

DISTRIBUTED SYSTEMS

Concepts and Design

Fifth Edition

George Coulouris
Jean Dollimore
Tim Kindberg
Gordon Blair



PEER-TO-PEER SYSTEMS

- 10.1 Introduction
- 10.2 Napster and its legacy
- 10.3 Peer-to-peer middleware
- 10.4 Routing overlays
- 10.5 Overlay case studies: Pastry, Tapestry
- 10.6 Application case studies: Squirrel, OceanStore, Ivy
- 10.7 Summary

Peer-to-peer systems represent a paradigm for the construction of distributed systems and applications in which data and computational resources are contributed by many hosts on the Internet, all of which participate in the provision of a uniform service. Their emergence is a consequence of the very rapid growth of the Internet, embracing many millions of computers and similar numbers of users requiring access to shared resources.

A key problem for peer-to-peer systems is the placement of data objects across many hosts and subsequent provision for access to them in a manner that balances the workload and ensures availability without adding undue overheads. We describe several recently developed systems and applications that are designed to achieve this.

Peer-to-peer middleware systems are emerging that have the capacity to share computing resources, storage and data present in computers 'at the edges of the Internet' on a global scale. They exploit existing naming, routing, data replication and security techniques in new ways to build a reliable resource-sharing layer over an unreliable and untrusted collection of computers and networks.

Peer-to-peer applications have been used to provide file sharing, web caching, information distribution and other services, exploiting the resources of tens of thousands of machines across the Internet. They are at their most effective when used to store very large collections of immutable data. Their design diminishes their effectiveness for applications that store and update mutable data objects.

10.1 Introduction

The demand for services in the Internet can be expected to grow to a scale that is limited only by the size of the world's population. The goal of peer-to-peer systems is to enable the sharing of data and resources on a very large scale by eliminating any requirement for separately managed servers and their associated infrastructure.

The scope for expanding popular services by adding to the number of the computers hosting them is limited when all the hosts must be owned and managed by the service provider. Administration and fault recovery costs tend to dominate. The network bandwidth that can be provided to a single server site over available physical links is also a major constraint. System-level services such as Sun NFS (Section 12.3), the Andrew File System (Section 12.4) or video servers (Section 20.6.1) and application-level services such as Google, Amazon or eBay all exhibit this problem to varying degrees.

Peer-to-peer systems aim to support useful distributed services and applications using data and computing resources available in the personal computers and workstations that are present in the Internet and other networks in ever-increasing numbers. This is increasingly attractive as the performance difference between desktop and server machines narrows and broadband network connections proliferate.

But there is another, broader aim: one author [Shirky 2000] has defined peer-to-peer applications as 'applications that exploit resources available at the edges of the Internet – storage, cycles, content, human presence'. Each type of resource sharing mentioned in that definition is already represented by distributed applications available for most types of personal computer. The purpose of this chapter is to describe some general techniques that simplify the construction of peer-to-peer applications and enhance their scalability, reliability and security.

Traditional client-server systems manage and provide access to resources such as files, web pages or other information objects located on a single server computer or a small cluster of tightly coupled servers. With such centralized designs, few decisions are required about the placement of the resources or the management of server hardware resources, but the scale of the service is limited by the server hardware capacity and network connectivity. Peer-to-peer systems provide access to information resources located on computers throughout a network (whether it be the Internet or a corporate network). Algorithms for the placement and subsequent retrieval of information objects are a key aspect of the system design. The aim is to deliver a service that is fully decentralized and self-organizing, dynamically balancing the storage and processing loads between all the participating computers as computers join and leave the service.

Peer-to-peer systems share these characteristics:

- Their design ensures that each user contributes resources to the system.
- Although they may differ in the resources that they contribute, all the nodes in a peer-to-peer system have the same functional capabilities and responsibilities.
- Their correct operation does not depend on the existence of any centrally administered systems.

- They can be designed to offer a limited degree of anonymity to the providers and users of resources.
- A key issue for their efficient operation is the choice of an algorithm for the placement of data across many hosts and subsequent access to it in a manner that balances the workload and ensures availability without adding undue overheads.

Computers and network connections owned and managed by a multitude of different users and organizations are necessarily volatile resources; their owners do not guarantee to keep them switched on, connected and fault-free. So the availability of the processes and computers participating in peer-to-peer systems is unpredictable. Peer-to-peer services therefore cannot rely on guaranteed access to individual resources, although they can be designed to make the probability of failure to access a copy of a replicated object arbitrarily small. It is worth noting that this weakness of peer-to-peer systems can be turned into a strength if the replication of resources that it calls for is exploited to achieve a degree of resistance to tampering by malicious nodes (for example, through Byzantine fault-tolerance techniques; see Chapter 18).

Several early Internet-based services, including DNS (Section 13.2.3) and Netnews/Usenet [Kantor and Lapsley 1986], adopted a multi-server scalable and fault-tolerant architecture. The Xerox Grapevine name registration and mail delivery service [Birrell *et al.* 1982, Schroeder *et al.* 1984] provides an interesting early example of a scalable, fault-tolerant distributed service. Lamport's part-time parliament algorithm for distributed consensus [Lamport 1989], the Bayou replicated storage system (see Section 18.4.2) and the classless interdomain IP routing algorithm (see Section 3.4.3) are all examples of distributed algorithms for the placement or location of information and can be considered as antecedents of peer-to-peer systems.

But the potential for the deployment of peer-to-peer services using resources at the edges of the Internet emerged only when a significant number of users had acquired always-on, broadband connections to the network, making their desktop computers suitable platforms for resource sharing. This occurred first in the United States around 1999. By mid-2004 the worldwide number of broadband Internet connections had comfortably exceeded 100 million [Internet World Stats 2004].

Three generations of peer-to-peer system and application development can be identified. The first generation was launched by the Napster music exchange service [OpenNap 2001], which we describe in the next section. A second generation of file-sharing applications offering greater scalability, anonymity and fault tolerance quickly followed including Freenet [Clarke *et al.* 2000, freenetproject.org], Gnutella, Kazaa [Leibowitz *et al.* 2003] and BitTorrent [Cohen 2003].

Peer-to-peer middleware • The third generation is characterized by the emergence of middleware layers for the application-independent management of distributed resources on a global scale. Several research teams have now completed the development, evaluation and refinement of peer-to-peer middleware platforms and demonstrated or deployed them in a range of application services. The best-known and most fully developed examples include Pastry [Rowstron and Druschel 2001], Tapestry [Zhao *et al.* 2004], CAN [Ratnasamy *et al.* 2001], Chord [Stoica *et al.* 2001] and Kademlia [Maymounkov and Mazieres 2002].

Figure 10.1 Distinctions between IP and overlay routing for peer-to-peer applications

	<i>IP</i>	<i>Application-level routing overlay</i>
<i>Scale</i>	IPv4 is limited to 2^{32} addressable nodes. The IPv6 namespace is much more generous (2^{128}), but addresses in both versions are hierarchically structured and much of the space is preallocated according to administrative requirements.	Peer-to-peer systems can address more objects. The GUID namespace is very large and flat ($>2^{128}$), allowing it to be much more fully occupied.
<i>Load balancing</i>	Loads on routers are determined by network topology and associated traffic patterns.	Object locations can be randomized and hence traffic patterns are divorced from the network topology.
<i>Network dynamics (addition/deletion of objects/nodes)</i>	IP routing tables are updated asynchronously on a best-effort basis with time constants on the order of 1 hour.	Routing tables can be updated synchronously or asynchronously with fractions-of-a-second delays.
<i>Fault tolerance</i>	Redundancy is designed into the IP network by its managers, ensuring tolerance of a single router or network connectivity failure. n -fold replication is costly.	Routes and object references can be replicated n -fold, ensuring tolerance of n failures of nodes or connections.
<i>Target identification</i>	Each IP address maps to exactly one target node.	Messages can be routed to the nearest replica of a target object.
<i>Security and anonymity</i>	Addressing is only secure when all nodes are trusted. Anonymity for the owners of addresses is not achievable.	Security can be achieved even in environments with limited trust. A limited degree of anonymity can be provided.

These platforms are designed to place resources (data objects, files) on a set of computers that are widely distributed throughout the Internet and to route messages to them on behalf of clients, relieving clients of any need to make decisions about placing resources and to hold information about the whereabouts of the resources they require. Unlike the second-generation systems, they provide guarantees of delivery for requests in a bounded number of network hops. They place replicas of resources on available host computers in a structured manner, taking account of their volatile availability, their variable trustworthiness and requirements for load balancing and locality of information storage and use.

Resources are identified by globally unique identifiers (GUIDs), usually derived as a secure hash (described in Section 11.4.3) from some or all of the resource's state. The use of a secure hash makes a resource 'self certifying' – clients receiving a resource can check the validity of the hash. This protects it against tampering by untrusted nodes on which it may be stored, but this technique requires that the states of resources are immutable, since a change to the state would result in a different hash value. Hence peer-to-peer storage systems are inherently best suited to the storage of immutable objects

(such as music or video files). Their use for objects with changing values is more challenging, but this can be accommodated by the addition of trusted servers to manage a sequence of versions and identify the current version (as is done for example in OceanStore and Ivy, described in Sections 10.6.2 and 10.6.3).

The use of peer-to-peer systems for applications that demand a high level of availability for the objects stored requires careful application design to avoid situations in which all of the replicas of an object are simultaneously unavailable. There is a risk of this for objects stored on computers with the same ownership, geographic location, administration, network connectivity, country or jurisdiction. The use of randomly distributed GUIDs assists by distributing the object replicas to randomly located nodes in the underlying network. If the underlying network spans many organizations across the globe, then the risk of simultaneous unavailability is much reduced.

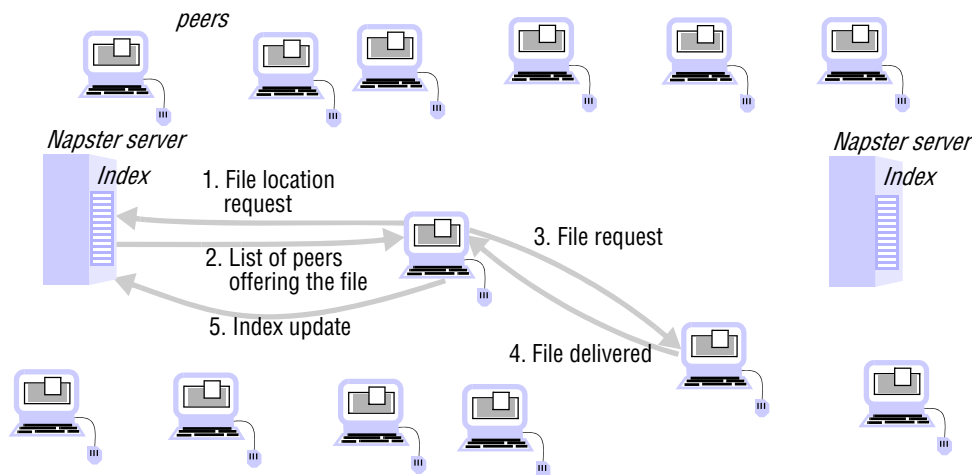
Overlay routing versus IP routing • At first sight, routing overlays share many characteristics with the IP packet routing infrastructure that constitutes the primary communication mechanism of the Internet (see Section 3.4.3). It is therefore legitimate to ask why an additional application-level routing mechanism is required in peer-to-peer systems. The answer lies in several distinctions that are identified in Figure 10.1. It may be argued that some of these distinctions arise from the ‘legacy’ nature of IP as the Internet’s primary protocol, but the legacy’s impact is too strong for it to be overcome in order to support peer-to-peer applications more directly.

Distributed computation • The exploitation of spare computing power on end-user computers has long been a subject of interest and experiment. Work with the first personal computers at Xerox PARC [Shoch and Hupp 1982] showed the feasibility of performing loosely coupled compute-intensive tasks by running background processes on ~100 personal computers linked by a local network. More recently, much larger numbers of computers have been put to use to perform several scientific calculations that require almost unlimited quantities of computing power.

The most widely known effort of this type is the *SETI@home* project [Anderson *et al.* 2002], which is part of a wider project called the Search for Extra-Terrestrial Intelligence. SETI@home partitions a stream of digitized radio telescope data into 107-second work units, each of about 350 kbytes and distributes them to client computers whose computing power is contributed by volunteers. Each work unit is distributed redundantly to 3–4 personal computers to guard against erroneous or malicious nodes and is examined for significant signal patterns. The distribution of work units and the coordination of results is handled by a single server that is responsible for communication with all of the clients. Anderson *et al.* [2002] reported that 3.91 million personal computers had participated in the SETI@home project by August 2002, resulting in the processing of 221 million work units and representing an average 27.36 teraflops of computational power during the 12 months to July 2002. The work completed to that date represented the largest single computation on record.

The SETI@home computation is unusual in that it does not involve any communication or coordination between computers while they are processing the work units; the results are communicated to a central server in a single short message that may be delivered whenever the client and server are available. Some other scientific tasks of this nature have been identified, including the search for large prime numbers and attempts at brute-force decryption, but the unleashing of the computational power in the

Figure 10.2 Napster: peer-to-peer file sharing with a centralized, replicated index



Internet for a broader range of tasks will depend upon the development of a distributed platform that supports data sharing and the coordination of computation between participating computers on a large scale. That is the goal of the Grid project, discussed in Chapter 19.

In this chapter we focus on algorithms and systems developed to date for the sharing of data in peer-to-peer networks. In Section 10.2 we summarize Napster's design and review the lessons learned from it. In Section 10.3 we describe the general requirements for peer-to-peer middleware layers. The following sections cover the design and application of peer-to-peer middleware platforms, starting with an abstract specification in Section 10.4, followed by detailed descriptions of two fully developed examples in Section 10.5 and some applications of them in Section 10.6.

10.2 Napster and its legacy

The first application in which a demand for a globally scalable information storage and retrieval service emerged was the downloading of digital music files. Both the need for and the feasibility of a peer-to-peer solution were first demonstrated by the Napster file-sharing system [OpenNap 2001] which provided a means for users to share files. Napster became very popular for music exchange soon after its launch in 1999. At its peak, several million users were registered and thousands were swapping music files simultaneously.

Napster's architecture included centralized indexes, but users supplied the files, which were stored and accessed on their personal computers. Napster's method of operation is illustrated by the sequence of steps shown in Figure 10.2. Note that in step 5 clients are expected to add their own music files to the pool of shared resources by

transmitting a link to the Napster indexing service for each available file. Thus the motivation for Napster and the key to its success was the making available of a large, widely distributed set of files to users throughout the Internet, fulfilling Shirky's dictum by providing access to 'shared resources at the edges of the Internet'.

Napster was shut down as a result of legal proceedings instituted against the operators of the Napster service by the owners of the copyright in some of the material (i.e., digitally encoded music) that was made available on it (see the box below).

Anonymity for the receivers and the providers of shared data and other resources is a concern for the designers of peer-to-peer systems. In systems with many nodes, the routing of requests and results can be made sufficiently tortuous to conceal their source and the contents of files can be distributed across multiple nodes, spreading the responsibility for making them available. Mechanisms for anonymous communication that are resistant to most forms of traffic analysis are available [Goldschlag *et al.* 1999]. If files are also encrypted before they are placed on servers, the owners of the servers can plausibly deny any knowledge of the contents. But these anonymity techniques add to the cost of resource sharing, and recent work has shown that the anonymity available is weak against some attacks [Wright *et al.* 2002].

The Freenet [Clarke *et al.* 2000] and FreeHaven [Dingledine *et al.* 2000] projects are focused on providing Internet-wide file services that offer anonymity for the providers and users of the shared files. Ross Anderson has proposed the Eternity Service [Anderson 1996], a storage service that provides long-term guarantees of data

Peer-to-peer systems and copyright ownership issues

The developers of Napster argued that they were not liable for the infringement of the copyright owners' rights because they were not participating in the copying process, which was performed entirely between users' machines. Their argument failed because the index servers were deemed an essential part of the process. Since the index servers were located at well-known addresses, their operators were unable to remain anonymous and so could be targeted in lawsuits.

A more fully distributed file-sharing service might have achieved a better separation of legal responsibilities, spreading the responsibility across all of the users and thus making the pursuit of legal remedies very difficult, if not impossible. Whatever view one takes about the legitimacy of file copying for the purpose of sharing copyright-protected material, there are legitimate social and political justifications for the anonymity of clients and servers in some application contexts. The most persuasive justification arises when anonymity is used to overcome censorship and maintain freedom of expression for individuals in oppressive societies or organizations.

It is known that email and web sites have played a significant role in achieving public awareness at times of political crisis in such societies; their role could be strengthened if the authors could be protected by anonymity. 'Whistle-blowing' is a related case: a 'whistle-blower' is an employee who publicizes or reports their employer's wrongdoings to authorities without revealing their own identity for fear of sanctions or dismissal. In some circumstances it is reasonable for such an action to be protected by anonymity.

availability through resistance to all sorts of accidental data loss and denial of service attacks. He bases the need for such a service on the observation that whereas publication is a permanent state for printed information – it is virtually impossible to delete material once it has been published and distributed to a few thousand libraries in diverse organizations and jurisdictions around the world – electronic publications cannot easily achieve the same level of resistance to censorship or suppression. Anderson covers the technical and economic requirements to ensure the integrity of the store and also points out that anonymity is often an essential requirement for the persistence of information, since it provides the best defence against legal challenges, as well as illegal actions such as bribes or attacks on the originators, owners or keepers of the data.

Lessons learned from Napster • Napster demonstrated the feasibility of building a useful large-scale service that depends almost wholly on data and computers owned by ordinary Internet users. To avoid swamping the computing resources of individual users (for example, the first user to offer a chart-topping song) and their network connections, Napster took account of network locality – the number of hops between the client and the server – when allocating a server to a client requesting a song. This simple load-distribution mechanism enabled the service to scale to meet the needs of large numbers of users.

Limitations: Napster used a (replicated) unified index of all available music files. For the application in question, the requirement for consistency between the replicas was not strong, so this did not hamper performance, but for many applications it would constitute a limitation. Unless the access path to the data objects is distributed, object discovery and addressing are likely to become a bottleneck.

Application dependencies: Napster took advantage of the special characteristics of the application for which it was designed in other ways:

- Music files are never updated, avoiding any need to make sure all the replicas of files remain consistent after updates.
- No guarantees are required concerning the availability of individual files – if a music file is temporarily unavailable, it can be downloaded later. This reduces the requirement for dependability of individual computers and their connections to the Internet.

10.3 Peer-to-peer middleware

A key problem in the design of peer-to-peer applications is providing a mechanism to enable clients to access data resources quickly and dependably wherever they are located throughout the network. Napster maintained a unified index of available files for this purpose, giving the network addresses of their hosts. Second-generation peer-to-peer file storage systems such as Gnutella and Freenet employ partitioned and distributed indexes, but the algorithms used are specific to each system.

This location problem existed in several services that predate the peer-to-peer paradigm as well. For example, Sun NFS addresses this need with the aid of a virtual file system abstraction layer at each client that accepts requests to access files stored on

multiple servers in terms of virtual file references (i.e., v-nodes, see Section 12.3). This solution relies on a substantial amount of preconfiguration at each client and manual intervention when file distribution patterns or server provision changes. It is clearly not scalable beyond a service managed by a single organization. AFS (Section 12.4) has similar properties.

Peer-to-peer middleware systems are designed specifically to meet the need for the automatic placement and subsequent location of the distributed objects managed by peer-to-peer systems and applications.

Functional requirements • The function of peer-to-peer middleware is to simplify the construction of services that are implemented across many hosts in a widely distributed network. To achieve this it must enable clients to locate and communicate with any individual resource made available to a service, even though the resources are widely distributed amongst the hosts. Other important requirements include the ability to add new resources and to remove them at will and to add hosts to the service and remove them. Like other middleware, peer-to-peer middleware should offer a simple programming interface to application programmers that is independent of the types of distributed resource that the application manipulates.

Non-functional requirements • To perform effectively, peer-to-peer middleware must also address the following non-functional requirements [cf. Kubiawicz 2003]:

Global scalability: One of the aims of peer-to-peer applications is to exploit the hardware resources of very large numbers of hosts connected to the Internet. Peer-to-peer middleware must therefore be designed to support applications that access millions of objects on tens of thousands or hundreds of thousands of hosts.

Load balancing: The performance of any system designed to exploit a large number of computers depends upon the balanced distribution of workload across them. For the systems we are considering, this will be achieved by a random placement of resources together with the use of replicas of heavily used resources.

Optimization for local interactions between neighbouring peers: The ‘network distance’ between nodes that interact has a substantial impact on the latency of individual interactions, such as client requests for access to resources. Network traffic loadings are also impacted by it. The middleware should aim to place resources close to the nodes that access them the most.

Accommodating to highly dynamic host availability: Most peer-to-peer systems are constructed from host computers that are free to join or leave the system at any time. The hosts and network segments used in peer-to-peer systems are not owned or managed by any single authority; neither their reliability nor their continuous participation in the provision of a service is guaranteed. A major challenge for peer-to-peer systems is to provide a dependable service despite these facts. As hosts join the system, they must be integrated into the system and the load must be redistributed to exploit their resources. When they leave the system whether voluntarily or involuntarily, the system must detect their departure and redistribute their load and resources.

Studies of peer-to-peer applications and systems such as Gnutella and Overnet have shown a considerable turnover of participating hosts [Saroju *et al.* 2002, Bhagwan *et al.* 2003]. For the Overnet peer-to-peer file-sharing system, with 85,000

infeasible to maintain a database at all client nodes giving the locations of all the resources (objects) of interest.

Knowledge of the locations of objects must be partitioned and distributed throughout the network. Each node is made responsible for maintaining detailed knowledge of the locations of nodes and objects in a portion of the namespace as well as a general knowledge of the topology of the entire namespace (Figure 10.3). A high degree of replication of this knowledge is necessary to ensure dependability in the face of the volatile availability of hosts and intermittent network connectivity. In the systems we describe below, replication factors as high as 16 are typically used.

10.4 Routing overlays

The development of middleware that meets the functional and non-functional requirements outlined in the previous section is an active area of research, and several significant middleware systems have already emerged. In this chapter we describe several of them in detail.

In peer-to-peer systems a distributed algorithm known as a *routing overlay* takes responsibility for locating nodes and objects. The name denotes the fact that the middleware takes the form of a layer that is responsible for routing requests from any client to a host that holds the object to which the request is addressed. The objects of interest may be placed at and subsequently relocated to any node in the network without client involvement. It is termed an overlay since it implements a routing mechanism in the application layer that is quite separate from any other routing mechanisms deployed at the network level such as IP routing. This approach to the management and location of replicated objects was first analyzed and shown to be effective for networks involving sufficiently many nodes in a groundbreaking paper by Plaxton *et al.* [1997].

The routing overlay ensures that any node can access any object by routing each request through a sequence of nodes, exploiting knowledge at each of them to locate the destination object. Peer-to-peer systems usually store multiple replicas of objects to ensure availability. In that case, the routing overlay maintains knowledge of the location of all the available replicas and delivers requests to the nearest ‘live’ node (i.e. one that has not failed) that has a copy of the relevant object.

The GUIDs used to identify nodes and objects are an example of the ‘pure’ names referred to in Section 13.1.1. These are also known as opaque identifiers, since they reveal nothing about the locations of the objects to which they refer.

The main task of a routing overlay is the following:

Routing of requests to objects: A client wishing to invoke an operation on an object submits a request including the object’s GUID to the routing overlay, which routes the request to a node at which a replica of the object resides.

But the routing overlay must also perform some other tasks:

Insertion of objects: A node wishing to make a new object available to a peer-to-peer service computes a GUID for the object and announces it to the routing overlay, which then ensures that the object is reachable by all other clients.

Figure 10.4 Basic programming interface for a distributed hash table (DHT) as implemented by the PAST API over Pastry

put(GUID, data)

Stores *data* in replicas at all nodes responsible for the object identified by *GUID*.

remove(GUID)

Deletes all references to *GUID* and the associated data.

value = *get*(GUID)

Retrieves the data associated with *GUID* from one of the nodes responsible for it.

Deletion of objects: When clients request the removal of objects from the service the routing overlay must make them unavailable.

Node addition and removal: Nodes (i.e., computers) may join and leave the service. When a node joins the service, the routing overlay arranges for it to assume some of the responsibilities of other nodes. When a node leaves (either voluntarily or as a result of a system or network fault), its responsibilities are distributed amongst the other nodes.

An object's GUID is computed from all or part of the state of the object using a function that delivers a value that is, with very high probability, unique. Uniqueness is verified by searching for another object with the same GUID. A hash function (such as SHA-1, see Section 11.4) is used to generate the GUID from the object's value. Because these randomly distributed identifiers are used to determine the placement of objects and to retrieve them, overlay routing systems are sometimes described as *distributed hash tables* (DHT). This is reflected by the simplest form of API used to access them, as shown in Figure 10.4. With this API, the *put()* operation is used to submit a data item to be stored together with its GUID. The DHT layer takes responsibility for choosing a location for it, storing it (with replicas to ensure availability) and providing access to it via the *get()* operation.

A slightly more flexible form of API is provided by a *distributed object location and routing* (DOLR) layer, as shown in Figure 10.5. With this interface objects can be stored anywhere and the DOLR layer is responsible for maintaining a mapping between object identifiers (GUIDs) and the addresses of the nodes at which replicas of the objects are located. Objects may be replicated and stored with the same GUID at different hosts, and the routing overlay takes responsibility for routing requests to the nearest available replica.

With the DHT model, a data item with GUID *X* is stored at the node whose GUID is numerically closest to *X* and at the *r* hosts whose GUIDs are next-closest to it numerically, where *r* is a replication factor chosen to ensure a very high probability of availability. With the DOLR model, locations for the replicas of data objects are decided outside the routing layer and the DOLR layer is notified of the host address of each replica using the *publish()* operation.

Figure 10.5 Basic programming interface for distributed object location and routing (DOLR) as implemented by Tapestry

publish(GUID)

GUID can be computed from the object (or some part of it, e.g., its name). This function makes the node performing a *publish* operation the host for the object corresponding to GUID.

unpublish(GUID)

Makes the object corresponding to GUID inaccessible.

sendToObj(msg, GUID, [n])

Following the object-oriented paradigm, an invocation message is sent to an object in order to access it. This might be a request to open a TCP connection for data transfer or to return a message containing all or part of the object's state. The final optional parameter [n], if present, requests the delivery of the same message to *n* replicas of the object.

The interfaces in Figures 10.4 and 10.5 are based on a set of abstract representations proposed by Dabek *et al.* [2003] to show that most peer-to-peer routing overlay implementations developed to date provide very similar functionality.

Research on the design of routing overlay systems began in 2000 and had reached fruition by 2005, with the development and evaluation of several successful prototypes. The evaluations demonstrated that their performance and dependability were adequate for use in many production environments. In the next section we describe two of these in detail: Pastry, which implements a distributed hash table API similar to the one presented in Figure 10.4, and Tapestry, which implements an API similar to that shown in Figure 10.5. Both Pastry and Tapestry employ a routing mechanism known as *prefix routing* to determine routes for the delivery of messages based on the values of the GUIDs to which they are addressed. Prefix routing narrows the search for the next node along the route by applying a binary mask that selects an increasing number of hexadecimal digits from the destination GUID after each hop. (This technique is also employed in classless interdomain routing for IP packets, as outlined in Section 3.4.3.)

Other routing schemes have been developed that exploit different measures of distance to narrow the search for the next hop destination. Chord [Stoica *et al.* 2001] bases the choice on the numerical difference between the GUIDs of the selected node and the destination node. CAN [Ratnasamy *et al.* 2001] uses distance in a *d*-dimensional hyperspace into which nodes are placed. Kademlia [Maymounkov and Mazieres 2002] uses the XOR of pairs of GUIDs as a metric for distance between nodes. Because XOR is symmetric, Kademlia can maintain participants' routing tables very simply; they always receive requests from the same nodes contained in their routing tables.

GUIDs are not human-readable, so client applications must obtain the GUIDs for resources of interest through some form of indexing service using human-readable names or search requests. Ideally, these indexes are also stored in a peer-to-peer manner to overcome the weaknesses of centralized indexes evidenced by Napster. But in simple cases, such as music files or publications available for peer-to-peer download, they can simply be indexed on web pages (*cf.* BitTorrent [Cohen 2003]). In BitTorrent a web

index search leads to a stub file containing details of the desired resource, including its GUID and the URL of a *tracker* – a host that holds an up-to-date list of network addresses for providers willing to supply the file (see Chapter 20 for more details of the BitTorrent protocol).

The foregoing description of routing overlays will probably have raised questions in the reader's mind about their performance and reliability. Answers to these questions will emerge from the descriptions of practical routing overlay systems in the next section.

10.5 Overlay case studies: Pastry, Tapestry

The prefix routing approach is adopted by both Pastry and Tapestry. Pastry is the message routing infrastructure deployed in several applications including PAST [Druschel and Rowstron 2001], an archival (immutable) file storage system implemented as a distributed hash table with the API in Figure 10.4, and Squirrel, a peer-to-peer web caching service described in Section 10.6.1. Pastry has a straightforward but effective design that makes it a good first example for us to study in detail.

Tapestry is the basis for the OceanStore storage system, which we describe in Section 10.6.2. It has a more complex architecture than Pastry because it aims to support a wider range of locality approaches. We describe Tapestry in Section 10.5.2.

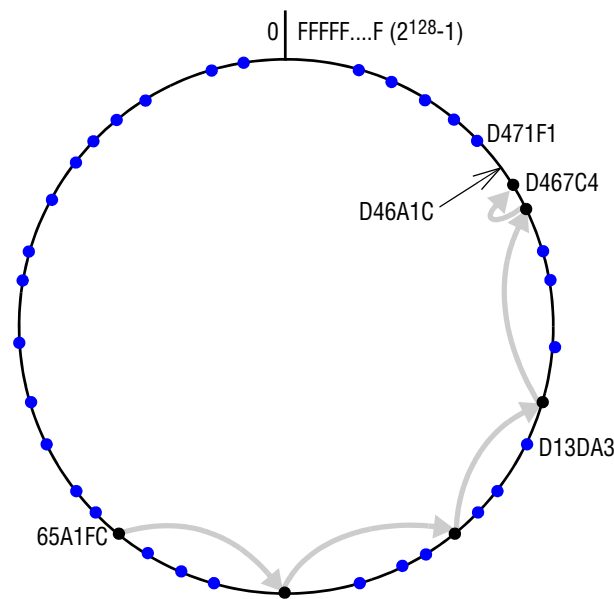
We also look at alternative unstructured approaches in Section 10.5.3, looking in detail at the overlay style adopted by Gnutella.

10.5.1 Pastry

Pastry [Rowstron and Druschel 2001, Castro *et al.* 2002a, freepastry.org] is a routing overlay with the characteristics that we outlined in Section 10.4. All the nodes and objects that can be accessed through Pastry are assigned 128-bit GUIDs. For nodes, these are computed by applying a secure hash function (such as SHA-1; see Section 11.4.3) to the public key with which each node is provided. For objects such as files, the GUID is computed by applying a secure hash function to the object's name or to some part of the object's stored state. The resulting GUIDs have the usual properties of secure hash values – that is, they are randomly distributed in the range 0 to $2^{128}-1$. They provide no clues as to the value from which they were computed, and clashes between GUIDs for different nodes or objects are extremely unlikely. (If a clash occurs, Pastry detects it and takes remedial action.)

In a network with N participating nodes, the Pastry routing algorithm will correctly route a message addressed to any GUID in $O(\log N)$ steps. If the GUID identifies a node that is currently active, the message is delivered to that node; otherwise, the message is delivered to the active node whose GUID is numerically closest to it. Active nodes take responsibility for processing requests addressed to all objects in their numerical neighbourhood.

Routing steps involve the use of an underlying transport protocol (normally UDP) to transfer the message to a Pastry node that is 'closer' to its destination. But note that the closeness referred to here is in an entirely artificial space – the space of GUIDs. The

Figure 10.6 Circular routing alone is correct but inefficient *Based on Rowstron and Druschel [2001]*

The dots depict live nodes. The space is considered as circular: node 0 is adjacent to node $(2^{128}-1)$. The diagram illustrates the routing of a message from node 65A1FC to D46A1C using leaf set information alone, assuming leaf sets of size 8 ($l=4$). This is a degenerate type of routing that would scale very poorly; it is not used in practice.

real transport of a message across the Internet between two Pastry nodes may require a substantial number of IP hops. To minimize the risk of unnecessarily extended transport paths, Pastry uses a locality metric based on network distance in the underlying network (such as a hop counts or round-trip latency measurements) to select appropriate neighbours when setting up the routing tables used at each node.

Thousands of hosts located at widely dispersed sites can participate in a Pastry overlay. It is fully self-organizing: when new nodes join the overlay they obtain the data needed to construct a routing table and other required state from existing members in $O(\log N)$ messages, where N is the number of hosts participating in the overlay. In the event of a node failure or departure, the remaining nodes can detect its absence and cooperatively reconfigure to reflect the required changes in the routing structure in a similar number of messages.

Routing algorithm • The full routing algorithm involves the use of a routing table at each node to route messages efficiently, but for the purposes of explanation, we describe the routing algorithm in two stages. The first stage describes a simplified form of the algorithm that routes messages correctly but inefficiently without a routing table, and

Figure 10.7 First four rows of a Pastry routing table

$p =$	GUID prefixes and corresponding node handles n															
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	n	n	n	n	n	n		n	n	n	n	n	n	n	n	n
1	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
	n	n	n	n	n		n	n	n	n	n	n	n	n	n	n
2	650	651	652	653	654	655	656	657	658	659	65A	65B	65C	65D	65E	65F
	n	n	n	n	n	n	n	n	n	n		n	n	n	n	n
3	65A0	65A1	65A2	65A3	65A4	65A5	65A6	65A7	65A8	65A9	65AA	65AB	65AC	65AD	65AE	65AF
	n		n	n	n	n	n	n	n	n	n	n	n	n	n	n

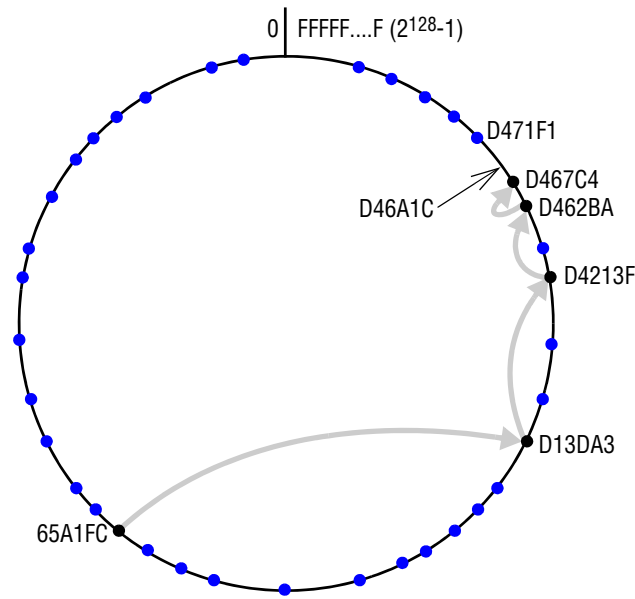
The routing table is located at a node whose GUID begins 65A1. Digits are in hexadecimal. The n s represent [GUID, IP address] pairs that act as node handles specifying the next hop to be taken by messages addressed to GUIDs that match each given prefix. Grey-shaded entries in the table body indicate that the prefix matches the current GUID up to the given value of p : the next row down or the leaf set should be examined to find a route. Although there are a maximum of 128 rows in the table, only $\log_{16} N$ rows will be populated on average in a network with N active nodes.

the second stage describes the full routing algorithm, which routes a request to any node in $O(\log N)$ messages:

Stage 1: Each active node stores a *leaf set* – a vector L (of size $2l$) containing the GUIDs and IP addresses of the nodes whose GUIDs are numerically closest on either side of its own (l above and l below). Leaf sets are maintained by Pastry as nodes join and leave. Even after a node failure, they will be corrected within a short time. (Fault recovery is discussed below.) It is therefore an invariant of the Pastry system that the leaf sets reflect a recent state of the system and that they converge on the current state in the face of failures up to some maximum rate of failure.

