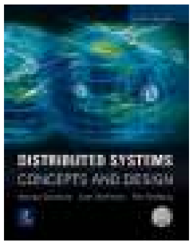
Distributed Systems Concepts and Design Excercise Solutions

data distriution (Sant Gadge Baba Amravati University)



Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001

Chapter 1 Exercise Solutions

1.1 Give five types of hardware resource and five types of data or software resource that can usefully be shared. Give examples of their sharing as it occurs in distributed systems.

1.1 Ans.

Hardware:

CPU: compute server (executes processor-intensive applications for clients), remote object server (executes methods on behalf of clients), worm program (shares cpu capacity of desktop machine with the local user). Most other servers, such as file servers, do some computation for their clients, hence their cpu is a shared resource.

memory: cache server (holds recently-accessed web pages in its RAM, for faster access by other local computers)

disk: file server, virtual disk server (see Chapter 8), video on demand server (see Chapter 15).

screen: Network window systems, such as X-11, allow processes in remote computers to update the content of windows.

printer: networked printers accept print jobs from many computers. managing them with a queuing system.

network capacity: packet transmission enables many simultaneous communication channels (streams of data) to be transmitted on the same circuits.

Data/software:

web page: web servers enable multiple clients to share read-only page content (usually stored in a file, but sometimes generated on-the-fly).

file: file servers enable multiple clients to share read-write files. Conflicting updates may result in inconsistent results. Most useful for files that change infrequently, such as software binaries.

object: possibilities for software objects are limitless. E.g. shared whiteboard, shared diary, room booking system, etc.

database: databases are intended to record the definitive state of some related sets of data. They have been shared ever since multi-user computers appeared. They include techniques to manage concurrent updates.

newsgroup content: The *netnews* system makes read-only copies of the recently-posted news items available to clients throughout the Internet. A copy of newsgroup content is maintained at each netnews server that is an approximate replica of those at other servers. Each server makes its data available to multiple clients.

video/audio stream: Servers can store entire videos on disk and deliver them at playback speed to multiple clients simultaneously.

exclusive lock: a system-level object provided by a lock server, enabling several clients to coordinate their use of a resource (such as printer that does not include a queuing scheme).

1.2 How might the clocks in two computers that are linked by a local network be synchronized without reference to an external time source? What factors limit the accuracy of the procedure you have described? How could the clocks in a large number of computers connected by the Internet be synchronized? Discuss the accuracy of that procedure.

1.2 Ans.

Several time synchronization protocols are described in Section 10.3. One of these is Cristian's protocol. Briefly, the round trip time *t* to send a message and a reply between computer A and computer B is measured by repeated tests; then computer A sends its clock setting *T* to computer B. B sets its clock to *T*+*t*/2. The setting can be refined by repetition. The procedure is subject to inaccuracy because of contention for the use of the local network from other computers and delays in the processing the messages in the operating systems of A and B. For a local network, the accuracy is probably within 1 ms.

For a large number of computers, one computer should be nominated to act as the time server and it should carry out Cristian's protocol with all of them. The protocol can be initiated by each in turn. Additional inaccuracies arise in the Internet because messages are delayed as they pass through switches in wider area networks. For a wide area network the accuracy is probably within 5-10 ms. These answers do not take into account the need for fault-tolerance. See Chapter 10 for further details.

1.3 A user arrives at a railway station that she has never visited before, carrying a PDA that is capable of wireless networking. Suggest how the user could be provided with information about the local services and amenities at that station, without entering the station's name or attributes. What technical challenges must be overcome?

1.3 Ans.

The user must be able to acquire the address of locally relevant information as automatically as possible. One method is for the local wireless network to provide the URL of web pages about the locality over a local wireless network.

For this to work: (1) the user must run a program on her device that listens for these URLs, and which gives the user sufficient control that she is not swamped by unwanted URLs of the places she passes through; and (2) the means of propagating the URL (e.g. infrared or an 802.11 wireless LAN) should have a reach that corresponds to the physical spread of the place itself.

1.4 What are the advantages and disadvantages of HTML, URLs and HTTP as core technologies for information browsing? Are any of these technologies suitable as a basis for client-server computing in general?

1.4 Ans.

HTML is a relatively straightforward language to parse and render but it confuses presentation with the underlying data that is being presented.

