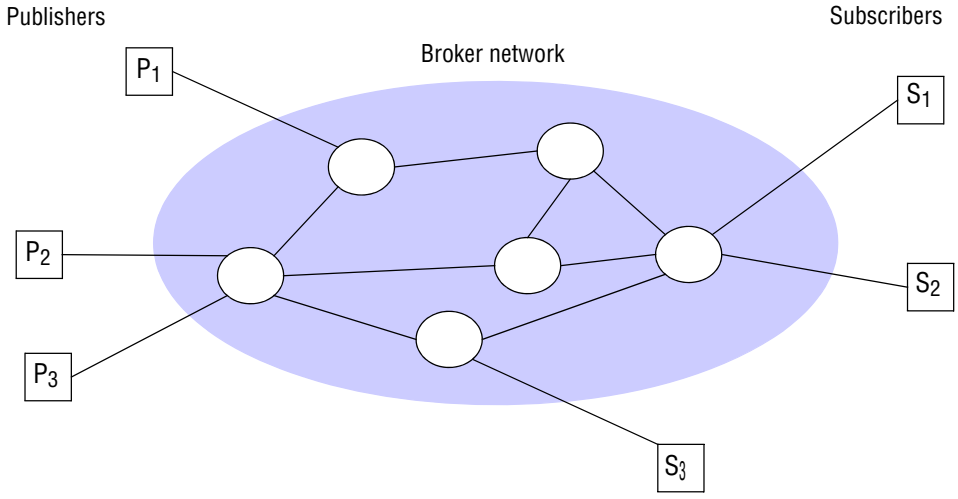
For some classes of application, such as the financial trading system described in Chapter 1, it is not enough for subscriptions to express queries over individual events. Rather, there is a need for more complex systems that can recognize complex event patterns. For example, Chapter 1 introduced the example of buying and selling shares based on observing temporal sequences of events related to share prices, demonstrating the need for *complex event processing* (or composite event detection, as it is sometimes called). Complex event processing allows the specification of patterns of events as they occur in the distributed environment – for example, 'inform me if water levels rise by at least 20% in the River Eden in at least three places and simulation models are also reporting a risk of flooding'. A further example of an event pattern arose in Chapter 1, concerned with detecting share price movements over a given time period. In general, patterns can be logical, temporal or spatial. For further information on complex event processing, refer to Muhl *et al*. [2006].

6.3.2 Implementation issues

From the description above, the task of a publish-subscribe system is clear: to ensure that events are delivered efficiently to all subscribers that have filters defined that match the event. Added to this, there may be additional requirements in terms of security, scalability, failure handling, concurrency and quality of service. This makes the implementation of publish-subscribe systems rather complex, and this has been an area of intense investigation in the research community. We consider key implementation issues below, examining centralized versus distributed implementations before moving on to consider the overall system architecture required to implement publish-subscribe systems (particularly distributed implementations of content-based approaches). We conclude the section by summarizing the design space of publish-subscribe systems, with associated pointers to the literature.

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

Figure 6.9 A network of brokers



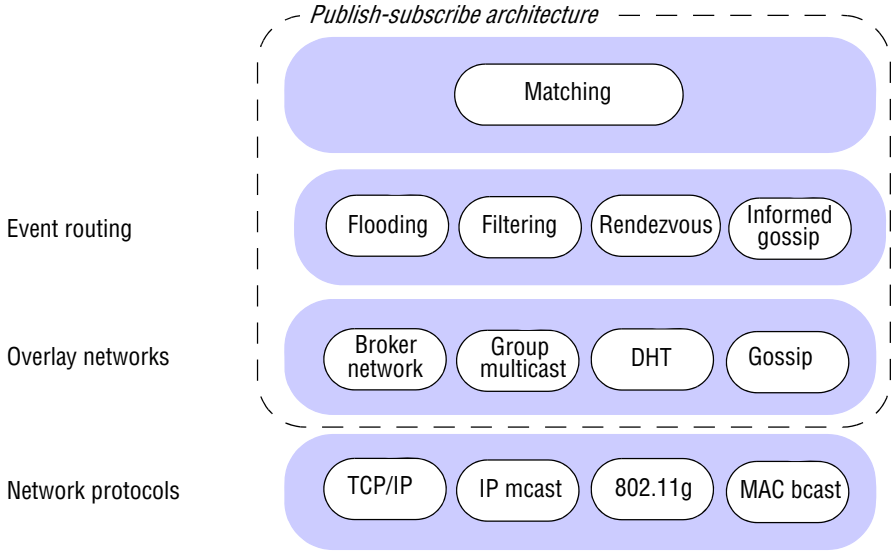
This approach is straightforward to implement, but the design lacks resilience and scalability, since the centralized broker represents a single point for potential system failure and a performance bottleneck. Consequently, distributed implementations of publish-subscribe systems are also available. In such schemes, the centralized broker is replaced by a *network of brokers* that cooperate to offer the desired functionality as illustrated in Figure 6.9. Such approaches have the potential to survive node failure and have been shown to be able to operate well in Internet-scale deployments.

Taking this a step further, it is possible to have a fully *peer-to-peer* implementation of a publish-subscribe system. This 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 (as discussed further below).

Overall systems architecture • As mentioned above, 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. For example, a distributed implementation can be achieved by mapping channels or topics onto associated groups (as defined in Section 6.2) and then using the underlying multicast communication facilities to deliver events to interested parties (using reliable and ordered variants, as appropriate). The distributed implementation of content-based (or by extrapolation, type-based) approaches is more complex and deserving of further consideration. The range of architectural choices for such approaches is captured in Figure 6.10 (adapted from Baldoni and Virgillito [2005]).

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

Figure 6.10 The architecture of publish-subscribe systems



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.

Within this overall architecture, there is a wide variety of implementation approaches. We step through a select set of implementations to illustrate the general principles behind content-based routing:

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 (effectively providing a multicast overlay, as discussed in Section 4.5.1). This approach has the benefit of simplicity but can result in a lot of unnecessary network

Figure 6.11 Filtering-based routing

```
upon receive publish(event e) from node x                    1
    matchlist := match(e, subscriptions)                     2
    send notify(e) to matchlist;                             3
    fwdlist := match(e, routing);                            4
    send publish(e) to fwdlist - x;                          5
upon receive subscribe(subscription s) from node x           6
    if x is client then                                      7
        add x to subscriptions;                              8
    else add(x, s) to routing;                               9
    send subscribe(s) to neighbours - x;                     10
```

traffic. Hence, the alternative schemes described below try to optimize the number of messages exchanged through consideration of content.

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. The specific algorithm for this filtering approach is captured in Figure 6.11 (taken from Baldoni and Virgillito [2005]). When a broker receives a publish request from a given node, it must pass this notification to all connected nodes where there is a corresponding matching subscription and also decide where to propagate this event through the network of brokers. Lines 2 and 3 achieve the first goal by matching the event against the subscription list and then forwarding the event to all the nodes with matching subscriptions (the *matchlist*). Lines 4 and 5 then use the match function again, this time matching the event against the routing table and forwarding only to the paths that lead to a subscription (the *fwdlist*). Brokers must also deal with incoming subscription events. If the subscription event is from an immediately connected subscriber, then this subscription must be entered in the subscriptions table (lines 7 and 8). Otherwise, the broker is an intermediary node; this node now knows that a pathway exists towards this subscription and hence an appropriate entry is added to the routing table (line 9). In both cases, this subscription event is then passed to all neighbours apart from the originating node (line 10).

