



# PRESENTATION POINTS



# CHARACTERIZATION OF DISTRIBUTED SYSTEMS

## Objectives

To place distributed systems in a realistic context through examples of their application, a summary of their history and a survey of their essential characteristics. To become aware of the inherently distributed nature of many key applications and be able to identify benefits and drawbacks of distribution in real-world applications. To gain a good understanding of the meanings of the terms *resource sharing*, *openness*, *concurrency*, *scalability*, *fault tolerance* and *transparency* as they apply to distributed systems.

## Points to emphasize

The wide scope of distributed applications, as illustrated by the application examples of Section 1.2, and the corresponding need for distributed systems that are general-purpose.

The benefits of distribution and the need to incorporate the key characteristics of Section 1.3 in order to achieve the benefits (e.g. through a discussion of distributed UNIX implementations).

We take a resource-based approach to the definition of many distributed system design problems throughout the book, so the material on resource-sharing, resource managers and object managers in Section 1.3 is important. This also provides an opportunity to link the study of distributed systems to programming methodology through abstract data types and object-orientation.

Transparency in general and the eight forms of transparency defined on pages 20-21. Note that these transparencies are discussed in relation to Sun NFS on pages 223-225.

## Possible difficulties

The biggest difficulty is probably the ‘bottom-up’, hardware-oriented approach that many students will bring from previous courses on computer networks. The chapter is determinedly ‘top-down’; it aims to present a view of distributed systems as the result of a major research, software design and engineering enterprise that has produced solutions to many problems previously considered intractable.

Section 1.3 is inevitably rather abstract and needs illustration, see below.

Section 1.4 on historical development has little conceptual content apart from the cyclic model of the development of technical ideas presented Figure 1.9. Once that has been covered, the section is probably best left for private study.

## Teaching hints

The top-down design ethos is perhaps best conveyed by emphasising that although distributed systems are now pervasive, there are many unresolved design problems, and by showing that some problems cannot easily be solved by bottom-up methods, e.g. openness, scalable naming schemes, fault-tolerant software design.

Since many of the concepts of Section 1.3 are likely to be new to students, they should be illustrated profusely, using the examples from the chapter or better, from local systems that provide suitable illustrations (positive or negative) of the key characteristics.

## Objectives

Like Chapter 1, this is a context-setting chapter. It introduces the key problems of distributed system design and discusses the nature of their solutions. It does not attempt to describe design solutions – these will emerge in later chapters. The objectives are to understand the issues involved in designing general-purpose solutions to the problems of naming, communication, software structure, workload allocation and consistency maintenance in distributed systems and to become aware of the constraints imposed on system designers by key design goals (Figure 2.1) and users' requirements (Section 2.3). Much of the vocabulary for the book will also be acquired from this chapter.

## Points to emphasize

We suggest a design-oriented approach to the chapter, aiming to achieve the realization that the current state-of-the-art in distributed systems is the result of myriad design decisions, not an immutable physical reality.

Thus we would emphasize that the design of computer systems (and distributed systems in particular) involves trade-offs. The design objectives listed in Figure 2.1 are the commodities to be traded. In order to trade them, we must first understand how they are achieved.

For example, communication is not cost-free. Communication times have an important effect on the performance of distributed systems. We must understand that there are many factors affecting the cost of communication beyond simple network performance. Once this has been appreciated, we can consider the trade-offs to be made between communication performance and reliability, scalability or security.

Trade-offs should not be the concern of the users. Users should be offered *guarantees* with respect to the design goals of Figure 2.1 and the user requirements of Section 2.3. The guarantees must be verifiable based on the design characteristics of the system. For example, in Chapter 8 we describe the guarantees of consistency offered by various file services in the face of multiple concurrent updates to a file's data.

## Possible difficulties

The notion of design against specific goals may be unfamiliar. Some students will have difficulty in placing themselves in the position of designers, considering alternative designs and the trade-offs that they represent.

We use the notion of *shared resource* in this chapter, so students should make sure that they have understood its definition in Chapter 1. Other terminology, such as *process* and *name space* may cause difficulties and should be revised if necessary.

## Teaching hints

There are many possible approaches to this chapter. Here we outline a design-oriented approach. This involves taking specific examples of distributed system design problems, considering each of the design issues defined in Section 2.2 and discussing possible trade-offs between the design goals of Figure 2.1 (and possibly some of the user requirements of Section 2.3). Our solution to Exercise 2.1 illustrates the application of this approach to the design of NFS.

Other exercises and examples of design problems in distributed systems or applications can be introduced; for example, the design of simple text-based conferencing system, a multimedia conferencing system or a network information service (similar to World Wide Web) could be considered. A design project such as those proposed in the project work section of this Guide will give students practical experience with designing against specific goals and trading-off goals against each other.

## Objectives

The chapter does not assume any prior knowledge of computer networks, but the objectives and pace of study will differ considerably for students who have taken an earlier networks course and those who have not.

*For students who have not previously studied computer networks:* To provide an understanding of the basic concepts and principles of local and wide-area computer networking and their extension to internetworking. The intention is to achieve comprehension of the principles of operation of the major network technologies at a level of detail sufficient to assess the impact of networks on distributed system performance and reliability and to evaluate and compare different network technologies for use in distributed systems.