URLs are efficient resource locators but they are not sufficiently rich as resource links. For example, they may point at a resource that has been relocated or destroyed; their granularity (a whole resource) is too coarse-grained for many purposes.

HTTP is a simple protocol that can be implemented with a small footprint, and which can be put to use in many types of content transfer and other types of service. Its verbosity (HTML messages tend to contain many strings) makes it inefficient for passing small amounts of data.

HTTP and URLs are acceptable as a basis for client-server computing except that (a) there is no strong type-checking (web services operate by-value type checking without compiler support), (b) there is the inefficiency that we have mentioned.

1.5 Use the World Wide Web as an example to illustrate the concept of resource sharing, client and server.

Resources in the World Wide Web and other services are named by URLs. What do the initials

URL denote? Give examples of three different sorts of web resources that can be named by URLs.

1.5 Ans.

Web Pages are examples of resources that are shared. These resources are managed by Web servers.

Client-server architecture. The Web Browser is a client program (e.g. Netscape) that runs on the user's computer. The Web server accesses local files containing the Web pages and then supplies them to client browser processes.

URL - Uniform Resource Locator

(3 of the following a file or a image, movies, sound, anything that can be rendered, a query to a database or to a search engine.

1.6 Give an example of a URL.

List the three main components of a URL, stating how their boundaries are denoted and illustrating each one from your example.

To what extent is a URL location transparent?

1.6 Ans.

http://www.dcs.qmw.ac.uk/research/distrib/index.html

- The protocol to use. the part before the colon, in the example the protocol to use is http ("HyperText Transport Protocol").
- The part between // and / is the Domain name of the Web server host www.dcs.qmw.ac.uk.
- The remainder refers to information on that host - named within the top level directory used by that Web server research/distrib/book.html.

The hostname *www* is location independent so we have location transparency in that the address of a particular computer is not included. Therefore the organisation may move the Web service to another computer.

But if the responsibility for providing a WWW-based information service moves to another organisation, the URL would need to be changed.

1.7 A server program written in one language (for example C++) provides the implementation of a BLOB object that is intended to be accessed by clients that may be written in a different language (for example Java). The client and server computers may have different hardware, but all of them are attached to an internet. Describe the problems due to each of the five aspects of heterogeneity that need to be solved to make it possible for a client object to invoke a method on the server object.

1.7 Ans.

As the computers are attached to an internet, we can assume that Internet protocols deal with differences in networks.

But the computers may have different hardware - therefore we have to deal with differences of representation of data items in request and reply messages from clients to objects. A common standard will be defined for each type of data item that must be transmitted between the object and its clients.

The computers may run different operating systems, therefore we need to deal with different operations to send and receive messages or to express invocations. Thus at the Java/C++ level a common operation would be used which will be translated to the particular operation according to the operating system it runs on.

We have two different programming languages C++ and Java, they use different representations for data structures such as strings, arrays, records. A common standard will be defined for each type of data structure that must be transmitted between the object and its clients and a way of translating between that data structure and each of the languages.

We may have different implementors, e.g. one for C++ and the other for Java. They will need to agree on the common standards mentioned above and to document them.

1.8 An open distributed system allows new resource sharing services such as the BLOB object in Exercise 1.7 to be added and accessed by a variety of client programs. Discuss in the context of this example, to what extent the needs of openness differ from those of heterogeneity.

1.8 Ans.

To add the BLOB object to an existing open distributed system, the standards mentioned in the answer to Exercise 1.7 must already have been agreed for the distributed system To list them again:

- the distributed system uses a common set of communication protocols (probably Internet protocols).
- it uses an defined standard for representing data items (to deal with heterogeneity of hardware).
- It uses a common standard for message passing operations (or for invocations).
- It uses a language independent standard for representing data structures.

But for the open distributed system the standards must have been agreed and documented before the BLOB object was implemented. The implementors must conform to those standards. In addition, the interface to the BLOB object must be published so that when it is added to the system, both existing and new clients will be able to access it. The publication of the standards allows parts of the system to be implemented by different vendors and to work together.

1.9 Suppose that the operations of the BLOB object are separated into two categories – public operations that are available to all users and protected operations that are available only to certain named users. State all of the problems involved in ensuring that only the named users can use a protected operation. Supposing that access to a protected operation provides information that should not be revealed to all users, what further problems arise?