The GUID space is treated as circular: GUID 0's lower neighbour is $2^{128}-1$. Figure 10.6 gives a view of active nodes distributed in this circular address space. Since every leaf set includes the GUIDs and IP addresses of the current node's immediate neighbours, a Pastry system with correct leaf sets of size at least 2 can route messages to any GUID trivially as follows: any node A that receives a message M with destination address D routes the message by comparing D with its own GUID A and with each of the GUIDs in its leaf set and forwarding M to the node amongst them that is numerically closest to D .

Figure 10.8 Pastry routing example

Based on Rowstron and Druschel [2001]

Routing a message from node 65A1FC to D46A1C. With the aid of a well-populated routing table the message can be delivered in $\sim \log_{16}(N)$ hops.

Figure 10.6 illustrates this for a Pastry system with $l = 4$. (In typical real installations of Pastry, $l = 8$.) Based on the definition of leaf sets we can conclude that at each step M is forwarded to a node that is closer to D than the current node and that this process will eventually deliver M to the active node closest to D . But such a routing scheme is clearly very inefficient, requiring $\sim N/2l$ hops to deliver a message in a network with N nodes.

Stage II: The second part of our explanation describes the full Pastry algorithm and shows how efficient routing is achieved with the aid of routing tables.

Each Pastry node maintains a tree-structured routing table giving GUIDs and IP addresses for a set of nodes spread throughout the entire range of 2^{128} possible GUID values, with increased density of coverage for GUIDs numerically close to its own.

Figure 10.7 shows the structure of the routing table for a specific node, and Figure 10.8 illustrates the actions of the routing algorithm. The routing table is structured as follows: GUIDs are viewed as hexadecimal values and the table classifies GUIDs based on their hexadecimal prefixes. The table has as many rows as there are hexadecimal digits in a GUID, so for the prototype Pastry system that we are describing, there are $128/4 = 32$ rows. Any row n contains 15 entries – one for each possible value of the n^{th} hexadecimal digit, excluding the value in the local

Figure 10.9 Pastry's routing algorithm

To handle a message M addressed to a node D (where $R[p,i]$ is the element at column i , row p of the routing table):

1. *If* ($L_i < D < L_i$) { // the destination is within the leaf set or is the current node.
2. Forward M to the element L_i of the leaf set with GUID closest to D or the current node A .
3. } *else* { // use the routing table to despatch M to a node with a closer GUID
4. Find p , the length of the longest common prefix of D and A ,. and i , the $(p+1)^{\text{th}}$ hexadecimal digit of D .
5. *If* ($R[p,i] \neq \text{null}$) forward M to $R[p,i]$ // route M to a node with a longer common prefix.
6. *else* { // there is no entry in the routing table.
7. Forward M to any node in L or R with a common prefix of length p but a GUID that is numerically closer.
- }
- }

node's GUID. Each entry in the table points to one of the potentially many nodes whose GUIDs have the relevant prefix.

The routing process at any node A uses the information in its routing table R and leaf set L to handle each request from an application and each incoming message from another node according to the algorithm shown in Figure 10.9.

We can be sure that the algorithm will succeed in delivering M to its destination because lines 1, 2 and 7 perform the actions described in Stage I of our description above, and we have shown this to be a complete, although inefficient, routing algorithm. The remaining steps are designed to use the routing table to improve the algorithm's performance by reducing the number of hops required.

Lines 4–5 come into play whenever D does not fall within the numeric range of the current node's leaf set and relevant routing table entries are available. The selection of a destination for the next hop involves comparing the hexadecimal digits of D with those of A (the GUID of the current node) from left to right to discover the length, p , of their longest common prefix. This length is then used as a row offset, together with the first non-matching digit of D as a column offset, to access the required element of the routing table. The construction of the table ensures that this element (if not empty) contains the IP address of a node whose GUID has $p+1$ prefix digits in common with D .

Line 7 is used when D falls outside the numeric range of the leaf set and there isn't a relevant routing table entry. This case is rare; it arises only when nodes have recently failed and the table hasn't yet been updated. The routing algorithm is able to proceed by scanning both the leaf set and the routing table and forwarding M to another node whose GUID has p matching prefix digits but is numerically closer to D . If that node is in L , then we are following the Stage I procedure illustrated in Figure 10.6. If it is in R , then it must be closer to D than any node in L , hence we are improving on Stage I.

Host integration • New nodes use a joining protocol in order to acquire their routing table and leaf set contents and notify other nodes of changes they must make to their

tables. First, the new node computes a suitable GUID (typically by applying the SHA-1 hash function to the node's public key), then it makes contact with a nearby Pastry node. (Here we use the term *nearby* to refer to network distance, i.e., a small number of network hops or low transmission delay; see the box below.)

Suppose that the new node's GUID is X and the nearby node it contacts has GUID A . Node X sends a special *join* request message to A , giving X as its destination. A despatches the *join* message via Pastry in the normal way. Pastry will route the *join* message to the existing node whose GUID is numerically closest to X ; let us call this destination node Z .

A , Z and all the nodes (B , C , ...) through which the *join* message is routed on its way to Z add additional steps to the normal Pastry routing algorithm, which result in the transmission of the contents of the relevant parts of their routing tables and leaf sets to X . X examines them and constructs its own routing table and leaf set from them, requesting some additional information from other nodes if necessary.

To see how X builds its routing table, note that the first row of the table depends on the value of X 's GUID, and to minimize routing distances, the table should be constructed to route messages via neighbouring nodes whenever possible. A is a neighbour of X , so the first row of A 's table is a good initial choice for the first row of X 's table, X_0 . On the other hand, A 's table is probably not relevant for the second row, X_1 , because X 's and A 's GUIDs may not share the same first hexadecimal digit. The routing algorithm ensures that X 's and B 's GUIDs do share the same first digit, though, which implies that the second row of B 's routing table, B_1 , is a suitable initial value for X_1 . Similarly, C_2 is suitable for X_2 , and so on.

Furthermore, recalling the properties of leaf sets, note that since Z 's GUID is numerically closest to X 's, X 's leaf set should be similar to Z 's. In fact, X 's ideal leaf set will differ from Z 's by just one member. Z 's leaf set is therefore taken as an adequate initial approximation, which will eventually be optimized through interaction with its neighbours as described in the fault tolerance subsection below.

Finally, once X has constructed its leaf set and routing table in the manner outlined above, it sends their contents to all the nodes identified in the leaf set and the routing table and they adjust their own tables to incorporate the new node. The entire task of incorporating a new node into the Pastry infrastructure requires the transmission of $O(\log N)$ messages.

Host failure or departure • Nodes in the Pastry infrastructure may fail or depart without warning. A Pastry node is considered failed when its immediate neighbours (in GUID space) can no longer communicate with it. When this occurs, it is necessary to repair the leaf sets that contain the failed node's GUID.

Nearest neighbour algorithm

The new node should have the address of at least one existing Pastry node, but it might not be nearby. To ensure that nearby nodes are known Pastry includes a 'nearest neighbour' algorithm to find a nearby node by recursively measuring the round-trip delay for a probe message sent periodically to each member of the leaf set of the nearest currently known Pastry node.

To repair its leaf set L , the node that discovers the failure looks for a live node close to the failed node in L and requests a copy of that node's leaf set, L' . L' will contain a sequence of GUIDs that partly overlap those in L , including one with an appropriate value to replace the failed node. Other neighbouring nodes are then informed of the failure and they perform a similar procedure. This repair procedure guarantees that leaf sets will be repaired unless l adjacently numbered nodes fail simultaneously.

Repairs to routing tables are made on a 'when discovered' basis. The routing of messages can proceed with some routing table entries that are no longer live – failed routing attempts result in the use of a different entry from the same row of a routing table.

Locality • The Pastry routing structure is highly redundant: there are many routes between each pair of nodes. The construction of the routing tables aims to take advantage of this redundancy to reduce actual message transmission times by exploiting the locality properties of nodes in the underlying transport network (which is normally a subset of nodes in the Internet).

Recall that each row in a routing table contains 16 entries. The entries in the i^{th} row give the addresses of 16 nodes with GUIDs with $i-1$ initial hexadecimal digits that match the current node's GUID and an i^{th} digit that takes each of the possible hexadecimal values. A well-populated Pastry overlay will contain many more nodes than can be contained in an individual routing table; whenever a new routing table is being constructed a choice is made for each position between several candidates (taken from routing information supplied by other nodes) based on a proximity neighbour selection algorithm [Gummadi *et al.* 2003]. A locality metric (number of IP hops or measured latency) is used to compare candidates and the closest available node is chosen. Since the information available is not comprehensive, this mechanism cannot produce globally optimal routings, but simulations have shown that it results in routes that are on average only about 30–50% longer than the optimum.

Fault tolerance • As described above, the Pastry routing algorithm assumes that all entries in routing tables and leaf sets refer to live, correctly functioning nodes. All nodes send 'heartbeat' messages (i.e., messages sent at fixed time intervals to indicate that the sender is alive) to neighbouring nodes in their leaf sets, but information about failed nodes detected in this manner may not be disseminated sufficiently rapidly to eliminate routing errors. Nor does it account for malicious nodes that may attempt to interfere with correct routing. To overcome these problems, clients that depend upon reliable message delivery are expected to employ an *at-least-once* delivery mechanism (see Section 5.3.1) and repeat their requests several times in the absence of a response. This will allow Pastry a longer time window to detect and repair node failures.

To deal with any remaining failures or malicious nodes, a small degree of randomness is introduced into the route selection algorithm described in Figure 10.9. Essentially, the step in line 5 of Figure 10.9 is modified in a randomly selected small proportion of cases to yield a common prefix that is less than the maximum length. This results in the use of a routing taken from an earlier row of the routing table, producing less optimal but different routing than the standard version of the algorithm. With this random variation in the routing algorithm, client retransmissions should eventually succeed even in the presence of a small number of malicious nodes.

Dependability • The authors of Pastry have developed an updated version called MSPastry [Castro *et al.* 2003] that uses the same routing algorithm and similar host management methods, but also includes some additional dependability measures and some performance optimizations in the host management algorithms.

Dependability measures include the use of acknowledgements at each hop in the routing algorithm. If the sending host does not receive an acknowledgement after a specified timeout, it selects an alternative route and retransmits the message. The node that failed to send an acknowledgement is then noted as a suspected failure.

As mentioned above, to detect failed nodes each Pastry node periodically sends a heartbeat message to its immediate neighbour to the left (i.e., with a lower GUID) in the leaf set. Each node also records the time of the last heartbeat message received from its immediate neighbour on the right (with a higher GUID). If the interval since the last heartbeat exceeds a timeout threshold, the detecting node starts a repair procedure that involves contacting the remaining nodes in the leaf set with a notification about the failed node and a request for suggested replacements. Even in the case of multiple simultaneous failures, this procedure terminates with all nodes on the left side of the failed node having leaf sets that contain the l live nodes with the closest GUIDs.

We have seen that the routing algorithm can function correctly using leaf sets alone; but the maintenance of the routing tables is important for performance. Suspected failed nodes in routing tables are probed in a similar manner to that used for the leaf set and if they fail to respond, their routing table entries are replaced with a suitable alternative obtained from a nearby node. In addition, a simple gossip protocol (see Section 18.4.1) is used to periodically exchange routing table information between nodes in order to repair failed entries and prevent slow deterioration of the locality properties. The gossip protocol is run about every 20 minutes.

Evaluation work • Castro and his colleagues have carried out an exhaustive performance evaluation of MSPastry, aimed at determining the impact on performance and dependability of the host join/leave rate and the associated dependability mechanisms [Castro *et al.* 2003].

The evaluation was performed by running the MSPastry system under control of a simulator running on a single machine that simulates a large network of hosts, with message passing replaced by simulated transmission delays. The simulation realistically modelled the join/leave behaviour of hosts and IP transmission delays based on parameters from real installations.

All of the dependability mechanisms of MSPastry were included, with realistic intervals for probe and heartbeat messages. The simulation work was validated by comparison with measurements taken with MSPastry running a real application load across an internal network with 52 nodes.

Here we summarize the key results.

Dependability: With an assumed IP message loss rate of 0%, MSPastry failed to deliver 1.5 in 100,000 requests (presumably due to the non-availability of destination hosts), and all requests that were delivered arrived at the correct node.

With an assumed IP message loss rate of 5%, MSPastry lost about 3.3 in 100,000 requests and 1.6 in 100,000 requests were delivered to the wrong node. The use of per-hop acknowledgements in MSPastry ensures that all lost or misdirected messages are eventually retransmitted and reach the correct node.

Performance: The metric used to evaluate the performance of MSPastry is called *relative delay penalty* (RDP) [Chu *et al.* 2000], or *stretch*. RDP is a direct measure of the extra cost incurred in employing an overlay routing layer. It is the ratio between the average delay in delivering a request by the routing overlay and in delivering a similar message between the same two nodes via UDP/IP. The RDP values observed for MSPastry under simulated loads ranged from ~ 1.8 with zero network message loss to ~ 2.2 with 5% network message loss.

Overheads: The extra network load generated by *control traffic* – messages involved in maintaining leaf sets and routing tables – was less than 2 messages per minute per node. The RDP and control traffic were both increased significantly for session lengths less than about 60 minutes due to initial setup overheads.

Overall these results show that overlay routing layers can be constructed that achieve good performance and high dependability with thousands of nodes operating in realistic environments. Even with mean session lengths shorter than 60 minutes and high network error rates the system degrades gracefully, continuing to provide an effective service.

Optimizing overlay lookup latency • Zhang *et al.* [2005a] have shown that the lookup performance of an important class of overlay networks (including Pastry, Chord and Tapestry) can be substantially enhanced by the inclusion of a simple learning algorithm that measures the latencies actually experienced in accessing the overlay nodes and thus incrementally modifies the overlay routing tables to optimize access latencies.

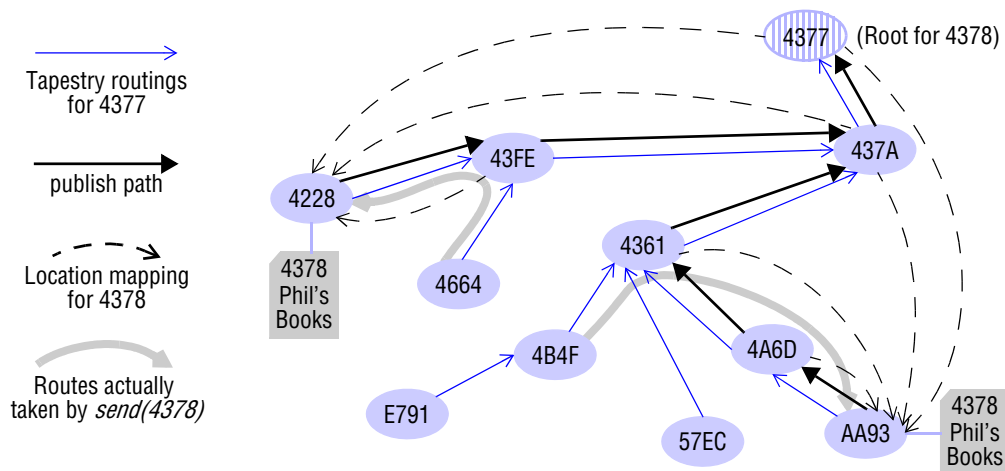
10.5.2 Tapestry

Tapestry implements a distributed hash table and routes messages to nodes based on GUIDs associated with resources using prefix routing in a manner similar to Pastry. But Tapestry's API conceals the distributed hash table from applications behind a DOLR interface like the one shown in Figure 10.5. Nodes that hold resources use the *publish(GUID)* primitive to make them known to Tapestry, and the holders of resources remain responsible for storing them. Replicated resources are published with the same GUID by each node that holds a replica, resulting in multiple entries in the Tapestry routing structure.

This gives Tapestry applications additional flexibility: they can place replicas close (in network distance) to frequent users of resources in order to reduce latency and minimize network load or to ensure tolerance of network and host failures. But this distinction between Pastry and Tapestry is not fundamental: Pastry applications can achieve similar flexibility by making the objects associated with GUIDs simply act as proxies for more complex application-level objects and Tapestry can be used to implement a distributed hash table in terms of its DOLR API [Dabek *et al.* 2003].

In Tapestry 160-bit identifiers are used to refer both to objects and to the nodes that perform routing actions. Identifiers are either *NodeIds*, which refer to computers that perform routing operations, or *GUIDs*, which refer to the objects. For any resource with GUID G there is a unique root node with GUID R_G that is numerically closest to G . Hosts H holding replicas of G periodically invoke *publish(G)* to ensure that newly arrived hosts become aware of the existence of G . On each invocation of *publish(G)* a

Figure 10.10 Tapestry routing

From Zhao *et al.* [2004]

Replicas of the file *Phil's Books* ($G=4378$) are hosted at nodes 4228 and AA93. Node 4377 is the root node for object 4378. The Tapestry routings shown are some of the entries in routing tables. The publish paths show routes followed by the publish messages laying down cached location mappings for object 4378. The location mappings are subsequently used to route messages sent to 4378.

publish message is routed from the invoker towards node R_G . On receipt of a publish message R_G enters (G, IP_H) , the mapping between G and the sending host's IP address in its routing table, and each node along the publication path caches the same mapping. This process is illustrated in Figure 10.10. When nodes hold multiple (G, IP) mappings for the same GUID, they are sorted by the network distance (round-trip time) to the IP address. For replicated objects this results in the selection of the nearest available replica of the object as the destination for subsequent messages sent to the object.

Zhao *et al.* [2004] give full details of the Tapestry routing algorithms and the management of Tapestry's routing tables in the face of node arrival and departure. Their paper includes comprehensive performance evaluation data based on simulation of large-scale Tapestry networks, showing that its performance is similar to Pastry's. In Section 10.6.2 we describe the OceanStore file store, which has been built and deployed over Tapestry.

10.5.3 From structured to unstructured peer-to-peer

The discussion so far has focused exclusively on what are known as *structured peer-to-peer* approaches. In structured approaches, there is an overall global policy governing the topology of the network, the placement of objects in the network and the routing or search functions used to locate objects in the network. In other words, there is a specific (distributed) data structure underpinning the associated overlay and a set of algorithms

Figure 10.11 Structured versus unstructured peer-to-peer systems

	<i>Structured peer-to-peer</i>	<i>Unstructured peer-to-peer</i>
<i>Advantages</i>	Guaranteed to locate objects (assuming they exist) and can offer time and complexity bounds on this operation; relatively low message overhead.	Self-organizing and naturally resilient to node failure.
<i>Disadvantages</i>	Need to maintain often complex overlay structures, which can be difficult and costly to achieve, especially in highly dynamic environments.	Probabilistic and hence cannot offer absolute guarantees on locating objects; prone to excessive messaging overhead which can affect scalability.

operating over that data structure. This can clearly be seen in the examples of Pastry and Tapestry based on the underlying distributed hash table and associated ring structures. Because of the structure imposed, such algorithms are efficient and offer time bounds on the location of objects, but at the cost of maintaining the underlying structures, often in highly dynamic environments.

Because of this maintenance argument, *unstructured peer-to-peer* approaches have also been developed. In unstructured approaches, there is no overall control over the topology or the placement of objects within the network. Rather, the overlay is created in an ad hoc manner, with each node that joins the network following some simple, local rules to establish connectivity. In particular, a joining node will establish contact with a set of *neighbours* knowing that the neighbours will also be connected to further neighbours and so on, forming a network that is fundamentally decentralized and self-organizing and hence resilient to node failure. To locate a given object, it is then necessary to carry out a search of the resultant network topology; clearly, this approach cannot offer any guarantees of being able to find the object and performance will be unpredictable. In addition, there is a real risk of generating excessive message traffic to locate objects.

A summary of the relative strengths of structured and unstructured peer-to-peer systems is provided in Figure 10.11. It is interesting to reflect that, despite the apparent drawbacks of unstructured peer-to-peer systems, this approach is dominant in the Internet, particularly in supporting peer-to-peer file sharing (with systems such as Gnutella, FreeNet and BitTorrent all adopting unstructured approaches). As will be seen, significant advancements have been made in such systems to improve the performance of unstructured approaches and this work is significant given the amount of traffic generated by peer-to-peer file sharing in the Internet (for example, a study carried out in the years 2008/9 indicates that peer-to-peer file-sharing applications account for between 43% and 70% of all Internet traffic, depending on the part of the world being considered [www.ipoque.com]).

Strategies for effective search • In peer-to-peer file sharing, all nodes in the network offer files to the greater environment. As mentioned above, the problem of locating a file then maps onto a search of the whole network to locate appropriate files. If implemented naively, this would be implemented by flooding the network with requests. This is precisely the strategy adopted in early versions of Gnutella. In particular, in Gnutella

0.4, every node forwarded a request to each of its neighbours, who then in turn passed this on to their neighbours, and so on until a match was found. Each search was also constrained with a time-to-live field limiting the number of hops. At the time Gnutella 0.4 was deployed, the average connectivity was around 5 neighbours per node. This approach is simple but does not scale and quickly floods the network with search-related traffic.

A number of refinements have been developed for search in unstructured networks [Lv *et al.* 2002, Tsoumakos and Roussopoulos 2006], including:

Expanded ring search: In this approach, the initiating node carries out a series of searches with increasing values in the time-to-live field, recognizing that a significant number of the requests will be met locally (especially if coupled with an effective replication strategy, as discussed below).

Random walks: With random walks, the initiating node sets off a number of walkers who follow their own random pathways through the interconnected graph offered by the unstructured overlay.

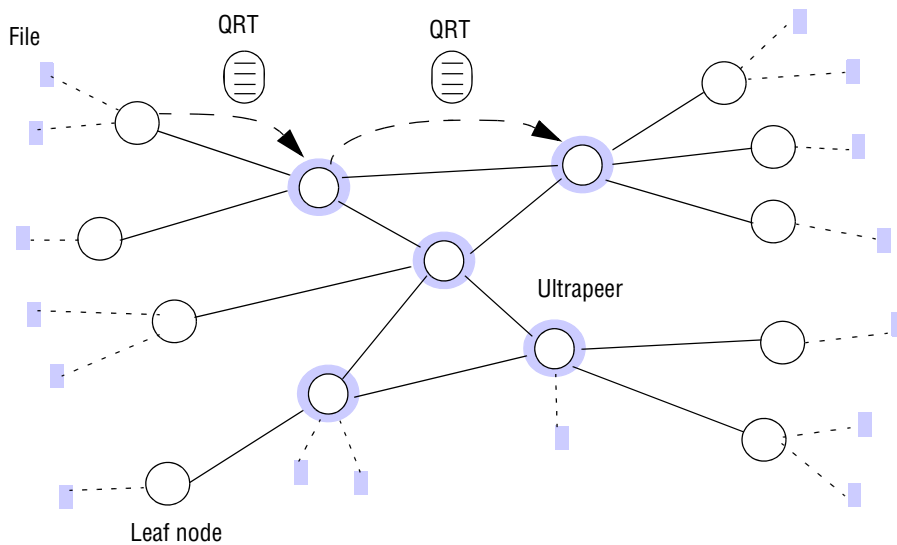
Gossiping: In gossiping approaches, a node sends a request to a given neighbour with a certain probability, and hence requests propagate through the network in a manner similar to a virus through a population (as such, gossip protocols are also referred to as *epidemic protocols*). The probability can either be fixed for a given network or calculated dynamically based on previous experience and/or the current context. (Note that gossiping is a common technique in distributed systems; further applications can be found in Chapters 6 and 18).

Such strategies can significantly reduce the overhead of search in unstructured networks and hence increase the scalability of the algorithms. Such strategies are also often supported by appropriate *replication* techniques. By replicating content across a number of peers, the probability of efficient discovery of particular files is significantly enhanced. Techniques include whole file replication and the scattering of fragments of files across the Internet – an approach that is used effectively in BitTorrent, for example, to reduce the burden on any one peer in downloading large files (see Chapter 20).

Case study: Gnutella • Gnutella was originally launched in 2000 and since then has grown to be one of the dominant and most influential peer-to-peer file-sharing applications. As mentioned above, initially the protocol adopted a rather simple flooding strategy that did not scale particularly well. In response to this, Gnutella 0.6 introduced a range of modifications that have significantly improved the performance of the protocol.

The first major amendment was to move from a pure peer-to-peer architecture where all nodes are equal to one where all peers still cooperate to offer the service but some nodes, designated to have additional resources, are elected as *ultrapeers*, and form the heart of the network, with other peers taking on the role of leaf nodes (or leaves). Leaves connect themselves to a small number of ultrapeers which are heavily connected to other ultrapeers (with over 32 connections each). This dramatically reduces the maximum number of hops required for exhaustive search. This style of peer-to-peer architecture is referred to as a hybrid architecture and is also the approach adopted in Skype (as discussed in Section 4.5.2).

Figure 10.12 Key elements in the Gnutella protocol



The other key enhancement was the introduction of a Query Routing Protocol (QRP) designed to reduce the number of queries issued by nodes. The protocol is based on exchanging information about files contained on nodes and hence only forwarding queries down paths where the system thinks there will be a positive outcome. Rather than sharing information about files directly, the protocol produces a set of numbers from hashing on the individual words in a file name. For example, a file name such as “Chapter ten on P2P” will be represented by four numbers, say $\langle 65, 47, 09, 76 \rangle$. A node produces a Query Routing Table (QRT) containing the hash values representing each of the files contained on that node which it then sends to all its associated ultrapeers. Ultrapeers then produce their own Query Routing Tables based on a union of all the entries from all the connected leaves together with entries for files contained in that node, and exchange these with other connected ultrapeers. In this way, ultrapeers can determine which paths offer a valid route for a given query, thus significantly reducing the amount of unnecessary traffic. More specifically, an ultrapeer will forward a query to a node if a match is found (indicating that node has the file) and will carry out the same check before passing it on to another ultrapeer if it is the last hop to the file. Note that, in order to avoid overloading of ultrapeers, nodes will send a query to one ultrapeer at a time and then wait for a specified period to see if they get a positive response.

Finally, a query in Gnutella contains the network address of the initiating ultrapeer, which implies that once a file is found it can be sent directly to the associated ultrapeer (using UDP), avoiding a reverse traversal through the graph.

The major elements associated with Gnutella 0.6 are summarized in Figure 10.12.

10.6 Application case studies: Squirrel, OceanStore, Ivy

The routing overlay layers described in the preceding section have been exploited in several application experiments and the resulting applications have been extensively evaluated. We have chosen three of them for further study, the Squirrel web caching service based on Pastry, and the OceanStore and Ivy file stores.

10.6.1 Squirrel web cache

The authors of Pastry have developed the Squirrel peer-to-peer web caching service for use in local networks of personal computers [Iyer *et al.* 2002]. In medium and large local networks web caching is typically performed using a dedicated server computer or cluster. The Squirrel system performs the same task by exploiting storage and computing resources already available on desktop computers in the local network. We first give a brief general description of the operation of a web caching service, then we outline the design of Squirrel and review its effectiveness.

Web caching • Web browsers generate HTTP *GET* requests for Internet objects like HTML pages, images, etc. These may be serviced from a browser cache on the client machine, from a *proxy web cache* (a service running on another computer in the same local network or on a nearby node in the Internet) or from the *origin web server* (the server whose domain name is included in the parameters of the *GET* request), depending on which contains a fresh copy of the object. The local and proxy caches each contain a set of recently retrieved objects organized for fast lookup by URL. Some objects are uncacheable because they are generated dynamically by the server in response to each request.

When a browser cache or proxy web cache receives a *GET* request, there are three possibilities: the requested object is uncacheable, there is a cache miss or the object is found in the cache. In the first two cases the request is forwarded to the next level towards the origin web server. When the requested object is found in a cache, the cached copy must be tested for freshness.

Web objects are stored in web servers and cache servers with some additional metadata values including a timestamp giving a *date of last modification* (T) and possibly a *time-to-live* (t) or an *eTag* (a hash computed from the contents of a web page). These metadata items are supplied by the origin server whenever an object is returned to a client.

Objects that have an associated time-to-live t , are considered fresh if $T+t$ is later than the current real time. For objects without a time-to-live, an estimated value for t is used (often only a few seconds). If the result of this freshness evaluation is positive, the cached object is returned to the client without contacting the origin web server. Otherwise, a conditional *GET* (*cGET*) request is issued to the next level for validation. There are two basic types of *cGET* requests: an *If-Modified-Since* request containing the timestamp of the last known modification, and an *If-None-Match* request containing an *eTag* representing the object contents. This *cGET* request can be serviced either by another web cache or by the origin server. A web cache that receives a *cGET* request and does not have a fresh copy of the object forwards the request towards the origin web

server. The response contains either the entire object, or a *not-modified* message if the cached object is unchanged.

Whenever a newly modified cacheable object is received from the origin server, it is added to the set of objects in the local cache (displacing older objects that are still valid if necessary) together with a timestamp, a time-to-live and an *eTag* if available.

The scheme described above is the basis of operation for the centralized proxy web caching services deployed in most local networks that support large numbers of web clients. Proxy web caches are typically implemented as a multi-threaded process running on a single dedicated host or a set of processes running on a cluster of computers and require a substantial quantity of dedicated computing resources in both cases.

Squirrel • The Squirrel web caching service performs the same functions using a small part of the resources of each client computer on a local network. The SHA-1 secure hash function is applied to the URL of each cached object to produce a 128-bit Pastry GUID. Since the GUID is not used to validate the contents, it need not be based on the entire object contents, as it is in other Pastry applications. The authors of Squirrel base their justification for this on the end-to-end argument (Section 2.3.3), arguing that the authenticity of a web page may be compromised at many points in its journey from the host to the client; authentication of cached pages adds little to any overall guarantee of authenticity and the HTTPS protocol (incorporating end-to-end Transport Layer Security, discussed in Section 11.6.3) should be used to achieve a much better guarantee for those interactions that require it.

In the simplest implementation of Squirrel – which proved to be the most effective one – the node whose GUID is numerically closest to the GUID of an object becomes that object's *home node*, responsible for holding the cached copy of the object when there is one.

Client nodes are configured to include a local Squirrel proxy process, which takes responsibility for both local and remote caching of web objects. If a fresh copy of a required object is not in the local cache, Squirrel routes a *Get* request or a *cGet* request (when there is a stale copy of the object in the local cache) via Pastry to the home node. If the home node has a fresh copy, it directly responds to the client with a *not-modified* message or a fresh copy, as appropriate. If the home node has a stale copy or no copy of the object, it issues a *cGet* or a *Get* to the origin server, respectively. The origin server may respond with a *not-modified* message or a copy of the object. In the former case, the home node revalidates its cache entry and forwards a copy of the object to the client. In the latter case, it forwards a copy of the new value to the client and places a copy in its local cache if the object is cacheable.

Evaluation of Squirrel • Squirrel was evaluated by simulation using modelled loads derived from traces of the activity of existing centralized proxy web caches in two real working environments within Microsoft, one with 105 active clients (in Cambridge) and the other with more than 36,000 (in Redmond). The evaluation compared the performance of a Squirrel web cache with a centralized one in three respects:

The reduction in total external bandwidth used: The total external bandwidth is inversely related to the hit ratio, since it is only cache misses that generate requests to external web servers. The hit ratios observed for centralized web cache servers were 29% (for Redmond) and 38% (for Cambridge). When the same activity logs

were used to generate a simulated load for the Squirrel cache, with each client contributing 100 Mbytes of disk storage, very similar hit ratios of 28% (Redmond) and 37% (Cambridge) were achieved. It follows that the external bandwidth would be reduced by a similar proportion.

The latency perceived by users for access to web objects: The use of a routing overlay results in several message transfers (routing hops) across the local network to transmit a request from a client to the host responsible for caching the relevant object (the home node). The mean numbers of routing hops observed in the simulation were 4.11 hops to deliver a *GET* request in the Redmond case and 1.8 hops in the Cambridge case, whereas only a single message transfer is required to access a centralized cache service.

However local transfers take only a few milliseconds with modern Ethernet hardware, including TCP connection setup time, whereas wide area TCP message transfers across the Internet require 10–100 ms. The Squirrel authors therefore argue that the latency for access to objects found in the cache is swamped by the much greater latency of access to objects not found in the cache, giving a similar user experience to that provided with a centralized cache.

The computational and storage load imposed on client nodes: The average number of cache requests served for other nodes by each node over the whole period of the evaluation was extremely low, at only 0.31 per minute (Redmond), indicating that the overall proportion of system resources consumed is extremely low.

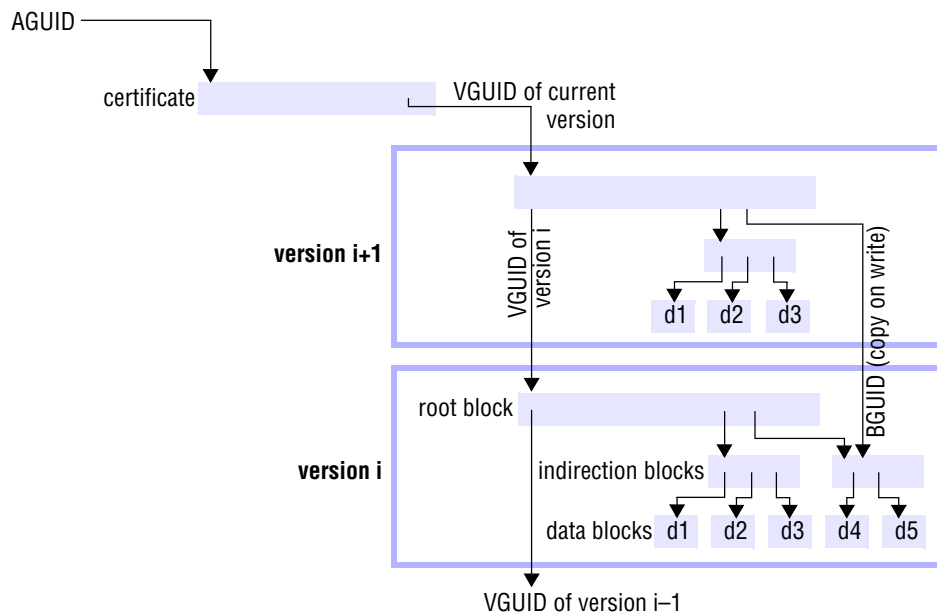
Based on the measurements described above, the authors of Squirrel concluded that its performance is comparable to that of a centralized cache. Squirrel achieves a reduction in the observed latency for web page access close to that achievable by a centralized cache server with a similarly sized dedicated cache. The additional load imposed on client nodes is low and likely to be imperceptible to users. The Squirrel system was subsequently deployed as the primary web cache in a local network with 52 client machines using Squirrel, and the results confirmed their conclusions.

10.6.2 OceanStore file store

The developers of Tapestry have designed and built a prototype for a peer-to-peer file store. Unlike PAST, it supports the storage of mutable files. The OceanStore design [Kubiatowicz *et al.* 2000; Kubiatowicz 2003; Rhea *et al.* 2001, 2003] aims to provide a very large scale, incrementally scalable persistent storage facility for mutable data objects with long-term persistence and reliability in an environment of constantly changing network and computing resources. OceanStore is intended for use in a variety of applications including the implementation of an NFS-like file service, electronic mail hosting, databases and other applications involving the sharing and persistent storage of large numbers of data objects.

The design includes provision for the replicated storage of both mutable and immutable data objects. The mechanism for maintaining consistency between replicas can be tailored to application needs in a manner that was inspired by the Bayou system (Section 18.4.2). Privacy and integrity are achieved through the encryption of data and the use of a Byzantine agreement protocol (see Section 15.5) for updates to replicated

Figure 10.13 Storage organization of OceanStore objects



Version $i+1$ has been updated in blocks $d1$, $d2$ and $d3$. The certificate and the root blocks include some metadata not shown. All unlabelled arrows are BGUIDs.

objects. This is needed because the trustworthiness of individual hosts cannot be assumed.