*For students who have previously studied computer networks:* To recapitulate and update their knowledge of the principles of operation of the major network technologies, including internetwork technologies. To relate that knowledge to the requirements of distributed systems. To consider the special requirements of distributed systems for simple, efficient protocol stacks.

## Points to emphasize

Latency of communication is at least as important for distributed system performance as data transfer rate.

The performance of distributed systems is determined by the speed and latency of end-to-end communication (sending process to receiving process) and not just the network performance. The software delays in the sending and receiving machines nearly always swamp the network transmission delays in determining the speed of communication, except for very heavily loaded networks or very high bandwidth applications.

Networks are highly reliable, whereas client and server computers and their software often aren't, so error detection and recovery is best performed end-to-end at the highest feasible level.

The OSI Reference Model and the ISO protocol standards that are based on it are a useful reference point, but not much used in practice. The TCP/IP and UDP suites are currently dominant, even for distributed systems implemented solely over local area networks. Since these are internetwork protocols, it is important to emphasize the layering of IP (the network layer protocol of the Internet) over various network-layer protocols for different physical networks.

To a considerable extent, the choice of underlying protocols is unimportant once good RPC and multicast communication services are available. But the FLIP case study in Section 3.5 and the description of the Firefly RPC system in Section 18.8 show how a root-and-branch approach to the design of a protocol stack optimized for distributed system requirements can yield optimal performance.

## Possible difficulties

Even students who have previously studied computer networks may not be very familiar with internetworking concepts and principles.

The conceptualization of the basic communication models – store-and-forward packet routing, broadcast with collisions (CSMA/CD) and without (the various ring technologies) – seems to cause some difficulty to students without previous networking knowledge. This leads to difficulty in relating the effect of a particular communication model on the potential performance of distributed systems – for example, routing delays in store-and-forward networks are variable and can be quite large. These distinctions should be emphasized and related to the case studies – for example, routing delays are still variable in ATM networks, but much smaller.

## Teaching hints

Devote plenty of time to the internetwork case studies. Comer [1991] is an excellent source of additional material and exercises on internetworking.

## Objectives

To study the building blocks for lightweight interprocess communication protocols for a distributed system. To describe Request-Reply protocols (RPC and language integration are left until Chapter 5). To introduce the main options for group communication (the implementation is left until Chapter 11 and Section 18.9).

## Points to emphasis

The material in Chapter 3 is relevant for networking, where the emphasis is on a single interconnection between a pair of components. Chapter 4 is concerned with distributed systems in which there are many components and interconnections and the emphasis is on the logical relationship between components.

Message passing is the best building block for lightweight interprocess communication protocols because it carries the minimum possible overheads for the resulting protocols.

System reconfigurability requires location independent identifiers for message destinations.

Request-reply protocols are designed to support client-server communication in which a client in an active role asks a passive server to perform an operation and return the result. This relationship determines the protocol as follows: (i) the client needs one primitive (*DoOperation*) and the server needs two (*GetRequest* and *SendReply*); (ii) since clients normally wait for replies *DoOperation* is synchronous; (iii) a server must be able to receive *GetRequest* messages while performing operations. To deal with failure, requests are re-transmitted. To ensure that an operation is performed at most once, duplicate requests are filtered and replies are saved for re-transmission.

Communication between the members of a group of processes is the basis of replicated services. Atomic multicast ensures that all replicas will receive the same message. It is not simple to implement because the originator may fail. Totally-ordered multicast ensures that all replicas receive messages in the same order. Recipients ‘hold back’ messages so as to deliver them in the correct order. Protocols entail considerable latency, large numbers of messages and storage costs. Causally ordered multicast is less expensive, because ‘hold-back’ is not needed.

## Possible difficulties

Students will already know about network ports from networking courses (or from Section 3.3). They may not perceive that process ports are different. Unfortunately, the Unix case study may reinforce such misconceptions. When students consider the design of a Request-Reply protocol, they are often unsure whether messages are queued at server ports by the underlying communication service (e.g. UDP)

Although the request-reply protocol appears to be straightforward, students do not always appreciate the effort required by the protocol to ensure that an operation is performed at most once. This is important for Chapter 5.

In our experience, students find it hard to understand the important options for multicast – in particular, they do not appear to register the forward reference to causal ordering.

## Teaching hints

The approach taken is to consider the roles of processes and the interactions between them in client-server communication and group multicast. The students could be asked to go back to Section 2.2. To widen the discussion, other patterns of communication such as producer consumer or peer groups of servers could be discussed. Revise location transparency (see Section 1.3) before discussing location independent destination identifiers. The details of the naming aspects of identifiers are deferred until Section 6.4.

Before discussing the request-reply protocol, consider all the effects of client or server failing independently. Assume that client-server communication is synchronous (like a procedure call) and avoid the issue of calls not requiring replies until Section 5.5.

For totally-ordered multicast, emphasize the fact that messages from different originators may arrive at common destinations in different orders. Use this to motivate the need for globally ordered message identifiers.

## Objectives

To study the integration of synchronous RPC into a programming language and to study its implementation by means of two case studies. To introduce asynchronous RPC. The performance of RPC is left to Section 6.5.

## Points to emphasize

An RPC system runs over a Request-Reply protocol and can be integrated relatively transparently with a programming language. The semantics of parameter passing is adjusted to suit the separation of client and server. Programmers must avoid passing addresses and using global variables and must be aware that new sorts of errors can arise due to distribution.