Figure 6.12 Rendezvous-based routing

```
upon receive publish(event e) from node x at node i
    rvlist := EN(e);
    if i in rvlist then begin
        matchlist <- match(e, subscriptions);
        send notify(e) to matchlist;
    end
    send publish(e) to rvlist - i;
upon receive subscribe(subscription s) from node x at node i
    rvlist := SN(s);
    if i in rvlist then
        add s to subscriptions;
    else
        send subscribe(s) to rvlist - i;
```

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. There are interesting trade-offs between the two approaches, and some systems adopt both approaches in tandem [Carzaniga *et al.* 2001].

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. Note that both *SN(s)* and *EN(e)* return more than one node if reliability is a concern. Note also that this approach only works if the intersection of *EN(e)* and *SN(s)* is non-empty for a given *e* that matches *s* (known as the mapping intersection rule, as defined by Baldoni and Virgillito [2005]). The corresponding code for rendezvous-based routing is shown in Figure 6.12 (again taken from Baldoni and Virgillito [2005]). This time, we leave the interpretation of the algorithm as an exercise for the reader (see Exercise 6.11).

One interesting interpretation of rendezvous-based routing is to map the event space onto a *distributed hash table* (DHT). Distributed hash tables were introduced briefly in Section 4.5.1 and are examined in more detail in Chapter 10. A distributed

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

hash table 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.

It is possible to employ other peer-to-peer middleware approaches to underpin event routing in publish-subscribe systems. Indeed, this is a very active area of research with many novel and interesting proposals emerging, particularly for very large-scale systems [Carzaniga *et al.* 2001]. One specific approach is to adopt *gossiping* as a means of supporting event routing. Gossip-based approaches are a popular mechanism for achieving multicast (including reliable multicast), as discussed in Section 18.4.1. They operate by nodes in the network periodically and probabilistically exchanging events (or data) with neighbouring nodes. Through this approach, it is possible to propagate events effectively through the network without the structure imposed by other approaches. A pure gossip approach is effectively an alternative strategy for implementing flooding, as described above. However, it is possible to take into account local information and, in particular, content to achieve what is referred to as *informed gossip*. Such approaches can be particularly attractive in highly dynamic environments where network or node churn can be high [Baldoni *et al*. 2005].

6.3.3 Examples of publish-subscribe systems

We conclude this section by listing some major examples of publish-subscribe systems, providing references for further reading (see Figure 6.13). This figure also captures the design space for publish-subscribe systems, illustrating how different designs can result from decisions on subscription and distribution models and, especially, the underlying event routing strategy. Note that event routing is not required for centralized schemes, hence the blank entry in the table.

6.4 Message queues

Message queues (or more accurately, distributed message queues) are a further important category of indirect communication systems. 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, discussed further in Section 6.4.1.

We examine message queues in more detail below, considering the programming model offered by message queueing systems before addressing implementation issues. The section then concludes by presenting the Java Messaging Service (JMS) as an example of a middleware specification supporting message queues (and also publish-subscribe).

6.4.1 The programming model

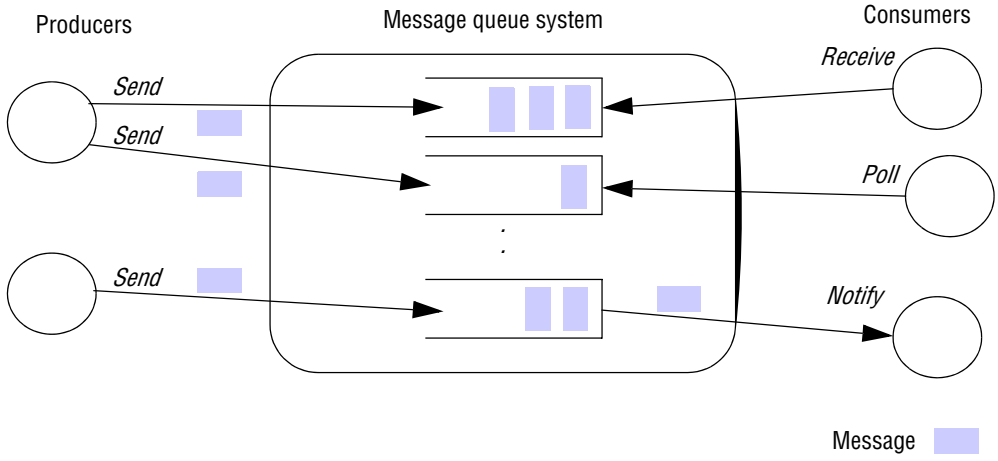
The programming model offered by message queues is very simple. It offers an approach to communication in distributed systems through queues. In particular, producer processes can *send* messages to a specific queue and other (consumer) processes can then receive messages from this queue. Three styles 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 a message is available in the associated queue.

This overall approach is captured pictorially in Figure 6.14.

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), *metadata* associated with the message, including fields such as the priority of the message and the delivery mode, and also the *body* of the message. The body is normally opaque and untouched by the message queue system. The associated

Figure 6.14 The message queue paradigm



content is serialized using any of the standard approaches described in Section 4.3; 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.

Oracle's AQ introduces an interesting twist on this basic idea to achieve better integration with (relational) databases; 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*. In particular, following the definition of reliable communication in Section 2.4.2, any message sent is eventually received (validity) and the message received is identical to the one sent, and no messages are delivered twice (integrity). 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. Detailed consideration of transactions is deferred until Chapter 16.

- 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. This could be as simple as

transforming from one byte order to another (big-endian to little-endian) or more complex, involving for example a transformation from one external data representation to another (such as SOAP to IIOP). Some systems also allow programmers to develop their own application-specific transformation in response to triggers from the underlying message queuing system. Message transformation is an important tool in dealing with heterogeneity generally and achieving Enterprise Application Integration in particular (as discussed above). Note that the term *message broker* is often used to denote a service responsible for message transformation.

- 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. See Chapter 11.

As a final word on the programming abstraction offered by message queues, it is helpful to compare the style of programming with other communication paradigms. Message queues are similar in many ways to the message-passing systems considered in Chapter 4. The difference is that whereas message-passing systems have implicit queues associated with senders and receivers (for example, the message buffers in MPI), message queuing systems have explicit queues that are third-party entities, separate from the sender and the receiver. It is this key difference that makes message queues an indirect communication paradigm with the crucial properties of space and time uncoupling.