An OceanStore prototype, called Pond [Rhea *et al.* 2003], has been built. It is sufficiently complete to support applications and its performance has been evaluated against a variety of benchmarks in order to validate the OceanStore design and compare its performance with more traditional approaches. In the remainder of this section we give an overview of the OceanStore/Pond design and summarize the evaluation results.

Pond uses the Tapestry routing overlay mechanism to place blocks of data at nodes distributed throughout the Internet and to despatch requests to them.

Storage organization • OceanStore/Pond data objects are analogous to files, with their data stored in a set of blocks. But each object is represented as an ordered sequence of immutable versions that are (in principle) kept forever. Any update to an object results in the generation of a new version. The versions share any unchanged blocks, following the copy-on-write technique for creating and updating objects described in Section 7.4.2. So a small difference between versions requires only a small amount of additional storage.

Objects are structured in a manner that is reminiscent of the Unix filing system, with the data blocks organized and accessed through a metadata block called the root block and additional indirection blocks if necessary (*cf.* Unix *i-nodes*). Another level of indirection is used to associate a persistent textual or other externally visible name (for example, the pathname for a file) with the sequence of versions of a data object. Figure

Figure 10.14 Types of identifier used in OceanStore

<i>Name</i>	<i>Meaning</i>	<i>Description</i>
BGUID	block GUID	Secure hash of a data block
VGUID	version GUID	BGUID of the root block of a version
AGUID	active GUID	Uniquely identifies all the versions of an object

10.13 illustrates this organization. GUIDs are associated with the object (an AGUID), the root block for each version of the object (a VGUID), the indirection blocks and the data blocks (BGUIDs). Several replicas of each block are stored at peer nodes selected according to locality and storage availability criteria, and their GUIDs are published (using the *publish()* primitive of Figure 10.5) by each of the nodes that holds a replica so that Tapestry can be used by clients to access the blocks.

Three types of GUIDs are used, as summarized in Figure 10.14. The first two are GUIDs of the type normally assigned to objects stored in Tapestry – they are computed from the contents of the relevant block using a secure hash function so that they can be used later to authenticate and verify the integrity of the contents. The blocks that they reference are necessarily immutable, since any change to the contents of a block would invalidate the use of the GUID as an authentication token.

The third type of identifier used is AGUIDs. These refer (indirectly) to the entire stream of versions of an object, enabling clients to access the current version of the object or any previous version. Since the objects stored are mutable, the GUIDs used to identify them cannot be derived from their contents, because that would render GUIDs held in indexes, etc., obsolete whenever an object changed.

Instead, whenever a new storage object is created a permanent AGUID is generated by applying a secure hash function to an application-specific name (e.g., a file name) supplied by the client creating the object and a public key that represents the object's owner (see Section 11.2.5). In a filing system application, an AGUID would be stored in the directories against each file name.

The association between an AGUID and the sequence of versions of the object that it identifies is recorded in a signed certificate that is stored and replicated by a primary copy replication scheme (also called passive replication; see Section 18.3.1). The certificate includes the VGUID of the current version and the root block for every version contains the VGUID of the previous version, so there is a chain of references enabling clients that hold a certificate to traverse the entire chain of versions (Figure 10.13). A signed certificate is needed to ensure that the association is authentic and has been made by an authorized principal. Clients are expected to check this. Whenever a new version of an object is created, a new certificate is generated holding the VGUID of the new version together with a timestamp and a version sequence number.

The trust model for peer-to-peer systems requires that construction of each new certificate is agreed (as described below) amongst a small set of hosts called the *inner ring*. Whenever a new object is stored in OceanStore, a set of hosts is selected to act as the inner ring for that object. They use Tapestry's *publish()* primitive to make the AGUID for the object known to Tapestry. Clients can then use Tapestry to route requests for the object's certificate to one of the nodes in the inner ring.

The new certificate replaces the old primary copy held at each inner ring node and is disseminated to a larger number of secondary copies. It is left to clients to determine how often they check for a new version (a similar decision has to be taken for cached copies of files in NFS; most installations operate with a consistency window of 30 seconds between client and server; see Section 12.3).

As usual in peer-to-peer systems, trust cannot be placed in any individual host. The updating of primary copies requires consensus agreement between the hosts in the inner ring. They use a version of a state-machine-based Byzantine agreement algorithm described by Castro and Liskov [2000] to update the object and sign the certificate. The use of a Byzantine agreement protocol ensures that the certificate is correctly maintained even if some members of the inner ring fail or behave maliciously. Because the computational and communication costs of Byzantine agreement rise with the square of the number of hosts involved, the number of hosts in the inner ring is kept small and the resulting certificate is replicated more widely using the primary copy scheme mentioned above.

Performing an update also involves checking access rights and serializing the update with any other pending writes. Once the update process is completed for the primary copy, the results are disseminated to secondary replicas stored on hosts outside the inner ring using a multicast routing tree that is managed by Tapestry.

Because of their read-only nature, data blocks are replicated by a different, more storage-efficient mechanism. This mechanism is based on the division of each block into m equal-sized fragments, which are encoded using *erasure codes* [Weatherspoon and Kubiatowicz 2002] to n fragments, where $n > m$. The key property of erasure coding is that it is possible to reconstruct a block from any m of its fragments. In a system that uses erasure coding all data objects remain available with the loss of up to $n - m$ hosts. In the Pond implementation $m = 16$ and $n = 32$, so for a doubling of the storage cost, the system can tolerate the failure of up to 16 hosts without loss of data. Tapestry is used to store fragments in and retrieve them from the network.

This high level of fault tolerance and data availability is achieved at some cost in terms of reconstructing blocks from erasure-coded fragments. To minimize the impact of this, the whole blocks are also stored in the network using Tapestry. Since they can be reconstructed from their fragments, these blocks are treated as a cache – they are not fault tolerant and they can be disposed of when storage space is required.

Performance • Pond was developed as a prototype to prove the feasibility of a scalable peer-to-peer file service, rather than as a production implementation. It is implemented in Java and includes almost all of the design outlined above. It was evaluated against several purpose-designed benchmarks and in a simple emulation of an NFS client and server in terms of OceanStore objects. The developers tested the NFS emulation against the Andrew benchmark [Howard *et al.* 1988], which emulates a software development workload. The table in Figure 10.15 shows the results for the latter. They were obtained using 1 GHz Pentium III PC running Linux. The LAN tests were performed using a Gigabit Ethernet and the WAN results were obtained using two sets of nodes linked by the Internet.

The conclusions drawn by the authors were that the performance of OceanStore/Pond when operating over a wide area network (i.e., the Internet) substantially exceeds that of NFS for reading and is within a factor of three of NFS for

Figure 10.15 Performance evaluation of the Pond prototype emulating NFS

Phase	LAN		WAN		Predominant operations in benchmark
	Linux NFS	Pond	Linux NFS	Pond	
1	0.0	1.9	0.9	2.8	Read and write
2	0.3	11.0	9.4	16.8	Read and write
3	1.1	1.8	8.3	1.8	Read
4	0.5	1.5	6.9	1.5	Read
5	2.6	21.0	21.5	32.0	Read and write
Total	4.5	37.2	47.0	54.9	

The figures show times in seconds to run different phases of the Andrew benchmark. It has five phases: (1) creates subdirectories recursively, (2) copies a source tree, (3) examines the status of all the files in the tree without examining their data, (4) examines every byte of data in all the files and (5) compiles and links the files.

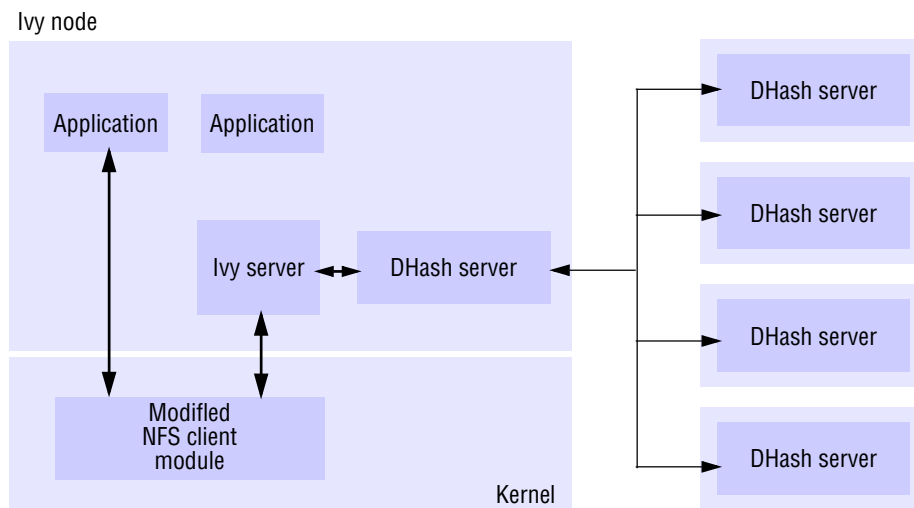
updating files and directories; the LAN results were substantially worse. Overall, the results suggest that an Internet-scale peer-to-peer file service based on the OceanStore design would be an effective solution for the distribution of files that do not change very rapidly (such as cached copies of web pages). Its potential for use as an alternative to NFS is questionable even for wide-area networks and is clearly uncompetitive for purely local use.

These results were obtained with data blocks stored without erasure-code-based fragmentation and replication. The use of public keys contributes substantially to the computational cost of Pond's operation. The figures shown are for 512-bit keys, whose security is good but less than perfect. The results for 1024-bit keys were substantially worse for the phases of those benchmarks that involved file updates. Other results obtained with purpose-designed benchmarks included measurement of the impact of the Byzantine agreement process on the latency of updates. These were in the range of 100 ms to 10 seconds. A test of update throughput achieved a maximum of 100 updates/second.

10.6.3 Ivy file system

Like OceanStore, Ivy [Muthitacharoen *et al.* 2002] is a read/write file system supporting multiple readers and writers implemented over an overlay routing layer and a distributed hash-addressed data store. Unlike OceanStore, the Ivy file system emulates a Sun NFS server. Ivy stores the state of files as logs of the file update requests issued by Ivy clients and reconstructs the files by scanning the logs whenever it is unable to satisfy an access request from its local cache. The log records are held in the DHash distributed hash-addressed storage service [Dabek *et al.* 2001]. (Logs were first used to record file updates in the Sprite distributed operating system [Rosenblum and Ooosterhout 1992], as described briefly in Section 12.5, but there they were used simply to optimize the update performance of the file system.)

Figure 10.16 Ivy system architecture



The design of Ivy resolves several previously unresolved issues arising from the need to host files in partially trusted or unreliable machines, including:

- The maintenance of consistent file metadata (*cf. i-node* contents in Unix/NFS file systems) with potentially concurrent file updates at different nodes. Locking is not used because the failure of nodes or network connectivity might cause indefinite blocking.
- Partial trust between participants and vulnerability to attacks of participants' machines. Recovery from integrity failures caused by such attacks is based on the notion of *views* of the file system. A view is a representation of the state constructed from logs of the updates made by a set of participants. Participants may be removed and a view recomputed without their updates. Thus a shared file system is seen as the result of merging all the updates performed by a (dynamically selected) set of participants.
- Continued operation during partitions in the network, which can result in conflicting updates to shared files. Conflicting updates are resolved using methods related to those used in the Coda file system (Section 18.4.3).

Ivy implements an API at each client node that is based on the NFS server protocol (similar to the set of operations listed in Section 12.3, Figure 12.9). Client nodes include an Ivy server process that uses DHash to store and access log records at nodes throughout a local or wide area network based on keys (GUIDs) that are computed as the hash of the record contents (see Figure 10.16). DHash implements a programming interface like the one shown in Figure 10.4 and replicates all entries at several nodes for

resilience and availability. The Ivy authors note that DHash could in principle be replaced by another distributed hash-addressed store such as Pastry, Tapestry or CAN.

An Ivy file store consists of a set of update logs, one log per participant. Each Ivy participant appends only to its own log but can read from all the logs that comprise the file system. Updates are stored in separate per-participant logs so that they can be rolled back in case of security breaches or consistency failures.

An Ivy log is a reverse time-ordered linked list of log entries. Each log entry is a timestamped record of a client request to change the contents or metadata of a file or directory. DHash uses the 160-bit SHA-1 hash of a record as a key for placing and retrieving the record. Each participant also maintains a mutable DHash block (called a *log-head*) that points to the participant's most recent log record. Mutable blocks are assigned a cryptographic public-key pair by their owner. The contents of the block are signed with the private key and can therefore be authenticated with the corresponding public key. Ivy uses version vectors (that is, vector timestamps; see Section 14.4) to impose a total order on log entries when reading from multiple logs.

DHash stores a log record using a SHA-1 hash of its contents as the key. Log records are chained in timestamp order using the DHash key as a link. The log-head holds the key for the most recent log entry. To store and retrieve log-heads, a public key pair is computed by the owner of the log. The public key value is used as its DHash key and the private key is used by the owner to sign the log-head. Any participant that has the public key can retrieve the log-head and use it to access all of the records in the log.

Assuming a file system composed of a single log for the moment, the canonical execution method for a request to read a sequence of bytes from a file requires a sequential scan of the log to find the log records that contain updates for the relevant portion of the file. Logs are of unlimited length, but the scan terminates when the first record or records are found that cover the required sequence of bytes.

The canonical algorithm to access a multi-user, multiple-log file system involves the comparison of vector timestamps in log records to determine the order of updates (since a global clock cannot be assumed).

The time taken to perform this process for an operation as simple as a *read* request is potentially very long. It is reduced to a more tolerable and predicable duration through the use of a combination of local caches and *snapshots*. Snapshots are representations of the file system computed and stored locally by each participant as a by-product of their use of the logs. They constitute a soft representation of the file system in the sense that they may be invalidated if a participant is ejected from the system.

Update consistency is *close-to-open*; that is, the updates performed on a file by an application are not visible to other processes until the file is closed. The use of a close-to-open consistency model enables *write* operations on a file to be saved at the client node until the file is closed; then the entire set of *write* operations is written as a single log record and a new log-head record is generated and written (an extension to the NFS protocol enables the occurrence of a *close* operation in the application to be notified to the Ivy server).

Since there is a separate Ivy server at each node and each autonomously stores its updates in a separate log without coordination with the other servers, the serialization of updates must be done at the time when logs are read in order to construct the content of files. The version vectors written into log records can be used to order most updates, but

conflicting updates are possible and they must be resolved by application-specific automatic or manual methods, as is done in Coda (see Section 18.4.3).

Data integrity is achieved by a combination of the mechanisms that we have already mentioned: log records are immutable and their address is a secure hash of their contents; log-heads are verified by checking a public-key signature of their contents. But the trust model allows for the possibility that a malicious participant may gain access to a file system. For example, they might delete files that they own maliciously. When this is detected, the malicious participant is ejected from the view; their log is no longer used to calculate the contents of the file system and files that they have deleted are once again visible in the new view.

The Ivy authors used a modified Andrew benchmark [Howard *et al.* 1988] to compare the performance of Ivy with a standard NFS server in local and wide area network environments. They considered (a) Ivy using local DHash servers compared to a single local NFS server and (b) Ivy using DHash servers located at several remote Internet sites compared to a single remote NFS server. They also considered the performance characteristics as a function of the numbers of participants in a view, the number of participants writing concurrently and the number of DHash servers used to store the logs.

They found that Ivy execution times were within a factor of two of NFS execution times for most of the tests in the benchmark and within a factor of three for all of them. The execution times for the wide area network deployment exceeded those for the local case by a factor of 10 or more, but similar ratios were obtained for a remote NFS server. Full details of the performance evaluation can be found in the Ivy paper [Muthitacharoen *et al.* 2002]. It should be noted, though, that NFS was not designed for wide area use; the Andrew File System and other more recently developed server-based systems such as xFS [Anderson *et al.* 1996] offer higher performance in wide area deployments and might have made better bases for the comparison. The primary contribution of Ivy is in its novel approach to the management of security and integrity in an environment of partial trust – an inevitable feature of very large distributed systems that span many organizations and jurisdictions.

10.7 Summary

Peer-to-peer architectures were first shown to support very large scale data sharing with the Internet-wide use of Napster and its descendants for digital music sharing. The fact that much of their use conflicted with copyright laws doesn't diminish their technical significance, although they did also have technical drawbacks that restricted their deployment to applications in which guarantees of data integrity and availability were unimportant.

Subsequent research resulted in the development of peer-to-peer middleware platforms that deliver requests to data objects wherever they are located in the Internet. In structured approaches, the objects are addressed using GUIDs, which are pure names containing no IP addressing information. Objects are placed at nodes according to some mapping function that is specific to each middleware system. Delivery is performed by a routing overlay in the middleware that maintains routing tables and forwards requests

along a route determined by calculating distance according to the chosen mapping function. In unstructured approaches, nodes form themselves into an ad hoc network and then propagate searches through neighbours to find appropriate resources. Several strategies have been developed to improve the performance of this search function and increase the overall scalability of the system.

The middleware platforms add integrity guarantees based on the use of a secure hash function to generate the GUIDs and availability guarantees based on the replication of objects at several nodes and on fault-tolerant routing algorithms.

The platforms have been deployed in several large-scale pilot applications, refined and evaluated. Recent evaluation results indicate that the technology is ready for deployment in applications involving large numbers of users sharing many data objects. The benefits of peer-to-peer systems include:

- their ability to exploit unused resources (storage, processing) in the host computers;
- their scalability to support large numbers of clients and hosts with excellent balancing of the loads on network links and host computing resources;
- the self-organizing properties of the middleware platforms which result in support costs that are largely independent of the numbers of clients and hosts deployed.

Weaknesses and subjects of current research include:

- their use for the storage of mutable data is relatively costly compared to a trusted, centralized service;
- the promising basis that they provide for client and host anonymity has not yet resulted in strong guarantees of anonymity.

EXERCISES

- 10.1 Early file-sharing applications such as Napster were restricted in their scalability by the need to maintain a central index of resources and the hosts that hold them. What other solutions to the indexing problem can you identify?
pages 428–430, 435, Section 18.4
- 10.2 The problem of maintaining indexes of available resources is application-dependent. Consider the suitability of each of your answers to Exercise 10.1 for:
- i) music and media file sharing;
 - ii) long-term storage of archived material such as journal or newspaper content;
 - iii) network storage of general-purpose read-write files.
- 10.3 What are the main guarantees that users expect conventional servers (e.g., web servers or file servers) to offer?
Section 1.5.5
- 10.4 The guarantees offered by conventional servers may be violated as a result of:
- i) physical damage to the host;
 - ii) Errors or inconsistencies introduced by system administrators or their managers;

- iii) successful attacks on the security of the system software;
- iv) hardware or software errors.

Give two examples of possible incidents for each type of violation. Which of them could be described as a breach of trust or a criminal act? Would they be breaches of trust if they occurred on a personal computer that was contributing some resources to a peer-to-peer service? Why is this relevant for peer-to-peer systems? *Section 11.1.1*

- 10.5 Peer-to-peer systems typically depend on *untrusted* and *volatile* computer systems for most of their resources. Trust is a social phenomenon with technical consequences. Volatility (i.e., unpredictable availability) also is often due to human actions. Elaborate your answers to Exercise 10.4 by discussing the possible ways in which each of them are likely to differ according to the following attributes of the computers used:

- i) ownership;
- ii) geographic location;
- iii) network connectivity;
- iv) country or jurisdiction.

What does this suggest about policies for the placement of data objects in a peer-to-peer storage service?

- 10.6 Assess the availability and trustworthiness of the personal computers in your environment. You should estimate:

Uptime: How many hours per day is the computer operating and connected to the Internet?

Software consistency: Is the software managed by a competent technician?

Security: Is the computer fully protected against tampering by its users or others?

Based on your assessment, discuss the feasibility of running a datasharing service on the set of computers you have assessed and outline the problems that must be addressed in a peer-to-peer data sharing service. *pages 431–432*

- 10.7 Explain why using the secure hash of an object to identify and route messages to it is tamper-proof. What properties are required of the hash function? How can integrity be maintained, even if a substantial proportion of peer nodes are subverted? *pages 426, 453, Section 11.4.3*

- 10.8 It is often argued that peer-to-peer systems can offer anonymity for:

- i) clients accessing resources;
- ii) the hosts providing access to resources.

Discuss each of these propositions. Suggest a way in which the resistance to attacks on anonymity might be improved. *page 429*

- 10.9 Routing algorithms choose a next hop according to an estimate of distance in some addressing space. Pastry and Tapestry both use circular linear address spaces in which a function based on the approximate numerical difference between GUIDs determines

their separation. Kademlia uses the XOR of the GUIDs. How does this help in the maintenance of routing tables? Does the XOR operation provide appropriate properties for a distance metric?
pages 435, [Maymounkov and Mazières 2002]

- 10.10 When the Squirrel peer-to-peer web caching service was evaluated by simulation, 4.11 hops were required on average to route a request for a cache entry when simulating the Redmond traffic, whereas only 1.8 were required for the Cambridge traffic. Explain this and show that it supports the theoretical performance claimed for Pastry.

pages 436, 450

- 10.11 In unstructured peer-to-peer systems, significant improvements on search results can be provided by the adoption of particular search strategies. Compare and contrast expanded ring search and random walk strategies, highlighting when each approach is likely to be effective.

page 446

TIME AND GLOBAL STATES

- 14.1 Introduction
- 14.2 Clocks, events and process states
- 14.3 Synchronizing physical clocks
- 14.4 Logical time and logical clocks
- 14.5 Global states
- 14.6 Distributed debugging
- 14.7 Summary

In this chapter, we introduce some topics related to the issue of time in distributed systems. Time is an important practical issue. For example, we require computers around the world to timestamp electronic commerce transactions consistently. Time is also an important theoretical construct in understanding how distributed executions unfold. But time is problematic in distributed systems. Each computer may have its own physical clock, but the clocks typically deviate, and we cannot synchronize them perfectly. We examine algorithms for synchronizing physical clocks approximately and then go on to explain logical clocks, including vector clocks, which are a tool for ordering events without knowing precisely when they occurred.

The absence of global physical time makes it difficult to find out the state of our distributed programs as they execute. We often need to know what state process A is in when process B is in a certain state, but we cannot rely on physical clocks to know what is true at the same time. The second half of the chapter examines algorithms to determine global states of distributed computations despite the lack of global time.

14.1 Introduction

This chapter introduces fundamental concepts and algorithms related to monitoring distributed systems as their execution unfolds, and to timing the events that occur in their executions.

Time is an important and interesting issue in distributed systems, for several reasons. First, time is a quantity we often want to measure accurately. In order to know at what time of day a particular event occurred at a particular computer it is necessary to synchronize its clock with an authoritative, external source of time. For example, an eCommerce transaction involves events at a merchant's computer and at a bank's computer. It is important, for auditing purposes, that those events are timestamped accurately.

Second, algorithms that depend upon clock synchronization have been developed for several problems in distribution [Liskov 1993]. These include maintaining the consistency of distributed data (the use of timestamps to serialize transactions is discussed in Section 16.6), checking the authenticity of a request sent to a server (a version of the Kerberos authentication protocol, discussed in Chapter 11, depends on loosely synchronized clocks) and eliminating the processing of duplicate updates (see, for example, Ladin *et al.* [1992]).

Measuring time can be problematic due to the existence of multiple frames of reference. Einstein demonstrated, in his Special Theory of Relativity, the intriguing consequences that follow from the observation that the speed of light is constant for all observers, regardless of their relative velocity. He proved from this assumption, among other things, that two events that are judged to be simultaneous in one frame of reference are not necessarily simultaneous according to observers in other frames of reference that are moving relative to it. For example, an observer on the Earth and an observer travelling away from the Earth in a spaceship will disagree on the time interval between events, the more so as their relative speed increases.

The relative order of two events can even be reversed for two different observers. But this cannot happen if one event causes the other to occur. In that case, the physical effect follows the physical cause for all observers, although the time elapsed between cause and effect can vary. The timing of physical events was thus proved to be relative to the observer, and Newton's notion of absolute physical time was shown to be without foundation. There is no special physical clock in the universe to which we can appeal when we want to measure intervals of time.

The notion of physical time is also problematic in a distributed system. This is not due to the effects of special relativity, which are negligible or nonexistent for normal computers (unless one counts computers travelling in spaceships!). Rather, the problem is based on a similar limitation in our ability to timestamp events at different nodes sufficiently accurately to know the order in which any pair of events occurred, or whether they occurred simultaneously. There is no absolute, global time to which we can appeal. And yet we sometimes need to observe distributed systems and establish whether certain states of affairs occurred at the same time. For example, in object-oriented systems we need to be able to establish whether references to a particular object no longer exist – whether the object has become garbage (in which case we can free its memory). Establishing this requires observations of the states of processes (to find out

whether they contain references) and of the communication channels between processes (in case messages containing references are in transit).

In the first half of this chapter, we examine methods whereby computer clocks can be approximately synchronized, using message passing. We go on to introduce logical clocks, including vector clocks, which are used to define an order of events without measuring the physical time at which they occurred.

In the second half, we describe algorithms whose purpose is to capture global states of distributed systems as they execute.

14.2 Clocks, events and process states

Chapter 2 presented an introductory model of interaction between the processes within a distributed system. Here we refine that model in order to help us to understand how to characterize the system's evolution as it executes, and how to timestamp the events in a system's execution that interest users. We begin by considering how to order and timestamp the events that occur at a single process.

We take a distributed system to consist of a collection \wp of N processes p_i , $i = 1, 2, \dots, N$. Each process executes on a single processor, and the processors do not share memory (Chapter 6 briefly considered the case of processes that share memory). Each process p_i in \wp has a state s_i that, in general, it transforms as it executes. The process's state includes the values of all the variables within it. Its state may also include the values of any objects in its local operating system environment that it affects, such as files. We assume that processes cannot communicate with one another in any way except by sending messages through the network. So, for example, if the processes operate robot arms connected to their respective nodes in the system, then they are not allowed to communicate by shaking one another's robot hands!

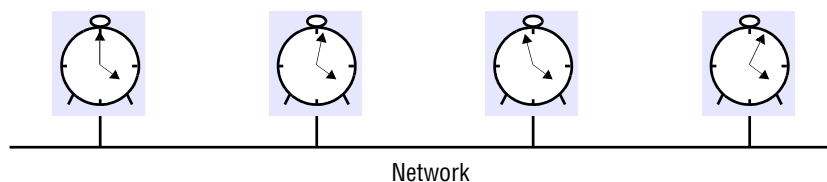
As each process p_i executes it takes a series of actions, each of which is either a message *send* or *receive* operation, or an operation that transforms p_i 's state – one that changes one or more of the values in s_i . In practice, we may choose to use a high-level description of the actions, according to the application. For example, if the processes in \wp are engaged in an eCommerce application, then the actions may be ones such as 'client dispatched order message' or 'merchant server recorded transaction to log'.

We define an event to be the occurrence of a single action that a process carries out as it executes – a communication action or a state-transforming action. The sequence of events within a single process p_i can be placed in a single, total ordering, which we denote by the relation \rightarrow_i between the events. That is, $e \rightarrow_i e'$ if and only if the event e occurs before e' at p_i . This ordering is well defined, whether or not the process is multi-threaded, since we have assumed that the process executes on a single processor.

Now we can define the *history* of process p_i to be the series of events that take place within it, ordered as we have described by the relation \rightarrow_i :

$$\text{history}(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \dots \rangle$$

Clocks • We have seen how to order the events at a process, but not how to timestamp them – i.e., to assign to them a date and time of day. Computers each contain their own physical clocks. These clocks are electronic devices that count oscillations occurring in

Figure 14.1 Skew between computer clocks in a distributed system

a crystal at a definite frequency, and typically divide this count and store the result in a counter register. Clock devices can be programmed to generate interrupts at regular intervals in order that, for example, timeslicing can be implemented; however, we shall not concern ourselves with this aspect of clock operation.

The operating system reads the node's hardware clock value, $H_i(t)$, scales it and adds an offset so as to produce a software clock $C_i(t) = \alpha H_i(t) + \beta$ that approximately measures real, physical time t for process p_i . In other words, when the real time in an absolute frame of reference is t , $C_i(t)$ is the reading on the software clock. For example, $C_i(t)$ could be the 64-bit value of the number of nanoseconds that have elapsed at time t since a convenient reference time. In general, the clock is not completely accurate, so $C_i(t)$ will differ from t . Nonetheless, if C_i behaves sufficiently well (we shall examine the notion of clock correctness shortly), we can use its value to timestamp any event at p_i . Note that successive events will correspond to different timestamps only if the *clock resolution* – the period between updates of the clock value – is smaller than the time interval between successive events. The rate at which events occur depends on such factors as the length of the processor instruction cycle.

Clock skew and clock drift • Computer clocks, like any others, tend not to be in perfect agreement (Figure 14.1). The instantaneous difference between the readings of any two clocks is called their *skew*. Also, the crystal-based clocks used in computers are, like any other clocks, subject to *clock drift*, which means that they count time at different rates, and so diverge. The underlying oscillators are subject to physical variations, with the consequence that their frequencies of oscillation differ. Moreover, even the same clock's frequency varies with temperature. Designs exist that attempt to compensate for this variation, but they cannot eliminate it. The difference in the oscillation period between two clocks might be extremely small, but the difference accumulated over many oscillations leads to an observable difference in the counters registered by two clocks, no matter how accurately they were initialized to the same value. A clock's *drift rate* is the change in the offset (difference in reading) between the clock and a nominal perfect reference clock per unit of time measured by the reference clock. For ordinary clocks based on a quartz crystal this is about 10^{-6} seconds/second, giving a difference of 1 second every 1,000,000 seconds, or 11.6 days. The drift rate of 'high-precision' quartz clocks is about 10^{-7} or 10^{-8} .

Coordinated Universal Time • Computer clocks can be synchronized to external sources of highly accurate time. The most accurate physical clocks use atomic oscillators, whose drift rate is about one part in 10^{13} . The output of these atomic clocks is used as the

standard for elapsed real time, known as *International Atomic Time*. Since 1967, the standard second has been defined as 9,192,631,770 periods of transition between the two hyperfine levels of the ground state of Caesium-133 (Cs^{133}).

Seconds and years and other time units that we use are rooted in astronomical time. They were originally defined in terms of the rotation of the Earth on its axis and its rotation about the Sun. However, the period of the Earth's rotation about its axis is gradually getting longer, primarily because of tidal friction; atmospheric effects and convection currents within the Earth's core also cause short-term increases and decreases in the period. So astronomical time and atomic time have a tendency to get out of step.

Coordinated Universal Time – abbreviated as UTC (from the French equivalent) – is an international standard for timekeeping. It is based on atomic time, but a so-called 'leap second' is inserted – or, more rarely, deleted – occasionally to keep it in step with astronomical time. UTC signals are synchronized and broadcast regularly from land-based radio stations and satellites covering many parts of the world. For example, in the USA, the radio station WWV broadcasts time signals on several shortwave frequencies. Satellite sources include the *Global Positioning System* (GPS).

Receivers are available commercially. Compared with 'perfect' UTC, the signals received from land-based stations have an accuracy on the order of 0.1–10 milliseconds, depending on the station used. Signals received from GPS satellites are accurate to about 1 microsecond. Computers with receivers attached can synchronize their clocks with these timing signals.

14.3 Synchronizing physical clocks

In order to know at what time of day events occur at the processes in our distributed system \wp – for example, for accountancy purposes – it is necessary to synchronize the processes' clocks, C_i , with an authoritative, external source of time. This is *external synchronization*. And if the clocks C_i are synchronized with one another to a known degree of accuracy, then we can measure the interval between two events occurring at different computers by appealing to their local clocks, even though they are not necessarily synchronized to an external source of time. This is *internal synchronization*. We define these two modes of synchronization more closely as follows, over an interval of real time I :

External synchronization: For a synchronization bound $D > 0$, and for a source S of UTC time, $|S(t) - C_i(t)| < D$, for $i = 1, 2, \dots, N$ and for all real times t in I . Another way of saying this is that the clocks C_i are *accurate* to within the bound D .

Internal synchronization: For a synchronization bound $D > 0$, $|C_i(t) - C_j(t)| < D$ for $i, j = 1, 2, \dots, N$, and for all real times t in I . Another way of saying this is that the clocks C_i *agree* within the bound D .

Clocks that are internally synchronized are not necessarily externally synchronized, since they may drift collectively from an external source of time even though they agree with one another. However, it follows from the definitions that if the system \wp is

externally synchronized with a bound D , then the same system is internally synchronized with a bound of $2D$.

Various notions of *correctness* for clocks have been suggested. It is common to define a hardware clock H to be correct if its drift rate falls within a known bound $\rho > 0$ (a value derived from one supplied by the manufacturer, such as 10^{-6} seconds/second). This means that the error in measuring the interval between real times t and t' ($t' > t$) is bounded:

$$(1 - \rho)(t' - t) \leq H(t') - H(t) \leq (1 + \rho)(t' - t)$$

This condition forbids jumps in the value of hardware clocks (during normal operation). Sometimes we also require our software clocks to obey the condition but a weaker condition of *monotonicity* may suffice. Monotonicity is the condition that a clock C only ever advances:

$$t' > t \Rightarrow C(t') > C(t)$$

For example, the UNIX *make* facility is a tool that is used to compile only those source files that have been modified since they were last compiled. The modification dates of each corresponding pair of source and object files are compared to determine this condition. If a computer whose clock was running fast set its clock back after compiling a source file but before the file was changed, the source file might appear to have been modified prior to the compilation. Erroneously, *make* will not recompile the source file.

We can achieve monotonicity despite the fact that a clock is found to be running fast. We need only change the rate at which updates are made to the time as given to applications. This can be achieved in software without changing the rate at which the underlying hardware clock ticks – recall that $C_i(t) = \alpha H_i(t) + \beta$, where we are free to choose the values of α and β .

A hybrid correctness condition that is sometimes applied is to require that a clock obeys the monotonicity condition, and that its drift rate is bounded between synchronization points, but to allow the clock value to jump ahead at synchronization points.

A clock that does not keep to whatever correctness conditions apply is defined to be *faulty*. A clock's *crash failure* is said to occur when the clock stops ticking altogether; any other clock failure is an *arbitrary failure*. A historical example of an arbitrary failure is that of a clock with the 'Y2K bug', which broke the monotonicity condition by registering the date after 31 December 1999 as 1 January 1900 instead of 2000; another example is a clock whose batteries are very low and whose drift rate suddenly becomes very large.

Note that clocks do not have to be accurate to be correct, according to the definitions. Since the goal may be internal rather than external synchronization, the criteria for correctness are only concerned with the proper functioning of the clock's 'mechanism', not its absolute setting.

We now describe algorithms for external synchronization and for internal synchronization.

14.3.1 Synchronization in a synchronous system

We begin by considering the simplest possible case: of internal synchronization between two processes in a synchronous distributed system. In a synchronous system, bounds are known for the drift rate of clocks, the maximum message transmission delay, and the time required to execute each step of a process (see Section 2.4.1).

One process sends the time t on its local clock to the other in a message m . In principle, the receiving process could set its clock to the time $t + T_{trans}$, where T_{trans} is the time taken to transmit m between them. The two clocks would then agree (since the aim is internal synchronization, it does not matter whether the sending process's clock is accurate).

Unfortunately, T_{trans} is subject to variation and is unknown. In general, other processes are competing for resources with the processes to be synchronized at their respective nodes, and other messages compete with m for the network resources. Nonetheless, there is always a minimum transmission time, min , that would be obtained if no other processes executed and no other network traffic existed; min can be measured or conservatively estimated.