Call semantics cannot be 'exactly once'. The semantics achieved depends on the approach to dealing with failures.

Services are encapsulated and their clients interact with them only via interfaces. Servers are designed, implemented and installed before clients can use them. Client programs access a service by means of RPCs to operations in its interface.

A 'service interface' defines the signatures of the procedures in a service. It forms the basis for generating 'client stub procedures' and marshalling procedures which together enable RPCs to have access transparency. It can also be used to generate server stub procedures and dispatcher.

Late binding of a client to a service is essential. This is achieved with the help of a binder which can also provide location transparency. Incremental development of services is supported by allowing for the selection of a versions. A binder should be relocatable.

The two case studies illustrate approaches to the design of an RPC system. It is worth discussing the different approaches to an interface definition language and a binder. These studies also provide an opportunity to discuss issues of access and location transparency.

Synchronous RPC may not provide adequate performance for some applications. Asynchronous RPC and buffered calls enhance the throughput. If the two are integrated in single communication mechanism (e.g. the call stream), servers need not be aware of whether their clients are making synchronous or asynchronous calls.

## Possible difficulties

In our experience, students find it hard to appreciate the variety of call semantics and how to achieve each one.

The section on asynchronous RPC is rather specialized and could be omitted without having any effect on the understanding of later chapters.

## Teaching hints

The approach taken is to consider RPC as a transparent form of local procedure call. If necessary revise the following topics which should have been covered in earlier courses on programming and operating systems. (i) the semantics of local procedure call and parameter passing; (ii) the use of modules with interfaces; (iii) the use of the standard input/output library in UNIX as an application programming interface to the system calls; (iv) the way that application programs handle the -1 error returns of UNIX system calls.

Access and location transparency could be revised (from Section 1.3). The students could be asked to consider the measures taken to deal with failures in the Request-Reply protocol from Section 4.3.

Practical work involving either (i) the use of an RPC system (e.g. ANSA or Sun RPC) or (ii) the implementation of a simple RPC system over UDP would help to reinforce this chapter. (See Project Work).

## Objectives

To reinforce the notion of a distributed system as a collection of resources managed by kernels and servers, and to identify the infrastructure requirements necessary for interworking between and within computers. To explain the rationale for microkernels and compare them to monolithic kernels. To understand the need for multi-threaded processes. To identify invocation mechanisms and appreciate their importance and costs. To examine required memory management features, including virtual memory. To prepare the ground for discussion of services and distributed shared memory described later in the book.

## Points to emphasize

Very few distributed systems are controlled by a homogeneous operating system. There is not always a clear distinction between the operating system and the applications that utilize it. The material in this chapter deals with an infrastructure for process management, memory management and invocation, on which further services can be based.

The relative advantages and disadvantages of microkernels and monolithic kernels should be given lively and critical discussion, including discussion of which features covered in the chapter, such as multi-threading, can and should be bolted on to conventional monolithic kernels. Microkernels have yet to be proven in large-scale commercial use.

Threads are actually familiar ‘processes’ from operating systems courses, which usually consider processes that can share memory. Threads are not a fundamentally new concept, but they are important for building efficient client-server systems.

Invocation is more than just communication. The need to handle multimedia data, in particular, shows that distributed operating systems have to provide more than just RPC. The cost of local invocations is significant. Communication is the key to heterogeneous interworking.

Large, sparse address spaces and copy-on-write memory sharing are needed in all modern operating systems, and are not connected with distribution *per se*. Virtual memory management has had to be re-thought for distributed systems; this has given us an opportunity to implement distributed shared memory.

## Possible difficulties

This chapter assumes a reasonable grasp of first courses in computer architecture and (centralized) operating systems.

Students seem to be able to appreciate the need for threads in servers, but examples of multi-threaded applications will help them appreciate the need for threads in clients.

The material requires illustration wherever possible with practical problems in server design. Chapters 7, 8 and 9 will reinforce the issues for file and name services.

It is possible to omit Section 6.6. on virtual memory, but it is needed for Chapter 17 on distributed shared memory and the case studies in Chapter 18.

## Teaching hints

Review the material on software structure in Section 2.2.

Link the material to current developments in commercial operating system development, for example Microsoft’s Windows NT and IBM’s Workplace OS.

One or more of the case studies in Chapter 18 can be used to illustrate the material.

## Objectives

The design of distributed file systems has been extensively researched since the earliest days of distributed system development (see Figures 1.10 and 1.11). Several successful products have emerged, with which many students will be familiar.

These products are largely based on a standard set of concepts and techniques. The purpose of this chapter is to lay down a framework for the discussion of distributed file systems that is somewhat abstracted from the products, enabling students more easily to understand their design and to compare and evaluate them. We achieve this by presenting a 'basic model' that embodies most of the concepts found in currently available distributed file service products.

After studying this chapter, students should have sufficient understanding of distributed file service design and implementation to design and construct a practical file service. A secondary objective is to consider in detail the design of a particular distributed service, applying the principles learned in all preceding chapters.

## Points to emphasize

The key role of file servers means that failure transparency is important and that the performance of file servers under load must be good.