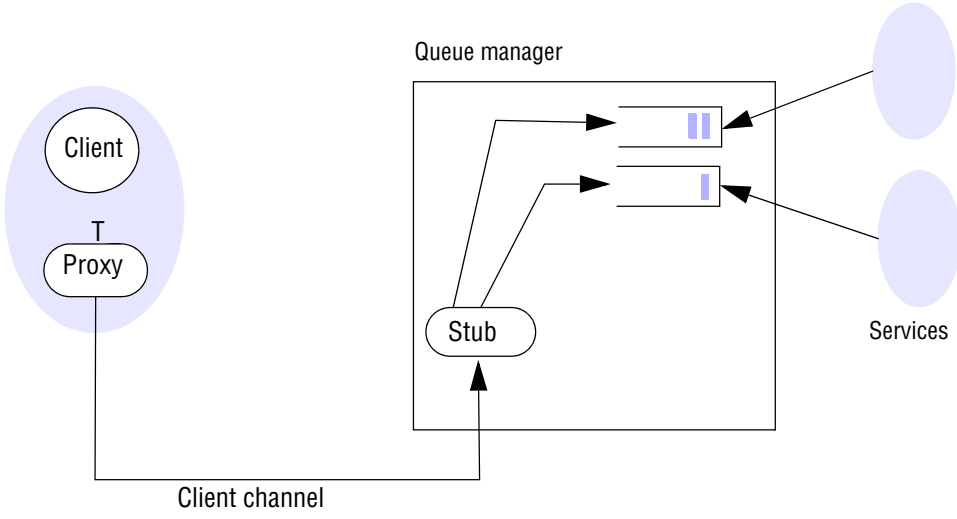
6.4.2 Implementation issues

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. To illustrate distributed architectures, we briefly consider the approach adopted in WebSphere MQ as representative of the state-of-the-art in this area.

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]. Queues in WebSphere MQ are managed by *queue managers* which host and manage queues and allow applications to access queues through the *Message Queue Interface* (MQI). The MQI is a relatively simple interface allowing applications to carry out operations such as connecting to or disconnecting from a queue (*MQCONN* and *MQDISC*) or sending/receiving messages to/from a queue (*MQPUT* and *MQGET*). Multiple queue managers can reside on a single physical server.

Client applications accessing a queue manager may reside on the same physical server. More generally, though, they will be on different machines and must then communicate with the queue manager through what is known as a *client channel*. Client channels adopt the rather familiar concept of a proxy, as introduced in Chapters 2 and 5,

Figure 6.15 A simple networked topology in WebSphere MQ



whereby MQI commands are issued on the proxy and then sent transparently to the queue manager for execution using RPC. An example of such a configuration is shown in Figure 6.15. 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). To achieve this, 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, for example to deliver certain requirements in terms of scalability and performance. Tools are provided for systems administrators to create suitable topologies and to hide the complexities of establishing message channels and routing strategies.

A wide range of topologies can be created, including trees, meshes or a bus-based configuration. To illustrate the concept of topologies further, we present one example topology often used in WebSphere MQ deployments, the hub-and-spoke topology.

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, particularly in large-scale deployments covering significant geographical areas (and possibly crossing organizational boundaries). The key to the approach is to be able to connect to a local spoke over a high-bandwidth connection, for example over a local area network (spokes may even be placed in the same physical machine as client applications to minimize latency).

Recall that communication between a client application and a queue manager uses RPC, whereas internal communication between queue managers is asynchronous (non-blocking). This means that the client application is only blocked until the message is deposited in the local queue manager (the local spoke); subsequent delivery, potentially over wide area networks, is asynchronous but guaranteed to be reliable by the WebSphere MQ middleware.

Clearly, the drawback of this architecture is that the hub can be a potential bottleneck and a single point of failure. WebSphere MQ also supports other facilities to overcome these problems, including queue manager clusters, which allow multiple instances of the same service to be supported by multiple queue managers with implicit load balancing across the different instantiations [www.redbooks.ibm.com].

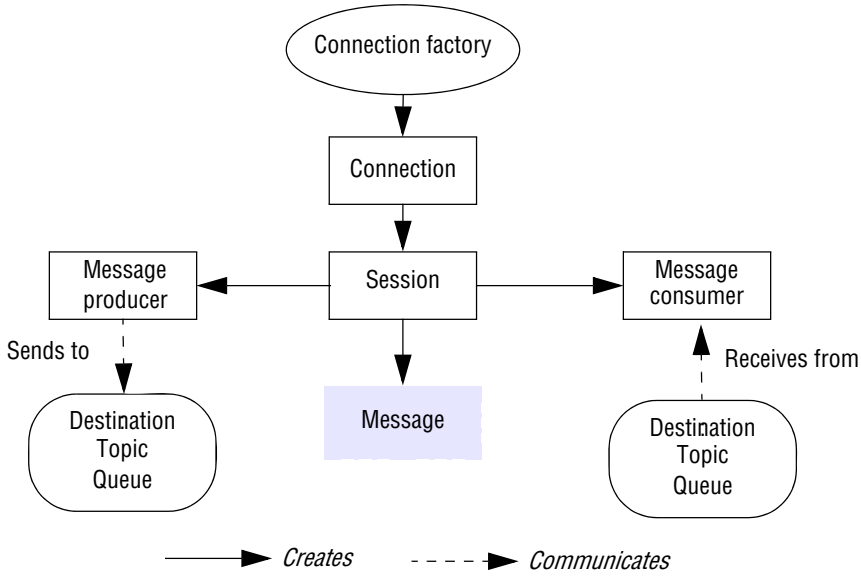
6.4.3 Case study: The Java Messaging Service (JMS)

The *Java Messaging Service* (JMS) [java.sun.com XI] is a specification of a standardized way for distributed Java programs to communicate indirectly. Most notably, as will be explained, the specification unifies the publish-subscribe and message queue paradigms at least superficially by supporting topics and queues as alternative destinations of messages. A wide variety of implementations of the common specification are now available, including Joram from OW2, Java Messaging from JBoss, Sun's Open MQ, Apache ActiveMQ and OpenJMS. Other platforms, including WebSphere MQ, also provide a JMS interface on to their underlying infrastructure.

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).

Figure 6.16 The programming model offered by JMS



- A JMS destination is an object supporting indirect communication in JMS. It is either a JMS topic or a JMS queue.

Programming with JMS • The programming model offered by the JMS API is captured in Figure 6.16. To interact with a JMS provider, it is first necessary to create a *connection* between a client program and the provider. This is created through a *connection factory* (a service responsible for creating connections with the required properties). The resultant connection is a logical channel between the client and provider; the underlying implementation may, for example, map onto a TCP/IP socket if implemented over the Internet. Note that two types of connection can be established, a *TopicConnection* or a *QueueConnection*, thus enforcing a clear separation between the two modes of operation within given connections.

Connections can be used to create one or more *sessions* – a session is a series of operations involving the creation, production and consumption of messages related to a logical task. The resultant session object also supports operations to create *transactions*, supporting all-or-nothing execution of a series of operations, as discussed in Section 6.4.1. There is a clear distinction between topic sessions and queue sessions in that a *TopicConnection* can support one or more topic sessions and a *QueueConnection* can support one or more queue sessions, but it is not possible to mix session styles in a connection. Thus, the two styles of operation are integrated in a rather superficial way.