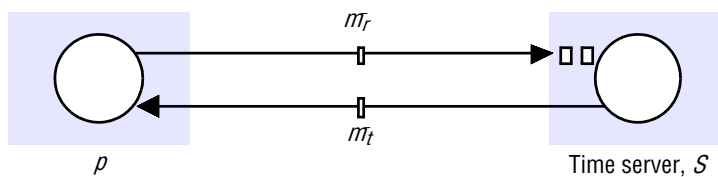
In a synchronous system, by definition, there is also an upper bound max on the time taken to transmit any message. Let the uncertainty in the message transmission time be u , so that $u = (max - min)$. If the receiver sets its clock to be $t + min$, then the clock skew may be as much as u , since the message may in fact have taken time max to arrive. Similarly, if it sets its clock to $t + max$, the skew may again be as large as u . If, however, it sets its clock to the halfway point, $t + (max + min)/2$, then the skew is at most $u/2$. In general, for a synchronous system, the optimum bound that can be achieved on clock skew when synchronizing N clocks is $u(1 - 1/N)$ [Lundelius and Lynch 1984].

Most distributed systems found in practice are asynchronous: the factors leading to message delays are not bounded in their effect, and there is no upper bound max on message transmission delays. This is particularly so for the Internet. For an asynchronous system, we may say only that $T_{trans} = min + x$, where $x \geq 0$. The value of x is not known in a particular case, although a distribution of values may be measurable for a particular installation.

14.3.2 Cristian's method for synchronizing clocks

Cristian [1989] suggested the use of a time server, connected to a device that receives signals from a source of UTC, to synchronize computers externally. Upon request, the server process S supplies the time according to its clock, as shown in Figure 14.2.

Figure 14.2 Clock synchronization using a time server



Cristian observed that while there is no upper bound on message transmission delays in an asynchronous system, the round-trip times for messages exchanged between pairs of processes are often reasonably short – a small fraction of a second. He describes the algorithm as *probabilistic*: the method achieves synchronization only if the observed round-trip times between client and server are sufficiently short compared with the required accuracy.

A process p requests the time in a message m_r , and receives the time value t in a message m_t (t is inserted in m_t at the last possible point before transmission from S 's computer). Process p records the total round-trip time T_{round} taken to send the request m_r and receive the reply m_t . It can measure this time with reasonable accuracy if its rate of clock drift is small. For example, the round-trip time should be on the order of 1–10 milliseconds on a LAN, over which time a clock with a drift rate of 10^{-6} seconds/second varies by at most 10^{-5} milliseconds.

A simple estimate of the time to which p should set its clock is $t + T_{round}/2$, which assumes that the elapsed time is split equally before and after S placed t in m_t . This is normally a reasonably accurate assumption, unless the two messages are transmitted over different networks. If the value of the minimum transmission time min is known or can be conservatively estimated, then we can determine the accuracy of this result as follows.

The earliest point at which S could have placed the time in m_t was min after p dispatched m_r . The latest point at which it could have done this was min before m_t arrived at p . The time by S 's clock when the reply message arrives is therefore in the range $[t + min, t + T_{round} - min]$. The width of this range is $T_{round} - 2min$, so the accuracy is $\pm(T_{round}/2 - min)$.

Variability can be dealt with to some extent by making several requests to S (spacing the requests so that transitory congestion can clear) and taking the minimum value of T_{round} to give the most accurate estimate. The greater the accuracy required, the smaller the probability of achieving it. This is because the most accurate results are those in which both messages are transmitted in a time close to min – an unlikely event in a busy network.

Discussion of Cristian's algorithm • As described, Cristian's method suffers from the problem associated with all services implemented by a single server: that the single time server might fail and thus render synchronization temporarily impossible. Cristian suggested, for this reason, that time should be provided by a group of synchronized time servers, each with a receiver for UTC time signals. For example, a client could multicast its request to all servers and use only the first reply obtained.

Note that a faulty time server that replied with spurious time values, or an imposter time server that replied with deliberately incorrect times, could wreak havoc in a computer system. These problems were beyond the scope of the work described by Cristian [1989], which assumes that sources of external time signals are self-checking. Cristian and Fetzer [1994] describe a family of probabilistic protocols for internal clock synchronization, each of which tolerates certain failures. Srikanth and Toueg [1987] first described an algorithm that is optimal with respect to the accuracy of the synchronized clocks, while tolerating some failures. Dolev *et al.* [1986] showed that if f is the number of faulty clocks out of a total of N , then we must have $N > 3f$ if the other, correct, clocks are still to be able to achieve agreement. The problem of dealing with

faulty clocks is partially addressed by the Berkeley algorithm, which is described next. The problem of malicious interference with time synchronization can be dealt with by authentication techniques.

14.3.3 The Berkeley algorithm

Gusella and Zatti [1989] describe an algorithm for internal synchronization that they developed for collections of computers running Berkeley UNIX. In it, a coordinator computer is chosen to act as the *master*. Unlike in Cristian's protocol, this computer periodically polls the other computers whose clocks are to be synchronized, called *slaves*. The slaves send back their clock values to it. The master estimates their local clock times by observing the round-trip times (similarly to Cristian's technique), and it averages the values obtained (including its own clock's reading). The balance of probabilities is that this average cancels out the individual clocks' tendencies to run fast or slow. The accuracy of the protocol depends upon a nominal maximum round-trip time between the master and the slaves. The master eliminates any occasional readings associated with larger times than this maximum.

Instead of sending the updated current time back to the other computers – which would introduce further uncertainty due to the message transmission time – the master sends the amount by which each individual slave's clock requires adjustment. This can be a positive or negative value.

The Berkeley algorithm eliminates readings from faulty clocks. Such clocks could have a significant adverse effect if an ordinary average was taken so instead the master takes a *fault-tolerant average*. That is, a subset is chosen of clocks that do not differ from one another by more than a specified amount, and the average is taken of readings from only these clocks.

Gusella and Zatti describe an experiment involving 15 computers whose clocks were synchronized to within about 20–25 milliseconds using their protocol. The local clocks' drift rates were measured to be less than 2×10^{-5} , and the maximum round-trip time was taken to be 10 milliseconds.

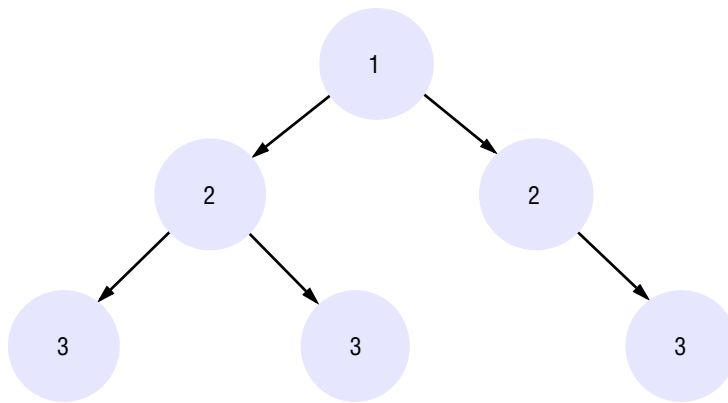
Should the master fail, then another can be elected to take over and function exactly as its predecessor. Section 15.3 discusses some general-purpose election algorithms. Note that these are not guaranteed to elect a new master in bounded time, so the difference between two clocks would be unbounded if they were used.

14.3.4 The Network Time Protocol

Cristian's method and the Berkeley algorithm are intended primarily for use within intranets. The Network Time Protocol (NTP) [Mills 1995] defines an architecture for a time service and a protocol to distribute time information over the Internet.

NTP's chief design aims and features are as follows:

To provide a service enabling clients across the Internet to be synchronized accurately to UTC: Although large and variable message delays are encountered in Internet communication, NTP employs statistical techniques for the filtering of timing data and it discriminates between the quality of timing data from different servers.

Figure 14.3 An example synchronization subnet in an NTP implementation

Arrows denote synchronization control, numbers denote strata.

To provide a reliable service that can survive lengthy losses of connectivity: There are redundant servers and redundant paths between the servers. The servers can reconfigure so as to continue to provide the service if one of them becomes unreachable.

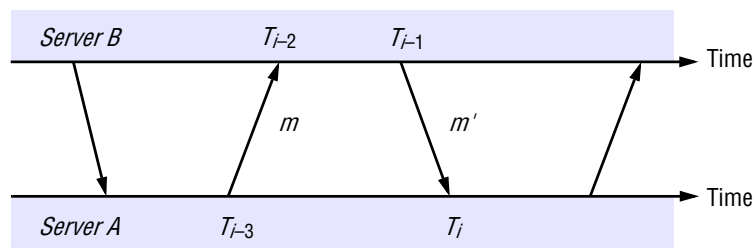
To enable clients to resynchronize sufficiently frequently to offset the rates of drift found in most computers: The service is designed to scale to large numbers of clients and servers.

To provide protection against interference with the time service, whether malicious or accidental: The time service uses authentication techniques to check that timing data originate from the claimed trusted sources. It also validates the return addresses of messages sent to it.

The NTP service is provided by a network of servers located across the Internet. *Primary servers* are connected directly to a time source such as a radio clock receiving UTC; *secondary servers* are synchronized, ultimately, with primary servers. The servers are connected in a logical hierarchy called a *synchronization subnet* (see Figure 14.3), whose levels are called *strata*. Primary servers occupy stratum 1: they are at the root. Stratum 2 servers are secondary servers that are synchronized directly with the primary servers; stratum 3 servers are synchronized with stratum 2 servers, and so on. The lowest-level (leaf) servers execute in users' workstations.

The clocks belonging to servers with high stratum numbers are liable to be less accurate than those with low stratum numbers, because errors are introduced at each level of synchronization. NTP also takes into account the total message round-trip delays to the root in assessing the quality of timekeeping data held by a particular server.

The synchronization subnet can reconfigure as servers become unreachable or failures occur. If, for example, a primary server's UTC source fails, then it can become

Figure 14.4 Messages exchanged between a pair of NTP peers

a stratum 2 secondary server. If a secondary server's normal source of synchronization fails or becomes unreachable, then it may synchronize with another server.

NTP servers synchronize with one another in one of three modes: multicast, procedure-call and symmetric mode. *Multicast mode* is intended for use on a high-speed LAN. One or more servers periodically multicasts the time to the servers running in other computers connected by the LAN, which set their clocks assuming a small delay. This mode can achieve only relatively low accuracies, but ones that nonetheless are considered sufficient for many purposes.

Procedure-call mode is similar to the operation of Cristian's algorithm, described in Section 14.3.2. In this mode, one server accepts requests from other computers, which it processes by replying with its timestamp (current clock reading). This mode is suitable where higher accuracies are required than can be achieved with multicast, or where multicast is not supported in hardware. For example, file servers on the same or a neighbouring LAN that need to keep accurate timing information for file accesses could contact a local server in procedure-call mode.

Finally, *symmetric mode* is intended for use by the servers that supply time information in LANs and by the higher levels (lower strata) of the synchronization subnet, where the highest accuracies are to be achieved. A pair of servers operating in symmetric mode exchange messages bearing timing information. Timing data are retained as part of an association between the servers that is maintained in order to improve the accuracy of their synchronization over time.

In all modes, messages are delivered unreliably, using the standard UDP Internet transport protocol. In procedure-call mode and symmetric mode, processes exchange pairs of messages. Each message bears timestamps of recent message events: the local times when the previous NTP message between the pair was sent and received, and the local time when the current message was transmitted. The recipient of the NTP message notes the local time when it receives the message. The four times T_{i-3} , T_{i-2} , T_{i-1} and T_i are shown in Figure 14.4 for the messages m and m' sent between servers A and B . Note that in symmetric mode, unlike in Cristian's algorithm, there can be a non-negligible delay between the arrival of one message and the dispatch of the next. Also, messages may be lost, but the three timestamps carried by each message are nonetheless valid.

For each pair of messages sent between two servers the NTP calculates an *offset* o_i , which is an estimate of the actual offset between the two clocks, and a *delay* d_i , which is the total transmission time for the two messages. If the true offset of the clock at B relative to that at A is o , and if the actual transmission times for m and m' are t and t' , respectively, then we have:

$$T_{i-2} = T_{i-3} + t + o \text{ and } T_i = T_{i-1} + t' - o$$

This leads to:

$$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

and:

$$o = o_i + (t' - t)/2, \text{ where } o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$$

Using the fact that $t, t' \geq 0$, it can be shown that $o_i - d_i/2 \leq o \leq o_i + d_i/2$. Thus o_i is an estimate of the offset, and d_i is a measure of the accuracy of this estimate.

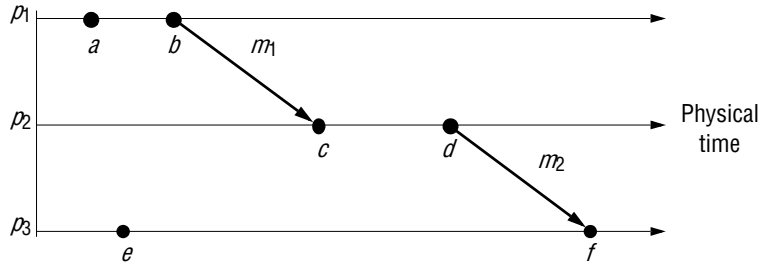
NTP servers apply a data filtering algorithm to successive pairs $\langle o_i, d_i \rangle$, which estimates the offset o and calculates the quality of this estimate as a statistical quantity called the *filter dispersion*. A relatively high filter dispersion represents relatively unreliable data. The eight most recent pairs $\langle o_i, d_i \rangle$ are retained. As with Cristian's algorithm, the value of o_j that corresponds to the minimum value d_j is chosen to estimate o .

The value of the offset derived from communication with a single source is not necessarily used by itself to control the local clock, however. In general, an NTP server engages in message exchanges with several of its peers. In addition to data filtering applied to exchanges with each single peer, NTP applies a peer-selection algorithm. This examines the values obtained from exchanges with each of several peers, looking for relatively unreliable values. The output from this algorithm may cause a server to change the peer that it primarily uses for synchronization.

Peers with lower stratum numbers are more favoured than those in higher strata because they are 'closer' to the primary time sources. Also, those with the lowest *synchronization dispersion* are relatively favoured. This is the sum of the filter dispersions measured between the server and the root of the synchronization subnet. (Peers exchange synchronization dispersions in messages, allowing this total to be calculated.)

NTP employs a phase lock loop model [Mills 1995], which modifies the local clock's update frequency in accordance with observations of its drift rate. To take a simple example, if a clock is discovered always to gain time at the rate of, say, four seconds per hour, then its frequency can be reduced slightly (in software or hardware) to compensate for this. The clock's drift in the intervals between synchronization is thus reduced.

Mills quotes synchronization accuracies on the order of tens of milliseconds over Internet paths, and one millisecond on LANs.

Figure 14.5 Events occurring at three processes

14.4 Logical time and logical clocks

From the point of view of any single process, events are ordered uniquely by times shown on the local clock. However, as Lamport [1978] pointed out, since we cannot synchronize clocks perfectly across a distributed system, we cannot in general use physical time to find out the order of any arbitrary pair of events occurring within it.

In general, we can use a scheme that is similar to physical causality but that applies in distributed systems to order some of the events that occur at different processes. This ordering is based on two simple and intuitively obvious points:

- If two events occurred at the same process p_i ($i = 1, 2, \dots, N$), then they occurred in the order in which p_i observes them – this is the order \rightarrow_i that we defined above.
- Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving the message.

Lamport called the partial ordering obtained by generalizing these two relationships the *happened-before* relation. It is also sometimes known as the relation of *causal ordering* or *potential causal ordering*.

We can define the happened-before relation, denoted by \rightarrow , as follows:

HB1: If \exists process p_i : $e \rightarrow_i e'$, then $e \rightarrow e'$.

HB2: For any message m , $send(m) \rightarrow receive(m)$
 – where $send(m)$ is the event of sending the message, and $receive(m)$ is the event of receiving it.

HB3: If e , e' and e'' are events such that $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

Thus, if e and e' are events, and if $e \rightarrow e'$, then we can find a series of events e_1, e_2, \dots, e_n occurring at one or more processes such that $e = e_1$ and $e' = e_n$, and for $i = 1, 2, \dots, N-1$ either HB1 or HB2 applies between e_i and e_{i+1} . That is, either they occur in succession at the same process, or there is a message m such that $e_i = send(m)$ and $e_{i+1} = receive(m)$. The sequence of events e_1, e_2, \dots, e_n need not be unique.

The relation \rightarrow is illustrated for the case of three processes, p_1 , p_2 and p_3 , in Figure 14.5. It can be seen that $a \rightarrow b$, since the events occur in this order at process p_1 ($a \rightarrow_i b$), and similarly $c \rightarrow d$. Furthermore, $b \rightarrow c$, since these events are the sending and

reception of message m_1 , and similarly $d \rightarrow f$. Combining these relations, we may also say that, for example, $a \rightarrow f$.

It can also be seen from Figure 14.5 that not all events are related by the relation \rightarrow . For example, $a \not\rightarrow e$ and $e \not\rightarrow a$, since they occur at different processes, and there is no chain of messages intervening between them. We say that events such as a and e that are not ordered by \rightarrow are *concurrent* and write this $a \parallel e$.

The relation \rightarrow captures a flow of data intervening between two events. Note, however, that in principle data can flow in ways other than by message passing. For example, if Smith enters a command to his process to send a message, then telephones Jones, who commands her process to issue another message, the issuing of the first message clearly happened-before that of the second. Unfortunately, since no network messages were sent between the issuing processes, we cannot model this type of relationship in our system.

Another point to note is that if the happened-before relation holds between two events, then the first might or might not actually have caused the second. For example, if a server receives a request message and subsequently sends a reply, then clearly the reply transmission is caused by the request transmission. However, the relation \rightarrow captures only potential causality, and two events can be related by \rightarrow even though there is no real connection between them. A process might, for example, receive a message and subsequently issue another message, but one that it issues every five minutes anyway and that bears no specific relation to the first message. No actual causality has been involved, but the relation \rightarrow would order these events.

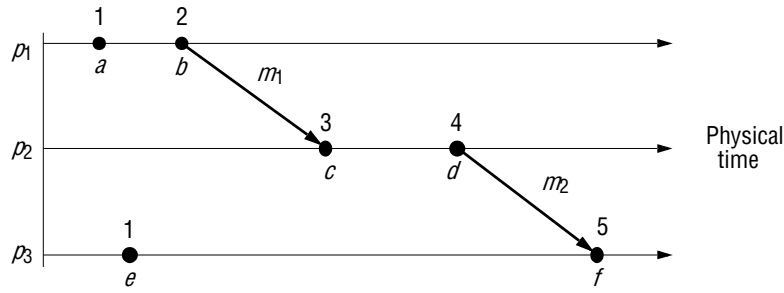
Logical clocks • Lamport [1978] invented a simple mechanism by which the happened-before ordering can be captured numerically, called a *logical clock*. A Lamport logical clock is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock. Each process p_i keeps its own logical clock, L_i , which it uses to apply so-called *Lamport timestamps* to events. We denote the timestamp of event e at p_i by $L_i(e)$, and by $L(e)$ we denote the timestamp of event e at whatever process it occurred at.

To capture the happened-before relation \rightarrow , processes update their logical clocks and transmit the values of their logical clocks in messages as follows:

- LC1: L_i is incremented before each event is issued at process p_i :
 $L_i := L_i + 1$.
- LC2: (a) When a process p_i sends a message m , it piggybacks on m the value $t = L_i$.
 (b) On receiving (m, t) , a process p_j computes $L_j := \max(L_j, t)$ and then applies LC1 before timestamping the event *receive*(m).

Although we increment clocks by 1, we could have chosen any positive value. It can easily be shown, by induction on the length of any sequence of events relating two events e and e' , that $e \rightarrow e' \Rightarrow L(e) < L(e')$.

Note that the converse is not true. If $L(e) < L(e')$, then we cannot infer that $e \rightarrow e'$. In Figure 14.6 we illustrate the use of logical clocks for the example given in Figure 14.5. Each of the processes p_1 , p_2 and p_3 has its logical clock initialized to 0. The clock values given are those immediately after the event to which they are adjacent. Note that, for example, $L(b) > L(e)$ but $b \parallel e$.

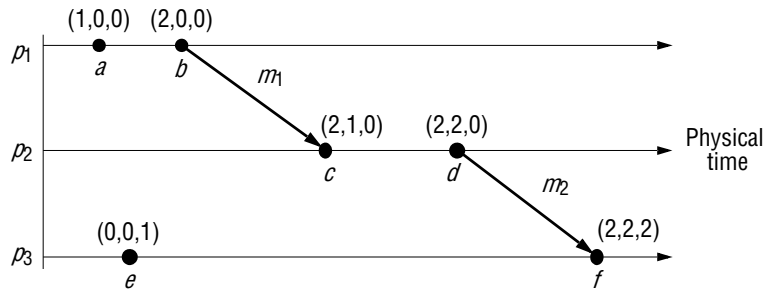
Figure 14.6 Lamport timestamps for the events shown in Figure 14.5

Totally ordered logical clocks • Some pairs of distinct events, generated by different processes, have numerically identical Lamport timestamps. However, we can create a total order on the set of events – that is, one for which all pairs of distinct events are ordered – by taking into account the identifiers of the processes at which events occur. If e is an event occurring at p_i with local timestamp T_i , and e' is an event occurring at p_j with local timestamp T_j , we define the global logical timestamps for these events to be (T_i, i) and (T_j, j) , respectively. And we define $(T_i, i) < (T_j, j)$ if and only if either $T_i < T_j$, or $T_i = T_j$ and $i < j$. This ordering has no general physical significance (because process identifiers are arbitrary), but it is sometimes useful. Lamport used it, for example, to order the entry of processes to a critical section.

Vector clocks • Mattern [1989] and Fidge [1991] developed vector clocks to overcome the shortcoming of Lamport's clocks: the fact that from $L(e) < L(e')$ we cannot conclude that $e \rightarrow e'$. A vector clock for a system of N processes is an array of N integers. Each process keeps its own vector clock, V_i , which it uses to timestamp local events. Like Lamport timestamps, processes piggyback vector timestamps on the messages they send to one another, and there are simple rules for updating the clocks:

- VC1: Initially, $V_i[j] = 0$, for $i, j = 1, 2, \dots, N$.
- VC2: Just before p_i timestamps an event, it sets $V_i[i] := V_i[i] + 1$.
- VC3: p_i includes the value $t = V_i$ in every message it sends.
- VC4: When p_i receives a timestamp t in a message, it sets $V_i[j] := \max(V_i[j], t[j])$, for $j = 1, 2, \dots, N$. Taking the component-wise maximum of two vector timestamps in this way is known as a *merge* operation.

For a vector clock V_i , $V_i[i]$ is the number of events that p_i has timestamped, and $V_i[j]$ ($j \neq i$) is the number of events that have occurred at p_j that have potentially affected p_i . (Process p_j may have timestamped more events by this point, but no information has flowed to p_i about them in messages as yet.)

Figure 14.7 Vector timestamps for the events shown in Figure 14.5

We may compare vector timestamps as follows:

$$V = V' \text{ iff } V[j] = V'[j] \text{ for } j = 1, 2, \dots, N$$

$$V \leq V' \text{ iff } V[j] \leq V'[j] \text{ for } j = 1, 2, \dots, N$$

$$V < V' \text{ iff } V \leq V' \wedge V \neq V'$$

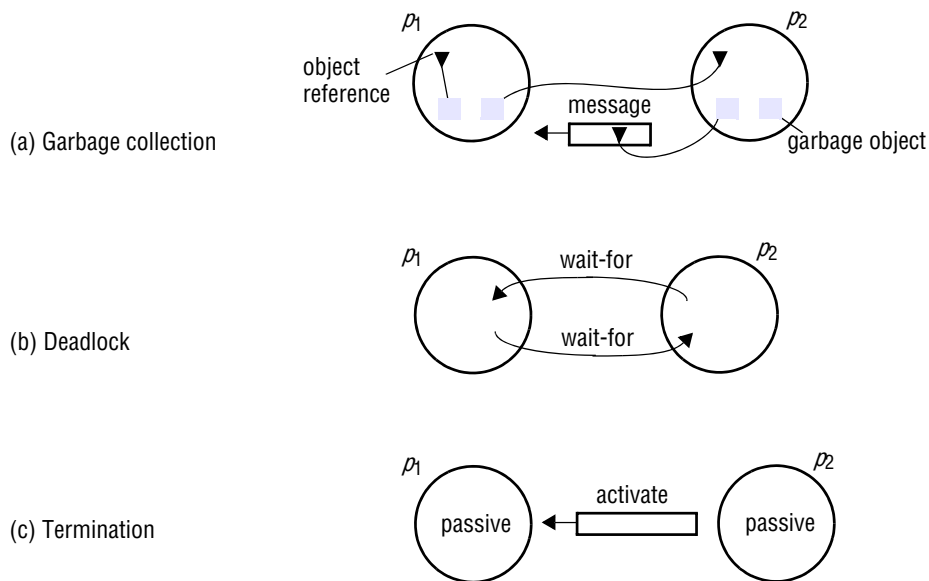
Let $V(e)$ be the vector timestamp applied by the process at which e occurs. It is straightforward to show, by induction on the length of any sequence of events relating two events e and e' , that $e \rightarrow e' \Rightarrow V(e) < V(e')$. Exercise 10.13 leads the reader to show the converse: if $V(e) < V(e')$, then $e \rightarrow e'$.

Figure 14.7 shows the vector timestamps of the events of Figure 14.5. It can be seen, for example, that $V(a) < V(f)$, which reflects the fact that $a \rightarrow f$. Similarly, we can tell when two events are concurrent by comparing their timestamps. For example, that $c \parallel e$ can be seen from the facts that neither $V(c) \leq V(e)$ nor $V(e) \leq V(c)$.

Vector timestamps have the disadvantage, compared with Lamport timestamps, of taking up an amount of storage and message payload that is proportional to N , the number of processes. Charron-Bost [1991] showed that, if we are to be able to tell whether or not two events are concurrent by inspecting their timestamps, then the dimension N is unavoidable. However, techniques exist for storing and transmitting smaller amounts of data, at the expense of the processing required to reconstruct complete vectors. Raynal and Singhal [1996] give an account of some of these techniques. They also describe the notion of *matrix clocks*, whereby processes keep estimates of other processes' vector times as well as their own.

14.5 Global states

In this and the next section we examine the problem of finding out whether a particular property is true of a distributed system as it executes. We begin by giving the examples of distributed garbage collection, deadlock detection, termination detection and debugging:

Figure 14.8 Detecting global properties

Distributed garbage collection: An object is considered to be garbage if there are no longer any references to it anywhere in the distributed system. The memory taken up by that object can be reclaimed once it is known to be garbage. To check that an object is garbage, we must verify that there are no references to it anywhere in the system. In Figure 14.8(a), process p_1 has two objects that both have references – one has a reference within p_1 itself, and p_2 has a reference to the other. Process p_2 has one garbage object, with no references to it anywhere in the system. It also has an object for which neither p_1 nor p_2 has a reference, but there is a reference to it in a message that is in transit between the processes. This shows that when we consider properties of a system, we must include the state of communication channels as well as the state of the processes.

Distributed deadlock detection: A distributed deadlock occurs when each of a collection of processes waits for another process to send it a message, and where there is a cycle in the graph of this ‘waits-for’ relationship. Figure 14.8(b) shows that processes p_1 and p_2 are each waiting for a message from the other, so this system will never make progress.

Distributed termination detection: The problem here is how to detect that a distributed algorithm has terminated. Detecting termination is a problem that sounds deceptively easy to solve: it seems at first only necessary to test whether each process has halted. To see that this is not so, consider a distributed algorithm executed by two processes p_1 and p_2 , each of which may request values from the other. Instantaneously, we may find that a process is either active or passive – a passive process is not engaged in any activity of its own but is prepared to respond with a value requested by the other. Suppose we discover that p_1 is passive and that p_2 is

passive (Figure 14.8c). To see that we may not conclude that the algorithm has terminated, consider the following scenario: when we tested p_1 for passivity, a message was on its way from p_2 , which became passive immediately after sending it. On receipt of the message, p_1 became active again – after we had found it to be passive. The algorithm had not terminated.

The phenomena of termination and deadlock are similar in some ways, but they are different problems. First, a deadlock may affect only a subset of the processes in a system, whereas all processes must have terminated. Second, process passivity is not the same as waiting in a deadlock cycle: a deadlocked process is attempting to perform a further action, for which another process waits; a passive process is not engaged in any activity.

Distributed debugging: Distributed systems are complex to debug [Bonnaire *et al.* 1995], and care needs to be taken in establishing what occurred during the execution. For example, suppose Smith has written an application in which each process p_i contains a variable x_i ($i = 1, 2, \dots, N$). The variables change as the program executes, but they are required always to be within a value δ of one another. Unfortunately, there is a bug in the program, and Smith suspects that under certain circumstances $|x_i - x_j| > \delta$ for some i and j , breaking her consistency constraints. Her problem is that this relationship must be evaluated for values of the variables that occur at the same time.

Each of the problems above has specific solutions tailored to it; but they all illustrate the need to observe a global state, and so motivate a general approach.

14.5.1 Global states and consistent cuts

It is possible in principle to observe the succession of states of an individual process, but the question of how to ascertain a global state of the system – the state of the collection of processes – is much harder to address.

The essential problem is the absence of global time. If all processes had perfectly synchronized clocks, then we could agree on a time at which each process would record its state – the result would be an actual global state of the system. From the collection of process states we could tell, for example, whether the processes were deadlocked. But we cannot achieve perfect clock synchronization, so this method is not available to us.

So we might ask whether we can assemble a meaningful global state from local states recorded at different real times. The answer is a qualified ‘yes’, but in order to see this we must first introduce some definitions.

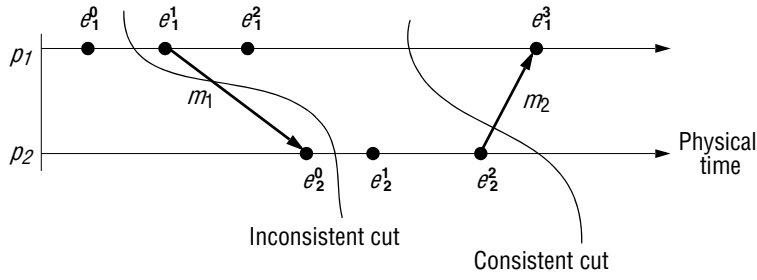
Let us return to our general system \wp of N processes p_i ($i = 1, 2, \dots, N$), whose execution we wish to study. We said above that a series of events occurs at each process, and that we may characterize the execution of each process by its history:

$$\text{history}(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \dots \rangle$$

Similarly, we may consider any finite prefix of the process’s history:

$$h_i^k = \langle e_i^0, e_i^1, \dots, e_i^k \rangle$$

Figure 14.9 Cuts



Each event either is an internal action of the process (for example, the updating of one of its variables), or is the sending or receipt of a message over the communication channels that connect the processes.

In principle, we can record what occurred in \wp 's execution. Each process can record the events that take place there, and the succession of states it passes through. We denote by s_i^k the state of process p_i immediately before the k th event occurs, so that s_i^0 is the initial state of p_i . We noted in the examples above that the state of the communication channels is sometimes relevant. Rather than introducing a new type of state, we make the processes record the sending or receipt of all messages as part of their state. If we find that process p_i has recorded that it sent a message m to process p_j ($i \neq j$), then by examining whether p_j has received that message we can infer whether or not m is part of the state of the channel between p_i and p_j .

We can also form the *global history* of \wp as the union of the individual process histories:

$$H = h_0 \cup h_1 \cup \dots \cup h_{N-1}$$

Mathematically, we can take any set of states of the individual processes to form a global state $S = (s_1, s_2, \dots, s_N)$. But which global states are meaningful – that is, which process states could have occurred at the same time? A global state corresponds to initial prefixes of the individual process histories. A *cut* of the system's execution is a subset of its global history that is a union of prefixes of process histories:

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \dots \cup h_N^{c_N}$$

The state s_i in the global state S corresponding to the cut C is that of p_i immediately after the last event processed by p_i in the cut – $e_i^{c_i}$ ($i = 1, 2, \dots, N$). The set of events $\{e_i^{c_i} : i = 1, 2, \dots, N\}$ is called the *frontier* of the cut.

Consider the events occurring at processes p_1 and p_2 shown in Figure 14.9. The figure shows two cuts, one with frontier $\langle e_1^0, e_2^0 \rangle$ and another with frontier $\langle e_1^2, e_2^2 \rangle$. The leftmost cut is *inconsistent*. This is because at p_2 it includes the receipt of the message m_1 , but at p_1 it does not include the sending of that message. This is showing an 'effect' without a 'cause'. The actual execution never was in a global state corresponding to the process states at that frontier, and we can in principle tell this by examining the \rightarrow relation between events. By contrast, the rightmost cut is *consistent*.

It includes both the sending and the receipt of message m_1 and the sending but not the receipt of message m_2 . That is consistent with the actual execution – after all, the message took some time to arrive.

A cut C is consistent if, for each event it contains, it also contains all the events that happened-before that event:

$$\text{For all events } e \in C, f \rightarrow e \Rightarrow f \in C$$

A *consistent global state* is one that corresponds to a consistent cut. We may characterize the execution of a distributed system as a series of transitions between global states of the system:

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \dots$$

In each transition, precisely one event occurs at some single process in the system. This event is either the sending of a message, the receipt of a message or an internal event. If two events happened simultaneously, we may nonetheless deem them to have occurred in a definite order – say, ordered according to process identifiers. (Events that occur simultaneously must be concurrent: neither happened-before the other.) A system evolves in this way through consistent global states.

A *run* is a total ordering of all the events in a global history that is consistent with each local history's ordering, \mathcal{O}_i ($i = 1, 2, \dots, N$). A *linearization* or *consistent run* is an ordering of the events in a global history that is consistent with this happened-before relation \rightarrow on H . Note that a linearization is also a run.

Not all runs pass through consistent global states, but all linearizations pass only through consistent global states. We say that a state S' is *reachable* from a state S if there is a linearization that passes through S and then S' .

Sometimes we may alter the ordering of concurrent events within a linearization, and derive a run that still passes through only consistent global states. For example, if two successive events in a linearization are the receipt of messages by two processes, then we may swap the order of these two events.

14.5.2 Global state predicates, stability, safety and liveness

Detecting a condition such as deadlock or termination amounts to evaluating a *global state predicate*. A global state predicate is a function that maps from the set of global states of processes in the system \wp to $\{True, False\}$. One of the useful characteristics of the predicates associated with the state of an object being garbage, of the system being deadlocked or the system being terminated is that they are all *stable*: once the system enters a state in which the predicate is *True*, it remains *True* in all future states reachable from that state. By contrast, when we monitor or debug an application we are often interested in non-stable predicates, such as that in our example of variables whose difference is supposed to be bounded. Even if the application reaches a state in which the bound obtains, it need not stay in that state.

We also note here two further notions relevant to global state predicates: safety and liveness. Suppose there is an undesirable property α that is a predicate of the system's global state – for example, α could be the property of being deadlocked. Let

S_0 be the original state of the system. *Safety* with respect to α is the assertion that α evaluates to *False* for all states S reachable from S_0 . Conversely, let β be a desirable property of a system's global state – for example, the property of reaching termination. *Liveness* with respect to β is the property that, for any linearization L starting in the state S_0 , β evaluates to *True* for some state S_L reachable from S_0 .

14.5.3 The 'snapshot' algorithm of Chandy and Lamport

Chandy and Lamport [1985] describe a 'snapshot' algorithm for determining global states of distributed systems, which we now present. The goal of the algorithm is to record a set of process and channel states (a 'snapshot') for a set of processes p_i ($i = 1, 2, \dots, N$) such that, even though the combination of recorded states may never have occurred at the same time, the recorded global state is consistent.

We shall see that the state that the snapshot algorithm records has convenient properties for evaluating stable global predicates.