**File service components:** The role of each of the components – flat file service, directory service and client module; the division of responsibilities between the components results in an exposed interface - the flat file service - that is hidden in conventional operating systems. The useful separation of concerns, openness and modularity achieved by the division of responsibilities. A forward reference can be made to the similarity that this structure bears to NFS (where the directory service can be seen as integrated with the client module, see Chapter 8)

**Design issues:** The need for unique file identifiers; file attributes; the benefits of statelessness.

**Interfaces:** The absence of an *Open* operation; the idempotent nature of all of the interfaces; comparison of the flat file service with Unix system calls; method for interpreting a multi-part pathname.

**Implementation techniques:** File groups; the significance of file groups is that they provide a basis for the distribution of groups of files between different servers and their replication (described in the Coda case study in Chapter 8).

Capabilities; space leaks; construction of UFIDs and their role in access control; file location methods; server cache; client cache; the practical importance of caching at both the server and the clients and the problems of consistency maintenance raised by client caching.

## Possible difficulties

The most difficult portion of the chapter is probably the material on the construction and use of capabilities for access control, pp. 212-215. Capabilities are introduced here because the model relies on their use to implement file protection but they are of much wider applicability; they are an important tool for open system design and are used extensively in modern microkernel-based systems. Their use in Chorus and Amoeba is discussed in Chapter 18. If students have difficulty with capabilities here, they should be recapitulated when Chapter 18 is studied.

## Teaching hints

The service interfaces defined in this chapter are quite straightforward. They are amongst the first service interfaces that the student will have encountered (there is one earlier example in Chapter 5 (Figure 5.3) and several others in later chapters); we recommend a review of these file service interfaces, comparing them with the Unix file system primitives (see page 206) and emphasizing their support for the design goal of stateless servers.

The model presented in this chapter is sufficiently detailed for it to be used as the basis for a coursework exercise on the implementation of a file service.

## Objectives

To show that the designs of NFS and the other file services discussed in this chapter are in accordance with the principles of distributed system design studied in earlier chapters and with the basic model for file servers presented in Chapter 7, thus grounding that material firmly in examples of real systems.

To provide a sufficiently detailed description of the file services covered to enable students to understand and evaluate their practical behaviour obtain an understanding of the reasons for the differences between their designs and for their success, commercial or otherwise, based on their design goals.

To give an early appreciation (through the study of Coda in Section 8.4) of the issues involved and the solutions available for the replication of data between servers. Replication will be covered more generally in Chapter 11.

## Points to emphasize

The three systems studied in this chapter provide examples of the fact that designs for services can differ radically even when the application programming interface is almost identical. It is for this reason that the design of distributed systems remains an important and interesting activity.

Meaning of one-copy file semantics and the problems that this introduces for caching and replication.

**NFS:** Remote mounting - what it is, and what it means for location transparency; statelessness of the NFS interface - benefits and drawbacks; software components and architecture - why it is implemented in the kernel; VFS - its role; path name translation; mount service; caching - implications of updates and cache consistency; performance. See also [Pawlowski *et al.* 1994]\* for a description of recent revisions and performance improvements, and [Duchamp 1994] for a discussion of a method for optimal pathname lookup.

**AFS:** Whole-file serving and caching - compare with ftp; observations of UNIX file usage - especially average and maximum file sizes and locality of reference to files; software structure - roles of Vice and Venus, user-level processes; Vice as a flat file service, with directory structure implemented in Venus - FID structure; implementation of file system calls in AFS.

Callback promises - comparison of AFS-2 with AFS-1; Vice service interface; update semantics - for AFS-1 and AFS-2, and semantics while file is open, potential lost updates; Unix kernel changes; read-only replicas; performance.

**Coda:** Lessons learned from AFS - requirements for scalability, fault-tolerance, detached working; disconnected operation - user-generated list of required files; VSGs and AVSGs - broadcast of updates to AVSG on close; optimistic replication strategy - use of CVV, with example; update semantics; use of multicast RPC to replicate updates; cache coherence - dealing with changes to the AVSG and lost callbacks; disconnected operation; performance.

## Possible difficulties and teaching hints

Coda version vectors usually cause difficulty on first reading. They are a special case of the vector timestamps found in Chapter 11.

Note that this chapter is by no means exhaustive in its coverage of research on file service design; many other designs have been developed and reported, some with novel approaches and solutions to problems that go beyond what can be achieved in a straightforward emulation of the Unix application programming interface. Had space and time allowed, we might have included a discussion of log-based file server design [Rosenblum and Ousterhout 1982] and of the recently-reported file services designed to support multimedia applications by the inclusion of Quality of Service specifications for time-based data [Anderson *et al.* 1992, Lo 1994].

---

\* See Additional References.

## Objectives

To give the reader an appreciation of the importance of naming as an issue in distributed systems. To understand the issues relevant to the design and implementation of a name service, including: the name space; the resolution mechanism; the division and replication of naming data between servers; and attribute caching. To understand the key features of the DNS, GNS and X.500 name services.

## Points to emphasize

Naming is intimately related to network (access and location) transparency and migration transparency.

We need a separate name service in addition to the naming schemes used in the context of specific services, for the reasons given in Section 9.1.

There are separate concerns to be addressed: designing the name space; meeting administrative requirements to partition the name space; and creating an implementation that scales.

A system such as the DNS that resolves names on a world-wide scale represents a considerable engineering achievement – as does the system of routing daemons that resolves IP addresses. Students should be given a feel for the engineering problems and strong motivation for getting naming right.

The SNS is only a paper design; encourage students to criticize it and think up alternatives.