The session object is central to the operation of JMS, supporting methods for the creation of messages, message producers and message consumers:

- In JMS, a *message* consists of three parts: a *header*, a set of *properties* and the *body* of the message. The header contains all the information needed to identify and route the message, including the destination (a reference to either a topic or a

Figure 6.17 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
}
```

queue), the priority of the message, the expiration date, a message ID and a timestamp. Most of these fields are created by the underlying system, but some can be filled in specifically through the associated constructor methods. Properties are all user-defined and can be used to associate other application-specific metadata elements with a message. For example, if implementing a context-aware system (as discussed in Chapter 19), the properties can be used to express additional context associated with the message, including a location field. As in the general description of message queue systems, this body is opaque and untouched by the system. In JMS, the body can be any one of a text message, a byte stream, a serialized Java object, a stream of primitive Java values or a more structured set of name/value pairs.

- A *message producer* is an object used to publish messages under a particular topic or to send messages to a queue.
- A *message consumer* is an object used to subscribe to messages concerned with a given topic or to receive messages from a queue. The consumer is more complicated than the producer, for two reasons. First, it is possible to associate filters with message consumers by specifying what is known as a *message selector* – a predicate defined over the values in the header and properties parts of a message (not the body). A subset of the database query language SQL is used to specify properties. This could be used, for example, to filter messages from a

Figure 6.18 Java class *FireAlarmConsumerJMS*

```
import javax.jms.*;
import javax.naming.*;

public class FireAlarmConsumerJMS {
    public String await() {
    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
        TopicSubscriber topicSub = topicSess.createSubscriber(topic);        10
        topicSub.start();                                                    11
        TextMessage msg = (TextMessage) topicSub.receive();                  12
        return msg.getText();                                                13
        } catch (Exception e) {                                              14
            return null;                                                     15
    }                                                                        16
}
```

given location in the context-aware example above. Second, there are two modes provided for receiving messages: the program either can block using a *receive* operation or it can establish a *message listener* object which must provide an *onMessage* method that is invoked whenever a suitable message is identified.

A simple example • To illustrate the use of JMS, we return to our example of Section 6.2.3, the fire alarm service, and show how this would be implemented in JMS. We choose the topic-based publish-subscribe service as this is intrinsically a one-to-many application, with the alarm producing alarm messages targeted towards many consumer applications.

The code for the fire alarm object is shown in Figure 6.17. It is more complicated than the equivalent JGroups example mainly because of the need to create a connection, session, publisher and message, as shown in lines 6–11. This is relatively straightforward apart from the parameters of *createTopicSession*, which are whether the session should be transactional (*false* in this case) and the mode of acknowledging messages (*AUTO_ACKNOWLEDGE* in this example, which means a session automatically acknowledges the receipt of a message). There is additional complexity associated with finding the connection factory and topic in the distributed environment (the complexity of connecting to a named channel in JGroups is all hidden in the *connect* method). This is achieved using JNDI (the Java Naming and Directory Interface) in lines 2 to 5. This is included for completeness and it is assumed that readers can appreciate

the purpose of these lines of code without further explanation. Lines 12 and 13 contain the crucial code to create a new message and then publish it to the appropriate topic. The code to create a new instance of the *FireAlarmJMS* class and then raise an alarm is:

```
FireAlarmJMS alarm = new FireAlarmJMS();
alarm.raise();
```

The corresponding code for the receiver end is very similar and is shown in Figure 6.18. Lines 2–9 are identical and create the required connection and session, respectively. This time, though, an object of type *TopicSubscriber* is created next (line 10), and the *start* method in line 11 starts this subscription, enabling messages to be received. The blocking *receive* in line 12 then awaits an incoming message and line 13 returns the textual contents of this message as a string. This class is used as follows by a consumer:

```
FireAlarmConsumerJMS alarmCall = new FireAlarmConsumerJMS();
String msg = alarmCall.await();
System.out.println("Alarm received: "+msg);
```

Overall this case study has illustrated how 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

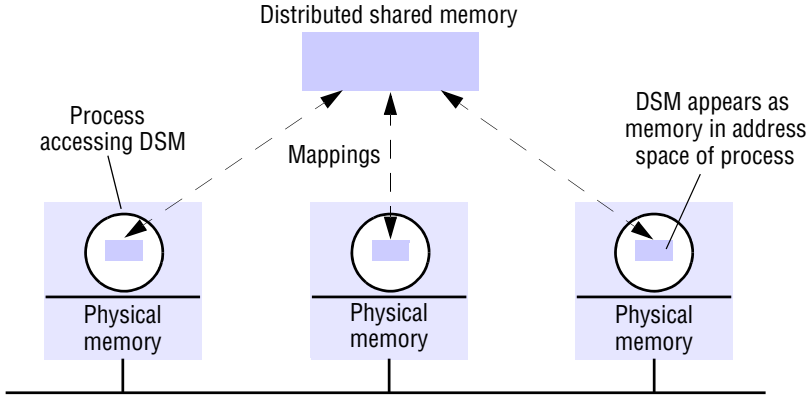
In this section, we examine indirect communication paradigms that offer an abstraction of shared memory. We look briefly at distributed shared memory techniques that were developed principally for parallel computing before moving on to tuple space communication, an approach that allows programmers to read and write tuples from a shared tuple space. Whereas distributed shared memory operates at the level of reading and writing bytes, tuple spaces offer a higher-level perspective in the form of semi-structured data. In addition, whereas distributed shared memory is accessed by address, tuple spaces are *associative*, offering a form of content-addressable memory [Gelernter 1985].

Chapter 18 of the fourth edition of this book provided in-depth coverage of distributed shared memory, including consistency models and several case studies. This chapter can be found on the companion web site for the book [www.cdk5.net/dsm].

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 (see Figure 6.19).

Figure 6.19 The distributed shared memory abstraction



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).

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 to be discussed in Chapter 18, as well as to those of caching shared files, discussed in Chapter 12.

One of the first notable examples of a DSM implementation was the Apollo Domain file system [Leach *et al*. 1983], in which processes hosted by different workstations share files by mapping them simultaneously into their address spaces. This example shows that distributed shared memory can be persistent. That is, it may outlast the execution of any process or group of processes that accesses it and be shared by different groups of processes over time.

The significance of DSM first grew alongside the development of shared-memory multiprocessors (discussed further in Section 7.3). Much research has gone into investigating algorithms suitable for parallel computation on these multiprocessors. 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 [Dubois *et al*. 1988]. 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. This is a hierarchical

architecture in which the four-processor boards are connected using a high-performance switch or higher-level bus. 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 (again, see Section 7.3), 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 shared-memory 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 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; but how can memory be shared between computers with, for example, different integer representations?

Synchronization between processes is achieved in the message model through message passing primitives themselves, using techniques such as the lock server implementation discussed in Chapter 16. 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). Chapter 7 briefly discusses such synchronization objects in the context of programming with threads.

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 [Carter *et al*. 1991] – 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.

There is no definitive answer as to whether DSM is preferable to message passing for any particular application. DSM remains a tool whose ultimate status depends upon the efficiency with which it can be implemented.

6.5.2 Tuple space communication

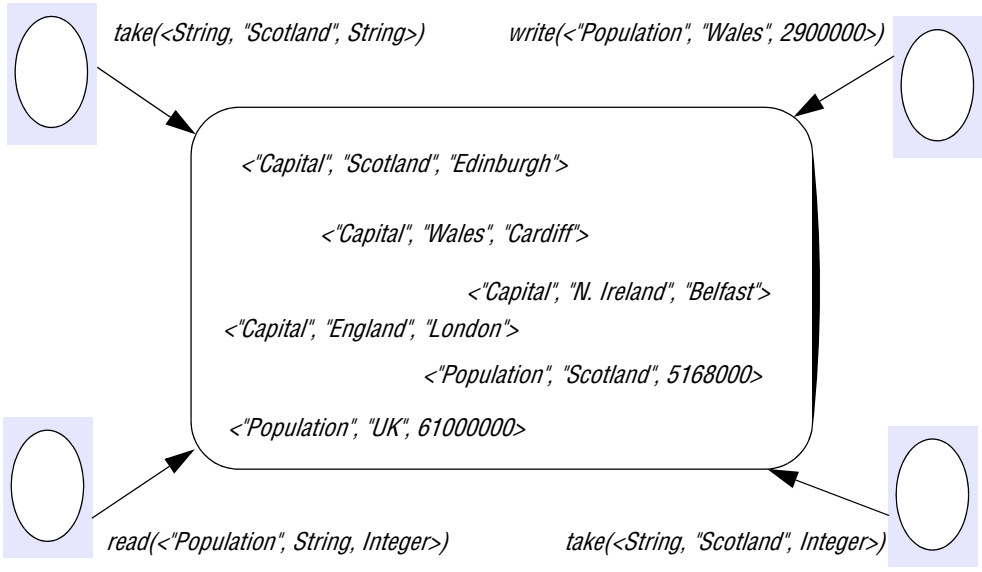
Tuple spaces were first introduced by David Gelernter from Yale University as a novel form of distributed computing based on what he refers to as *generative communication* [Gelernter 1985]. In this approach, processes communicate indirectly by placing tuples in a tuple space, from which other processes can read or remove them. Tuples do not have an address but are accessed by pattern matching on content (content-addressable memory, as discussed by Gelernter [1985]). The resultant Linda programming model has been highly influential and has led to significant developments in distributed programming including systems such as Agora [Bisiani and Forin 1988] and, more significantly, JavaSpaces from Sun (discussed below) and IBM's TSpaces. Tuple space communication has also been influential in the field of ubiquitous computing, for reasons that are explored in depth in Chapter 19.

This section provides an examination of the tuple space paradigm as it applies to distributed computing. We start by examining the programming model offered by tuple spaces before briefly considering the associated implementation issues. The section then concludes by examining the JavaSpaces specification as a case study, illustrating how tuple spaces have evolved to embrace the object-oriented world.

The 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. To enable processes to

Figure 6.20 The tuple space abstraction



synchronize their activities, the read and take operations both block until there is a matching tuple in the tuple space. A tuple specification includes the number of fields and the required values or types of the fields. For example, *take*(<*String*, *integer*>) could extract either <"*fred*", *1958*> or <"*sid*", *1964*>; *take*(<*String*, *1958*>) would extract only <"*fred*", *1958*> of those two.

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>);
```