The algorithm records state locally at processes; it does not give a method for gathering the global state at one site. An obvious method for gathering the state is for all processes to send the state they recorded to a designated collector process, but we shall not address this issue further here.

The algorithm assumes that:

- Neither channels nor processes fail – communication is reliable so that every message sent is eventually received intact, exactly once.
- Channels are unidirectional and provide FIFO-ordered message delivery.
- The graph of processes and channels is strongly connected (there is a path between any two processes).
- Any process may initiate a global snapshot at any time.
- The processes may continue their execution and send and receive normal messages while the snapshot takes place.

For each process p_i , let the *incoming channels* be those at p_i over which other processes send it messages; similarly, the *outgoing channels* of p_i are those on which it sends messages to other processes. The essential idea of the algorithm is as follows. Each process records its state and also, for each incoming channel, a set of messages sent to it. The process records, for each channel, any messages that arrived after it recorded its state and before the sender recorded its own state. This arrangement allows us to record the states of processes at different times but to account for the differentials between process states in terms of messages transmitted but not yet received. If process p_i has sent a message m to process p_j , but p_j has not received it, then we account for m as belonging to the state of the channel between them.

The algorithm proceeds through use of special *marker* messages, which are distinct from any other messages the processes send and which the processes may send and receive while they proceed with their normal execution. The marker has a dual role: as a prompt for the receiver to save its own state, if it has not already done so; and as a means of determining which messages to include in the channel state.

Figure 14.10 Chandy and Lamport’s ‘snapshot’ algorithm

Marker receiving rule for process p_i

On receipt of a marker message at p_i over channel c :

if (p_i has not yet recorded its state) it

records its process state now;
records the state of c as the empty set;
turns on recording of messages arriving over other incoming channels;

else

p_i records the state of c as the set of messages it has received over c since it saved its state.

end if

Marker sending rule for process p_i

After p_i has recorded its state, for each outgoing channel c :

p_i sends one marker message over c
(before it sends any other message over c).

The algorithm is defined through two rules, the *marker receiving rule* and the *marker sending rule* (Figure 14.10). The marker sending rule obligates processes to send a marker after they have recorded their state, but before they send any other messages.

The marker receiving rule obligates a process that has not recorded its state to do so. In that case, this is the first marker that it has received. It notes which messages subsequently arrive on the other incoming channels. When a process that has already saved its state receives a marker (on another channel), it records the state of that channel as the set of messages it has received on it since it saved its state.

Any process may begin the algorithm at any time. It acts as though it has received a marker (over a nonexistent channel) and follows the marker receiving rule. Thus it records its state and begins to record messages arriving over all its incoming channels. Several processes may initiate recording concurrently in this way (as long as the markers they use can be distinguished).

We illustrate the algorithm for a system of two processes, p_1 and p_2 , connected by two unidirectional channels, c_1 and c_2 . The two processes trade in ‘widgets’. Process p_1 sends orders for widgets over c_2 to p_2 , enclosing payment at the rate of \$10 per widget. Some time later, process p_2 sends widgets along channel c_1 to p_1 . The

Figure 14.11 Two processes and their initial states

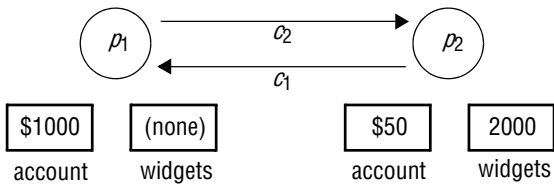
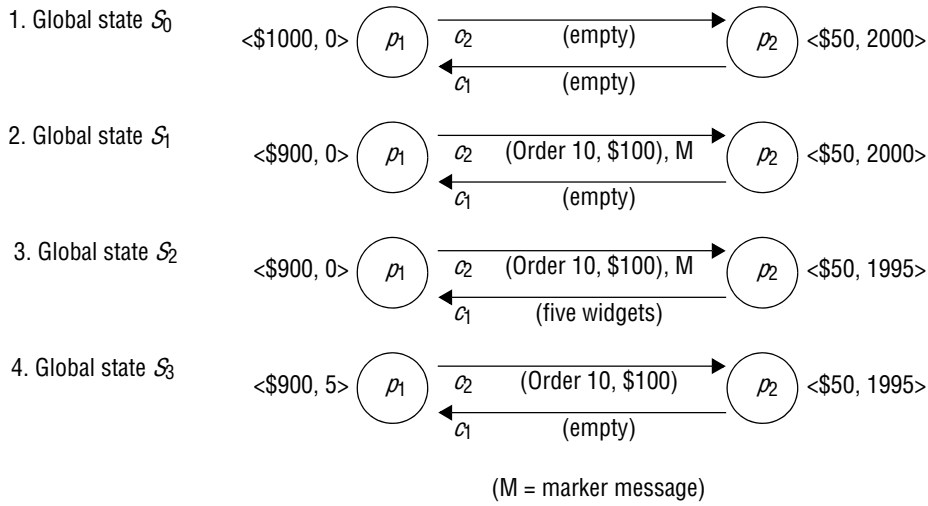


Figure 14.12 The execution of the processes in Figure 14.11

processes have the initial states shown in Figure 14.11. Process p_2 has already received an order for five widgets, which it will shortly dispatch to p_1 .

Figure 14.12 shows an execution of the system while the state is recorded. Process p_1 records its state in the actual global state S_0 , when the state of p_1 is $\langle \$1000, 0 \rangle$. Following the marker sending rule, process p_1 then emits a marker message over its outgoing channel c_2 before it sends the next application-level message: (Order 10, \$100), over channel c_2 . The system enters actual global state S_1 .

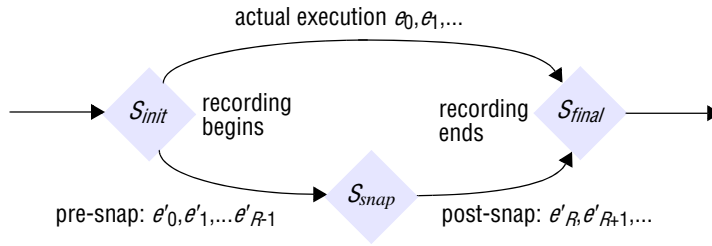
Before p_2 receives the marker, it emits an application message (five widgets) over c_1 in response to p_1 's previous order, yielding a new actual global state S_2 .

Now process p_1 receives p_2 's message (five widgets), and p_2 receives the marker. Following the marker receiving rule, p_2 records its state as $\langle \$50, 1995 \rangle$ and that of channel c_2 as the empty sequence. Following the marker sending rule, it sends a marker message over c_1 .

When process p_1 receives p_2 's marker message, it records the state of channel c_1 as the single message (five widgets) that it received after it first recorded its state. The final actual global state is S_3 .

The final recorded state is $p_1: \langle \$1000, 0 \rangle$; $p_2: \langle \$50, 1995 \rangle$; $c_1: \langle \text{(five widgets)} \rangle$; $c_2: \langle \rangle$. Note that this state differs from all the global states through which the system actually passed.

Termination of the snapshot algorithm • We assume that a process that has received a marker message records its state within a finite time and sends marker messages over each outgoing channel within a finite time (even when it no longer needs to send application messages over these channels). If there is a path of communication channels and processes from a process p_i to a process p_j ($j \neq i$), then it is clear on these assumptions that p_j will record its state a finite time after p_i recorded its state. Since we are assuming the graph of processes and channels to be strongly connected, it follows

Figure 14.13 Reachability between states in the snapshot algorithm

that all processes will have recorded their states and the states of incoming channels a finite time after some process initially records its state.

Characterizing the observed state • The snapshot algorithm selects a cut from the history of the execution. The cut, and therefore the state recorded by this algorithm, is consistent. To see this, let e_i and e_j be events occurring at p_i and p_j , respectively, such that $e_i \rightarrow e_j$. We assert that if e_j is in the cut, then e_i is in the cut. That is, if e_j occurred before p_j recorded its state, then e_i must have occurred before p_i recorded its state. This is obvious if the two processes are the same, so we shall assume that $j \neq i$. Assume, for the moment, the opposite of what we wish to prove: that p_i recorded its state before e_i occurred. Consider the sequence of H messages m_1, m_2, \dots, m_H ($H \geq 1$), giving rise to the relation $e_i \rightarrow e_j$. By FIFO ordering over the channels that these messages traverse, and by the marker sending and receiving rules, a marker message would have reached p_j ahead of each of m_1, m_2, \dots, m_H . By the marker receiving rule, p_j would therefore have recorded its state before the event e_j . This contradicts our assumption that e_j is in the cut, and we are done.

We may further establish a reachability relation between the observed global state and the initial and final global states when the algorithm runs. Let $Sys = e_0, e_1, \dots$ be the linearization of the system as it executed (where two events occurred at exactly the same time, we order them according to process identifiers). Let S_{init} be the global state immediately before the first process recorded its state; let S_{final} be the global state when the snapshot algorithm terminates, immediately after the last state-recording action; and let S_{snap} be the recorded global state.

We shall find a permutation of Sys , $Sys' = e'_0, e'_1, e'_2, \dots$ such that all three states S_{init} , S_{snap} and S_{final} occur in Sys' , S_{snap} is reachable from S_{init} in Sys' , and S_{final} is reachable from S_{snap} in Sys' . Figure 14.13 shows this situation, in which the upper linearization is Sys and the lower linearization is Sys' .

We derive Sys' from Sys by first categorizing all events in Sys as *pre-snap* events or *post-snap* events. A pre-snap event at process p_i is one that occurred at p_i before it recorded its state; all other events are post-snap events. It is important to understand that a post-snap event may occur before a pre-snap event in Sys , if the events occur at different processes. (Of course, no post-snap event may occur before a pre-snap event at the same process.)

We shall show how we may order all pre-snap events before post-snap events to obtain Sys' . Suppose that e_j is a post-snap event at one process, and e_{j+1} is a pre-snap

event at a different process. It cannot be that $e_j \rightarrow e_{j+1}$ for then these two events would be the sending and receiving of a message, respectively. A marker message would have to have preceded the message, making the reception of the message a post-snap event, but by assumption e_{j+1} is a pre-snap event. We may therefore swap the two events without violating the happened-before relation (that is, the resultant sequence of events remains a linearization). The swap does not introduce new process states, since we do not alter the order in which events occur at any individual process.

We continue swapping pairs of adjacent events in this way as necessary until we have ordered all pre-snap events $e'_0, e'_1, e'_2, \dots, e'_{R-1}$ prior to all post-snap events $e'_R, e'_{R+1}, e'_{R+2}, \dots$ with Sys' the resulting execution. For each process, the set of events in $e'_0, e'_1, e'_2, \dots, e'_{R-1}$ that occurred at it is exactly the set of events that it experienced before it recorded its state. Therefore the state of each process at that point, and the state of the communication channels, is that of the global state S_{snap} recorded by the algorithm. We have disturbed neither of the states S_{init} or S_{final} with which the linearization begins and ends. So we have established the reachability relationship.

Stability and the reachability of the observed state • The reachability property of the snapshot algorithm is useful for detecting stable predicates. In general, any non-stable predicate we establish as being *True* in the state S_{snap} may or may not have been *True* in the actual execution whose global state we recorded. However, if a stable predicate is *True* in the state S_{snap} then we may conclude that the predicate is *True* in the state S_{final} , since by definition a stable predicate that is *True* of a state S is also *True* of any state reachable from S . Similarly, if the predicate evaluates to *False* for S_{snap} , then it must also be *False* for S_{init} .

14.6 Distributed debugging

We now examine the problem of recording a system's global state so that we may make useful statements about whether a transitory state – as opposed to a stable state – occurred in an actual execution. This is what we require, in general, when debugging a distributed system. We gave an example above in which each of a set of processes p_i has a variable x_i . The safety condition required in this example is $|x_i - x_j| \leq \delta$ ($i, j = 1, 2, \dots, N$); this constraint is to be met even though a process may change the value of its variable at any time. Another example is a distributed system controlling a system of pipes in a factory where we are interested in whether all the valves (controlled by different processes) were open at some time. In these examples, we cannot in general observe the values of the variables or the states of the valves simultaneously. The challenge is to monitor the system's execution over time – to capture 'trace' information rather than a single snapshot – so that we can establish *post hoc* whether the required safety condition was or may have been violated.

Chandy and Lamport's [1985] snapshot algorithm collects state in a distributed fashion, and we pointed out how the processes in the system could send the state they gather to a monitor process for collection. The algorithm we describe next (due to Marzullo and Neiger [1991]) is centralized. The observed processes send their states to a process called a *monitor*, which assembles globally consistent states from what it receives. We consider the monitor to lie outside the system, observing its execution.

Our aim is to determine cases where a given global state predicate ϕ was definitely *True* at some point in the execution we observed, and cases where it was possibly *True*. The notion ‘possibly’ arises as a natural concept because we may extract a consistent global state S from an executing system and find that $\phi(S)$ is *True*. No single observation of a consistent global state allows us to conclude whether a non-stable predicate ever evaluated to *True* in the actual execution. Nevertheless, we may be interested to know whether they *might* have occurred, as far as we can tell by observing the execution.

The notion ‘definitely’ does apply to the actual execution and not to a run that we have extrapolated from it. It may sound paradoxical for us to consider what happened in an actual execution. However, it is possible to evaluate whether ϕ was definitely *True* by considering all linearizations of the observed events.

We now define the notions of *possibly* ϕ and *definitely* ϕ for a predicate ϕ in terms of linearizations of H , the history of the system’s execution:

possibly ϕ : The statement *possibly* ϕ means that there is a consistent global state S through which a linearization of H passes such that $\phi(S)$ is *True*.

definitely ϕ : The statement *definitely* ϕ means that for all linearizations L of H , there is a consistent global state S through which L passes such that $\phi(S)$ is *True*.

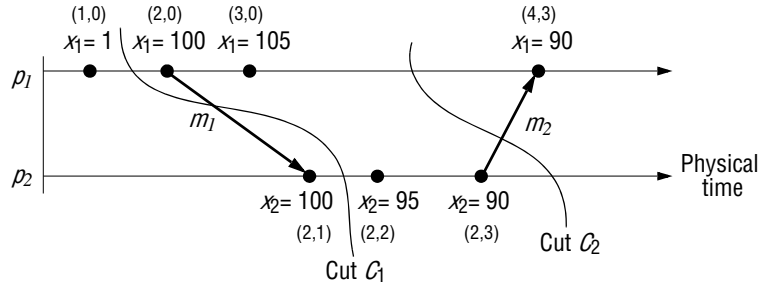
When we use Chandy and Lamport’s snapshot algorithm and obtain the global state S_{snap} , we may assert *possibly* ϕ if $\phi(S_{snap})$ happens to be *True*. But in general evaluating *possibly* ϕ entails a search through all consistent global states derived from the observed execution. Only if $\phi(S)$ evaluates to *False* for all consistent global states S is it not the case that *possibly* ϕ . Note also that while we may conclude *definitely* $(\neg\phi)$ from \neg *possibly* ϕ , we may not conclude \neg *possibly* ϕ from *definitely* $(\neg\phi)$. The latter is the assertion that $\neg\phi$ holds at some state on every linearization: ϕ may hold for other states.

We now describe: how the process states are collected; how the monitor extracts consistent global states; and how the monitor evaluates *possibly* ϕ and *definitely* ϕ in both asynchronous and synchronous systems.

14.6.1 Collecting the state

The observed processes p_i ($i = 1, 2, \dots, N$) send their initial state to the monitor initially, and thereafter from time to time, in *state messages*. The monitor records the state messages from each process p_i in a separate queue Q_i , for each $i = 1, 2, \dots, N$.

The activity of preparing and sending state messages may delay the normal execution of the observed processes, but it does not otherwise interfere with it. There is no need to send the state except initially and when it changes. There are two optimizations to reduce the state-message traffic to the monitor. First, the global state predicate may depend only on certain parts of the processes’ states – for example, only on the states of particular variables – so the observed processes need only send the relevant state to the monitor. Second, they need only send their state at times when the predicate ϕ may become *True* or cease to be *True*. There is no point in sending changes to the state that do not affect the predicate’s value.

Figure 14.14 Vector timestamps and variable values for the execution of Figure 14.9

For example, in the example system of processes p_i that are supposed to obey the constraint $|x_i - x_j| \leq \delta$, ($i, j = 1, 2, \dots, N$), each process need only notify the monitor when the values of its own variable x_i changes. When they send their state, they supply the value of x_i but do not need to send any other variables.

14.6.2 Observing consistent global states

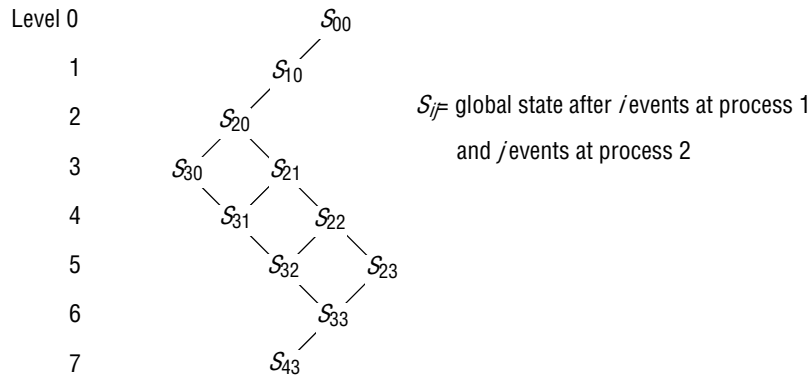
The monitor must assemble consistent global states against which it evaluates ϕ . Recall that a cut C is consistent if and only if for all events e in the cut C , $f \rightarrow e \Rightarrow f \in C$.

For example, Figure 14.14 shows two processes p_1 and p_2 with variables x_1 and x_2 , respectively. The events shown on the timelines (with vector timestamps) are adjustments to the values of the two variables. Initially, $x_1 = x_2 = 0$. The requirement is $|x_1 - x_2| \leq 50$. The processes make adjustments to their variables, but ‘large’ adjustments cause a message containing the new value to be sent to the other process. When either of the processes receives an adjustment message from the other, it sets its variable equal to the value contained in the message.

Whenever one of the processes p_1 or p_2 adjusts the value of its variable (whether it is a ‘small’ adjustment or a ‘large’ one), it sends the value in a state message to the monitor. The latter keeps the state messages in the per-process queues for analysis. If the monitor were to use values from the inconsistent cut C_1 in Figure 14.14, then it would find that $x_1 = 1, x_2 = 100$, breaking the constraint $|x_1 - x_2| \leq 50$. But this state of affairs never occurred. On the other hand, values from the consistent cut C_2 show $x_1 = 105, x_2 = 90$.

In order that the monitor can distinguish consistent global states from inconsistent global states, the observed processes enclose their vector clock values with their state messages. Each queue Q_i is kept in sending order, which can immediately be established by examining the i th component of the vector timestamps. Of course, the monitor may deduce nothing about the ordering of states sent by different processes from their arrival order, because of variable message latencies. It must instead examine the vector timestamps of the state messages.

Let $S = (s_1, s_2, \dots, s_N)$ be a global state drawn from the state messages that the monitor has received. Let $V(s_i)$ be the vector timestamp of the state s_i received from p_i . Then it can be shown that S is a consistent global state if and only if:

Figure 14.15 The lattice of global states for the execution of Figure 14.14

$$V(s_i)[i] \geq V(s_j)[i] \text{ for } i, j = 1, 2, \dots, N - (\text{Condition CGS})$$

This says that the number of p_i 's events known at p_j when it sent s_j is no more than the number of events that had occurred at p_i when it sent s_i . In other words, if one process's state depends upon another (according to happened-before ordering), then the global state also encompasses the state upon which it depends.

In summary, we now possess a method whereby the monitor may establish whether a given global state is consistent, using the vector timestamps kept by the observed processes and piggybacked on the state messages that they send to it.

Figure 14.15 shows the lattice of consistent global states corresponding to the execution of the two processes in Figure 14.14. This structure captures the relation of reachability between consistent global states. The nodes denote global states, and the edges denote possible transitions between these states. The global state S_{00} has both processes in their initial state; S_{10} has p_2 still in its initial state and p_1 in the next state in its local history. The state S_{01} is not consistent, because of the message m_1 sent from p_1 to p_2 , so it does not appear in the lattice.

The lattice is arranged in levels with, for example, S_{00} in level 0 and S_{10} in level 1. In general, S_{ij} is in level $(i + j)$. A linearization traverses the lattice from any global state to any global state reachable from it on the next level – that is, in each step some process experiences one event. For example, S_{22} is reachable from S_{20} , but S_{22} is not reachable from S_{30} .

The lattice shows us all the linearizations corresponding to a history. It is now clear in principle how a monitor should evaluate *possibly* ϕ and *definitely* ϕ . To evaluate *possibly* ϕ , the monitor starts at the initial state and steps through all consistent states reachable from that point, evaluating ϕ at each stage. It stops when ϕ evaluates to *True*. To evaluate *definitely* ϕ , the monitor must attempt to find a set of states through which all linearizations must pass, and at each of which ϕ evaluates to *True*. For example, if $\phi(S_{30})$ and $\phi(S_{21})$ in Figure 14.15 are both *True* then, since all linearizations pass through these states, *definitely* ϕ holds.

Figure 14.16 Algorithms to evaluate *possibly* ϕ and *definitely* ϕ

```

1. Evaluating possibly  $\phi$  for global history  $H$  of  $N$  processes
    $L := 0$ ;
    $States := \{ (s_1^0, s_2^0, \dots, s_N^0) \}$ ;
   while ( $\phi(S) = False$  for all  $S \in States$ )
      $L := L + 1$ ;
      $Reachable := \{ S' : S' \text{ reachable in } H \text{ from some } S \in States \wedge level(S') = L \}$ ;
      $States := Reachable$ 
   end while
   output "possibly  $\phi$ ";

2. Evaluating definitely  $\phi$  for global history  $H$  of  $N$  processes
    $L := 0$ ;
   if ( $\phi(s_1^0, s_2^0, \dots, s_N^0)$ ) then  $States := \{\}$  else  $States := \{ (s_1^0, s_2^0, \dots, s_N^0) \}$ ;
   while ( $States \neq \{\}$ )
      $L := L + 1$ ;
      $Reachable := \{ S' : S' \text{ reachable in } H \text{ from some } S \in States \wedge level(S') = L \}$ ;
      $States := \{ S \in Reachable : \phi(S) = False \}$ 
   end while
   output "definitely  $\phi$ ";

```

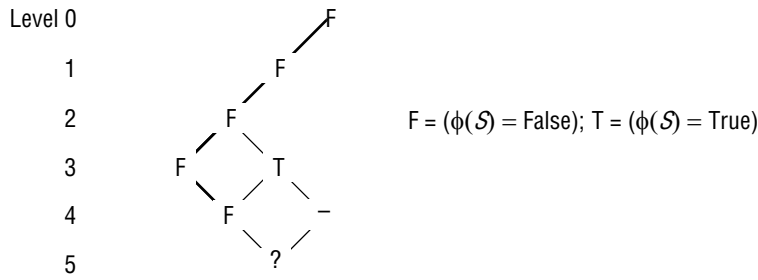
14.6.3 Evaluating *possibly* ϕ

To evaluate *possibly* ϕ , the monitor must traverse the lattice of reachable states, starting from the initial state $(s_1^0, s_2^0, \dots, s_N^0)$. The algorithm is shown in Figure 14.16. The algorithm assumes that the execution is infinite. It may easily be adapted for a finite execution.

The monitor may discover the set of consistent states in level $L + 1$ reachable from a given consistent state in level L by the following method. Let $S = (s_1, s_2, \dots, s_N)$ be a consistent state. Then a consistent state in the next level reachable from S is of the form $S' = (s_1, s_2, \dots, s'_i, \dots, s_N)$, which differs from S only by containing the next state (after a single event) of some process p_i . The monitor can find all such states by traversing the queues of state messages Q_i ($i = 1, 2, \dots, N$). The state S' is reachable from S if and only if:

$$\text{for } j = 1, 2, \dots, N, j \neq i: V(s_j)[j] \geq V(s'_i)[j]$$

This condition comes from condition CGS above and from the fact that S was already a consistent global state. A given state may in general be reached from several states at the previous level, so the monitor should take care to evaluate the consistency of each state only once.

Figure 14.17 Evaluating *definitely* ϕ 

14.6.4 Evaluating *definitely* ϕ

To evaluate *definitely* ϕ , the monitor again traverses the lattice of reachable states a level at a time, starting from the initial state $(s_1^0, s_2^0, \dots, s_N^0)$. The algorithm (shown in Figure 14.16) again assumes that the execution is infinite but may easily be adapted for a finite execution. It maintains the set *States*, which contains those states at the current level that may be reached on a linearization from the initial state by traversing only states for which ϕ evaluates to *False*. As long as such a linearization exists, we may not assert *definitely* ϕ : the execution could have taken this linearization, and ϕ would be *False* at every stage along it. If we reach a level for which no such linearization exists, we may conclude *definitely* ϕ .

In Figure 14.17, at level 3 the set *States* consists of only one state, which is reachable by a linearization on which all states are *False* (marked in bold lines). The only state considered at level 4 is the one marked 'F'. (The state to its right is not considered, since it can only be reached via a state for which ϕ evaluates to *True*.) If ϕ evaluates to *True* in the state at level 5, then we may conclude *definitely* ϕ . Otherwise, the algorithm must continue beyond this level.

Cost • The algorithms we have just described are combinatorially explosive. Suppose that k is the maximum number of events at a single process. Then the algorithms we have described entail $O(k^N)$ comparisons (the monitor compares the states of each of the N observed processes with one another).

There is also a space cost to these algorithms of $O(kN)$. However, we observe that the monitor may delete a message containing state s_i from queue Q_i when no other item of state arriving from another process could possibly be involved in a consistent global state containing s_i . That is, when:

$$V(s_j^{last})[i] > V(s_i)[i] \text{ for } j = 1, 2, \dots, N, j \neq i$$

where s_j^{last} is the last state that the monitor has received from process p_j .

14.6.5 Evaluating possibly ϕ and definitely ϕ in synchronous systems

The algorithms we have given so far work in an asynchronous system: we have made no timing assumptions. But the price paid for this is that the monitor may examine a consistent global state $S = (s_1, s_2, \dots, s_N)$ for which any two local states s_i and s_j occurred an arbitrarily long time apart in the actual execution of the system. Our requirement, by contrast, is to consider only those global states that the actual execution could in principle have traversed.

In a synchronous system, suppose that the processes keep their physical clocks internally synchronized within a known bound, and that the observed processes provide physical timestamps as well as vector timestamps in their state messages. Then the monitor need consider only those consistent global states whose local states could possibly have existed simultaneously, given the approximate synchronization of the clocks. With good enough clock synchronization, these will number many less than all globally consistent states.

We now give an algorithm to exploit synchronized clocks in this way. We assume that each observed process p_i ($i = 1, 2, \dots, N$) and the monitor, which we shall call p_0 , keep a physical clock C_i ($i = 0, 1, \dots, N$). These are synchronized to within a known bound $D > 0$; that is, at the same real time:

$$|C_i(t) - C_j(t)| < D \text{ for } i, j = 0, 1, \dots, N$$

The observed processes send both their vector time and physical time with their state messages to the monitor. The monitor now applies a condition that not only tests for consistency of a global state $S = (s_1, s_2, \dots, s_N)$, but also tests whether each pair of states could have happened at the same real time, given the physical clock values. In other words, for $i, j = 1, 2, \dots, N$:

$$V(s_i)[i] \geq V(s_j)[i] \text{ and } s_i \text{ and } s_j \text{ could have occurred at the same real time.}$$

The first clause is the condition that we used earlier. For the second clause, note that p_i is in the state s_i from the time it first notifies the monitor, $C_i(s_i)$, to some later local time $L_i(s_i)$ – say, when the next state transition occurs at p_i . For s_i and s_j to have obtained at the same real time we thus have, allowing for the bound on clock synchronization:

$$C_i(s_i) - D \leq C_j(s_j) \leq L_i(s_i) + D \text{ – or vice versa (swapping } i \text{ and } j).$$

The monitor must calculate a value for $L_i(s_i)$, which is measured against p_i 's clock. If the monitor has received a state message for p_i 's next state s'_i , then $L_i(s_i)$ is $C_i(s'_i)$. Otherwise, the monitor estimates $L_i(s_i)$ as $C_0 - \text{max} + D$, where C_0 is the monitor's current local clock value and max is the maximum transmission time for a state message.

14.7 Summary

This chapter began by describing the importance of accurate timekeeping for distributed systems. It then described algorithms for synchronizing clocks despite the drift between them and the variability of message delays between computers.

The degree of synchronization accuracy that is practically obtainable fulfils many requirements but is nonetheless not sufficient to determine the ordering of an arbitrary pair of events occurring at different computers. The happened-before relation is a partial order on events that reflects a flow of information between them – within a process, or via messages between processes. Some algorithms require events to be ordered in happened-before order, for example, successive updates made to separate copies of data. Lamport clocks are counters that are updated in accordance with the happened-before relationship between events. Vector clocks are an improvement on Lamport clocks, in that it is possible to determine by examining their vector timestamps whether two events are ordered by happened-before or are concurrent.

We introduced the concepts of events, local and global histories, cuts, local and global states, runs, consistent states, linearizations (consistent runs) and reachability. A consistent state or run is one that is in accord with the happened-before relation.

We went on to consider the problem of recording a consistent global state by observing a system's execution. Our objective was to evaluate a predicate on this state. An important class of predicates are the stable predicates. We described the snapshot algorithm of Chandy and Lamport, which captures a consistent global state and allows us to make assertions about whether a stable predicate holds in the actual execution. We went on to give Marzullo and Neiger's algorithm for deriving assertions about whether a predicate held or may have held in the actual run. This algorithm employs a monitor process to collect states. The monitor examines vector timestamps to extract consistent global states, and it constructs and examines the lattice of all consistent global states. This algorithm involves great computational complexity but is valuable for understanding and can be of some practical benefit in real systems where relatively few events change the global predicate's value. The algorithm has a more efficient variant in synchronous systems, where clocks may be synchronized.

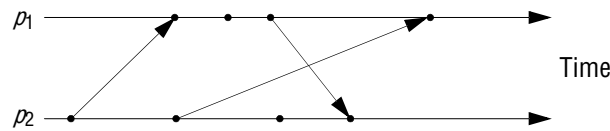
EXERCISES

- 14.1 Why is computer clock synchronization necessary? Describe the design requirements for a system to synchronize the clocks in a distributed system. *page 596*
- 14.2 A clock is reading 10:27:54.0 (hr:min:sec) when it is discovered to be 4 seconds fast. Explain why it is undesirable to set it back to the right time at that point and show (numerically) how it should be adjusted so as to be correct after 8 seconds have elapsed. *page 600*
- 14.3 A scheme for implementing *at-most-once* reliable message delivery uses synchronized clocks to reject duplicate messages. Processes place their local clock value (a ‘timestamp’) in the messages they send. Each receiver keeps a table giving, for each sending process, the largest message timestamp it has seen. Assume that clocks are synchronized to within 100 ms, and that messages can arrive at most 50 ms after transmission.
- When may a process ignore a message bearing a timestamp T , if it has recorded the last message received from that process as having timestamp T' ?
 - When may a receiver remove a timestamp 175,000 (ms) from its table? (Hint: use the receiver’s local clock value.)
 - Should the clocks be internally synchronized or externally synchronized?
- page 601*
- 14.4 A client attempts to synchronize with a time server. It records the round-trip times and timestamps returned by the server in the table below.
- Which of these times should it use to set its clock? To what time should it set it? Estimate the accuracy of the setting with respect to the server’s clock. If it is known that the time between sending and receiving a message in the system concerned is at least 8 ms, do your answers change?

Round-trip (ms)	Time (hr:min:sec)
22	10:54:23.674
25	10:54:25.450
20	10:54:28.342

- page 601*
- 14.5 In the system of Exercise 14.4 it is required to synchronize a file server’s clock to within ± 1 millisecond. Discuss this in relation to Cristian’s algorithm. *page 601*
- 14.6 What reconfigurations would you expect to occur in the NTP synchronization subnet? *page 604*
- 14.7 An NTP server B receives server A’s message at 16:34:23.480, bearing a timestamp of 16:34:13.430, and replies to it. A receives the message at 16:34:15.725, bearing B’s timestamp, 16:34:25.7. Estimate the offset between B and A and the accuracy of the estimate. *page 605*

- 14.8 Discuss the factors to be taken into account when deciding to which NTP server a client should synchronize its clock. page 606
- 14.9 Discuss how it is possible to compensate for clock drift between synchronization points by observing the drift rate over time. Discuss any limitations to your method. page 607
- 14.10 By considering a chain of zero or more messages connecting events e and e' and using induction, show that $e \rightarrow e' \Rightarrow L(e) < L(e')$. page 608
- 14.11 Show that $V_j[i] \leq V_i[i]$. page 609
- 14.12 In a similar fashion to Exercise 14.10, show that $e \rightarrow e' \Rightarrow V(e) < V(e')$. page 610
- 14.13 Using the result of Exercise 14.11, show that if events e and e' are concurrent then neither $V(e) \leq V(e')$ nor $V(e') \leq V(e)$. Hence show that if $V(e) < V(e')$ then $e \rightarrow e'$. page 610
- 14.14 Two processes P and Q are connected in a ring using two channels, and they constantly rotate a message m . At any one time, there is only one copy of m in the system. Each process's state consists of the number of times it has received m , and P sends m first. At a certain point, P has the message and its state is 101. Immediately after sending m , P initiates the snapshot algorithm. Explain the operation of the algorithm in this case, giving the possible global state(s) reported by it. page 615



- 14.15 The figure above shows events occurring for each of two processes, p_1 and p_2 . Arrows between processes denote message transmission.
Draw and label the lattice of consistent states (p_1 state, p_2 state), beginning with the initial state $(0,0)$. page 622
- 14.16 Jones is running a collection of processes p_1, p_2, \dots, p_N . Each process p_i contains a variable v_i . She wishes to determine whether all the variables v_1, v_2, \dots, v_N were ever equal in the course of the execution.
- Jones' processes run in a synchronous system. She uses a monitor process to determine whether the variables were ever equal. When should the application processes communicate with the monitor process, and what should their messages contain?
 - Explain the statement *possibly* ($v_1 = v_2 = \dots = v_N$). How can Jones determine whether this statement is true of her execution? page 623

COORDINATION AND AGREEMENT

- 15.1 Introduction
- 15.2 Distributed mutual exclusion
- 15.3 Elections
- 15.4 Coordination and agreement in group communication
- 15.5 Consensus and related problems
- 15.6 Summary

In this chapter, we introduce some topics and algorithms related to the issue of how processes coordinate their actions and agree on shared values in distributed systems, despite failures. The chapter begins with algorithms to achieve mutual exclusion among a collection of processes, so as to coordinate their accesses to shared resources. It goes on to examine how an election can be implemented in a distributed system – that is, how a group of processes can agree on a new coordinator of their activities after the previous coordinator has failed.

The second half of the chapter examines the related problems of group communication, consensus, Byzantine agreement and interactive consistency. In the context of group communication, the issue is how to agree on such matters as the order in which messages are to be delivered. Consensus and the other problems generalize from this: how can any collection of processes agree on some value, no matter what the domain of the values in question? We encounter a fundamental result in the theory of distributed systems: that under certain conditions – including surprisingly benign failure conditions – it is impossible to guarantee that processes will reach consensus.