1.9 Ans.

Each request to access a protected operation must include the identity of the user making the request. The problems are:

- defining the identities of the users. Using these identities in the list of users who are allowed to access the protected operations at the implementation of the BLOB object. And in the request messages.
- ensuring that the identity supplied comes from the user it purports to be and not some other user pretending to be that user.
- preventing other users from replaying or tampering with the request messages of legitimate users.

Further problems.

- the information returned as the result of a protected operation must be hidden from unauthorised users. This means that the messages containing the information must be encrypted in case they are intercepted by unauthorised users.

1.10 The INFO service manages a potentially very large set of resources, each of which can be accessed by users throughout the Internet by means of a key (a string name). Discuss an approach to the design of the names of the resources that achieves the minimum loss of performance as the number of resources in the service increases. Suggest how the INFO service can be implemented so as to avoid performance bottlenecks when the number of users becomes very large.

1.10 Ans.

Algorithms that use hierarchic structures scale better than those that use linear structures. Therefore the solution should suggest a hierarchic naming scheme. e.g. that each resource has an name of the form 'A.B.C' etc. where the time taken is O(log n) where there are n resources in the system.

To allow for large numbers of users, the resources are partitioned amongst several servers, e.g. names starting with A at server 1, with B at server 2 and so forth. There could be more than one level of partitioning as in DNS. To avoid performance bottlenecks the algorithm for looking up a name must be decentralised. That is, the same server must not be involved in looking up every name. (A centralised solution would use a single root server that holds a location database that maps parts of the information onto particular servers). Some replication is required to avoid such centralisation. For example: i) the location database might be replicated at multiple root servers or ii) the location database might be replicated in every server. In both cases, different clients must access different servers (e.g. local ones or randomly).

1.11 List the three main software components that may fail when a client process invokes a method in a server object, giving an example of a failure in each case. To what extent are these failures independent of one another? Suggest how the components can be made to tolerate one another's failures.

1.11 Ans.

The three main software components that may fail are:

- the client process e.g. it may crash
- the server process e.g. the process may crash
- the communication software e.g. a message may fail to arrive

The failures are generally caused independently of one another. Examples of dependent failures:

- if the loss of a message causes the client or server process to crash. (The crashing of a server would cause a client to perceive that a reply message is missing and might indirectly cause it to fail).
- if clients crashing cause servers problems.
- if the crash of a process causes a failures in the communication software.

Both processes should be able to tolerate missing messages. The client must tolerate a missing reply message after it has sent an invocation request message. Instead of making the user wait forever for the reply, a client process could use a timeout and then tell the user it has not been able to contact the server.

A simple server just waits for request messages, executes invocations and sends replies. It should be absolutely immune to lost messages. But if a server stores information about its clients it might eventually fail if clients crash without informing the server (so that it can remove redundant information). (See stateless servers in chapter 4/5/8).

The communication software should be designed to tolerate crashes in the communicating processes. For example, the failure of one process should not cause problems in the communication between the surviving processes.

1.12 A server process maintains a shared information object such as the BLOB object of Exercise 1.7. Give arguments for and against allowing the client requests to be executed concurrently by the server. In the case that they are executed concurrently, give an example of possible 'interference' that can occur between the operations of different clients. Suggest how such interference may be prevented.

1.12 Ans.

For concurrent executions - more throughput in the server (particularly if the server has to access a disk or another service)

Against - problems of interference between concurrent operations

Example:

Client A's thread reads value of variable X

Client B's thread reads value of variable X

Client A's thread adds 1 to its value and stores the result in X

Client B's thread subtracts 1 from its value and stores the result in X

Result: X := X-1; imagine that X is the balance of a bank account, and clients A and B are implementing credit and debit transactions, and you can see immediately that the result is incorrect.

To overcome interference use some form of concurrency control. For example, for a Java server use synchronized operations such as credit and debit.

1.13 A service is implemented by several servers. Explain why resources might be transferred between them. Would it be satisfactory for clients to multicast all requests to the group of servers as a way of achieving mobility transparency for clients?

1.13 Ans.

Migration of resources (information objects) is performed: to reduce communication delays (place objects in a server that is on the same local network as their most frequent users); to balance the load of processing and or storage utilisation between different servers.