Caching and replication are key to name service implementation.

## Possible difficulties

Students may be confused because a name service appears to be just a database service. Point out the ways in which a name service differs from a conventional database: the resolution mechanism, the physical scale of its use, and the slow-changing nature of the data.

Students sometimes find confusing the models of navigation outlined in Sections 9.2 and 9.3 and discussed in the DNS case study. They should be encouraged to think of reasons to use one method rather than another.

## Teaching hints

Review the brief discussion of naming in Sections 2.2 and 6.4. Review the way names arise in interprocess communication and remote procedure call. Encourage the students to think of examples of how naming is important in other areas of Computer Science, for example program compilation and linking.

It is advisable to cover at least DNS as a case study, since this will be familiar to most students.

Encourage use of *nslookup* to query local DNS servers, and querying of X.500 servers.

Ask students to investigate and analyse the Uniform Resource Locators used in the World Wide Web.

## Objectives

To give students an appreciation of why we need synchronized clocks in distributed systems, and of the variability in network delays that stands in the way of accurate synchronization. To understand the key features of Cristian's synchronization algorithm, the Berkeley algorithm and the Network Time Protocol. To understand the utility of logical clocks, the rules for updating them and their limitations. To introduce algorithms for distributed mutual exclusion and election algorithms.

## Points to emphasize

The lack of universal time is a fundamental characteristic of loosely-coupled distributed systems. However, it is possible to synchronize clocks to reasonable degrees of accuracy, which are adequate for many purposes. Internal synchronization algorithms synchronize clocks together, without reference to UTC; external synchronization algorithms synchronize them to UTC.

Where clock accuracy is an important requirement rather than a convenience (that is, in real-time distributed systems), a more critical analysis is required than that given in the chapter – in particular where clocks may report false values, and where time servers must be assumed to fail occasionally. Questions then arise about the guarantees that can be placed on clock values returned to the user.

Logical clocks enable us to reason about the order in which events occur without reference to real time. The happened-before relation captures events that are related at the level of the application semantics; unfortunately, it can also capture events that are unrelated at this level. The students should appreciate that sometimes hidden causal channels (such as telephones) exist, which do not appear in our system model.

Distributed mutual exclusion is needed when processes access a resource or collection of resources and we require their updates to be consistent. It is preferable for the service that manages the resources to provide mutual exclusion itself, since this does not require extra communication; but file servers, for example, do not provide synchronization and we require a separate synchronization service. Algorithms to achieve distributed mutual exclusion must be evaluated on the basis of their efficiency and their behaviour under failure conditions.

Election algorithms are required when there is a need to distinguish one of a collection of processes as a coordinator. These algorithms are designed to select a unique process, even under failure conditions.

Unfortunately, network partitions may mean that a collection of processes is split into several subgroups, each of which cannot know whether the other continues to function.

## Possible difficulties

Students may find it hard to see the point of logical clocks. Concentrate on the happened-before relation, show the application of logical clocks in Ricart and Agrawala's algorithm, and encourage students to think of examples of actions that depend on prior actions.

## Teaching hints

Review Coda's version vectors in Section 8.4 as measures of logical time.

Encourage students to state and prove the properties of the algorithms mathematically (see [Raynal 1988]).

Get the students to investigate the design of UNIX file locking, which is implemented in distributed systems via the *lockd* daemon.

If you have already introduced the concept of distributed shared memory or talked about shared files, point out that these require a distributed mutual exclusion service.

## Objectives

To appreciate the advantages and costs of replicating data: the potential improvement in response times and reliability, and the extra communication costs involved in keeping data consistent. To understand the concepts of atomicity and causal, total and sync-ordering, and to appreciate when they are required. To understand the gossip architecture and process group systems (ISIS in particular): their functionality and implementation.

## Points to emphasize

Replication should be transparent to clients of a replicated service, but it is not normally transparent to front ends and replica managers.

It is applications that generate ordering requirements; a system for replication needs to provide a toolkit of causal, total and sync-ordering for implementors to meet application requirements. Sync-ordering is not as fundamental as causal and total ordering, but is needed for events such as group joins. The systems in this chapter only order individual operations: transactions (Chapter 12) are needed to serialize groups of operations.

Vector timestamps are natural generalizations of logical clocks. Unlike the latter, they accurately reflect causality or concurrency through the partial order we have defined on them. Total ordering is most easily obtained through use of a sequencer, but the disadvantages of this approach make it desirable to find other methods (for example, the original ISIS protocol and, for further study, the Trans protocol [Melliard-Smith *et al.* 1990]<sup>†</sup>).

The gossip architecture is not widely used but was chosen for its elegant implementation of ordering constraints and to show that, in general, replication is about more than multicast. The architecture assumes that the timeliness of delivery is not critical and that it is acceptable always to perform reads and updates separately (see the answer to Exercise 11.20). These conditions do not always meet application requirements. Students should be encouraged to question them. Apart from its use of multicast, ISIS differs from the gossip architecture mainly in its notions of group membership and virtual synchrony. Multicast and group membership protocols are tightly coupled, because of virtual synchrony.

The chapter has not seriously dealt with network partitions, and the students should be aware of this gap.

## Possible difficulties

The bulletin board example is used as a familiar example for illustration only. It should not be taken too literally as an application with definite ordering requirements, or as one for which the protocols in this chapter are suited. Over a wide area network, a lightweight approach is needed.