15.1 Introduction

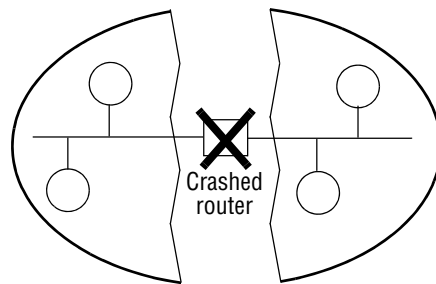
This chapter introduces a collection of algorithms whose goals vary but that share an aim that is fundamental in distributed systems: for a set of processes to coordinate their actions or to agree on one or more values. For example, in the case of a complex piece of machinery such as a spaceship, it is essential that the computers controlling it agree on such conditions as whether the spaceship's mission is proceeding or has been aborted. Furthermore, the computers must coordinate their actions correctly with respect to shared resources (the spaceship's sensors and actuators). The computers must be able to do so even where there is no fixed master-slave relationship between the components (which would make coordination particularly simple). The reason for avoiding fixed master-slave relationships is that we often require our systems to keep working correctly even if failures occur, so we need to avoid single points of failure, such as fixed masters.

An important distinction for us, as in Chapter 14, will be whether the distributed system under study is asynchronous or synchronous. In an asynchronous system we can make no timing assumptions. In a synchronous system, we shall assume that there are bounds on the maximum message transmission delay, on the time taken to execute each step of a process, and on clock drift rates. The synchronous assumptions allow us to use timeouts to detect process crashes.

Another important aim of the chapter is to consider failures, and how to deal with them when designing algorithms. Section 2.4.2 introduced a failure model, which we shall use in this chapter. Coping with failures is a subtle business, so we begin by considering some algorithms that tolerate no failures and progress through benign failures before exploring how to tolerate arbitrary failures. Along the way, we encounter a fundamental result in the theory of distributed systems: even under surprisingly benign failure conditions, it is impossible to guarantee in an asynchronous system that a collection of processes can agree on a shared value – for example, for all of a spaceship's controlling processes to agree 'mission proceed' or 'mission abort'.

Section 15.2 examines the problem of distributed mutual exclusion. This is the extension to distributed systems of the familiar problem of avoiding race conditions in kernels and multi-threaded applications. Since much of what occurs in distributed systems is resource sharing, this is an important problem to solve. Next, Section 15.3 introduces the related but more general issue of how to 'elect' one of a collection of processes to perform a special role. For example, in Chapter 14 we saw how processes synchronize their clocks to a designated time server. If this server fails and several surviving servers can fulfil that role, then for the sake of consistency it is necessary to choose just one server to take over.

Coordination and agreement related to group communication is the subject of Section 15.4. As Section 4.4.1 explained, the ability to multicast a message to a group is a very useful communication paradigm, with applications from locating resources to coordinating the updates to replicated data. Section 15.4 examines multicast reliability and ordering semantics, and gives algorithms to achieve the variations. Multicast delivery is essentially a problem of agreement between processes: the recipients agree on which messages they will receive, and in which order they will receive them. Section 15.5 discusses the problem of agreement more generally, primarily in the forms known as consensus and Byzantine agreement.

Figure 15.1 A network partition

The treatment followed in this chapter involves stating the assumptions and the goals to be met, and giving an informal account of why the algorithms presented are correct. There is insufficient space to provide a more rigorous approach. For that, we refer the reader to a text that gives a thorough account of distributed algorithms, such as Attiya and Welch [1998] and Lynch [1996].

Before presenting the problems and algorithms, we discuss failure assumptions and the practical matter of detecting failures in distributed systems.

15.1.1 Failure assumptions and failure detectors

For the sake of simplicity, this chapter assumes that each pair of processes is connected by reliable channels. That is, although the underlying network components may suffer failures, the processes use a reliable communication protocol that masks these failures – for example, by retransmitting missing or corrupted messages. Also for the sake of simplicity, we assume that no process failure implies a threat to the other processes' ability to communicate. This means that none of the processes depends upon another to forward messages.

Note that a reliable channel *eventually* delivers a message to the recipient's input buffer. In a synchronous system, we suppose that there is hardware redundancy where necessary, so that a reliable channel not only eventually delivers each message despite underlying failures, but does so within a specified time bound.

In any particular interval of time, communication between some processes may succeed while communication between others is delayed. For example, the failure of a router between two networks may mean that a collection of four processes is split into two pairs, such that intra-pair communication is possible over their respective networks; but inter-pair communication is not possible while the router has failed. This is known as a *network partition* (Figure 15.1). Over a point-to-point network such as the Internet, complex topologies and independent routing choices mean that connectivity may be *asymmetric*: communication is possible from process p to process q , but not vice versa. Connectivity may also be *intransitive*: communication is possible from p to q and from q to r , but p cannot communicate directly with r . Thus our reliability assumption entails that eventually any failed link or router will be repaired or circumvented. Nevertheless, the processes may not all be able to communicate at the same time.

The chapter assumes, unless we state otherwise, that processes fail only by crashing – an assumption that is good enough for many systems. In Section 15.5, we shall consider how to treat the cases where processes have arbitrary (Byzantine) failures. Whatever the type of failure, a *correct* process is one that exhibits no failures at any point in the execution under consideration. Note that correctness applies to the whole execution, not just to a part of it. So a process that suffers a crash failure is ‘non-failed’ before that point, not ‘correct’ before that point.

One of the problems in the design of algorithms that can overcome process crashes is that of deciding when a process has crashed. A *failure detector* [Chandra and Toueg 1996, Stelling *et al.* 1998] is a service that processes queries about whether a particular process has failed. It is often implemented by an object local to each process (on the same computer) that runs a failure-detection algorithm in conjunction with its counterparts at other processes. The object local to each process is called a *local failure detector*. We outline how to implement failure detectors shortly, but first we concentrate on some of the properties of failure detectors.

A failure ‘detector’ is not necessarily accurate. Most fall into the category of *unreliable failure detectors*. An unreliable failure detector may produce one of two values when given the identity of a process: *Unsuspected* or *Suspected*. Both of these results are hints, which may or may not accurately reflect whether the process has actually failed. A result of *Unsuspected* signifies that the detector has recently received evidence suggesting that the process has not failed; for example, a message was recently received from it. But of course, the process may have failed since then. A result of *Suspected* signifies that the failure detector has some indication that the process may have failed. For example, it may be that no message from the process has been received for more than a nominal maximum length of silence (even in an asynchronous system, practical upper bounds can be used as hints). The suspicion may be misplaced: for example, the process could be functioning correctly but be on the other side of a network partition, or it could be running more slowly than expected.

A *reliable failure detector* is one that is always accurate in detecting a process’s failure. It answers processes’ queries with either a response of *Unsuspected* – which, as before, can only be a hint – or *Failed*. A result of *Failed* means that the detector has determined that the process has crashed. Recall that a process that has crashed stays that way, since by definition a process never takes another step once it has crashed.

It is important to realize that, although we speak of one failure detector acting for a collection of processes, the response that the failure detector gives to a process is only as good as the information available at that process. A failure detector may sometimes give different responses to different processes, since communication conditions vary from process to process.

We can implement an unreliable failure detector using the following algorithm. Each process p sends a ‘ p is here’ message to every other process, and it does this every T seconds. The failure detector uses an estimate of the maximum message transmission time of D seconds. If the local failure detector at process q does not receive a ‘ p is here’ message within $T + D$ seconds of the last one, then it reports to q that p is *Suspected*. However, if it subsequently receives a ‘ p is here’ message, then it reports to q that p is *OK*.

In a real distributed system, there are practical limits on message transmission times. Even email systems give up after a few days, since it is likely that communication

links and routers will have been repaired in that time. If we choose small values for T and D (so that they total 0.1 second, say), then the failure detector is likely to suspect non-crashed processes many times, and much bandwidth will be taken up with ‘ p is here’ messages. If we choose a large total timeout value (a week, say), then crashed processes will often be reported as *Unsuspected*.

A practical solution to this problem is to use timeout values that reflect the observed network delay conditions. If a local failure detector receives a ‘ p is here’ in 20 seconds instead of the expected maximum of 10 seconds, it can reset its timeout value for p accordingly. The failure detector remains unreliable, and its answers to queries are still only hints, but the probability of its accuracy increases.

In a synchronous system, our failure detector can be made into a reliable one. We can choose D so that it is not an estimate but an absolute bound on message transmission times; the absence of a ‘ p is here’ message within $T + D$ seconds entitles the local failure detector to conclude that p has crashed.

The reader may wonder whether failure detectors are of any practical use. Unreliable failure detectors may suspect a process that has not failed (they may be *inaccurate*), and they may not suspect a process that has in fact failed (they may be *incomplete*). Reliable failure detectors, on the other hand, require that the system is synchronous (and few practical systems are).

We have introduced failure detectors because they help us to think about the nature of failures in a distributed system. And any practical system that is designed to cope with failures must detect them – however imperfectly. But it turns out that even unreliable failure detectors with certain well-defined properties can help us to provide practical solutions to the problem of coordinating processes in the presence of failures. We return to this point in Section 15.5.

15.2 Distributed mutual exclusion

Distributed processes often need to coordinate their activities. If a collection of processes share a resource or collection of resources, then often mutual exclusion is required to prevent interference and ensure consistency when accessing the resources. This is the *critical section* problem, familiar in the domain of operating systems. In a distributed system, however, neither shared variables nor facilities supplied by a single local kernel can be used to solve it, in general. We require a solution to *distributed mutual exclusion*: one that is based solely on message passing.

In some cases shared resources are managed by servers that also provide mechanisms for mutual exclusion – Chapter 16 describes how some servers synchronize client accesses to resources. But in some practical cases, a separate mechanism for mutual exclusion is required.

Consider users who update a text file. A simple means of ensuring that their updates are consistent is to allow them to access it only one at a time, by requiring the editor to lock the file before updates can be made. NFS file servers, described in Chapter 12, are designed to be stateless and therefore do not support file locking. For this reason, UNIX systems provide a separate file-locking service, implemented by the daemon *lockd*, to handle locking requests from clients.

A particularly interesting example is where there is no server, and a collection of peer processes must coordinate their accesses to shared resources amongst themselves. This occurs routinely on networks such as Ethernets and IEEE 802.11 wireless networks in ‘ad hoc’ mode, where network interfaces cooperate as peers so that only one node transmits at a time on the shared medium. Consider, also, a system monitoring the number of vacancies in a car park with a process at each entrance and exit that tracks the number of vehicles entering and leaving. Each process keeps a count of the total number of vehicles within the car park and displays whether or not it is full. The processes must update the shared count of the number of vehicles consistently. There are several ways of achieving that, but it would be convenient for these processes to be able to obtain mutual exclusion solely by communicating among themselves, eliminating the need for a separate server.

It is useful to have a generic mechanism for distributed mutual exclusion at our disposal – one that is independent of the particular resource management scheme in question. We now examine some algorithms for achieving that.

15.2.1 Algorithms for mutual exclusion

We consider a system of N processes $p_i, i = 1, 2, \dots, N$, that do not share variables. The processes access common resources, but they do so in a critical section. For the sake of simplicity, we assume that there is only one critical section. It is straightforward to extend the algorithms we present to more than one critical section.

We assume that the system is asynchronous, that processes do not fail and that message delivery is reliable, so that any message sent is eventually delivered intact, exactly once.

The application-level protocol for executing a critical section is as follows:

```

enter()           // enter critical section – block if necessary
resourceAccesses() // access shared resources in critical section
exit()           // leave critical section – other processes may now enter

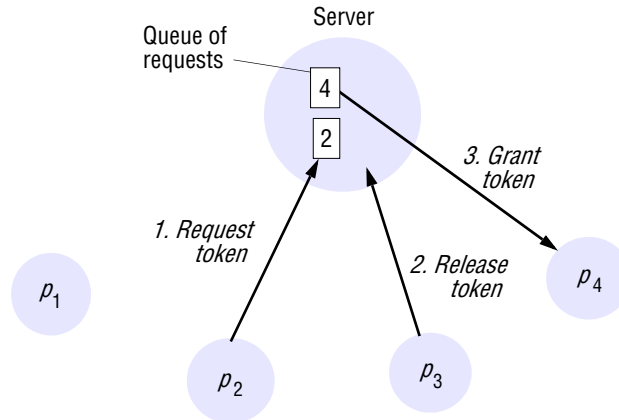
```

Our essential requirements for mutual exclusion are as follows:

- ME1: (safety) At most one process may execute in the critical section (CS) at a time.
- ME2: (liveness) Requests to enter and exit the critical section eventually succeed.

Condition ME2 implies freedom from both deadlock and starvation. A deadlock would involve two or more of the processes becoming stuck indefinitely while attempting to enter or exit the critical section, by virtue of their mutual interdependence. But even without a deadlock, a poor algorithm might lead to *starvation*: the indefinite postponement of entry for a process that has requested it.

The absence of starvation is a *fairness* condition. Another fairness issue is the order in which processes enter the critical section. It is not possible to order entry to the critical section by the times that the processes requested it, because of the absence of global clocks. But a useful fairness requirement that is sometimes made makes use of the happened-before ordering (Section 14.4) between messages that request entry to the critical section:

Figure 15.2 Server managing a mutual exclusion token for a set of processes

ME3: (\rightarrow ordering) If one request to enter the CS happened-before another, then entry to the CS is granted in that order.

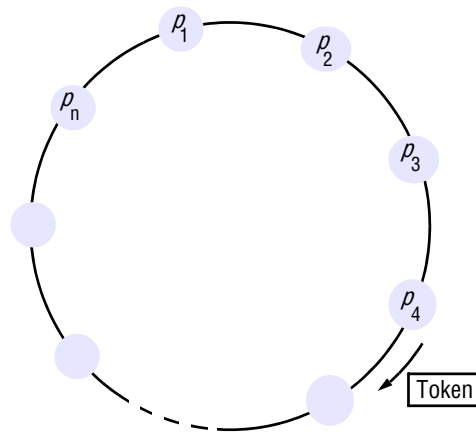
If a solution grants entry to the critical section in happened-before order, and if all requests are related by happened-before, then it is not possible for a process to enter the critical section more than once while another waits to enter. This ordering also allows processes to coordinate their accesses to the critical section. A multi-threaded process may continue with other processing while a thread waits to be granted entry to a critical section. During this time, it might send a message to another process, which consequently also tries to enter the critical section. ME3 specifies that the first process be granted access before the second.

We evaluate the performance of algorithms for mutual exclusion according to the following criteria:

- the *bandwidth* consumed, which is proportional to the number of messages sent in each *entry* and *exit* operation;
- the *client delay* incurred by a process at each *entry* and *exit* operation;
- the algorithm's effect upon the *throughput* of the system. This is the rate at which the collection of processes as a whole can access the critical section, given that some communication is necessary between successive processes. We measure the effect using the *synchronization delay* between one process exiting the critical section and the next process entering it; the throughput is greater when the synchronization delay is shorter.

We do not take the implementation of resource accesses into account in our descriptions. We do, however, assume that the client processes are well behaved and spend a finite time accessing resources within their critical sections.

The central server algorithm • The simplest way to achieve mutual exclusion is to employ a server that grants permission to enter the critical section. Figure 15.2 shows the use of this server. To enter a critical section, a process sends a request message to

Figure 15.3 A ring of processes transferring a mutual exclusion token

the server and awaits a reply from it. Conceptually, the reply constitutes a token signifying permission to enter the critical section. If no other process has the token at the time of the request, then the server replies immediately, granting the token. If the token is currently held by another process, then the server does not reply, but queues the request. When a process exits the critical section, it sends a message to the server, giving it back the token.

If the queue of waiting processes is not empty, then the server chooses the oldest entry in the queue, removes it and replies to the corresponding process. The chosen process then holds the token. In the figure, we show a situation in which p_2 's request has been appended to the queue, which already contained p_4 's request. p_3 exits the critical section, and the server removes p_4 's entry and grants permission to enter to p_4 by replying to it. Process p_1 does not currently require entry to the critical section.

Given our assumption that no failures occur, it is easy to see that the safety and liveness conditions are met by this algorithm. The reader should verify, however, that the algorithm does not satisfy property ME3.

We now evaluate the performance of this algorithm. Entering the critical section – even when no process currently occupies it – takes two messages (a *request* followed by a *grant*) and delays the requesting process by the time required for this round-trip. Exiting the critical section takes one *release* message. Assuming asynchronous message passing, this does not delay the exiting process.

The server may become a performance bottleneck for the system as a whole. The synchronization delay is the time taken for a round-trip: a *release* message to the server, followed by a *grant* message to the next process to enter the critical section.

A ring-based algorithm • One of the simplest ways to arrange mutual exclusion between the N processes without requiring an additional process is to arrange them in a logical ring. This requires only that each process p_i has a communication channel to the next process in the ring, $p_{(i+1) \bmod N}$. The idea is that exclusion is conferred by obtaining a token in the form of a message passed from process to process in a single direction –

Figure 15.4 Ricart and Agrawala's algorithm

```

On initialization
  state := RELEASED;

To enter the section
  state := WANTED;
  Multicast request to all processes;
  T := request's timestamp;
  Wait until (number of replies received = (N - 1));
  state := HELD;
} Request processing deferred here

On receipt of a request <Ti, pi> at pj (i ≠ j)
  if (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))
  then
    queue request from pi without replying;
  else
    reply immediately to pi;
  end if

To exit the critical section
  state := RELEASED;
  reply to any queued requests;

```

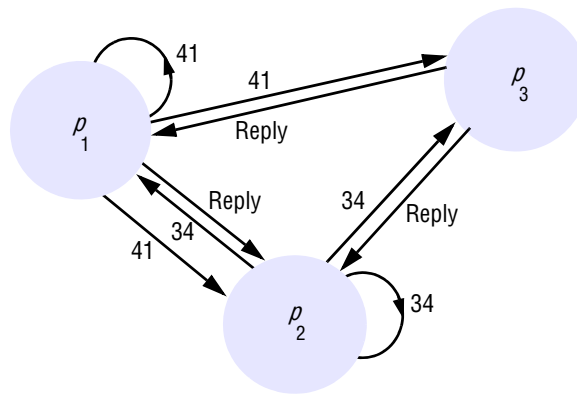
clockwise, say – around the ring. The ring topology may be unrelated to the physical interconnections between the underlying computers.

If a process does not require to enter the critical section when it receives the token, then it immediately forwards the token to its neighbour. A process that requires the token waits until it receives it, but retains it. To exit the critical section, the process sends the token on to its neighbour.

The arrangement of processes is shown in Figure 15.3. It is straightforward to verify that the conditions ME1 and ME2 are met by this algorithm, but that the token is not necessarily obtained in happened-before order. (Recall that the processes may exchange messages independently of the rotation of the token.)

This algorithm continuously consumes network bandwidth (except when a process is inside the critical section): the processes send messages around the ring even when no process requires entry to the critical section. The delay experienced by a process requesting entry to the critical section is between 0 messages (when it has just received the token) and N messages (when it has just passed on the token). To exit the critical section requires only one message. The synchronization delay between one process's exit from the critical section and the next process's entry is anywhere from 1 to N message transmissions.

An algorithm using multicast and logical clocks • Ricart and Agrawala [1981] developed an algorithm to implement mutual exclusion between N peer processes that is based upon multicast. The basic idea is that processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have

Figure 15.5 Multicast synchronization

replied to this message. The conditions under which a process replies to a request are designed to ensure that conditions ME1–ME3 are met.

The processes p_1, p_2, \dots, p_N bear distinct numeric identifiers. They are assumed to possess communication channels to one another, and each process p_i keeps a Lamport clock, updated according to the rules LC1 and LC2 of Section 14.4. Messages requesting entry are of the form $\langle T, p_i \rangle$, where T is the sender's timestamp and p_i is the sender's identifier.

Each process records its state of being outside the critical section (*RELEASED*), wanting entry (*WANTED*) or being in the critical section (*HELD*) in a variable *state*. The protocol is given in Figure 15.4.

If a process requests entry and the state of all other processes is *RELEASED*, then all processes will reply immediately to the request and the requester will obtain entry. If some process is in the state *HELD*, then that process will not reply to requests until it has finished with the critical section, and so the requester cannot gain entry in the meantime. If two or more processes request entry at the same time, then whichever process's request bears the lowest timestamp will be the first to collect $N - 1$ replies, granting it entry next. If the requests bear equal Lamport timestamps, the requests are ordered according to the processes' corresponding identifiers. Note that, when a process requests entry, it defers processing requests from other processes until its own request has been sent and it has recorded the timestamp T of the request. This is so that processes make consistent decisions when processing requests.

This algorithm achieves the safety property ME1. If it were possible for two processes p_i and p_j ($i \neq j$) to enter the critical section at the same time, then both of those processes would have to have replied to the other. But since the pairs $\langle T_i, p_i \rangle$ are totally ordered, this is impossible. We leave the reader to verify that the algorithm also meets requirements ME2 and ME3.

To illustrate the algorithm, consider a situation involving three processes, p_1, p_2 and p_3 , shown in Figure 15.5. Let us assume that p_3 is not interested in entering the critical section, and that p_1 and p_2 request entry concurrently. The timestamp of p_1 's request is 41, and that of p_2 is 34. When p_3 receives their requests, it replies

immediately. When p_2 receives p_1 's request, it finds that its own request has the lower timestamp and so does not reply, holding p_1 off. However, p_1 finds that p_2 's request has a lower timestamp than that of its own request and so replies immediately. On receiving this second reply, p_2 can enter the critical section. When p_2 exits the critical section, it will reply to p_1 's request and so grant it entry.

Gaining entry takes $2(N-1)$ messages in this algorithm: $N-1$ to multicast the request, followed by $N-1$ replies. Or, if there is hardware support for multicast, only one message is required for the request; the total is then N messages. It is thus a more expensive algorithm, in terms of bandwidth consumption, than the algorithms just described. However, the client delay in requesting entry is again a round-trip time (ignoring any delay incurred in multicasting the request message).

The advantage of this algorithm is that its synchronization delay is only one message transmission time. Both the previous algorithms incurred a round-trip synchronization delay.

The performance of the algorithm can be improved. First, note that the process that last entered the critical section and that has received no other requests for it still goes through the protocol as described, even though it could simply decide locally to reenter the critical section. Second, Ricart and Agrawala refined this protocol so that it requires N messages to obtain entry in the worst (and common) case, without hardware support for multicast. This is described in Raynal [1988].

Maekawa's voting algorithm • Maekawa [1985] observed that in order for a process to enter a critical section, it is not necessary for all of its peers to grant it access. Processes need only obtain permission to enter from *subsets* of their peers, as long as the subsets used by any two processes overlap. We can think of processes as voting for one another to enter the critical section. A 'candidate' process must collect sufficient votes to enter. Processes in the intersection of two sets of voters ensure the safety property ME1, that at most one process can enter the critical section, by casting their votes for only one candidate.

Maekawa associated a *voting set* V_i with each process p_i ($i = 1, 2, \dots, N$), where $V_i \subseteq \{p_1, p_1, \dots, p_N\}$. The sets V_i are chosen so that, for all $i, j = 1, 2, \dots, N$:

- $p_i \in V_i$
- $V_i \cap V_j \neq \emptyset$ – there is at least one common member of any two voting sets
- $|V_i| = K$ – to be fair, each process has a voting set of the same size
- Each process p_j is contained in M of the voting sets V_i .

Maekawa showed that the optimal solution, which minimizes K and allows the processes to achieve mutual exclusion, has $K \sim \sqrt{N}$ and $M = K$ (so that each process is in as many of the voting sets as there are elements in each one of those sets). It is non-trivial to calculate the optimal sets R_i . As an approximation, a simple way of deriving sets R_i such that $|R_i| \sim 2\sqrt{N}$ is to place the processes in a \sqrt{N} by \sqrt{N} matrix and let V_i be the union of the row and column containing p_i .

Maekawa's algorithm is shown in Figure 15.6. To obtain entry to the critical section, a process p_i sends *request* messages to all K members of V_i (including itself). p_i cannot enter the critical section until it has received all K *reply* messages. When a process p_j in V_i receives p_i 's *request* message, it sends a *reply* message immediately,

Figure 15.6 Maekawa's algorithm

```

On initialization
  state := RELEASED;
  voted := FALSE;

For  $p_i$  to enter the critical section
  state := WANTED;
  Multicast request to all processes in  $V_i$ ;
  Wait until (number of replies received =  $K$ );
  state := HELD;

On receipt of a request from  $p_i$  at  $p_j$ 
  if (state = HELD or voted = TRUE)
  then
    queue request from  $p_i$  without replying;
  else
    send reply to  $p_i$ ;
    voted := TRUE;
  end if

For  $p_i$  to exit the critical section
  state := RELEASED;
  Multicast release to all processes in  $V_i$ ;

On receipt of a release from  $p_i$  at  $p_j$ 
  if (queue of requests is non-empty)
  then
    remove head of queue – from  $p_k$ , say;
    send reply to  $p_k$ ;
    voted := TRUE;
  else
    voted := FALSE;
  end if

```

unless either its state is *HELD* or it has already replied ('voted') since it last received a *release* message. Otherwise, it queues the request message (in the order of its arrival) but does not yet reply. When a process receives a *release* message, it removes the head of its queue of outstanding requests (if the queue is nonempty) and sends a *reply* message (a 'vote') in response to it. To leave the critical section, p_i sends *release* messages to all K members of V_i (including itself).

This algorithm achieves the safety property, ME1. If it were possible for two processes p_i and p_j to enter the critical section at the same time, then the processes in $V_i \cap V_j \neq \emptyset$ would have to have voted for both. But the algorithm allows a process to make at most one vote between successive receipts of a *release* message, so this situation is impossible.

Unfortunately, the algorithm is deadlock-prone. Consider three processes, p_1 , p_2 and p_3 , with $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$ and $V_3 = \{p_3, p_1\}$. If the three

processes concurrently request entry to the critical section, then it is possible for p_1 to reply to itself and hold off p_2 , for p_2 to reply to itself and hold off p_3 , and for p_3 to reply to itself and hold off p_1 . Each process has received one out of two replies, and none can proceed.

The algorithm can be adapted [Sanders 1987] so that it becomes deadlock-free. In the adapted protocol, processes queue outstanding requests in happened-before order, so that requirement ME3 is also satisfied.

The algorithm's bandwidth utilization is $2\sqrt{N}$ messages per entry to the critical section and \sqrt{N} messages per exit (assuming no hardware multicast facilities). The total of $3\sqrt{N}$ is superior to the $2(N-1)$ messages required by Ricart and Agrawala's algorithm, if $N > 4$. The client delay is the same as that of Ricart and Agrawala's algorithm, but the synchronization delay is worse: a round-trip time instead of a single message transmission time.

Fault tolerance • The main points to consider when evaluating the above algorithms with respect to fault tolerance are:

- What happens when messages are lost?
- What happens when a process crashes?

None of the algorithms that we have described would tolerate the loss of messages, if the channels were unreliable. The ring-based algorithm cannot tolerate a crash failure of any single process. As it stands, Maekawa's algorithm can tolerate some process crash failures: if a crashed process is not in a voting set that is required, then its failure will not affect the other processes. The central server algorithm can tolerate the crash failure of a client process that neither holds nor has requested the token. The Ricart and Agrawala algorithm as we have described it can be adapted to tolerate the crash failure of such a process, by taking it to grant all requests implicitly.

We invite the reader to consider how to adapt the algorithms to tolerate failures, on the assumption that a reliable failure detector is available. Even with a reliable failure detector, care is required to allow for failures at any point (including during a recovery procedure), and to reconstruct the state of the processes after a failure has been detected. For example, in the central-server algorithm, if the server fails it must be established whether it or one of the client processes held the token.

We examine the general problem of how processes should coordinate their actions in the presence of faults in Section 15.5.

15.3 Elections

An algorithm for choosing a unique process to play a particular role is called an *election algorithm*. For example, in a variant of our central-server algorithm for mutual exclusion, the 'server' is chosen from among the processes p_i , ($i = 1, 2, \dots, N$) that need to use the critical section. An election algorithm is needed for this choice. It is essential that all the processes agree on the choice. Afterwards, if the process that plays the role of server wishes to retire then another election is required to choose a replacement.

We say that a process *calls the election* if it takes an action that initiates a particular run of the election algorithm. An individual process does not call more than one election at a time, but in principle the N processes could call N concurrent elections. At any point in time, a process p_i is either a *participant* – meaning that it is engaged in some run of the election algorithm – or a *non-participant* – meaning that it is not currently engaged in any election.

An important requirement is for the choice of elected process to be unique, even if several processes call elections concurrently. For example, two processes could decide independently that a coordinator process has failed, and both call elections.

Without loss of generality, we require that the elected process be chosen as the one with the largest identifier. The ‘identifier’ may be any useful value, as long as the identifiers are unique and totally ordered. For example, we could elect the process with the lowest computational load by having each process use $\langle 1/load, i \rangle$ as its identifier, where $load > 0$ and the process index i is used to order identifiers with the same load.

Each process p_i ($i = 1, 2, \dots, N$) has a variable $elected_i$, which will contain the identifier of the elected process. When the process first becomes a participant in an election it sets this variable to the special value ‘ \perp ’ to denote that it is not yet defined.

Our requirements are that, during any particular run of the algorithm:

- | | |
|----------------|--|
| E1: (safety) | A participant process p_i has $elected_i = \perp$ or $elected_i = P$, where P is chosen as the non-crashed process at the end of the run with the largest identifier. |
| E2: (liveness) | All processes p_i participate and eventually either set $elected_i \neq \perp$ – or crash. |

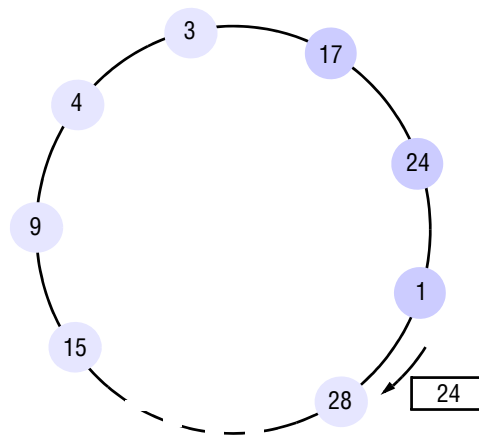
Note that there may be processes p_j that are not yet participants, which record in $elected_j$ the identifier of the previous elected process.

We measure the performance of an election algorithm by its total network bandwidth utilization (which is proportional to the total number of messages sent), and by the *turnaround time* for the algorithm: the number of serialized message transmission times between the initiation and termination of a single run.

A ring-based election algorithm • The algorithm of Chang and Roberts [1979] is suitable for a collection of processes arranged in a logical ring. Each process p_i has a communication channel to the next process in the ring, $p_{(i+1) \bmod N}$, and all messages are sent clockwise around the ring. We assume that no failures occur, and that the system is asynchronous. The goal of this algorithm is to elect a single process called the *coordinator*, which is the process with the largest identifier.

Initially, every process is marked as a *non-participant* in an election. Any process can begin an election. It proceeds by marking itself as a *participant*, placing its identifier in an *election* message and sending it to its clockwise neighbour.

When a process receives an *election* message, it compares the identifier in the message with its own. If the arrived identifier is greater, then it forwards the message to its neighbour. If the arrived identifier is smaller and the receiver is not a *participant*, then it substitutes its own identifier in the message and forwards it; but it does not forward the message if it is already a *participant*. On forwarding an *election* message in any case, the process marks itself as a *participant*.

Figure 15.7 A ring-based election in progress

Note: The election was started by process 17. The highest process identifier encountered so far is 24. Participant processes are shown in a darker tint.

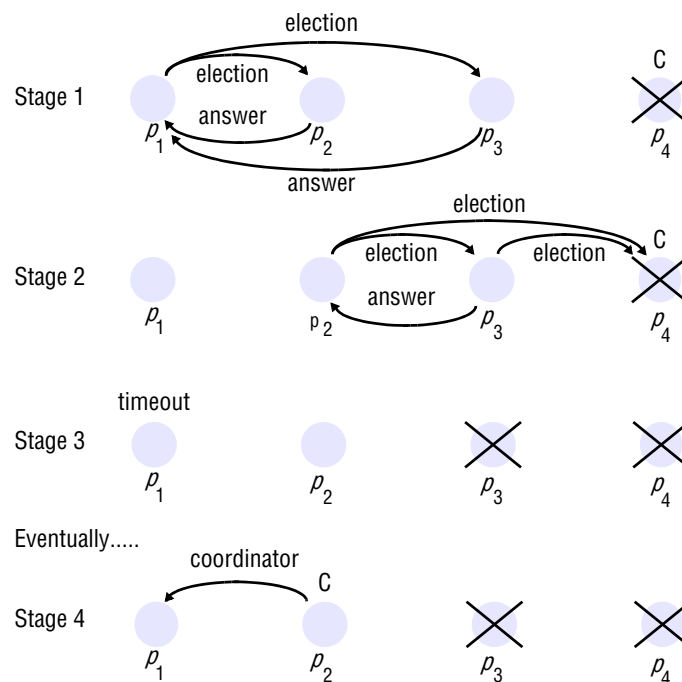
If, however, the received identifier is that of the receiver itself, then this process's identifier must be the greatest, and it becomes the coordinator. The coordinator marks itself as a *non-participant* once more and sends an *elected* message to its neighbour, announcing its election and enclosing its identity.

When a process p_i receives an *elected* message, it marks itself as a *non-participant*, sets its variable $elector_i$ to the identifier in the message and, unless it is the new coordinator, forwards the message to its neighbour.

It is easy to see that condition E1 is met. All identifiers are compared, since a process must receive its own identifier back before sending an *elected* message. For any two processes, the one with the larger identifier will not pass on the other's identifier. It is therefore impossible that both should receive their own identifier back.

Condition E2 follows immediately from the guaranteed traversals of the ring (there are no failures). Note how the *non-participant* and *participant* states are used so that duplicate messages arising when two processes start an election at the same time are extinguished as soon as possible, and always before the 'winning' election result has been announced.

If only a single process starts an election, then the worst-performing case is when its anti-clockwise neighbour has the highest identifier. A total of $N - 1$ messages are then required to reach this neighbour, which will not announce its election until its identifier has completed another circuit, taking a further N messages. The *elected* message is then sent N times, making $3N - 1$ messages in all. The turnaround time is also $3N - 1$, since these messages are sent sequentially.

Figure 15.8 The bully algorithm

The election of coordinator p_2 , after the failure of p_4 and then p_3

An example of a ring-based election in progress is shown in Figure 15.7. The *election* message currently contains 24, but process 28 will replace this with its identifier when the message reaches it.

While the ring-based algorithm is useful for understanding the properties of election algorithms in general, the fact that it tolerates no failures makes it of limited practical value. However, with a reliable failure detector it is in principle possible to re-constitute the ring when a process crashes.

The bully algorithm • The bully algorithm [Garcia-Molina 1982] allows processes to crash during an election, although it assumes that message delivery between processes is reliable. Unlike the ring-based algorithm, this algorithm assumes that the system is synchronous: it uses timeouts to detect a process failure. Another difference is that the ring-based algorithm assumed that processes have minimal *a priori* knowledge of one another: each knows only how to communicate with its neighbour, and none knows the identifiers of the other processes. The bully algorithm, on the other hand, assumes that each process knows which processes have higher identifiers, and that it can communicate with all such processes.

There are three types of message in this algorithm: an *election* message is sent to announce an election; an *answer* message is sent in response to an election message and a *coordinator* message is sent to announce the identity of the elected process – the new

‘coordinator’. A process begins an election when it notices, through timeouts, that the coordinator has failed. Several processes may discover this concurrently.

Since the system is synchronous, we can construct a reliable failure detector. There is a maximum message transmission delay, T_{trans} , and a maximum delay for processing a message $T_{process}$. Therefore, we can calculate a time $T = 2T_{trans} + T_{process}$ that is an upper bound on the time that can elapse between sending a message to another process and receiving a response. If no response arrives within time T , then the local failure detector can report that the intended recipient of the request has failed.

The process that knows it has the highest identifier can elect itself as the coordinator simply by sending a *coordinator* message to all processes with lower identifiers. On the other hand, a process with a lower identifier can begin an election by sending an *election* message to those processes that have a higher identifier and awaiting *answer* messages in response. If none arrives within time T , the process considers itself the coordinator and sends a *coordinator* message to all processes with lower identifiers announcing this. Otherwise, the process waits a further period T' for a *coordinator* message to arrive from the new coordinator. If none arrives, it begins another election.