If all servers receive all requests, the communication load on the network is much increased and servers must do unnecessary work filtering out requests for objects that they do not hold.

Distributed Systems: Concepts and Design

Edition 3

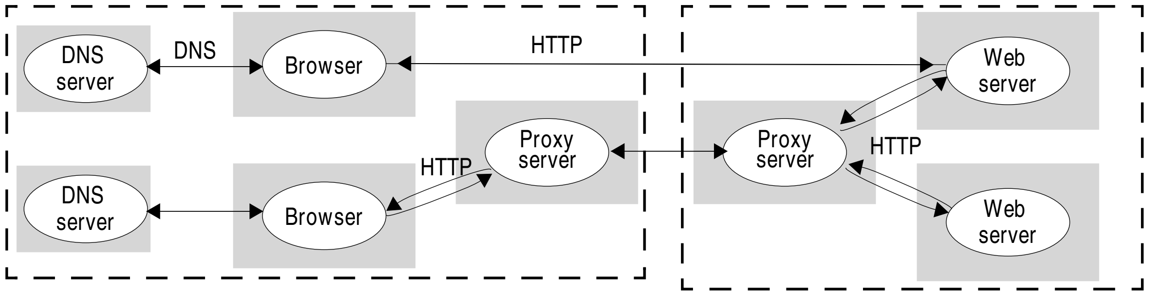
By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001.

Chapter 2 Exercise Solutions

2.1 Describe and illustrate the client-server architecture of one or more major Internet applications (for example the Web, email or netnews).

2.1 Ans.

Web:



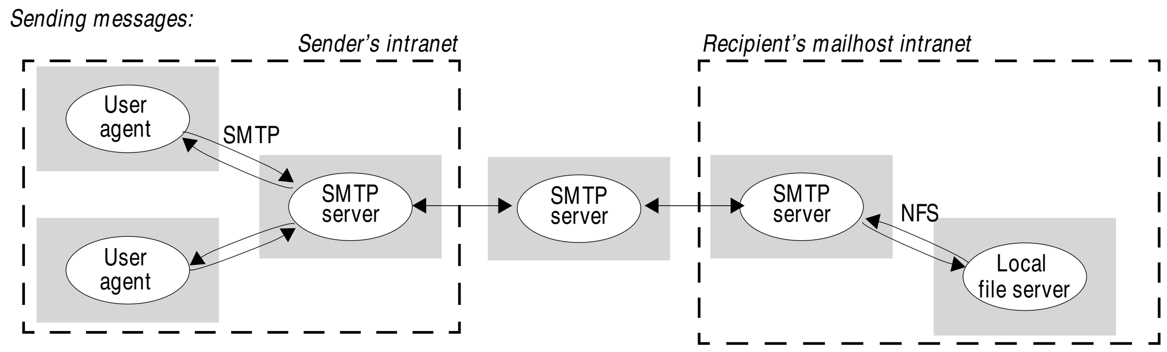
Browsers are clients of Domain Name Servers (DNS) and web servers (HTTP). Some intranets are configured to interpose a Proxy server. Proxy servers fulfil several purposes – when they are located at the same site as the client, they reduce network delays and network traffic. When they are at the same site as the server, they form a security checkpoint (see pp. 107 and 271) and they can reduce load on the server.

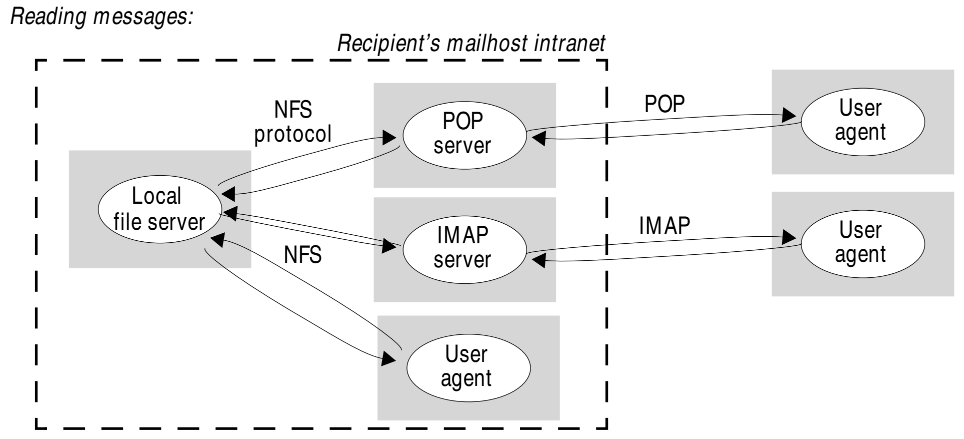
N.B. DNS servers are also involved in all of the application architectures described below, but they ore omitted from the discussion for clarity.