Students find total ordering reasonably straightforward to understand, but they find it hard to see the motivation for causal ordering at first.

The concept of virtual synchrony is a difficult one for somebody who has not tried to program an application that relies upon it.

## Teaching hints

Relate the material on vector timestamps to that on logical clocks in Chapter 10 and to Coda Version Vectors in Chapter 8.

Make vector timestamps concrete by pointing out their role as counters of messages. More ambitiously, it may help (Masters!) students to gain a more concrete idea of vector timestamps if they work through their application to consistent global states (see the answer to Exercise 10.10). See [Mattern 1988]\* and Chapter 4 of [Mullender 1993]\*.

If you have the ISIS toolkit installed, give the students some exercises using it. At least get them to go through a paper design (see the Projects section).

---

<sup>†</sup> See Additional References.

## Objectives

To study the design of services whose long-lived data items are intended to be shared by multiple clients. To establish a model of a single-process server whose data items are abstract data types. To set up the all-or-nothing and isolation properties of transactions within this model, so as to be able to study how they are maintained in the presence of concurrent clients and server failures. Concurrency control is studied in detail in Chapter 13. Distributed and replicated services are left until Chapter 14. Recovery is left until Chapter 15.

## Points to emphasize

In our model, a server provides operations on its data items which can be made long lived by keeping copies of its data items in a recovery file. Mutual exclusion mechanisms can be used to make the operations atomic and the *Signal* and *Wait* operations can be used to synchronize the operations of different clients.

A transaction consists of the execution of a sequence of client requests that access or update one or more of the data items. A transaction may commit or abort.

Transactions have two crucial properties: the ‘all-or-nothing’ property and the ‘isolation’ property.

The ‘all-or-nothing’ property requires that the effects of transactions that commit must be made permanent and when a transactions aborts, the result must be the same as if it had never existed. Recovery is concerned with ensuring the ‘all-or-nothing’ property which has two aspects: failure-atomicity and durability. Failure-atomicity requires that the ‘all-or-nothing’ property is maintained in the presence of server crashes and concurrent transactions that may abort at any time.

Concurrency control deals with the isolation property by ensuring that the effects of concurrent transactions are isolated from one another.

Serial execution of transactions maintains isolation. But doing one transaction at a time does not provide adequate performance, particularly for interactive applications.

Serial equivalence is defined (on page 364) as interleaving transactions having the same effect as if they are done one-at-a-time. To achieve it, the accesses to a data item should be serialized with respect to accesses by other transactions. This requires that all pairs of conflicting transactions should be executed in the same order.

Strict executions ensure the isolation property in the presence of aborting transactions.

## Possible difficulties

Students seem to find it difficult to think about servers of ‘data items’ with their own operations after having thought about file server.

The idea of serial equivalence is difficult. It might help to look ahead at the definition of conflicting operations (on Page 379).

## Teaching hints

Motivate the idea that transactions are necessary, i.e. that a service interface cannot always provide atomic operations which correspond to all of a client’s requirements for atomicity. In the Bank service example, the provision of a ‘Transfer’ operation might be useful, but would not remove the need for transactions in Figure 12.1.

The ACID mnemonic (page 360) is a useful teaching aid.

It is important for Chapters 13-15 that students should understand serial equivalence, strict executions and cascading aborts. Students could be asked to discuss solutions to exercises 12.6, 12.7, 12.9, 12.13 and 12.14.

The synchronisation between clients operations in a server provides a useful background for thinking about the implementation of locks. The exercises 12.3 and 12.4 address this issue. A practical exercise on the same topic is provided in Project Work.

Nested transactions could be omitted.

## Objectives

To study the three main approaches to concurrency control, all of which maintain the isolation property of transactions in the presence of their concurrent execution at a single server. The extension of these concurrency control methods to distributed services is left until Chapter 14.

## Points to emphasize

Two-phase locking uses locks on data items to ensure serial equivalence of transactions. The operation conflict rules give us the lock compatibility tables. Rules must be added for lock promotion. Strict two-phase locking prevents cascading aborts. Lock managers should use threads and synchronisation operations. Any locking scheme must address the issue of deadlock.

Deadlock detection is relatively simple at a single server.

Optimistic concurrency control assumes that conflict is unlikely and allows concurrent transactions to perform their operations on tentative versions without checking for conflicts between them. Each transaction is validated before it is allowed to commit. If the validation fails, the transaction is aborted. Starvation is an associated problem. The validation rules are derived from operation conflict rules. These rules are simplified by allowing only one transaction at a time to validate and write its permanent data items. There are two alternatives: Backward validation and forward validation.

Timestamp ordering ensures serial equivalence by ordering transactions according to the times at which they start. Each request to perform an operation is validated for its conflicts with other operations on the same data item. If the validation fails, the transaction is immediately aborted. Strict executions may be ensured by delaying *Read* operations. Deadlock cannot occur.

## Possible difficulties

Some of the ideas are quite subtle, in particular the interaction between two-phase locking and serial equivalence and the interaction between strict executions and recovery. These ideas can be tested in Exercises 13.1-13.6.

Although the idea of read/write locks is relatively straightforward, students experience difficulty in implementing the lock manager outlined in Figure 13.5. The task becomes even more difficult when lock timeouts are added.