A further illustration of the tuple space paradigm is given in Figure 6.20. This tuple space contains a range of tuples representing geographical information about countries in the United Kingdom, including populations and capital cities. The *take* operation *take(<String, "Scotland", String>)* will match <*"Capital", "Scotland", "Edinburgh"*>, whereas *take(<String, "Scotland", Integer>)* will match <*"Population", "Scotland", 5168000*>. The *write* operation *write(<"Population", "Wales, 2900000>)* will insert a new tuple in the tuple space with information on the population of Wales. Finally, *read(<"Population", String, Integer)* can match the equivalent tuples for the populations of the UK, Scotland or indeed Wales, if this operation is executed after the corresponding *write* operation. One will be selected nondeterministically by the tuple

space implementation and, with this being a *read* operation, the tuple will remain in the tuple space.

Note that *write*, *read* and *take* are known as *out*, *rd* and *in* in Linda; we use the more descriptive former names throughout this book. This terminology is also used in JavaSpaces, discussed in a case study below.

Properties associated with tuple spaces: Gelernter [1985] presents some interesting properties associated with tuple space communication, highlighting in particular both space and time uncoupling as discussed in Section 6.1:

> *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.

Gelernter [1985] also explores a range of other properties associated with the rather flexible style of naming employed in Linda (referred to as *free naming*). The interested reader is directed to Gelernter's paper for more information on this topic.

Variations on a theme: Since the introduction of Linda, refinements have been proposed to the original model:

- The original Linda model proposed a single, global tuple space. This is not optimal in large systems, as it leads to the danger of unintended aliasing of tuples: as the number of tuples in a tuple space increases, there is an increasing chance of a *read* or *take* matching a tuple by accident. This is particularly likely when matching on types, such as with *take*(<*String, integer*>), as mentioned above. Given this, a number of systems have proposed *multiple tuple spaces*, including the ability to dynamically create tuple spaces, introducing a degree of scoping into the system (see, for example, the JavaSpaces case study below).
- Linda was anticipated to be implemented as a centralized entity but later systems have experimented with *distributed* implementations of tuple spaces (including strategies to provide more fault tolerance). Given the importance of this topic to this book, we focus on this in the implementation issues subsection below.
- Researchers have also experimented with modifying or extending the operations provided in tuple spaces and adapting the underlying semantics. One rather interesting proposal is to unify the concepts of tuples and tuple spaces by modelling everything as (unordered) sets – that is, tuple spaces are sets of tuples and tuples are sets of values, which may now also include tuples. This variant is known as Bauhaus Linda [Carriero *et al.* 1995].
- Perhaps most interestingly, recent implementations of tuple spaces have moved from tuples of typed data items to data objects (with attributes), turning the tuple space into an *object space*. This proposal is adopted, for example, in the influential system JavaSpaces, discussed in more detail below.

Implementation issues • Many of the implementations of tuple spaces adopt a centralized solution where the tuple space resource is managed by a single server. This has advantages in terms of simplicity, but such solutions are clearly not fault tolerant and also will not scale. Because of this, distributed solutions have been proposed.

Replication: Several systems have proposed the use of *replication* to overcome the problems identified above [Bakken and Schlichting 1995, Bessani *et al.* 2008, Xu and Liskov 1989].

The proposals from Bakken and Schlichting [1995] and Bessani *et al*. [2008] adopt a similar approach to replication, referred to as the *state machine approach* and discussed further in Chapter 18. This approach assumes that a tuple space behaves like a state machine, maintaining state and changing this state in response to events received from other replicas or from the environment. To ensure consistency the replicas (i) must start in the same state (an empty tuple space), (ii) must execute events in the same order and (iii) must react deterministically to each event. The key second property can be guaranteed by adopting a totally ordered multicast algorithm, as discussed in Section 6.2.2.