Email:

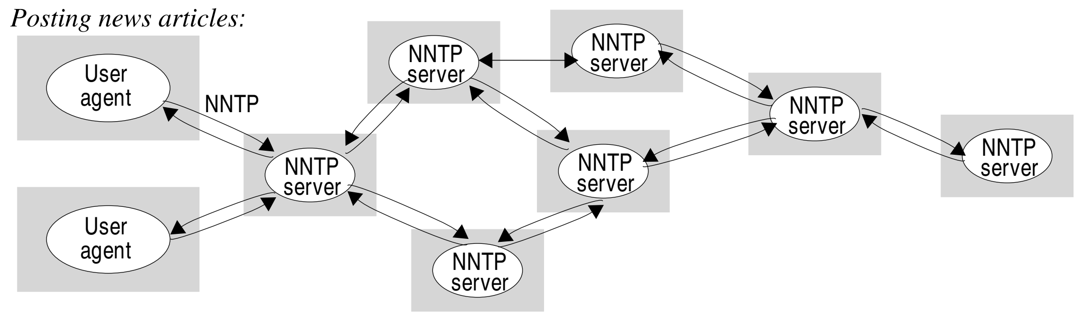
Sending messages: User Agent (the user’s mail composing program) is a client of a local SMTP server and passes each outgoing message to the SMTP server for delivery. The local SMTP server uses mail routing tables to determine a route for each message and then forwards the message to the next SMTP server on the chosen route. Each SMTP server similarly processes and forwards each incoming message unless the domain name in the message address matches the local domain. In the latter case, it attempts to deliver the message to local recipient by storing it in a mailbox file on a local disk or file server.

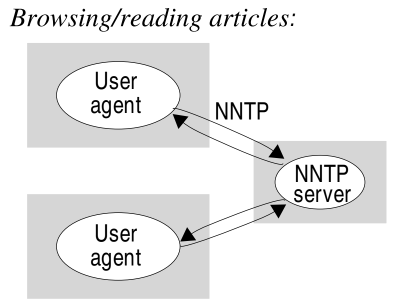
Reading messages: User Agent (the user’s mail reading program) is *either* a client of the local file server or a client of a mail delivery server such as a POP or IMAP server. In the former case, the User Agent reads messages directly form the mailbox file in which they were placed during the message delivery. (Exampes of such user agents are the UNIX *mail* and *pine* commands.) In the latter case, the User Agent requests information about the contents of the user’s mailbox file from a POP or IMAP server and receives messages from those servers for presentation to the user. POP and IMAP are protocols specifically designed to support mail access over wide areas and slow network connections, so a user can continue to access her home mailbox while travelling.





Netnews:





Posting news articles: User Agent (the user's news composing program) is a client of a local NNTP server and passes each outgoing article to the NNTP server for delivery. Each article is assigned a unique identifier. Each NNTP server holds a list of other NNTP servers for which it is a newsfeed – they are registered to receive articles from it. It periodically contacts each of the registered servers, delivers any new articles to them and requests any that they have which it has not (using the articles' unique id's to determine which they are). To ensure delivery of every article to every Netnews destination, there must be a path of newsfeed connections from that reaches every NNTP server.

Browsing/reading articles: User Agent (the user's news reading program) is a client of a local NNTP server. The User Agent requests updates for all of the newsgroups to which the user subscribes and presents them to the user.

2.2 For the applications discussed in Exercise 2.1 state how the servers cooperate in providing a service.

2.2 Ans.

Web: Web servers cooperate with Proxy servers to minimize network traffic and latency. Responsibility for consistency is taken by the proxy servers - they check the modification dates of pages frequently with the originating web server.

Mail: SMTP servers do not necessarily hold mail delivery routing tables to all destinations. Instead, they simply route messages addressed to unknown destinations to another server that is likely to have the relevant tables.