The section on locking may be taught on its own – it is independent of the rest of the chapter.

## Teaching hints

The approach is to consider transactions at a server of data items following the model established in Chapter 12. The methods for concurrency control are based on those used in databases but they are valid whether the data items are permanent or not.

To simplify the discussion of concurrency control, it is assumed that the high-level operations provided by a service can be translated into *Read* and *Write* operations on simple data items. We do not discuss semantic locks.

The three methods for concurrency control are derived from the interaction between operation conflicts and serial equivalence. It may help to discuss these ideas.

The students could be asked to discuss the provisions for concurrency control in the file services discussed in Chapters 7 and 8 which were designed under the assumption that most files are read and written by one user.

## Objectives

To extend the ideas of Chapter 12 to deal with distributed transactions. To extend the three methods of concurrency control given in Chapter 13 for use with multiple servers. To study how distributed deadlock may be detected To study transactions with replicated data.

## Points to emphasize

The two phase commit protocol ensures that all the servers in a transaction reach the same decision: to commit or to abort. Timeout actions are included in case servers fail. The protocol has considerable communication costs and can cause severe delays.

Distributed concurrency control is based on local concurrency control at each server. Locks will be held at each server until the coordinator announces the decision and the local commit is completed. As distributed deadlock detection is complex, timeouts on locks are a useful alternative.

A centralized solution to detection of distributed deadlocks is not scalable. Phantom deadlocks are a problem with non-centralized approaches to detection. Edge chasing is a technique for finding cyclic dependencies without storing the entire wait-for graph.

A service with replicated data items provide replication transparency. Interleaved operations on replicated data items should be equivalent to serial executions on a single copy. One-copy serializability is the goal for concurrency control on replicated data items.

Failures and recoveries must be serialized with respect to transactions – see available copies replication.

Replication schemes must deal with network partitions. Pessimistic schemes permit operations to continue in only one partition. Quorum consensus schemes allow partitions to decide independently which one can continue. The virtual partition algorithm combines the performance of available copies replication with the ability to handle network partitions.

## Possible difficulties

Students find it hard to remember that servers cannot change their mind after having agreed to commit in the two-phase commit protocol.

The distributed deadlock algorithms are very complicated – they could be omitted.

The application of one-copy serializability in the available copies algorithm is hard.

## Teaching hints

The approach is to consider an architecture for a distributed transaction in which the servers operate independently until the transaction is ready to commit, at which point a coordinator is required. Students could be asked to say why atomic commitment is difficult and to suggest a solution. They can be asked to consider each of the places in Figure 14.6 where a timeout might be required and to consider the problem of coordinator failure. Discussions of alternative protocols can be based on Exercises 14.1-14.4. (Recovery of the two-phase commit protocol is left for Chapter 15).

The idea of independent servers can be extended to the discussion of distributed concurrency control.

Locking, optimistic concurrency control and timestamp ordering should be revized before asking students to say how serial equivalence between independent servers can break down in each method. Discussion may be based on Exercises 14.6-14.10.

This discussion can be extended to discuss how distributed deadlock arises.

The architecture for transactions with replicated data can be compared with the architectures of Chapter 11. The algorithms for quorum consensus and virtual partition can be reinforced by Exercises 14.14-14.16.

## Objectives

To study how the all-or-nothing properties of transactions can be ensured in the presence of server failures.

To introduce other approaches to fault tolerance in distributed systems.

## Points to emphasize

Recovery is concerned with ensuring that a transaction service has the properties of failure atomicity and durability.

The recovery file consists of a log of all the transactions at a server and can be used to restore its data items. Checkpoints are used to reduce the size of the recovery file.

If the two-phase commit protocol has reached a decision to commit, the servers involved must complete it even if they fail repeatedly. The recovery file includes entries to ensure that this is possible.

Transaction recovery can take unbounded time and is not appropriate for services with real-time requirements.

The assumption that a server is fail-stop simplifies the design of components that use it. The fault model for transactions assumes that servers are fail-stop. For life-critical systems, it is safer to assume that arbitrary (or Byzantine) failures may occur.

The specification of a service should include its failure semantics. The classification of faults (e.g. response, timing, crash, arbitrary) is an aid to describing failure semantics.

Fault-tolerant services are implemented as groups of servers running on different computers. Closely synchronized groups can provide a continuous service even when some servers fail and they can mask Byzantine failures.

Loosely synchronized groups (e.g. primary/backup) are less expensive in processing resources but recovery can cause delays. They cannot be used to mask Byzantine failures.

## Possible difficulties

Understanding fault-tolerant distributed systems is very difficult because complex situations can arise when components fail.

Earlier chapters have assumed that value failures are masked and students may have difficulty in believing that a service can be anything other than fail stop. It may help to give an example of a component that breaks down and then emits garbage signals, rather than failing quietly. Difficulties can also arise with understanding timing failures, because students' experience is mainly with systems like Unix that do not offer real-time guarantees.

The difficulties with arbitrary failures and timing failures can lead to further difficulties in assessing the usefulness of closely synchronized group services. (see Exercise 15.13, 15.14)

## Teaching hints

The approach is that transaction recovery is well understood and has a well-defined fault model, but is not suitable for applications with strict requirements for performance and reliability in the presence of arbitrary faults.