If a process p_i receives a *coordinator* message, it sets its variable *elected_i* to the identifier of the coordinator contained within it and treats that process as the coordinator.

If a process receives an *election* message, it sends back an *answer* message and begins another election – unless it has begun one already.

When a process is started to replace a crashed process, it begins an election. If it has the highest process identifier, then it will decide that it is the coordinator and announce this to the other processes. Thus it will become the coordinator, even though the current coordinator is functioning. It is for this reason that the algorithm is called the ‘bully’ algorithm.

The operation of the algorithm is shown in Figure 15.8. There are four processes, $p_1 - p_4$. Process p_1 detects the failure of the coordinator p_4 and announces an election (stage 1 in the figure). On receiving an *election* message from p_1 , processes p_2 and p_3 send *answer* messages to p_1 and begin their own elections; p_3 sends an *answer* message to p_2 , but p_3 receives no *answer* message from the failed process p_4 (stage 2). It therefore decides that it is the coordinator. But before it can send out the *coordinator* message, it too fails (stage 3). When p_1 ’s timeout period T' expires (which we assume occurs before p_2 ’s timeout expires), it deduces the absence of a *coordinator* message and begins another election. Eventually, p_2 is elected coordinator (stage 4).

This algorithm clearly meets the liveness condition E2, by the assumption of reliable message delivery. And if no process is replaced, then the algorithm meets condition E1. It is impossible for two processes to decide that they are the coordinator, since the process with the lower identifier will discover that the other exists and defer to it.

But the algorithm is *not* guaranteed to meet the safety condition E1 if processes that have crashed are replaced by processes with the same identifiers. A process that replaces a crashed process p may decide that it has the highest identifier just as another process (which has detected p ’s crash) decides that *it* has the highest identifier. Two processes will therefore announce themselves as the coordinator concurrently. Unfortunately, there are no guarantees on message delivery order, and the recipients of these messages may reach different conclusions on which is the coordinator process.

Furthermore, condition E1 may be broken if the assumed timeout values turn out to be inaccurate – that is, if the processes' failure detector is unreliable.

Taking the example just given, suppose that either p_3 had not failed but was just running unusually slowly (that is, that the assumption that the system is synchronous is incorrect), or that p_3 had failed but was then replaced. Just as p_2 sends its *coordinator* message, p_3 (or its replacement) does the same. p_2 receives p_3 's *coordinator* message after it has sent its own and so sets $elect_2 = p_3$. Due to variable message transmission delays, p_1 receives p_2 's *coordinator* message after p_3 's and so eventually sets $elect_1 = p_2$. Condition E1 has been broken.

With regard to the performance of the algorithm, in the best case the process with the second-highest identifier notices the coordinator's failure. Then it can immediately elect itself and send $N - 2$ coordinator messages. The turnaround time is one message. The bully algorithm requires $O(N^2)$ messages in the worst case – that is, when the process with the lowest identifier first detects the coordinator's failure. For then $N - 1$ processes altogether begin elections, each sending messages to processes with higher identifiers.

15.4 Coordination and agreement in group communication

This chapter examines the key coordination and agreement problems related to group communication – that is, how to achieve the desired reliability and ordering properties across all members of a group. Chapter 6 introduced group communication as an example of an indirect communication technique whereby processes can send messages to a group. This message is propagated to all members of the group with certain guarantees in terms of reliability and ordering. We are particularly seeking reliability in terms of the properties of validity, integrity and agreement, and ordering in terms of FIFO ordering, causal ordering and total ordering.

In this chapter, we study multicast communication to groups of processes whose membership is known. Chapter 18 will expand our study to fully fledged group communication, including the management of dynamically varying groups.

System model • The system under consideration contains a collection of processes, which can communicate reliably over one-to-one channels. As before, processes may fail only by crashing.

The processes are members of groups, which are the destinations of messages sent with the *multicast* operation. It is generally useful to allow processes to be members of several groups simultaneously – for example, to enable processes to receive information from several sources by joining several groups. But to simplify our discussion of ordering properties, we shall sometimes restrict processes to being members of at most one group at a time.

The operation *multicast*(g, m) sends the message m to all members of the group g of processes. Correspondingly, there is an operation *deliver*(m) that delivers a message sent by multicast to the calling process. We use the term *deliver* rather than *receive* to make clear that a multicast message is not always handed to the application layer inside

the process as soon as it is received at the process's node. This is explained when we discuss multicast delivery semantics shortly.

Every message m carries the unique identifier of the process $sender(m)$ that sent it, and the unique destination group identifier $group(m)$. We assume that processes do not lie about the origin or destinations of messages.

Some algorithms assume that groups are closed (as defined in Chapter 6).

15.4.1 Basic multicast

It is useful to have at our disposal a basic multicast primitive that guarantees, unlike IP multicast, that a correct process will eventually deliver the message, as long as the multicaster does not crash. We call the primitive *B-multicast* and its corresponding basic delivery primitive *B-deliver*. We allow processes to belong to several groups, and each message is destined for some particular group.

A straightforward way to implement *B-multicast* is to use a reliable one-to-one *send* operation, as follows:

To *B-multicast*(g, m): for each process $p \in g$, *send*(p, m);

On *receive*(m) at p : *B-deliver*(m) at p .

The implementation may use threads to perform the *send* operations concurrently, in an attempt to reduce the total time taken to deliver the message. Unfortunately, such an implementation is liable to suffer from a so-called *ack-implosion* if the number of processes is large. The acknowledgements sent as part of the reliable *send* operation are liable to arrive from many processes at about the same time. The multicasting process's buffers will rapidly fill, and it is liable to drop acknowledgements. It will therefore retransmit the message, leading to yet more acknowledgements and further waste of network bandwidth. A more practical basic multicast service can be built using IP multicast, and we invite the reader to show this in Exercise 15.10.

15.4.2 Reliable multicast

Chapter 6 discussed reliable multicast in terms of validity, integrity and agreement. This section builds on this informal discussion, presenting a more complete definition.

Following Hadzilacos and Toueg [1994] and Chandra and Toueg [1996], we define a *reliable multicast* with corresponding operations *R-multicast* and *R-deliver*. Properties analogous to integrity and validity are clearly highly desirable in reliable multicast delivery, but we add another: a requirement that *all* correct processes in the group must receive a message if *any* of them does. It is important to realize that this is not a property of the *B-multicast* algorithm that is based on a reliable one-to-one *send* operation. The sender may fail at any point while *B-multicast* proceeds, so some processes may deliver a message while others do not.

A reliable multicast is one that satisfies the following properties:

Integrity: A correct process p delivers a message m at most once. Furthermore, $p \in group(m)$ and m was supplied to a *multicast* operation by $sender(m)$. (As with one-to-one communication, messages can always be distinguished by a sequence number relative to their sender.)

Figure 15.9 Reliable multicast algorithm

```

On initialization
  Received := {};

For process  $p$  to  $R$ -multicast message  $m$  to group  $g$ 
   $B$ -multicast( $g, m$ );      //  $p \in g$  is included as a destination

On  $B$ -deliver( $m$ ) at process  $q$  with  $g = \text{group}(m)$ 
  if ( $m \notin \text{Received}$ )
  then
    Received := Received  $\cup$  { $m$ };
    if ( $q \neq p$ ) then  $B$ -multicast( $g, m$ ); end if
     $R$ -deliver  $m$ ;
  end if

```

Validity: If a correct process multicasts message m , then it will eventually deliver m .

Agreement: If a correct process delivers message m , then all other correct processes in $\text{group}(m)$ will eventually deliver m .

The integrity property is analogous to that for reliable one-to-one communication. The validity property guarantees liveness for the sender. This may seem an unusual property, because it is asymmetric (it mentions only one particular process). But notice that validity and agreement together amount to an overall liveness requirement: if one process (the sender) eventually delivers a message m , since the correct processes agree on the set of messages they deliver, it follows that m will eventually be delivered to all the group's correct members.

The advantage of expressing the validity condition in terms of self-delivery is simplicity. What we require is that the message be delivered eventually by *some* correct member of the group.

The agreement condition is related to atomicity, the property of 'all or nothing', applied to delivery of messages to a group. If a process that multicasts a message crashes before it has delivered it, then it is possible that the message will not be delivered to any process in the group; but if it is delivered to some correct process, then all other correct processes will deliver it. Many papers in the literature use the term 'atomic' to include a total ordering condition; we define this shortly.

Implementing reliable multicast over B -multicast • Figure 15.9 gives a reliable multicast algorithm, with primitives R -multicast and R -deliver, that allows processes to belong to several closed groups simultaneously. To R -multicast a message, a process B -multicasts the message to the processes in the destination group (including itself). When the message is B -delivered, the recipient in turn B -multicasts the message to the group (if it is not the original sender), and then R -delivers the message. Since a message may arrive more than once, duplicates of the message are detected and not delivered.

This algorithm clearly satisfies the validity property, since a correct process will eventually B -deliver the message to itself. By the integrity property of the underlying communication channels used in B -multicast, the algorithm also satisfies the integrity property.

Agreement follows from the fact that every correct process *B-multicasts* the message to the other processes after it has *B-delivered* it. If a correct process does not *R-deliver* the message, then this can only be because it never *B-delivered* it. That in turn can only be because no other correct process *B-delivered* it either; therefore none will *R-deliver* it.

The reliable multicast algorithm that we have described is correct in an asynchronous system, since we made no timing assumptions. But the algorithm is inefficient for practical purposes. Each message is sent $|g|$ times to each process.

Reliable multicast over IP multicast • An alternative realization of *R-multicast* is to use a combination of IP multicast, piggybacked acknowledgements (that is, acknowledgements attached to other messages) and negative acknowledgements. This *R-multicast* protocol is based on the observation that IP multicast communication is often successful. In the protocol, processes do not send separate acknowledgement messages; instead, they piggyback acknowledgements on the messages that they send to the group. Processes send a separate response message only when they detect that they have missed a message. A response indicating the absence of an expected message is known as a *negative acknowledgement*.

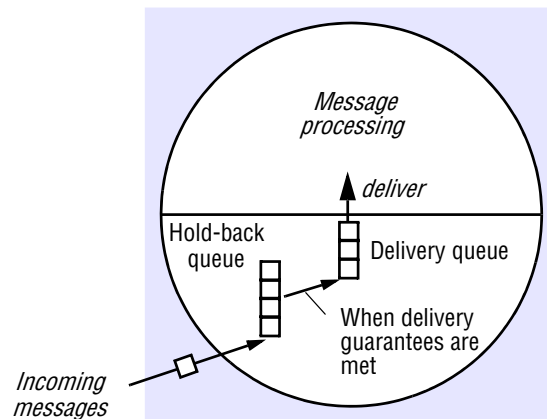
The description assumes that groups are closed. Each process p maintains a sequence number S_g^p for each group g to which it belongs. The sequence number is initially zero. Each process also records R_g^q , the sequence number of the latest message it has delivered from process q that was sent to group g .

For p to *R-multicast* a message to group g , it piggybacks onto the message the value S_g^p and acknowledgements, of the form $\langle q, R_g^q \rangle$. An acknowledgement states, for some sender q , the sequence number of the latest message from q destined for g that p has delivered since it last multicast a message. The multicaster p then IP-multicasts the message with its piggybacked values to g , and increments S_g^p by one.

The piggybacked values in a multicast message enable the recipients to learn about messages that they have not received. A process *R-delivers* a message destined for g bearing the sequence number S from p if and only if $S = R_g^p + 1$, and it increments R_g^p by one immediately after delivery. If an arriving message has $S \leq R_g^p$, then r has delivered the message before and it discards it. If $S > R_g^p + 1$, or if $R > R_g^q$ for an enclosed acknowledgement $\langle q, R \rangle$, then there are one or more messages that it has not yet received (and which are likely to have been dropped, in the first case). It keeps any message for which $S > R_g^p + 1$ in a *hold-back queue* (Figure 15.10) – such queues are often used to meet message delivery guarantees. It requests missing messages by sending negative acknowledgements, either to the original sender or to a process q from which it has received an acknowledgement $\langle q, R_g^q \rangle$ with R_g^q no less than the required sequence number.

The hold-back queue is not strictly necessary for reliability, but it simplifies the protocol by enabling us to use sequence numbers to represent sets of delivered messages. It also provides us with a guarantee of delivery order (see Section 15.4.3).

The integrity property follows from the detection of duplicates and the underlying properties of IP multicast (which uses checksums to expunge corrupted messages). The validity property holds because IP multicast has that property. For agreement we require, first, that a process can always detect missing messages. That in turn means that it will always receive a further message that enables it to detect the omission. As this

Figure 15.10 The hold-back queue for arriving multicast messages

simplified protocol stands, we guarantee detection of missing messages only in the case where correct processes multicast messages indefinitely. Second, the agreement property requires that there is always an available copy of any message needed by a process that did not receive it. We therefore assume that processes retain copies of the messages they have delivered – indefinitely, in this simplified protocol.

Neither of the assumptions we made to ensure agreement is practical (see Exercise 15.15). However, agreement is practically addressed in the protocols from which ours is derived: the Psync protocol [Peterson *et al.* 1989], Trans protocol [Melliar-Smith *et al.* 1990] and scalable reliable multicast protocol [Floyd *et al.* 1997]. Psync and Trans also provide further delivery ordering guarantees.

Uniform properties • The definition of agreement given above refers only to the behaviour of *correct* processes – processes that never fail. Consider what would happen in the algorithm of Figure 15.9 if a process was not correct and crashed after it had *R-delivered* a message. Since any process that *R-delivers* the message must first *B-multicast* it, it follows that all correct processes will still eventually deliver the message.

Any property that holds whether or not processes are correct is called a *uniform* property. We define uniform agreement as follows:

Uniform agreement: If a process, whether it is correct or fails, delivers message m , then all correct processes in $group(m)$ will eventually deliver m .

Uniform agreement allows a process to crash after it has delivered a message, while still ensuring that all correct processes will deliver the message. We have argued that the algorithm of Figure 15.9 satisfies this property, which is stronger than the non-uniform agreement property defined above.

Uniform agreement is useful in applications where a process may take an action that produces an observable inconsistency before it crashes. For example, suppose that the processes are servers that manage copies of a bank account, and that updates to the account are sent using reliable multicast to the group of servers. If the multicast does not satisfy uniform agreement, then a client that accesses a server just before it crashes may observe an update that no other server will process.

It is interesting to note that if we reverse the lines ‘*R-deliver m*’ and ‘*if (q ≠ p) then B-multicast(g, m); end if*’ in Figure 15.9, then the resultant algorithm does not satisfy uniform agreement.

Just as there is a uniform version of agreement, there are also uniform versions of any multicast property, including validity and integrity and the ordering properties that we are about to define.

15.4.3 Ordered multicast

The basic multicast algorithm of Section 15.4.1 delivers messages to processes in an arbitrary order, due to arbitrary delays in the underlying one-to-one *send* operations. This lack of an ordering guarantee is not satisfactory for many applications. For example, in a nuclear power plant it may be important that events signifying threats to safety conditions and events signifying actions by control units are observed in the same order by all processes in the system.

As discussed in Chapter 6, the common ordering requirements are total ordering, causal ordering and FIFO ordering, together with hybrid solutions (in particular, total-causal and total-FIFO). To simplify our discussion, we define these orderings under the assumption that any process belongs to at most one group (later we discuss the implications of allowing groups to overlap):

FIFO ordering: If a correct process issues *multicast(g, m)* and then *multicast(g, m')*, then every correct process that delivers *m'* will deliver *m* before *m'*.

Causal ordering: If *multicast(g, m) → multicast(g, m')*, where \rightarrow is the happened-before relation induced only by messages sent between the members of *g*, then any correct process that delivers *m'* will deliver *m* before *m'*.

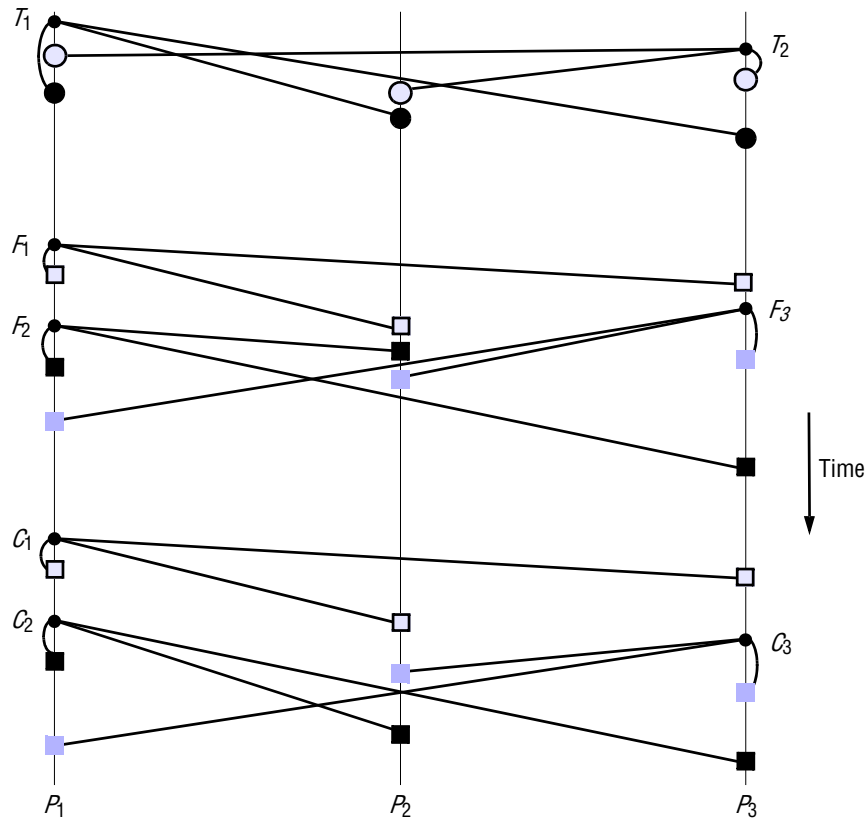
Total ordering: If a correct process delivers message *m* before it delivers *m'*, then any other correct process that delivers *m'* will deliver *m* before *m'*.

Causal ordering implies FIFO ordering, since any two multicasts by the same process are related by happened-before. Note that FIFO ordering and causal ordering are only partial orderings: not all messages are sent by the same process, in general; similarly, some multicasts are concurrent (not ordered by happened-before).

Figure 15.11 illustrates the orderings for the case of three processes. Close inspection of the figure shows that the totally ordered messages are delivered in the opposite order to the physical time at which they were sent. In fact, the definition of total ordering allows message delivery to be ordered arbitrarily, as long as the order is the same at different processes. Since total ordering is not necessarily also a FIFO or causal ordering, we define the hybrid of *FIFO-total* ordering as one for which message delivery obeys both FIFO and total ordering; similarly, under *causal-total* ordering message delivery obeys both causal and total ordering.

The definitions of ordered multicast do not assume or imply reliability. For example, the reader should check that, under total ordering, if correct process *p* delivers message *m* and then delivers *m'*, then a correct process *q* can deliver *m* without also delivering *m'* or any other message ordered after *m*.

We can also form hybrids of ordered and reliable protocols. A reliable totally ordered multicast is often referred to in the literature as an *atomic multicast*. Similarly,

Figure 15.11 Total, FIFO and causal ordering of multicast messages

Notice the consistent ordering of totally ordered messages T_1 and T_2 , the FIFO-related messages F_1 and F_2 and the causally related messages C_1 and C_2 – and the otherwise arbitrary delivery ordering of messages

we may form reliable FIFO multicast, reliable causal multicast and reliable versions of the hybrid ordered multicasts.

Ordering the delivery of multicast messages, as we shall see, can be expensive in terms of delivery latency and bandwidth consumption. The ordering semantics that we have described may delay the delivery of messages unnecessarily. That is, at the application level, a message may be delayed for another message that it does not in fact depend upon. For this reason, some have proposed multicast systems that use the application-specific message semantics alone to determine the order of message delivery [Cheriton and Skeen 1993, Pedone and Schiper 1999].

The example of the bulletin board • To make multicast delivery semantics more concrete, consider an application in which users post messages to bulletin boards. Each user runs a bulletin-board application process. Every topic of discussion has its own process group. When a user posts a message to a bulletin board, the application

Figure 15.12 Display from bulletin board program

Bulletin board: <i>os.interesting</i>		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

multicasts the user's posting to the corresponding group. Each user's process is a member of the group for the topic in which that user is interested, so they will receive just the postings concerning that topic.

Reliable multicast is required if every user is to receive every posting eventually. The users also have ordering requirements. Figure 15.12 shows the postings as they appear to a particular user. At a minimum, FIFO ordering is desirable, since then every posting from a given user – 'A.Hanlon', say – will be received in the same order, and users can talk consistently about A.Hanlon's second posting.

Note that the messages whose subjects are 'Re: Microkernels' (25) and 'Re: Mach' (27) appear after the messages to which they refer. A causally ordered multicast is needed to guarantee this relationship. Otherwise, arbitrary message delays could mean that, say, the message 'Re: Mach' could appear before the original message about Mach.

If the multicast delivery was totally ordered, then the numbering in the lefthand column would be consistent between users. Users could refer unambiguously, for example, to 'message 24'.

In practice, the USENET bulletin board system implements neither causal nor total ordering. The communication costs of achieving these orderings on a large scale outweigh their advantages.

Implementing FIFO ordering • FIFO-ordered multicast (with operations *FO-multicast* and *FO-deliver*) is achieved with sequence numbers, much as we would achieve it for one-to-one communication. We shall consider only non-overlapping groups. The reader should verify that the reliable multicast protocol that we defined on top of IP multicast in Section 15.4.2 also guarantees FIFO ordering, but we shall show how to construct a FIFO-ordered multicast on top of any given basic multicast. We use the variables S_g^p and R_g^q held at process p from the reliable multicast protocol of Section 15.4.2: S_g^p is a count of how many messages p has sent to g and, for each q , R_g^q is the sequence number of the latest message p has delivered from process q that was sent to group g .

For p to *FO-multicast* a message to group g , it piggybacks the value S_g^p onto the message, *B-multicasts* the message to g and then increments S_g^p by 1. Upon receipt of a message from q bearing the sequence number S , p checks whether $S = R_g^q + 1$. If so, this message is the next one expected from the sender q and p *FO-delivers* it, setting

$R_g^q := S$. If $S > R_g^q + 1$, it places the message in the hold-back queue until the intervening messages have been delivered and $S = R_g^q + 1$.

Since all messages from a given sender are delivered in the same sequence, and since a message's delivery is delayed until its sequence number has been reached, the condition for FIFO ordering is clearly satisfied. But this is so only under the assumption that groups are non-overlapping.

Note that we can use any implementation of *B-multicast* in this protocol. Moreover, if we use a reliable *R-multicast* primitive instead of *B-multicast*, then we obtain a reliable FIFO multicast.

Implementing total ordering • The basic approach to implementing total ordering is to assign totally ordered identifiers to multicast messages so that each process makes the same ordering decision based upon these identifiers. The delivery algorithm is very similar to the one we described for FIFO ordering; the difference is that processes keep group-specific sequence numbers rather than process-specific sequence numbers. We only consider how to totally order messages sent to non-overlapping groups. We call the multicast operations *TO-multicast* and *TO-deliver*.

We discuss two main methods for assigning identifiers to messages. The first of these is for a process called a *sequencer* to assign them (Figure 15.13). A process wishing to *TO-multicast* a message m to group g attaches a unique identifier $id(m)$ to it. The messages for g are sent to the sequencer for g , $sequencer(g)$, as well as to the members of g . (The sequencer may be chosen to be a member of g .) The process $sequencer(g)$ maintains a group-specific sequence number s_g , which it uses to assign increasing and consecutive sequence numbers to the messages that it *B-delivers*. It announces the sequence numbers by *B-multicasting order* messages to g (see Figure 15.13 for the details).

A message will remain in the hold-back queue indefinitely until it can be *TO-delivered* according to the corresponding sequence number. Since the sequence numbers are well defined (by the sequencer), the criterion for total ordering is met. Furthermore, if the processes use a FIFO-ordered variant of *B-multicast*, then the totally ordered multicast is also causally ordered. We leave the reader to show this.

The obvious problem with a sequencer-based scheme is that the sequencer may become a bottleneck and is a critical point of failure. Practical algorithms exist that address the problem of failure. Chang and Maxemchuk [1984] first suggested a multicast protocol employing a sequencer (which they called a *token site*). Kaashoek *et al.* [1989] developed a sequencer-based protocol for the Amoeba system. These protocols ensure that a message is in the hold-back queue at $f + 1$ nodes before it is delivered; up to f failures can thus be tolerated. Like Chang and Maxemchuk, Birman *et al.* [1991] also employ a token-holding site that acts as a sequencer. The token can be passed from process to process so that, for example, if only one process sends totally ordered multicasts that process can act as the sequencer, saving communication.

The protocol of Kaashoek *et al.* uses hardware-based multicast – available on an Ethernet, for example – rather than reliable point-to-point communication. In the simplest variant of their protocol, processes send the message to be multicast to the sequencer, one-to-one. The sequencer multicasts the message itself, as well as the identifier and sequence number. This has the advantage that the other members of the

Figure 15.13 Total ordering using a sequencer

1. Algorithm for group member p

On initialization: $r_g := 0$;

To TO-multicast message m to group g
 $B\text{-multicast}(g \cup \{\text{sequencer}(g)\}, \langle m, i \rangle)$;

On B-deliver($\langle m, i \rangle$) *with* $g = \text{group}(m)$
Place $\langle m, i \rangle$ in hold-back queue;

On B-deliver($m_{\text{order}} = \langle \text{"order"}, i, S \rangle$) *with* $g = \text{group}(m_{\text{order}})$
wait until $\langle m, i \rangle$ in hold-back queue and $S = r_g$;
 $TO\text{-deliver } m$; // (after deleting it from the hold-back queue)
 $r_g := S + 1$;
2. Algorithm for sequencer of g

On initialization: $s_g := 0$;

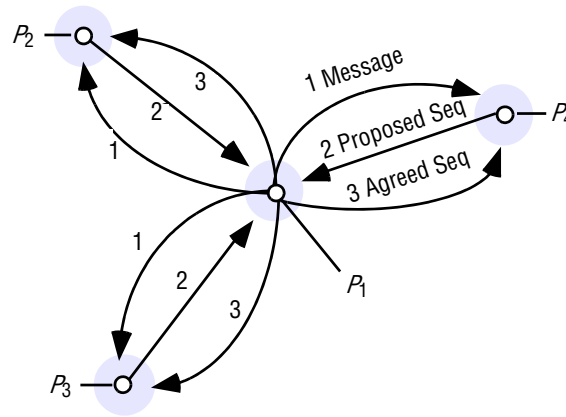
On B-deliver($\langle m, i \rangle$) *with* $g = \text{group}(m)$
 $B\text{-multicast}(g, \langle \text{"order"}, i, s_g \rangle)$;
 $s_g := s_g + 1$;

group receive only one message per multicast; its disadvantage is increased bandwidth utilization. The protocol is described in full at www.cdk5.net/coordination.

The second method that we examine for achieving totally ordered multicast is one in which the processes collectively agree on the assignment of sequence numbers to messages in a distributed fashion. A simple algorithm – similar to one that was originally developed to implement totally ordered multicast delivery for the ISIS toolkit [Birman and Joseph 1987a] – is shown in Figure 15.14. Once more, a process $B\text{-multicasts}$ its message to the members of the group. The group may be open or closed. The receiving processes propose sequence numbers for messages as they arrive and return these to the sender, which uses them to generate *agreed* sequence numbers.

Each process q in group g keeps A_g^q , the largest agreed sequence number it has observed so far for group g , and P_g^q , its own largest proposed sequence number. The algorithm for process p to multicast a message m to group g is as follows:

1. p $B\text{-multicasts}$ $\langle m, i \rangle$ to g , where i is a unique identifier for m .
2. Each process q replies to the sender p with a proposal for the message's agreed sequence number of $P_g^q := \text{Max}(A_g^q, P_g^q) + 1$. In reality, we must include process identifiers in the proposed values P_g^q to ensure a total order, since otherwise different processes could propose the same integer value; but for the sake of simplicity we shall not make that explicit here. Each process provisionally assigns the proposed sequence number to the message and places it in its hold-back queue, which is ordered with the *smallest* sequence number at the front.

Figure 15.14 The ISIS algorithm for total ordering

3. p collects all the proposed sequence numbers and selects the largest one, a , as the next agreed sequence number. It then B -multicasts $\langle i, a \rangle$ to g . Each process q in g sets $A_g^q := \text{Max}(A_g^q, a)$ and attaches a to the message (which is identified by i). It reorders the message in the hold-back queue if the agreed sequence number differs from the proposed one. When the message at the front of the hold-back queue has been assigned its agreed sequence number, it is transferred to the tail of the delivery queue. Messages that have been assigned their agreed sequence number but are not at the head of the hold-back queue are not yet transferred, however.

If every process agrees the same set of sequence numbers and delivers them in the corresponding order, then total ordering is satisfied. It is clear that correct processes ultimately agree on the same set of sequence numbers, but we must show that they are monotonically increasing and that no correct process can deliver a message prematurely.

Assume that a message m_1 has been assigned an agreed sequence number and has reached the front of the hold-back queue. By construction, a message that is received after this stage will and should be delivered after m_1 : it will have a larger proposed sequence number and thus a larger agreed sequence number than m_1 . So let m_2 be any other message that has not yet been assigned its agreed sequence number but that is on the same queue. We have that:

$$\text{agreedSequence}(m_2) \geq \text{proposedSequence}(m_2)$$

by the algorithm just given. Since m_1 is at the front of the queue:

$$\text{proposedSequence}(m_2) > \text{agreedSequence}(m_1)$$

Therefore:

$$\text{agreedSequence}(m_2) > \text{agreedSequence}(m_1)$$

Figure 15.15 Causal ordering using vector timestamps

Algorithm for group member p_i ($i = 1, 2, \dots, N$)

On initialization

$$V_i^g[j] := 0 \quad (j = 1, 2, \dots, N);$$

To CO-multicast message m to group g

$$V_i^g[i] := V_i^g[i] + 1;$$

$$B\text{-multicast}(g, \langle V_i^g, m \rangle);$$

On B-deliver($\langle V_j^g, m \rangle$) from p_j ($j \neq i$), with $g = \text{group}(m)$

place $\langle V_j^g, m \rangle$ in hold-back queue;

wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ ($k \neq j$);

CO-deliver m ; // after removing it from the hold-back queue

$$V_i^g[j] := V_i^g[j] + 1;$$

and total ordering is assured.

This algorithm has higher latency than the sequencer-based multicast algorithm: three messages are sent serially between the sender and the group before a message can be delivered.

Note that the total ordering chosen by this algorithm is not also guaranteed to be causally or FIFO-ordered: any two messages are delivered in an essentially arbitrary total order, influenced by communication delays.

For other approaches to implementing total ordering, see Melliar-Smith *et al.* [1990], Garcia-Molina and Spauster [1991] and Hadzilacos and Toueg [1994].

Implementing causal ordering • Next we give an algorithm for non-overlapping closed groups based on that developed by Birman *et al.* [1991], shown in Figure 15.15, in which the causally ordered multicast operations are *CO-multicast* and *CO-deliver*. The algorithm takes account of the happened-before relationship only as it is established by *multicast* messages. If the processes send one-to-one messages to one another, then these will not be accounted for.

Each process p_i ($i = 1, 2, \dots, N$) maintains its own vector timestamp (see Section 14.4). The entries in the timestamp count the number of multicast messages from each process that happened-before the next message to be multicast.

To *CO-multicast* a message to group g , the process adds 1 to its entry in the timestamp and *B-multicasts* the message along with its timestamp to g .

When a process p_i *B-delivers* a message from p_j , it must place it in the hold-back queue before it can *CO-deliver* it – that is, until it is assured that it has delivered any messages that causally preceded it. To establish this, p_i waits until (a) it has delivered any earlier message sent by p_j , and (b) it has delivered any message that p_j had delivered at the time it multicast the message. Both of those conditions can be detected by examining vector timestamps, as shown in Figure 15.15. Note that a process can immediately *CO-deliver* to itself any message that it *CO-multicasts*, although this is not described in Figure 15.15.

Each process updates its vector timestamp upon delivering any message, to maintain the count of causally precedent messages. It does this by incrementing the j th entry in its timestamp by one. This is an optimization of the *merge* operation that appears in the rules for updating vector clocks in Section 14.4. We can make the optimization in view of the delivery condition in the algorithm of Figure 15.15, which guarantees that only the j th entry will increase.

We outline the proof of the correctness of this algorithm as follows. Suppose that $\text{multicast}(g, m) \rightarrow \text{multicast}(g, m')$. Let V and V' be the vector timestamps of m and m' , respectively. It is straightforward to prove inductively from the algorithm that $V < V'$. In particular, if process p_k multicast m , then $V[k] \leq V'[k]$.

Consider what happens when some correct process p_i *B-delivers* m' (as opposed to *CO-delivering* it) without first *CO-delivering* m . By the algorithm, $V_i[k]$ can increase only when p_i delivers a message from p_k , when it increases by 1. But p_i has not received m , and therefore $V_i[k]$ cannot increase beyond $V[k] - 1$. It is therefore not possible for p_i to *CO-deliver* m' , since this would require that $V_i[k] \geq V'[k]$, and therefore that $V_i[k] \geq V[k]$.

The reader should check that if we substitute the reliable *R-multicast* primitive in place of *B-multicast*, then we obtain a multicast that is both reliable and causally ordered.

Furthermore, if we combine the protocol for causal multicast with the sequencer-based protocol for totally ordered delivery, then we obtain message delivery that is both total and causal. The sequencer delivers messages according to the causal order and multicasts the sequence numbers for the messages in the order in which it receives them. The processes in the destination group do not deliver a message until they have received an *order* message from the sequencer and the message is next in the delivery sequence.

Since the sequencer delivers messages in causal order, and since all other processes deliver messages in the same order as the sequencer, the ordering is indeed both total and causal.

Overlapping groups • We have considered only non-overlapping groups in the preceding definitions and algorithms for FIFO, total and causal ordering semantics. This simplifies the problem, but it is not satisfactory, since in general processes need to be members of multiple overlapping groups. For example, a process may be interested in events from multiple sources and thus join a corresponding set of event-distribution groups.

We can extend the ordering definitions to global orders [Hadzilacos and Toueg 1994], in which we have to consider that if message m is multicast to g , and if message m' is multicast to g' , then both messages are addressed to the members of $g \cap g'$:

Global FIFO ordering: If a correct process issues $\text{multicast}(g, m)$ and then $\text{multicast}(g', m')$, then every correct process in $g \cap g'$ that delivers m' will deliver m before m' .

Global causal ordering: If $\text{multicast}(g, m) \rightarrow \text{multicast}(g', m')$, where \rightarrow is the happened-before relation induced by any chain of multicast messages, then any correct process in $g \cap g'$ that delivers m' will deliver m before m' .

Pairwise total ordering: If a correct process delivers message m sent to g before it delivers m' sent to g' , then any other correct process in $g \cap g'$ that delivers m' will deliver m before m' .

Global total ordering: Let ' $<$ ' be the relation of ordering between delivery events. We require that ' $<$ ' obeys pairwise total ordering and that it is acyclic – under pairwise total ordering, ' $<$ ' is not acyclic by default.

One way of implementing these orders would be to multicast each message m to the group of *all* processes in the system. Each process either discards or delivers the message according to whether it belongs to $group(m)$. This would be an inefficient and unsatisfactory implementation: a multicast should involve as few processes as possible beyond the members of the destination group. Alternatives are explored in Birman *et al.* [1991], Garcia-Molina and Spauster [1991], Hadzilacos and Toueg [1994], Kindberg [1995] and Rodrigues *et al.* [1998].