Netnews: All NNTP servers cooperate in the manner described above to provide the newsfeed mechanism.

2.3 How do the applications discussed in Exercise 2.1 involve the partitioning and/or replication (or caching) of data amongst servers?

2.3 Ans.

Web: Web page masters are held in a file system at a single server. The information on the web as a whole is therefore partitioned amongst many web servers.

Replication is not a part of the web protocols, but a heavily-used web site may provide several servers with identical copies of the relevant file system using one of the well-known means for replicating slowly-changing data (Chapter 14). HTTP requests can be multiplexed amongst the identical servers using the (fairly basic) DNS load sharing mechanism described on page 169. In addition, web proxy servers support replication through the use of cached replicas of recently-used pages and browsers support replication by maintaining a local cache of recently accessed pages.

Mail: Messages are stored only at their destinations. That is, the mail service is based mainly on partitioning, although a message to multiple recipients is replicated at several destinations.

Netnews: Each group is replicated only at sites requiring it.

2.4 A search engine is a web server that responds to client requests to search in its stored indexes and (concurrently) runs several web crawler tasks to build and update the indexes. What are the requirements for synchronization between these concurrent activities?

2.4 Ans.

The crawler tasks could build partial indexes to new pages incrementally, then merge them with the active index (including deleting invalid references). This merging operation could be done on an off-line copy. Finally, the environment for processing client requests is changed to access the new index. The latter might need some concurrency control, but in principle it is just a change to one reference to the index which should be atomic.

2.5 Suggest some applications for the peer process model, distinguishing between cases when the state of all peers needs to be identical and cases that demand less consistency.

2.5 Ans.

Cooperative work (groupware) applications that provide a peer process near to each user.

Applications that need to present all users with identical state - shared whiteboard, shared view of a textual discussion

Less consistency: where a group of users are working on a shared document, but different users access different parts or perhaps one user locks part of the document and the others are shown the new version when it is ready.

Some services are effectively groups of peer processes to provide availability or fault tolerance. If they partition data then they don't need to keep consistent at all. If they replicate then they do.

2.6 List the types of local resource that are vulnerable to an attack by an untrusted program that is downloaded from a remote site and run in a local computer.

2.6 Ans.

Objects in the file system e.g. files, directories can be read/written/created/deleted using the rights of the local user who runs the program.

Network communication - the program might attempt to create sockets, connect to them, send messages etc.

Access to printers.

It may also impersonate the user in various ways, for example, sending/receiving email

---

2.7 Give some examples of applications where the use of mobile code is beneficial.

2.7 Ans.

Doing computation close to the user, as in Applets example

Enhancing browser- as described on page 70 e.g. to allow server initiated communication.

Cases where objects are sent to a process and the code is required to make them usable. (e.g. as in RMI in Chapter 5)

2.8 What factors affect the responsiveness of an application that accesses shared data managed by a server? Describe remedies that are available and discuss their usefulness.

2.8 Ans.

When the client accesses a server, it makes an invocation of an operation in a server running in a remote computer. The following can affect responsiveness:

1. server overloaded;
2. latency in exchanging request and reply messages (due to layers of OS and middleware software in client and server);
3. load on network.

The use of caching helps with all of the above problems. In particular client caching reduces all of them. Proxy server caches help with (1). Replication of the service also helps with 1. The use of lightweight communication protocols helps with (2).

2.9 Distinguish between buffering and caching.

2.9 Ans.

Buffering: a technique for storing data transmitted from a sending process to a receiving process in local memory or secondary (disk) storage until the receiving process is ready to consume it. For example, when reading data from a file or transmitting messages through a network, it is beneficial to handle it in large blocks. The blocks are held in buffer storage in the receiving process' memory space. The buffer is released when the data has been consumed by the process.

Caching: a technique for optimizing access to remote data objects by holding a copy of them in local memory or secondary (disk) storage. Accesses to parts of the remote object are translated into accesses to the corresponding parts of the local copy. Unlike buffering, the local copy may be retained as long as there is local memory available to hold it. A cache management algorithm and a release strategy are needed to manage the use of the memory allocated to the cache. (If we interpret the word 'remote' in the sense of 'further from the processor', then this definition is valid not only for client caches in distributed systems but also for disk block caches in operating systems and processor caches in cpu chips.)