Xu and Liskov [1989] adopt a different approach, which optimizes the replication strategy by using the semantics of the particular tuple space operations. In this proposal, updates are carried out in the context of the current *view* (the agreed set of replicas) and tuples are also partitioned into distinct *tuple sets* based on their associated logical names (designated as the first field in the tuple). The system consists of a set of workers carrying out computations on the tuple space, and a set of tuple space replicas. A given physical node can contain any number of workers, replicas or indeed both; a given worker therefore may or may not have a local replica. Nodes are connected by a communications network that may lose, duplicate or delay messages and can deliver messages out of order. Network partitions can also occur.

A *write* operation is implemented by sending a multicast message over the unreliable communications channel to all members of the view. On receipt, members place this tuple into their replica and acknowledge receipt. The *write* request is repeated until all acknowledgements are received. For the correct operation of the protocol, replicas must detect and acknowledge duplicate requests, but not carry out the associated *write* operations.

The *read* operation consists of sending a multicast message to all replicas. Each replica seeks a match and returns this match to the requesting site. The first tuple returned is delivered as the result of the *read*. This may come from a local node, but given that many workers will not have a local replica, this is not guaranteed.

The *take* operation is more complex because of the need to agree on the tuple to be selected and to remove this agreed tuple from all copies. The algorithm proceeds in two phases. In phase 1, the tuple specification is sent to all replicas, and the replica attempts to acquired the lock on the associated tuple set to serialize *take* requests on the replicas (*write* and *read* operations are unaffected by the lock); if the lock cannot be acquired, the *take* request is refused. Each replica that succeeds in obtaining the lock responds with the *set* of matching tuples. This step is repeated until all replicas have accepted the request and responded. The initiating process can then select one tuple from the intersection of all the replies and return this as the result of the *take* request. If it is

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.

not possible to obtain a majority of locks, the replicas are asked to release their locks and phase 1 repeats.

In phase 2, this tuple must be removed from all replicas. This is achieved by repeated multicasts to the replicas in the view until all have acknowledged deletion. As with *write* requests, it is necessary for replicas to detect repeat requests in phase 2 and to simply send another acknowledgement without carrying out another deletion (otherwise multiple tuples could erroneously be deleted at this stage).

The steps involved for each operation are summarized in Figure 6.21. Note that a separate algorithm is required to manage view changes if node failures occur or the network partitions (see Xu and Liskov [1989] for details).

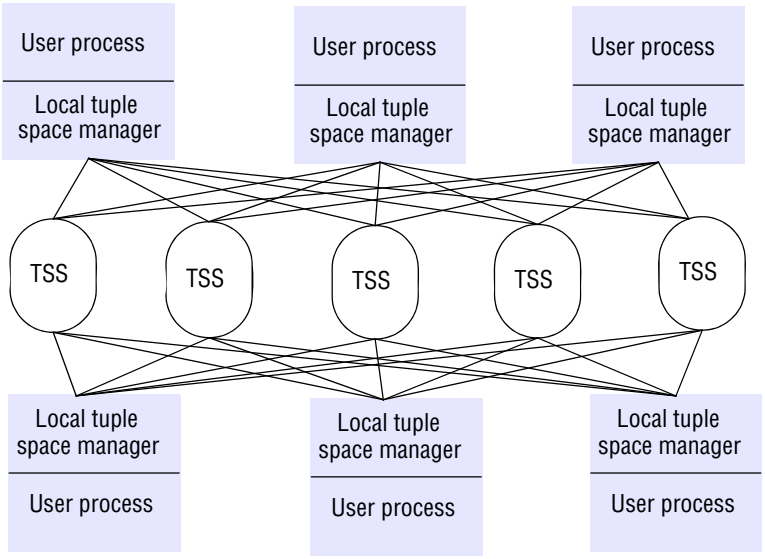
This algorithm is designed to minimize delay given the semantics of the three tuple space operations:

read operations only block until the first replica responds to the request.

take operations block until the end of phase 1, when the tuple to be deleted has been agreed.

write operations can return immediately.

Figure 6.22 Partitioning in the York Linda Kernel



This, though, introduces unacceptable levels of concurrency. For example, a *read* operation may access a tuple that should have been deleted in the second phase of a *take* operation. Therefore additional levels of concurrency control are required. In particular, Xu and Liskov [1989] introduce the following additional constraints:

- The operations of each worker must be executed at each replica in the same order as they were issued by the worker;.
- A *write* operation must not be executed at any replica until all previous *take* operations issued by the same worker have completed at all replicas in the worker's view.

A further example of using replication is provided in Chapter 19, where we present the L^{2}imbo approach, which uses replication to provide high availability in mobile environments [Davies *et al.* 1998].

Other approaches: A range of other approaches have been employed in the implementation of the tuple space abstraction, including partitioning of the tuple space over a number of nodes and mapping onto peer-to-peer overlays:

- The Linda Kernel developed at the University of York [Rowstron and Wood 1996] adopts an approach in which tuples are partitioned across a range of available tuple space servers (TSSs), as illustrated in Figure 6.22. There is no replication of tuples; that is, there is only one copy of each tuple. The motivation is to increase performance of the tuple space, especially for highly parallel computation.When a tuple is placed in tuple space, a hashing algorithm is used to select one of the tuple space servers to be used. The implementation of *read* or *take* is slightly more complex, as a tuple specification is provided that may specify types or values of the associated fields. The hashing algorithm uses this

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

specification to generate a set of possible servers that may contain matching tuples, and a linear search must then be employed until a matching tuple is discovered. Note that because there is only a single copy of a given tuple, the implementation of *take* is greatly simplified.

- Some implementations of tuple spaces have adopted peer-to-peer approaches in which all nodes cooperate to provide the tuple space service. This approach is particularly attractive given the intrinsic availability and scalability of peer-to-peer solutions. Examples of peer-to-peer implementations include PeerSpaces [Busi *et al.* 2003], which is developed using the JXTA peer-to-peer middleware [jxta.dev.java.net], LIME and TOTA (the latter two systems feature in Chapter 19).

Case study: JavaSpaces • JavaSpaces is a tool for tuple space communication developed by Sun [java.sun.com X, [java.sun.com VI]. More specifically, Sun provides the specification of a JavaSpaces service, and third-party developers are then free to offer implementations of JavaSpaces (significant implementations include GigaSpaces [www.gigaspaces.com] and Blitz [www.dancres.org]). The tool is strongly dependent on Jini (Sun's discovery service, discussed further in Section 19.2.1), as will become apparent below. The Jini Technology Starter Kit also includes an implementation of JavaSpaces, referred to as Outrigger.

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).

Figure 6.24 Java class *AlarmTupleJS*

```
import net.jini.core.entry.*;

public class AlarmTupleJS implements Entry {
    public String alarmType;

    public AlarmTupleJS() {
    }

    public AlarmTupleJS(String alarmType) {
            this.alarmType = alarmType;
    }
}
```

Programming with JavaSpaces: JavaSpaces allows the programmer to create any number of instances of a *space*, where a space is a shared, persistent repository of *objects* (thus offering an object space in the terminology introduced above). More specifically, an item in a JavaSpace is referred to as an *entry*: a group of objects contained in a class that implements *net.jini.core.entry.Entry*. Note that with entries containing objects (rather than tuples), it is possible to associate arbitrary behaviour with entries, thus significantly increasing the expressive power of the approach.