Multicast in synchronous and asynchronous systems • In this section, we have described algorithms for reliable unordered multicast, (reliable) FIFO-ordered multicast, (reliable) causally ordered multicast and totally ordered multicast. We also indicated how to achieve a multicast that is both totally and causally ordered. We leave the reader to devise an algorithm for a multicast primitive that guarantees both FIFO and total ordering. All the algorithms that we have described work correctly in asynchronous systems.

We did not, however, give an algorithm that guarantees both reliable and totally ordered delivery. Surprising though it may seem, while possible in a *synchronous* system, a protocol with these guarantees is impossible in an *asynchronous* distributed system – even one that has at worst suffered a single process crash failure. We return to this point in the next section.

15.5 Consensus and related problems

This section introduces the problem of consensus [Pease *et al.* 1980, Lamport *et al.* 1982] and the related problems of Byzantine generals and interactive consistency. We refer to these collectively as problems of *agreement*. Roughly speaking, the problem is for processes to agree on a value after one or more of the processes has proposed what that value should be.

For example, in Chapter 2 we described a situation in which two armies should decide consistently to attack the common enemy or retreat. Similarly, we may require that all the correct processes controlling a spaceship's engines should decide to either 'proceed' or 'abort' after each has proposed one action or the other, and in a transaction to transfer funds from one account to another the processes involved must consistently agree to perform the respective debit and credit. In mutual exclusion, the processes agree on which process can enter the critical section. In an election, the processes agree on which is the elected process. In totally ordered multicast, the processes agree on the order of message delivery.

Protocols exist that are tailored to these individual types of agreement. We described some of them above, and Chapters 16 and 17 examine transactions. But it is

useful for us to consider more general forms of agreement, in a search for common characteristics and solutions.

This section defines consensus more precisely and relates it to three related agreement problems: Byzantine generals, interactive consistency and totally ordered multicast. We go on to examine under what circumstances the problems can be solved, and to sketch some solutions. In particular, we discuss the well-known impossibility result of Fischer *et al.* [1985], which states that in an asynchronous system a collection of processes containing only one faulty process cannot be guaranteed to reach consensus. Finally, we consider how it is that practical algorithms exist despite the impossibility result.

15.5.1 System model and problem definitions

Our system model includes a collection of processes p_i ($i = 1, 2, \dots, N$) communicating by message passing. An important requirement that applies in many practical situations is for consensus to be reached even in the presence of faults. We assume, as before, that communication is reliable but that processes may fail. In this section we consider Byzantine (arbitrary) process failures, as well as crash failures. We sometimes specify an assumption that up to some number f of the N processes are faulty – that is, they exhibit some specified types of fault; the remainder of the processes are correct.

If arbitrary failures can occur, then another factor in specifying our system is whether the processes digitally sign the messages that they send (see Section 11.4). If processes sign their messages, then a faulty process is limited in the harm it can do. Specifically, during an agreement algorithm it cannot make a false claim about the values that a correct process has sent to it. The relevance of message signing will become clearer when we discuss solutions to the Byzantine generals problem. By default, we assume that signing does not take place.

Definition of the consensus problem • To reach consensus, every process p_i begins in the *undecided* state and *proposes* a single value v_i , drawn from a set D ($i = 1, 2, \dots, N$). The processes communicate with one another, exchanging values. Each process then sets the value of a *decision variable*, d_i . In doing so it enters the *decided* state, in which it may no longer change d_i ($i = 1, 2, \dots, N$). Figure 15.16 shows three processes engaged in a consensus algorithm. Two processes propose ‘proceed’ and a third proposes ‘abort’ but then crashes. The two processes that remain correct each decide ‘proceed’.

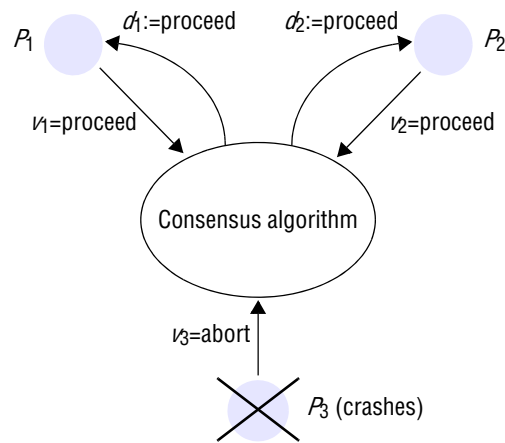
The requirements of a consensus algorithm are that the following conditions should hold for every execution of it:

Termination: Eventually each correct process sets its decision variable.

Agreement: The decision value of all correct processes is the same: if p_i and p_j are correct and have entered the *decided* state, then $d_i = d_j$ ($i, j = 1, 2, \dots, N$).

Integrity: If the correct processes all proposed the same value, then any correct process in the *decided* state has chosen that value.

Variations on the definition of integrity may be appropriate, according to the application. For example, a weaker type of integrity would be for the decision value to

Figure 15.16 Consensus for three processes

equal a value that some correct process proposed – not necessarily all of them. We use the definition above except where stated otherwise. Integrity is also known as *validity* in the literature.

To help in understanding how the formulation of the problem translates into an algorithm, consider a system in which processes cannot fail. It is then straightforward to solve consensus. For example, we can collect the processes into a group and have each process reliably multicast its proposed value to the members of the group. Each process waits until it has collected all N values (including its own). It then evaluates the function $\text{majority}(v_1, v_2, \dots, v_N)$, which returns the value that occurs most often among its arguments, or the special value $\perp \notin D$ if no majority exists. Termination is guaranteed by the reliability of the multicast operation. Agreement and integrity are guaranteed by the definition of *majority* and the integrity property of a reliable multicast. Every process receives the same set of proposed values, and every process evaluates the same function of those values. So they must all agree, and if every process proposed the same value, then they all decide on this value.

Note that *majority* is only one possible function that the processes could use to agree upon a value from the candidate values. For example, if the values are ordered, then the functions *minimum* and *maximum* may be appropriate.

If processes can crash this introduces the complication of detecting failures, and it is not immediately clear that a run of the consensus algorithm can terminate. In fact, if the system is asynchronous, then it may not; we shall return to this point shortly.

If processes can fail in *arbitrary* (Byzantine) ways, then faulty processes can in principle communicate random values to the others. This may seem unlikely in practice, but it is not beyond the bounds of possibility for a process with a bug to fail in this way. Moreover, the fault may not be accidental, but the result of mischievous or malevolent operation. Someone could deliberately make a process send different values to different peers in an attempt to thwart the others, which are trying to reach consensus. In case of inconsistency, correct processes must compare what they have received with what other processes claim to have received.

The Byzantine generals problem • In the informal statement of the *Byzantine generals problem* [Lamport *et al.* 1982], three or more generals are to agree to attack or to retreat. One, the commander, issues the order. The others, lieutenants to the commander, must decide whether to attack or retreat. But one or more of the generals may be ‘treacherous’ – that is, faulty. If the commander is treacherous, he proposes attacking to one general and retreating to another. If a lieutenant is treacherous, he tells one of his peers that the commander told him to attack and another that they are to retreat.

The Byzantine generals problem differs from consensus in that a distinguished process supplies a value that the others are to agree upon, instead of each of them proposing a value. The requirements are:

Termination: Eventually each correct process sets its decision variable.

Agreement: The decision value of all correct processes is the same: if p_i and p_j are correct and have entered the *decided* state, then $d_i = d_j$ ($i, j = 1, 2, \dots, N$).

Integrity: If the commander is correct, then all correct processes decide on the value that the commander proposed.

Note that, for the Byzantine generals problem, integrity implies agreement when the commander is correct; but the commander need not be correct.

Interactive consistency • The interactive consistency problem is another variant of consensus, in which every process proposes a single value. The goal of the algorithm is for the correct processes to agree on a *vector* of values, one for each process. We call this the ‘decision vector’. For example, the goal could be for each of a set of processes to obtain the same information about their respective states.

The requirements for interactive consistency are:

Termination: Eventually each correct process sets its decision variable.

Agreement: The decision vector of all correct processes is the same.

Integrity: If p_i is correct, then all correct processes decide on v_i as the i th component of their vector.

Relating consensus to other problems • Although it is common to consider the Byzantine generals problem with arbitrary process failures, in fact each of the three problems – consensus, Byzantine generals and interactive consistency – is meaningful in the context of either arbitrary or crash failures. Similarly, each can be framed assuming either a synchronous or an asynchronous system.

It is sometimes possible to derive a solution to one problem using a solution to another. This is a very useful property, both because it increases our understanding of the problems and because by reusing solutions we can potentially save on implementation effort and complexity.

Suppose that there exist solutions to consensus (C), Byzantine generals (BG) and interactive consistency (IC) as follows:

$C_i(v_1, v_2, \dots, v_N)$ returns the decision value of p_i in a run of the solution to the consensus problem, where v_1, v_2, \dots, v_N are the values that the processes proposed.

$BG_i(j, v)$ returns the decision value of p_i in a run of the solution to the Byzantine generals problem, where p_j , the commander, proposes the value v .

$IC_i(v_1, v_2, \dots, v_N)[j]$ returns the j th value in the decision vector of p_i in a run of the solution to the interactive consistency problem, where v_1, v_2, \dots, v_N are the values that the processes proposed.

The definitions of C_i , BG_i and IC_i assume that a faulty process proposes a single notional value, even though it may have given different proposed values to each of the other processes. This is only a convenience: the solutions will not rely on any such notional value.

It is possible to construct solutions out of the solutions to other problems. We give three examples:

IC from BG: We construct a solution to IC from BG by running BG N times, once with each process p_i ($i, j = 1, 2, \dots, N$) acting as the commander:

$$IC_i(v_1, v_2, \dots, v_N)[j] = BG_i(j, v_j) \quad (i, j = 1, 2, \dots, N)$$

C from IC: For the case where a majority of processes are correct, we construct a solution to C from IC by running IC to produce a vector of values at each process, then applying an appropriate function on the vector's values to derive a single value:

$$C_i(v_1, \dots, v_N) = \text{majority}(IC_i(v_1, \dots, v_N)[1], \dots, IC_i(v_1, \dots, v_N)[N])$$

where $i = 1, 2, \dots, N$ and *majority* is as defined above.

BG from C: We construct a solution to BG from C as follows:

- The commander p_j sends its proposed value v to itself and each of the remaining processes.
- All processes run C with the values v_1, v_2, \dots, v_N that they receive (p_j may be faulty).
- They derive $BG_i(j, v) = C_i(v_1, v_2, \dots, v_N)$ ($i = 1, 2, \dots, N$).

The reader should check that the termination, agreement and integrity conditions are preserved in each case. Fischer [1983] relates the three problems in more detail.

In systems with crash failures, consensus is equivalent to solving reliable and totally ordered multicast: given a solution to one, we can solve the other. Implementing consensus with a reliable and totally ordered multicast operation *RTO-multicast* is straightforward. We collect all the processes into a group, g . To achieve consensus, each process p_i performs *RTO-multicast*(g, v_i). Then each process p_i chooses $d_i = m_i$, where m_i is the *first* value that p_i *RTO-delivers*. The termination property follows from the reliability of the multicast. The agreement and integrity properties follow from the reliability and total ordering of multicast delivery. Chandra and Toueg [1996] demonstrate how reliable and totally ordered multicast can be derived from consensus.

15.5.2 Consensus in a synchronous system

This section describes an algorithm to solve consensus in a synchronous system, although it is based on a modified form of the integrity requirement. The algorithm uses only a basic multicast protocol. It assumes that up to f of the N processes exhibit crash failures.

Figure 15.17 Consensus in a synchronous system

Algorithm for process $p_i \in g$; algorithm proceeds in $f+1$ rounds

On initialization

$$Values_i^1 := \{v_i\}; Values_i^0 = \{\};$$

In round r ($1 \leq r \leq f+1$)

$$B\text{-multicast}(g, Values_i^r - Values_i^{r-1}); // \text{ Send only values that have not been sent}$$

$$Values_i^{r+1} := Values_i^r;$$

while (in round r)

$$\{$$

$$\quad \text{On } B\text{-deliver}(V_j) \text{ from some } p_j$$

$$\quad Values_i^{r+1} := Values_i^{r+1} \cup V_j;$$

$$\}$$

After $(f+1)$ rounds

$$\text{Assign } d_i = \text{minimum}(Values_i^{f+1});$$

To reach consensus, each correct process collects proposed values from the other processes. The algorithm proceeds in $f+1$ rounds, in each of which the correct processes *B-multicast* the values between themselves. At most f processes may crash, by assumption. At worst, all f crashes will occur during the rounds, but the algorithm guarantees that at the end of the rounds all the correct processes that have survived will be in a position to agree.

The algorithm, shown in Figure 15.17, is based on that by Dolev and Strong [1983] and its presentation by Attiya and Welch [1998]. Their modified form of the integrity requirement applies to the proposed values of all processes, not just the correct ones: if all processes, whether correct or not, proposed the same value, then any correct process in the *decided* state would choose that value. Given that the algorithm assumes crash failures at worst, the proposed values of correct and non-correct processes would not be expected to differ, at least not on the basis of failures. The revised form of integrity enables the convenient use of the *minimum* function to choose a decision value from those proposed.

The variable $Values_i^r$ holds the set of proposed values known to process p_i at the beginning of round r . Each process multicasts the set of values that it has not sent in previous rounds. It then takes delivery of similar multicast messages from other processes and records any new values. Although this is not shown in Figure 15.17, the duration of a round is limited by setting a timeout based on the maximum time for a correct process to multicast a message. After $f+1$ rounds, each process chooses the minimum value it has received as its decision value.

Termination is obvious from the fact that the system is synchronous. To check the correctness of the algorithm, we must show that each process arrives at the same set of values at the end of the final round. Agreement and integrity (in its modified form) will then follow, because the processes apply the *minimum* function to this set.

Assume, to the contrary, that two processes differ in their final set of values. Without loss of generality, some correct process p_i possesses a value v that another correct process p_j ($i \neq j$) does not possess. The only explanation for p_i possessing a proposed value v at the end that p_j does not possess is that any third process, p_k , say, that managed to send v to p_i crashed before v could be delivered to p_j . In turn, any process sending v in the previous round must have crashed, to explain why p_k possesses v in that round but p_j did not receive it. Proceeding in this way, we have to posit at least one crash in each of the preceding rounds. But we have assumed that at most f crashes can occur, and there are $f+1$ rounds. We have arrived at a contradiction.

It turns out that *any* algorithm to reach consensus despite up to f crash failures requires at least $f+1$ rounds of message exchanges, no matter how it is constructed [Dolev and Strong 1983]. This lower bound also applies in the case of Byzantine failures [Fischer and Lynch 1982].

15.5.3 The Byzantine generals problem in a synchronous system

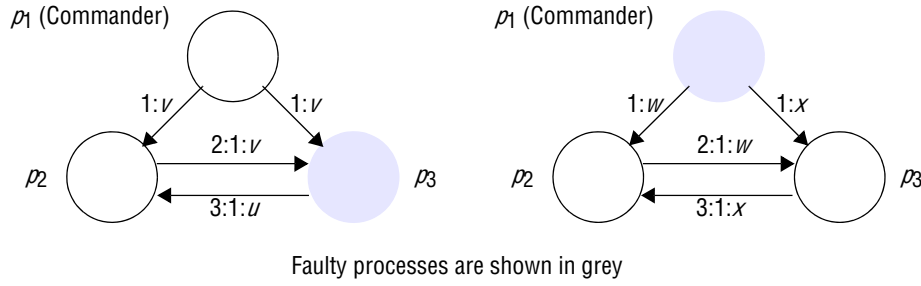
Now we discuss the Byzantine generals problem in a synchronous system. Unlike the algorithm for consensus described in the previous section, here we assume that processes can exhibit arbitrary failures. That is, a faulty process may send any message with any value at any time; and it may omit to send any message. Up to f of the N processes may be faulty. Correct processes can detect the absence of a message through a timeout; but they cannot conclude that the sender has crashed, since it may be silent for some time and then send messages again.

We assume that the communication channels between pairs of processes are private. If a process could examine all the messages that other processes sent, then it could detect the inconsistencies in what a faulty process sends to different processes. Our default assumption of channel reliability means that no faulty process can inject messages into the communication channel between correct processes.

Lamport *et al.* [1982] considered the case of three processes that send unsigned messages to one another. They showed that there is no solution that guarantees to meet the conditions of the Byzantine generals problem if one process is allowed to fail. They generalized this result to show that no solution exists if $N \leq 3f$. We shall demonstrate these results shortly. They went on to give an algorithm that solves the Byzantine generals problem in a synchronous system if $N \geq 3f+1$, for unsigned (they call them ‘oral’) messages.

Impossibility with three processes • Figure 15.18 shows two scenarios in which just one of three processes is faulty. In the lefthand configuration one of the lieutenants, p_3 , is faulty; on the right the commander, p_1 , is faulty. Each scenario in Figure 15.18 shows two rounds of messages: the values the commander sends, and the values that the lieutenants subsequently send to each other. The numeric prefixes serve to specify the sources of messages and to show the different rounds. Read the ‘:’ symbol in messages as ‘says’; for example, ‘3:1:u’ is the message ‘3 says 1 says u’.

In the lefthand scenario, the commander correctly sends the same value v to each of the other two processes, and p_2 correctly echoes this to p_3 . However, p_3 sends a value $u \neq v$ to p_2 . All p_2 knows at this stage is that it has received differing values; it cannot tell which were sent out by the commander.

Figure 15.18 Three Byzantine generals

In the righthand scenario, the commander is faulty and sends differing values to the lieutenants. After p_3 has correctly echoed the value x that it received, p_2 is in the same situation as it was in when p_3 was faulty: it has received two differing values.

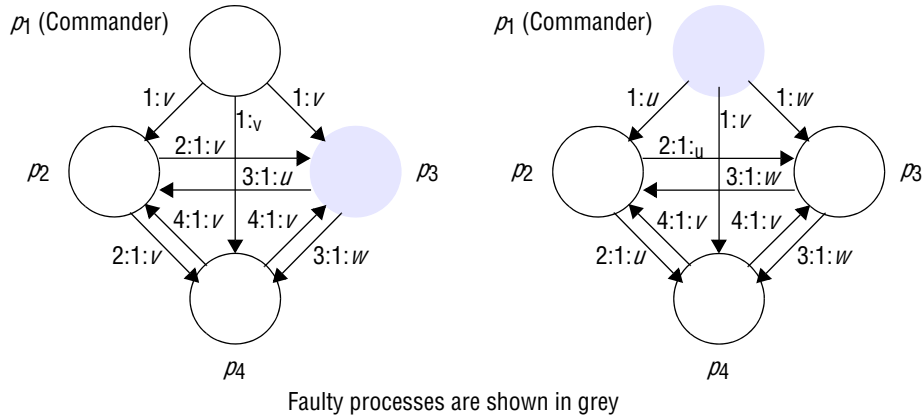
If a solution exists, then process p_2 is bound to decide on value v when the commander is correct, by the integrity condition. If we accept that no algorithm can possibly distinguish between the two scenarios, p_2 must also choose the value sent by the commander in the righthand scenario.

Following exactly the same reasoning for p_3 , assuming that it is correct, we are forced to conclude (by symmetry) that p_3 also chooses the value sent by the commander as its decision value. But this contradicts the agreement condition (the commander sends differing values if it is faulty). So no solution is possible.

Note that this argument rests on our intuition that nothing can be done to improve a correct general's knowledge beyond the first stage, where it cannot tell which process is faulty. It is possible to prove the correctness of this intuition [Pease *et al.* 1980]. Byzantine agreement *can* be reached for three generals, with one of them faulty, if the generals digitally sign their messages.

Impossibility with $N \leq 3f$ • Pease *et al.* generalized the basic impossibility result for three processes, to prove that no solution is possible if $N \leq 3f$. In outline, the argument is as follows. Assume that a solution exists with $N \leq 3f$. Let each of three processes p_1 , p_2 and p_3 use the solution to simulate the behaviour of n_1 , n_2 and n_3 generals, respectively, where $n_1 + n_2 + n_3 = N$ and $n_1, n_2, n_3 \leq N/3$. Assume, furthermore, that one of the three processes is faulty. Those of p_1 , p_2 and p_3 that are correct simulate correct generals: they simulate the interactions of their own generals internally and send messages from their generals to those simulated by other processes. The faulty process's simulated generals are faulty: the messages that it sends as part of the simulation to the other two processes may be spurious. Since $N \leq 3f$ and $n_1, n_2, n_3 \leq N/3$, at most f simulated generals are faulty.

Because the algorithm that the processes run is assumed to be correct, the simulation terminates. The correct simulated generals (in the two correct processes) agree and satisfy the integrity property. But now we have a means for the two correct processes out of the three to reach consensus: each decides on the value chosen by all of their simulated generals. This contradicts our impossibility result for three processes, with one faulty.

Figure 15.19 Four Byzantine generals

Solution with one faulty process • There is not sufficient space to describe fully the algorithm of Pease *et al.* that solves the Byzantine generals problem in a synchronous system with $N \geq 3f + 1$. Instead, we give the operation of the algorithm for the case $N \geq 4$, $f = 1$ and illustrate it for $N = 4$, $f = 1$.

The correct generals reach agreement in two rounds of messages:

- In the first round, the commander sends a value to each of the lieutenants.
- In the second round, each of the lieutenants sends the value it received to its peers.

A lieutenant receives a value from the commander, plus $N - 2$ values from its peers. If the commander is faulty, then all the lieutenants are correct and each will have gathered exactly the set of values that the commander sent out. Otherwise, one of the lieutenants is faulty; each of its correct peers receives $N - 2$ copies of the value that the commander sent, plus a value that the faulty lieutenant sent to it.

In either case, the correct lieutenants need only apply a simple majority function to the set of values they receive. Since $N \geq 4$, $(N - 2) \geq 2$. Therefore, the *majority* function will ignore any value that a faulty lieutenant sent, and it will produce the value that the commander sent if the commander is correct.

We now illustrate the algorithm that we have just outlined for the case of four generals. Figure 15.19 shows two scenarios similar to those in Figure 15.18, but in this case there are four processes, one of which is faulty. As in Figure 15.18, in the lefthand configuration one of the lieutenants, p_3 , is faulty; on the right, the commander, p_1 , is faulty.

In the lefthand case, the two correct lieutenant processes agree, deciding on the commander's value:

p_2 decides on $\text{majority}(v, u, v) = v$

p_4 decides on $\text{majority}(v, v, w) = v$

In the righthand case the commander is faulty, but the three correct processes agree:

p_2 , p_3 and p_4 decide on $\text{majority}(u, v, w) = \perp$ (the special value \perp applies where no majority of values exists).

The algorithm takes account of the fact that a faulty process may omit to send a message. If a correct process does not receive a message within a suitable time limit (the system is synchronous), it proceeds as though the faulty process had sent it the value \perp .

Discussion • We can measure the efficiency of a solution to the Byzantine generals problem – or any other agreement problem – by asking:

- How many message rounds does it take? (This is a factor in how long it takes for the algorithm to terminate.)
- How many messages are sent, and of what size? (This measures the total bandwidth utilization and has an impact on the execution time.)

In the general case ($f \geq 1$) the Lamport *et al.* [1982] algorithm for unsigned messages operates over $f + 1$ rounds. In each round, a process sends to a subset of the other processes the values that it received in the previous round. The algorithm is very costly: it involves sending $O(N^{f+1})$ messages.

Fischer and Lynch [1982] proved that any deterministic solution to consensus assuming Byzantine failures (and hence to the Byzantine generals problem, as Section 15.5.1 showed) will take at least $f + 1$ message rounds. So no algorithm can operate faster in this respect than that of Lamport *et al.* But there have been improvements in the message complexity, for example Garay and Moses [1993].

Several algorithms, such as that of Dolev and Strong [1983], take advantage of signed messages. Dolev and Strong's algorithm again takes $f + 1$ rounds, but the number of messages sent is only $O(N^2)$.

The complexity and cost of the solutions suggest that they are applicable only where the threat is great. Solutions that are based on more detailed knowledge of the fault model may be more efficient [Barborak *et al.* 1993]. If malicious users are the source of the threat, then a system to counter them is likely to use digital signatures; a solution without signatures is impractical.

15.5.4 Impossibility in asynchronous systems

We have provided solutions to consensus and the Byzantine generals problem (and hence, by derivation, to interactive consistency). However, all these solutions relied upon the system being synchronous. The algorithms assume that message exchanges take place in rounds, and that processes are entitled to time out and assume that a faulty process has not sent them a message within the round, because the maximum delay has been exceeded.

Fischer *et al.* [1985] proved that no algorithm can guarantee to reach consensus in an asynchronous system, even with one process crash failure. In an asynchronous system, processes can respond to messages at arbitrary times, so a crashed process is indistinguishable from a slow one. Their proof, which is beyond the scope of this book, involves showing that there is always some continuation of the processes' execution that avoids consensus being reached.

We immediately know from the result of Fischer *et al.* that there is no guaranteed solution in an asynchronous system to the Byzantine generals problem, to interactive consistency or to totally ordered and reliable multicast. If there were such a solution

then, by the results of Section 15.5.1, we would have a solution to consensus – contradicting the impossibility result.

Note the word ‘guarantee’ in the statement of the impossibility result. The result does not mean that processes can *never* reach distributed consensus in an asynchronous system if one is faulty. It allows that consensus can be reached with some probability greater than zero, confirming what we know in practice. For example, despite the fact that our systems are often effectively asynchronous, transaction systems have been reaching consensus regularly for many years.

One approach to working around the impossibility result is to consider *partially synchronous* systems, which are sufficiently weaker than synchronous systems to be useful as models of practical systems, and sufficiently stronger than asynchronous systems for consensus to be solvable in them [Dwork *et al.* 1988]. That approach is beyond the scope of this book. However, we shall now outline three other techniques for working around the impossibility result: fault masking, and reaching consensus by exploiting failure detectors and by randomizing aspects of the processes’ behaviour.

Masking faults • The first technique is to avoid the impossibility result altogether by masking any process failures that occur (see Section 2.4.2 for an introduction to fault masking). For example, transaction systems employ persistent storage, which survives crash failures. If a process crashes, then it is restarted (automatically, or by an administrator). The process places sufficient information in persistent storage at critical points in its program so that if it should crash and be restarted, it will find sufficient data to be able to continue correctly with its interrupted task. In other words, it will behave like a process that is correct, but that sometimes takes a long time to perform a processing step.

Of course, fault masking is generally applicable in system design. Chapter 16 discusses how transactional systems take advantage of persistent storage. Chapter 18 describes how process failures can also be masked by replicating software components.

Consensus using failure detectors • Another method for circumventing the impossibility result is to employ failure detectors. Some practical systems employ ‘perfect by design’ failure detectors to reach consensus. No failure detector in an asynchronous system that works solely by message passing can really be perfect. However, processes can agree to *deem* a process that has not responded for more than a bounded time to have failed. An unresponsive process may not really have failed, but the remaining processes act as if it had done. They make the failure ‘fail-silent’ by discarding any subsequent messages that they do in fact receive from a ‘failed’ process. In other words, we have effectively turned an asynchronous system into a synchronous one. This technique is used in the ISIS system [Birman 1993].

This method relies upon the failure detector usually being accurate. When it is inaccurate, then the system has to proceed without a group member that otherwise could potentially have contributed to the system’s effectiveness. Unfortunately, making the failure detector reasonably accurate involves using long timeout values, forcing processes to wait a relatively long time (and not perform useful work) before concluding that a process has failed. Another issue that arises for this approach is network partitioning, which we discuss in Chapter 18.

A quite different approach is to use imperfect failure detectors, and to reach consensus while allowing suspected processes to behave correctly instead of excluding

them. Chandra and Toueg [1996] analyzed the properties that a failure detector must have in order to solve the consensus problem in an asynchronous system. They showed that consensus can be solved in an asynchronous system, even with an unreliable failure detector, if fewer than $N/2$ processes crash and communication is reliable. The weakest type of failure detector for which this is so is called an *eventually weak failure detector*. This is one that is both:

Eventually weakly complete: Each faulty process is eventually suspected permanently by some correct process.

Eventually weakly accurate: After some point in time, at least one correct process is never suspected by any correct process.

Chandra and Toueg show that we cannot implement an eventually weak failure detector in an asynchronous system by message passing alone. However, we described a message-based failure detector in Section 15.1 that adapts its timeout values according to observed response times. If a process or the connection to it is very slow, then the timeout value will grow so that cases of falsely suspecting a process become rare. In the case of many real systems, this algorithm behaves sufficiently closely to an eventually weak failure detector for practical purposes.

Chandra and Toueg's consensus algorithm allows falsely suspected processes to continue their normal operations and allows processes that have suspected them to receive messages from them and process those messages normally. This makes the application programmer's life complicated, but it has the advantage that correct processes are not wasted by being falsely excluded. Moreover, timeouts for detecting failures can be set less conservatively than with the ISIS approach.

Consensus using randomization • The result of Fischer *et al.* [1985] depends on what we can consider to be an 'adversary'. This is a 'character' (actually, just a collection of random events) who can exploit the phenomena of asynchronous systems so as to foil the processes' attempts to reach consensus. The adversary manipulates the network to delay messages so that they arrive at just the wrong time, and similarly it slows down or speeds up the processes just enough so that they are in the 'wrong' state when they receive a message.

The third technique that addresses the impossibility result is to introduce an element of chance in the processes' behaviour, so that the adversary cannot exercise its thwarting strategy effectively. Consensus might still not be reached in some cases, but this method enables processes to reach consensus in a finite *expected* time. A probabilistic algorithm that solves consensus even with Byzantine failures can be found in Canetti and Rabin [1993].

15.6 Summary

We began this chapter by discussing the need for processes to access shared resources under conditions of mutual exclusion. Locks are not always implemented by the servers that manage the shared resources, and a separate distributed mutual exclusion service is then required. Three algorithms were considered that achieve mutual exclusion: one employing a central server, a ring-based algorithm and a multicast-based algorithm using logical clocks. None of these mechanisms can withstand failure as we described them, although they can be modified to tolerate some faults.

Next we explored elections, considering a ring-based algorithm and the bully algorithm, whose common aim is to elect a process uniquely from a given set – even if several elections take place concurrently. The bully algorithm could be used, for example, to elect a new master time server, or a new lock server, when the previous one fails.

The following section described coordination and agreement in group communication. It discussed reliable multicast, in which the correct processes agree on the set of messages to be delivered, and multicast with FIFO, causal and total delivery ordering. We gave algorithms for reliable multicast and for all three types of delivery ordering.

Finally, we described the three problems of consensus, Byzantine generals and interactive consistency. We defined the conditions for their solution and we showed relationships between these problems – including the relationship between consensus and reliable, totally ordered multicast.

Solutions exist in a synchronous system, and we described some of them. In fact, solutions exist even when arbitrary failures are possible. We outlined part of the solution to the Byzantine generals problem of Lamport *et al.* [1982]. More recent algorithms have lower complexity, but in principle none can better the $f + 1$ rounds taken by this algorithm, unless messages are digitally signed.

The chapter ended by describing the fundamental result of Fischer *et al.* [1982] concerning the impossibility of guaranteeing consensus in an asynchronous system. We discussed how it is that, nonetheless, systems regularly do reach agreement in asynchronous systems.

EXERCISES

- 15.1 Is it possible to implement either a reliable or an unreliable (process) failure detector using an unreliable communication channel? page 632
- 15.2 If all client processes are single-threaded, is mutual exclusion condition ME3, which specifies entry in happened-before order, relevant? page 635
- 15.3 Give a formula for the maximum throughput of a mutual exclusion system in terms of the synchronization delay. page 635
- 15.4 In the central server algorithm for mutual exclusion, describe a situation in which two requests are not processed in happened-before order. page 636

- 15.5 Adapt the central server algorithm for mutual exclusion to handle the crash failure of any client (in any state), assuming that the server is correct and given a reliable failure detector. Comment on whether the resultant system is fault-tolerant. What would happen if a client that possesses the token is wrongly suspected to have failed? *page 636*
- 15.6 Give an example execution of the ring-based algorithm to show that processes are not necessarily granted entry to the critical section in happened-before order. *page 637*
- 15.7 In a certain system, each process typically uses a critical section many times before another process requires it. Explain why Ricart and Agrawala's multicast-based mutual exclusion algorithm is inefficient for this case, and describe how to improve its performance. Does your adaptation satisfy liveness condition ME2? *page 639*
- 15.8 In the bully algorithm, a recovering process starts an election and will become the new coordinator if it has a higher identifier than the current incumbent. Is this a necessary feature of the algorithm? *page 644*
- 15.9 Suggest how to adapt the bully algorithm to deal with temporary network partitions (slow communication) and slow processes. *page 646*
- 15.10 Devise a protocol for basic multicast over IP multicast. *page 647*
- 15.11 How, if at all, should the definitions of integrity, agreement and validity for reliable multicast change for the case of open groups? *page 647*
- 15.12 Explain why reversing the order of the lines '*R-deliver m*' and '*if ($q \neq p$) then B-multicast(g, m); end if*' in Figure 15.9 makes the algorithm no longer satisfy uniform agreement. Does the reliable multicast algorithm based on IP multicast satisfy uniform agreement? *page 648*
- 15.13 Explain whether the algorithm for reliable multicast over IP multicast works for open as well as closed groups. Given any algorithm for closed groups, how, simply, can we derive an algorithm for open groups? *page 649*
- 15.14 Explain how to adapt the algorithm for reliable multicast over IP multicast to eliminate the hold-back queue – so that a received message that is not a duplicate can be delivered immediately, but without any ordering guarantees. Hint: use sets of sequence numbers to represent the messages that have been delivered so far. *page 649*
- 15.15 Consider how to address the impractical assumptions we made in order to meet the validity and agreement properties for the reliable multicast protocol based on IP multicast. Hint: add a rule for deleting retained messages when they have been delivered everywhere, and consider adding a dummy 'heartbeat' message, which is never delivered to the application, but which the protocol sends if the application has no message to send. *page 649*
- 15.16 Show that the FIFO-ordered multicast algorithm does not work for overlapping groups, by considering two messages sent from the same source to two overlapping groups, and considering a process in the intersection of those groups. Adapt the protocol to work for this case. Hint: processes should include with their messages the latest sequence numbers of messages sent to *all* groups. *page 654*

-
- 15.17 Show that, if the basic multicast that we use in the algorithm of Figure 15.13 is also FIFO-ordered, then the resultant totally-ordered multicast is also causally ordered. Is it the case that any multicast that is both FIFO-ordered and totally ordered is thereby causally ordered? *page 655*
- 15.18 Suggest how to adapt the causally ordered multicast protocol to handle overlapping groups. *page 657*
- 15.19 In discussing Maekawa's mutual exclusion algorithm, we gave an example of three subsets of a set of three processes that could lead to a deadlock. Use these subsets as multicast groups to show how a pairwise total ordering is not necessarily acyclic. *page 658*
- 15.20 Construct a solution to reliable, totally ordered multicast in a synchronous system, using a reliable multicast and a solution to the consensus problem. *page 659*
- 15.21 We gave a solution to consensus from a solution to reliable and totally ordered multicast, which involved selecting the first value to be delivered. Explain from first principles why, in an asynchronous system, we could not instead derive a solution by using a reliable but not totally ordered multicast service and the 'majority' function. (Note that, if we could, this would contradict the impossibility result of Fischer *et al.* [1985]!) Hint: consider slow/failed processes. *page 663*
- 15.22 Consider the algorithm given in Figure 15.17 for consensus in a synchronous system, which uses the following integrity definition: if all processes, whether correct or not, proposed the same value, then any correct process in the decided state would chose that value. Now consider an application in which correct processes may propose different results, e.g., by running different algorithms to decide which action to take in a control system's operation. Suggest an appropriate modification to the integrity definition and thus to the algorithm. *page 664*
- 15.23 Show that Byzantine agreement can be reached for three generals, with one of them faulty, if the generals digitally sign their messages. *page 665*