2.10 Give some examples of faults in hardware and software that can/cannot be tolerated by the use of redundancy in a distributed system. To what extent does the use of redundancy in the appropriate cases make a system fault-tolerant?

2.10 Ans.

- Hardware faults - processors, disks, network connections can use redundancy e.g. run process on multiple computers, write to two disks, have two separate routes in the network available.
- Software bugs, crashes. Redundancy is no good with bugs because they will be replicated. Replicated processes help with crashes which may be due to bugs in unrelated parts of the system. Retransmitted messages help with lost messages.

Redundancy makes faults less likely to occur. e.g. if the probability of failure in a single component is p then the probability of a single independent failure in k replicas is p^k.

2.11 Consider a simple server that carries out client requests without accessing other servers. Explain why it is generally not possible to set a limit on the time taken by such a server to respond to a client request. What would need to be done to make the server able to execute requests within a bounded time? Is this a practical option?

2.11 Ans.

The rate of arrival of client requests is unpredictable.

If the server uses threads to execute the requests concurrently, it may not be able to allocate sufficient time to a particular request within any given time limit.

If the server queues the request and carries them out one at a time, they may wait in the queue for an unlimited amount of time.

To execute requests within bounded time, limit the number of clients to suit its capacity. To deal with more clients, use a server with more processors. After that, (or instead) replicate the service....

The solution may be costly and in some cases keeping the replicas consistent may take up useful processing cycles, reducing those available for executing requests.

2.12 For each of the factors that contribute to the time taken to transmit a message between two processes over a communication channel, state what measures would be needed to set a bound on its contribution to the total time. Why are these measures not provided in current general-purpose distributed systems?

2.12 Ans.

Time taken by OS communication services in the sending and receiving processes - these tasks would need to be guaranteed sufficient processor cycles.

Time taken to access network. The pair of communicating processes would need to be given guaranteed network capacity.

The time to transmit the data is a constant once the network has been accessed.

To provide the above guarantees we would need more resources and associated costs. The guarantees associated with accessing the network can for example be provided with ATM networks, but they are expensive for use as LANs.

To give guarantees for the processes is more complex. For example, for a server to guarantee to receive and send messages within a time limit would mean limiting the number of clients.

2.13 The Network Time Protocol service can be used to synchronize computer clocks. Explain why, even with this service, no guaranteed bound given for the difference between two clocks.

2.13 Ans.

Any client using the ntp service must communicate with it by means of messages passed over a communication channel. If a bound can be set on the time to transmit a message over a communication channel, then the difference between the client's clock and the value supplied by the ntp service would also be bounded. With unbounded message transmission time, clock differences are necessarily unbounded.

2.14 Consider two communication services for use in asynchronous distributed systems. In service A, messages may be lost, duplicated or delayed and checksums apply only to headers. In service B, messages may be lost. delayed or delivered too fast for the recipient to handle them, but those that are delivered arrive order and with the correct contents.

Describe the classes of failure exhibited by each service. Classify their failures according to their effect on the properties of validity and integrity. Can service B be described as a reliable communication service?

2.14 Ans.

Service A can have:

arbitrary failures:

- as checksums do not apply to message bodies, message bodies can corrupted.
- duplicated messages,

omission failures (lost messages).

Because the distributed system in which it is used is asynchronous, it cannot suffer from timing failures.
Validity - is denied by lost messages
Integrity - is denied by corrupted messages and duplicated messages.

Service B can have:

omission failures (lost messages, dropped messages).

Because the distributed system in which it is used is asynchronous, it cannot suffer from timing failures.

It passes the integrity test, but not the validity test, therefore it cannot be called reliable.

2.15 Consider a pair of processes X and Y that use the communication service B from Exercise 2.14 to communicate with one another. Suppose that X is a client and Y a server and that an invocation consists of a request message from X to Y (that carries out the request) followed by a reply message from Y to X. Describe the classes of failure that may be exhibited by an invocation.

2.15 Ans.

An invocation may suffer from the following failures:

- *crash failures*: X or Y may crash. Therefore an invocation may suffer from crash failures.
- *omission failures*: as SB suffers from omission failures the request or reply message may be lost.