The operations defined on JavaSpaces are summarized in Figure 6.23 (showing the full signatures of each of the operations) and can be described as follows:

- A process can place an entry into a JavaSpace instance with the *write* method. As with Jini, an entry can have an associated *lease* (see Section 5.4.3), which is the time for which access is granted to the associated objects. This can be forever (*Lease.FOREVER*) or can be a numerical value specified in milliseconds. After this period, the entry is destroyed. The *write* operation can also be used in the context of a *transaction*, as discussed below (a value of *null* indicates that this is not a transactional operation). The *write* operation returns a *Lease* value representing the lease granted by the JavaSpace (which may be less than the time requested).
- A process can access an entry in a JavaSpace with either the *read* or *take* operation; *read* returns a copy of a matching entry and *take* removes a matching entry from the JavaSpace (as in the general programming model presented above). The matching requirements are specified by a *template*, which is of type *entry*. Particular fields in the template may be set to specific values and others can be left unspecified. A match is then defined as an entry that is of the same class as the template (or a valid subclass) and where there is an exact match for the set of specified values. As with *write*, *read* and *take* can be carried out in the context of a specified transaction (discussed below). The two operations are also blocking; the final parameter specifies a timeout representing the maximum length of time that a particular process or thread will block, for example to deal with the failure of a process supplying a given entry. The *readIfExists* and *takeIfExists* operations

Figure 6.25 Java class *FireAlarmJS*

```
import net.jini.space.JavaSpace;

public class FireAlarmJS {

    public void raise() {
        try {
            JavaSpace space = SpaceAccessor.findSpace("AlarmSpace");
            AlarmTupleJS tuple = new AlarmTupleJS("Fire!");
            space.write(tuple, null, 60*60*1000);
        catch (Exception e) {
        }
    }
}
```

are equivalent to *read* and *take*, respectively, but these operations will return a matching entry if one exists; otherwise, they will return *null*.

- The *notify* operation uses Jini distributed event notification, mentioned in Section 6.3 to register an interest in a given event – in this case, the arrival of entries matching a given template. This registration is governed by a lease, that is, the length of time the registration should persist in the JavaSpace. Notification is via a specified *RemoteEventListener* interface. Once again, this operation can be carried out in the context of a specified transaction.

As mentioned throughout the discussion above, operations in JavaSpaces can take place in the context of a *transaction*, ensuring that either all or none of the operations will be executed. Transactions are distributed entities and can span multiple JavaSpaces and multiple participating processes. Discussion of the general concept of transactions is deferred until Chapter 16.

A simple example: We conclude this examination of JavaSpaces by presenting an example, the intelligent fire alarm example first introduced in Section 6.2.3 and revisited in Section 6.4.3. In this example, there is a need to disseminate an emergency message to all recipients when a fire event is detected.

We start by defining an entry object of type *AlarmTupleJS*, as shown in Figure 6.24. This is relatively straightforward and shows the creation of a new entry with one field, the *alarmType*. The associated fire alarm code is shown in Figure 6.25. The first step in raising an alarm is to gain access to an appropriate instance of a JavaSpace (called *"AlarmSpace"*), which we assume is already created. Most implementations of JavaSpaces provide utility functions for this and, for simplicity, this is what we show in this code, using a utility class *SpaceAccessor* and method *findSpace* as provided in GigaSpaces (for convenience, a copy of this class is provided on the companion web site for the book [www.cdk5.net]). An entry is then created as an instance of the previously defined *AlarmTupleJS*. This entry has only one field, a string called *alarmType*, and this

Figure 6.26 Java class *FireAlarmReceiverJS*

```
import net.jini.space.JavaSpace;

public class FireAlarmConsumerJS {

    public String await() {
        try {
            JavaSpace space = SpaceAccessor.findSpace();
            AlarmTupleJS template = new AlarmTupleJS("Fire!");
            AlarmTupleJS recvd = (AlarmTupleJS) space.read(template, null,
                                                    Long.MAX_VALUE);
            return recvd.alarmType;
        }
        catch (Exception e) {
            return null;
        }
    }
}
```

is set to *"Fire!"*". Finally, this entry is placed into the JavaSpace using the *write* method, where it will remain for one hour. This code can then be called using the following:

```
FireAlarmJS alarm = new FireAlarmJS();
alarm.raise();
```

The corresponding code for the consumer end is shown in Figure 6.26. Access to the appropriate JavaSpace is obtained in the same manner. Following this, a template is created, the single field is set to *"Fire!"*, and an associated *read* method is invoked. Note that by setting the field to *"Fire!"*, we ensure that only entries with this type *and* this value will be returned (leaving the field blank would make any entry of type *AlarmTupleJS* a valid match). This is called as follows in a consumer:

```
FireAlarmConsumerJS alarmCall = new FireAlarmConsumerJS();
String msg = alarmCall.await();
System.out.println("Alarm received: " + msg);
```

This simple example illustrates how easy it is to write multiparty applications using JavaSpaces that are both time- and space-uncoupled.

6.6 Summary

This chapter has examined indirect communication in detail, complementing the study of remote invocation paradigms in the previous chapter. We defined indirect communication in terms of communication through an intermediary, with a resultant uncoupling between producers and consumers of messages. This leads to interesting properties, particularly in terms of dealing with change and establishing fault-tolerant strategies.

We have considered five styles of indirect communication in this chapter:

- group communication;
- publish-subscribe systems;
- message queues;
- distributed shared memory;
- tuple spaces.

The discussion has emphasized their commonalities in terms of all supporting indirect communication through forms of intermediary including groups, channels or topics, queues, shared memory or tuple spaces. Content-based publish-subscribe systems communicate through the publish-subscribe system as a whole, with subscriptions effectively defining logical channels managed by content-based routing.

As well as focusing on the commonalities, it is instructive to consider the key differences between the various approaches. We start by reconsidering the level of space and time uncoupling, picking up on the discussion in Section 6.1. All the techniques considered in this chapter exhibit space uncoupling in that messages are directed to an intermediary and not to any specific recipient or recipients. The position with respect to time uncoupling is more subtle and dependent on the level of persistency in the paradigm. Message queues, distributed shared memory and tuple spaces all exhibit time uncoupling. The other paradigms may, depending on the implementation. For example, in group communication, it is possible in some implementations for a receiver to join a group at an arbitrary point in time and to be brought up-to-date with respect to previous message exchanges (this is an optional feature in JGroups, for example, selected by constructing an appropriate protocol stack). Many publish-subscribe systems do not support persistency of events and hence are not time-uncoupled, but there are exceptions. JMS, for example, does support persistent events, in keeping with its integration of publish-subscribe and message queues.

The next observation is that the initial three techniques (groups, publish-subscribe and message queues) offer a programming model that emphasizes *communication* (through messages or events), whereas distributed shared memory and tuple spaces offer a more *state-based abstraction*. This is a fundamental difference and one that has significant repercussions in terms of scalability; in general terms, the communication-based abstractions have the potential to scale to very large scale systems with appropriate routing infrastructure (although this is not the case for group communication because of the need to maintain group membership, as discussed in Section 6.2.2). In contrast, the two state-based approaches have limitations with respect to scaling. This stems from the need to maintain consistent views of the shared state, for example between multiple readers and writers of shared memory. The situation with tuple spaces is a bit more subtle given the immutable nature of tuples. The key problem rests with implementing the destructive read operation, *take*, in a large-scale system; it is an interesting observation that without this operation, tuple spaces look very much like publish-subscribe systems (and hence are potentially highly scalable).