Revise Section 12.4 on the all-or-nothing properties of transactions and motivate the need for an intentions list. To explain the logging technique, discuss the transfer of the intentions list and associated data items to the recovery file. emphasize that entries from different transactions can be interleaved – see Exercises 15.1- 15.5 which also explore the interaction of recovery with timestamp ordering and optimistic concurrency control.

Revise the two-phase commit protocol and ask students to suggest what must be written to permanent storage at each step in Figure 14.6. Then compare the results with the table in Figure 15.3 and do Exercise 15.7.

Study the fault model for transactions, the primary/backup service and stable storage to illustrate the point that services should have well-defined failure semantics, given in terms of the classes of faults (pages 462-464).

Use the Auragen example on page 469-71 to illustrate the technique of primary and backup server and emphasize the point about not re-sending messages during recovery. Compare with master/slave (see Exercise 15.16).

## Objectives

To introduce security threats and demonstrate how the openness of distributed systems renders them vulnerable to attack unless security measures are incorporated at the system level. To review the security technologies developed to date: encryption, authentication and digital signatures, and to illustrate their application in general-purpose security tools. To illustrate the usefulness of formal methods in validating security procedures and protocols.

## Points to emphasize

Most serious applications of distributed systems demand secrecy and/or integrity. The design of secure applications requires an analysis of the security of all underlying system components.

The techniques in this chapter offer a reasonable basis for the design of secure systems, but great care is required to avoid loopholes. Formal methods for the verification of security protocols are still under development.

## Possible difficulties

The material is not inherently difficult, but there are many new concepts (many of them from the cryptography and security world) and some new notation. Taken all together, they are somewhat daunting. There may be something to be said for teaching it in two or more chunks, allowing time for the concepts and notation to sink in after the first chunk.

## Teaching hints

If a two-part approach is desired, a break could be taken after Section 15, leaving all aspects of authentication and digital signatures to a later stage.

Some exercises such as 16.1, 16.2 and 16.4 may help to bring the practical implications of security and the lack of it home.

## Objectives

To understand the motivation for preferring the shared memory programming model to message passing, but at the same time to appreciate its limitations. To understand the concept of distributed shared memory, and in particular the page-based and library-based approaches to it. To appreciate the design and implementation issues associated with DSM. To be familiar with write-update and write-invalidation protocols. To understand Ivy's use of write-invalidation to achieve sequential consistency. To appreciate the costs involved in implementing sequential consistency, and therefore the motivation for release consistency and other weaker forms.

## Points to emphasize

The goal of porting programs written for shared memory multiprocessors to clusters of workstations or distributed memory multiprocessors, using the DSM abstraction, is not straightforward. The programmer needs to understand the operation of DSM, and may need to annotate or alter her program (for example, she may have to change the layout of data items).

The consistency model is a quite different concept to the programmer's synchronization model; but the two are related in release consistency (and other weaker forms of consistency).

Use of write-update vs. write-invalidate depends upon the sharing pattern between readers and writers. Apart from the protocol and the consistency semantics, avoidance of thrashing, choice of granularity and avoidance of false sharing are strong determinants of the efficiency of a DSM implementation.

Sequential consistency is expensive to implement. Weaker forms of consistency, which depend upon programmers' use of synchronization objects, can be more efficiently implemented.

## Possible difficulties

The distinction between synchronization model and consistency model takes some getting used to.

Students are liable to find difficult the notions of sequential consistency and weaker forms of consistency.

## Teaching hints

Review the topic of mutual exclusion, as taught on undergraduate OS courses.

To implement (mutable) DSM is to replicate data with appropriate consistency semantics, and this is a problem that the students also meet in Chapter 11. That chapter's study of multicast semantics should throw light on ways to implement the different consistency models.

If the students are familiar with mapped files, then appeal to shared mapped files is likely to make the problems concrete for them.

Encourage the students to find examples of executions on sequentially inconsistent and consistent memories. Use the notation introduced in Exercise 17.3. Do the same for the other consistency models.

## Objectives

To give the student insight into the design of distributed operating systems by examining prominent examples of existing systems. In particular, we examine the features provided by three general-purpose microkernels, an operating system to support distributed objects, and examples of RPC and multicast implementations.

## Points to emphasize

The main emphasis when discussing the kernel case studies should be on the degree to which they have common features, and on the other hand how they differ in some significant ways.

Relate the common features to common design goals, and relate the differences to differences in their design goals and choices of trade-offs.

Distributed objects are a powerful paradigm for programming distributed systems. The Clouds system only addresses heavyweight objects, and dealing with fine-grained objects is a considerable further challenge.

The performance of RPC systems is dependent on software costs to a great extent.

## Possible difficulties

We assume that students are familiar with some UNIX system calls, such as *fork()*.

Students need a thorough understanding of the basic operation of paged MMUs, and of the concept of virtual memory. They also need to understand fully the costs involved in switching between MMU contexts.

## Teaching hints

Review the topics of processes, scheduling and virtual memory, as taught on undergraduate OS courses.

Review the material in Chapter 6, and relate the topics here to that material.

Although the case studies may be covered in isolation, it is instructive to cover at least two kernel case studies and compare them.

Appeal to the students' knowledge of UNIX (or another OS), and consider how abstractions such as processes, signals and files can be manufactured from microkernel abstractions.

The discussion of Clouds addresses only some of the issues raised by distributed object systems. Encourage the students to follow up the references and issues raised in the discussion.