Most of the above systems also offer *one-to-many* styles of communication, that is, multicast in terms of the communication-based services and global access to shared values in the state-based abstractions. The exceptions are message queuing, which is fundamentally *point-to-point* (and hence often offered in combination with publish-

Figure 6.27 Summary of indirect communication styles

	Groups	*Publish-subscribe systems*	*Message queues*	*DSM*	*Tuple spaces*
Space-uncoupled	Yes	Yes	Yes	Yes	Yes
Time-uncoupled	Possible	Possible	Yes	Yes	Yes
Style of service	Communication-based	Communication-based	Communication-based	State-based	State-based
Communication pattern	1-to-many	1-to-many	1-to-1	1-to-many	1-1 or 1-to-many
Main intent	Reliable distributed computing	Information dissemination or EAI; mobile and ubiquitous systems	Information dissemination or EAI; commercial transaction processing	Parallel and distributed computation	Parallel and distributed computation; mobile and ubiquitous systems
Scalability	Limited	Possible	Possible	Limited	Limited
Associative	No	Content-based publish-subscribe only	No	No	Yes

subscribe systems in commercial middleware), tuple spaces, which can be either one-to-many or point-to-point depending on whether receiving processes use the *read* or *take* operations, respectively.

There are also differences in *intent* in the various systems. Group communication is mainly designed to support reliable distributed systems, and hence the emphasis is on providing algorithmic support for reliability and ordering of message delivery. Interestingly, the algorithms to ensure reliability and ordering (especially the latter) can have a significant negative effect on scalability for similar reasons to maintaining consistent views of shared state. Publish-subscribe systems have largely been targeted at information dissemination (for example, in financial systems) and for Enterprise Application Integration. Finally, the shared memory approaches have generally been applied in parallel and distributed processing, including in the Grid community (although tuple spaces have been used effectively across a variety of application domains). Both publish-subscribe systems and tuple space communication have found favour in mobile and ubiquitous computing due to their support for volatile environments (as discussed in Chapter 19).

One other key issue associated with the five schemes is that both content-based publish-subscribe and tuple spaces offer a form of *associative addressing* based on content, allowing pattern matching between subscriptions and events or templates against tuples, respectively. The other approaches do not.

This discussion is summarized in Figure 6.27.

We have not considered issues related to quality of service in this analysis. Many message queue systems do offer intrinsic support for reliability in the form of transactions. More generally, however, quality of service remains a key challenge for indirect communication paradigms. Indeed, space and time uncoupling by their very nature make it difficult to reason about end-to-end properties of the system, such as real-time behaviour or security, and hence this is an important area for further research.

EXERCISES

6.1 Construct an argument as to why indirect communication may be appropriate in volatile environments. To what extent can this be traced to time uncoupling, space uncoupling or indeed a combination of both? *page 230*

6.2 Section 6.1 states that message passing is both time- and space-coupled – that is, messages are both directed towards a particular entity and require the receiver to be present at the time of the message send. Consider the case, though, where messages are directed towards a name rather than an address and this name is resolved using DNS. Does such a system exhibit the same level of indirection? *page 231, Section 13.2.3*

6.3 Section 6.1 refers to systems that are space-coupled but time- uncoupled – that is, messages are directed towards a given receiver (or receivers), but that receiver can have a lifetime independent from the sender's. Can you construct a communication paradigm with these properties? For example, does email fall into this category? *page 231*

6.4 As a second example, consider the communication paradigm referred to as queued RPC, as introduced in Rover [Joseph *et al*. 1997]. Rover is a toolkit to support distributed systems programming in mobile environments where participants in communication may become disconnected for periods of time. The system offers the RPC paradigm and hence calls are directed towards a given server (clearly space-coupled). The calls, though, are routed through an intermediary, a queue at the *sending* side, and are maintained in the queue until the receiver is available. To what extent is this time-uncoupled? Hint: consider the almost philosophical question of whether a recipient that is temporarily unavailable exists at that point in time. *page 231, Chapter 19*

6.5 If a communication paradigm is asynchronous, is it also time-uncoupled? Explain your answer with examples as appropriate. *page 232*

6.6 In the context of a group communication service, provide example message exchanges that illustrate the difference between causal and total ordering. *page 236*

6.7 Consider the *FireAlarm* example as written using JGroups (Section 6.2.3). Suppose this was generalized to support a variety of alarm types, such as fire, flood, intrusion and so on. What are the requirements of this application in terms of reliability and ordering? *page 230, page 240*

6.8 Suggest a design for a notification mailbox service that is intended to store notifications on behalf of multiple subscribers, allowing subscribers to specify when they require notifications to be delivered. Explain how subscribers that are not always active can make use of the service you describe. How will the service deal with subscribers that crash while they have delivery turned on? *page 245*

6.9 In publish-subscribe systems, explain how channel-based approaches can trivially be implemented using a group communication service? Why is this a less optimal strategy for implementing a content-based approach? *page 245*

6.10 Using the filtering-based routing algorithm in Figure 6.11 as a starting point, develop an alternative algorithm that illustrates how the use of advertisements can result in significant optimization in terms of message traffic generated. *page 251*

6.11 Construct a step-by-step guide explaining the operation of the alternative rendezvous-based routing algorithm shown in Figure 6.12. *page 252*

6.12 Building on your answer to Exercise 6.11, discuss two possible implementations of *EN(e)* and *SN(s)*. Why must the intersection of *EN(e)* and *SN(s)* be non-null for a given *e* that matches *s* (the intersection rule)? Does this apply in your possible implementations? *page 252*

6.13 Explain how the loose coupling inherent in message queues can aid with Enterprise Application Integration. As in Exercise 6.1, consider to what extent this can be traced to time uncoupling, space uncoupling or a combination of both. *page 254*

6.14 Consider the version of the *FireAlarm* program written in JMS (Section 6.4.3). How would you extend the consumer to receive alarms only from a given location? *page 261*

6.15 Explain in which respects DSM is suitable or unsuitable for client-server systems. *page 262*

6.16 Discuss whether message passing or DSM is preferable for fault-tolerant applications. *page 262*

6.17 Assuming a DSM system is implemented in middleware without any hardware support and in a platform-neutral manner, how would you deal with the problem of differing data representations on heterogeneous computers? Does your solution extend to pointers? *page 262*

6.18 How would you implement the equivalent of a remote procedure call using a tuple space? What are the advantages and disadvantages of implementing a remote procedure call–style interaction in this way? *page 265*

6.19 How would you implement a semaphore using a tuple space? *page 265*

6.20 Implement a replicated tuple space using the algorithm of Xu and Liskov [1989]. Explain how this algorithm uses the semantics of tuple space operations to optimize the replication strategy. *page 269*