2.16 Suppose that a basic disk read can sometimes read values that are different from those written. State the type of failure exhibited by a basic disk read. Suggest how this failure may be masked in order to produce a different benign form of failure. Now suggest how to mask the benign failure.

2.16 Ans.

The basic disk read exhibit arbitrary failures.

This can be masked by using a checksum on each disk block (making it unlikely that wrong values will go undetected) - when an incorrect value is detected, the read returns no value instead of a wrong value - an omission failure.

The omission failures can be masked by replicating each disk block on two independent disks. (Making omission failures unlikely).

.

2.17 Define the integrity property of reliable communication and list all the possible threats to integrity from users and from system components. What measures can be taken to ensure the integrity property in the face of each of these sources of threats

2.17 Ans.

Integrity - the message received is identical to the one sent and no messages are delivered twice.

threats from users:

- injecting spurious messages, replaying old messages, altering messages during transmission

threats from system components:

- messages may get corrupted en route
- messages may be duplicated by communication protocols that retransmit messages.

For threats from users - at the Chapter 2 stage they might just say use secure channels. If they have looked at Chapter 7 they may be able to suggest the use of authentication techniques and nonces.

For threats from system components. Checksums to detect corrupted messages - but then we get a validity problem (dropped message). Duplicated messages can be detected if sequence numbers are attached to messages.

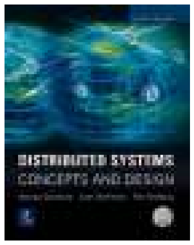
2.18 Describe possible occurrences of each of the main types of security threat (threats to processes, threats to communication channels, denial of service) that might occur in the Internet.

2.18 Ans.

Threats to processes: without authentication of principals and servers, many threats exist. An enemy could access other user's files or mailboxes, or set up 'spoof' servers. E.g. a server could be set up to 'spoof' a bank's service and receive details of user's financial transactions.

Threats to communication channels: IP spoofing - sending requests to servers with a false source address, man-in-the-middle attacks.

Denial of service: flooding a publicly-available service with irrelevant messages.



Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001

Chapter 3 Exercise Solutions

3.1 A client sends a 200 byte request message to a service, which produces a response containing 5000 bytes. Estimate the total time to complete the request in each of the following cases, with the performance assumptions listed below:

i) Using connectionless (datagram) communication (for example, UDP);

ii) Using connection-oriented communication (for example, TCP);

iii) The server process is in the same machine as the client.

[Latency per packet (local or remote,
incurred on both send and receive):5 milliseconds
Connection setup time (TCP only):5 milliseconds
Data transfer rate:10 megabits per second
MTU:1000 bytes
Server request processing time:2 milliseconds
Assume that the network is lightly loaded.]

3.1 Ans.

The send and receive latencies include (operating system) software overheads as well as network delays. Assuming that the former dominate, then the estimates are as below. If network overheads dominate, then the times may be reduced because the multiple response packets can be transmitted and received right after each other.

i) UDP: 5 + 2000/10000 + 2 + 5(5 + 10000/10000) = 37.2 milliseconds

ii) TCP: 5 + 5 + 2000/10000 + 2 + 5(5 + 10000/10000) = 42.2 milliseconds

iii)same machine: the messages can be sent by a single in memory copy; estimate interprocess data transfer rate at 40 megabits/second. Latency/message ~5 milliseconds. Time for server call:

5 + 2000/40000 + 5 + 50000/40000 = 11.3 milliseconds

3.2 The Internet is far too large for any router to hold routing information for all destinations. How does the Internet routing scheme deal with this issue?

3.2 Ans.

If a router does not find the network id portion of a destination address in its routing table, it despatches the packet to a *default address* an adjacent gateway or router that is designated as responsible for routing packets for which there is no routing information available. Each router's default address carries such packets towards a router than has more complete routing information, until one is encountered that has a specific entry for the relevant network id.

3.3 What is the task of an Ethernet switch? What tables does it maintain?

3.3 Ans.

An Ethernet switch must maintain routing tables giving the Ethernet addresses and network id for all hosts on the local network (connected set of Ethernets accessible from the switch). It does this by 'learning' the host addresses from the source address fields on each network. The switch receives all the packets transmitted on the Ethernets to which it is connected. It looks up the destination of each packet in its routing tables. If the destination is not found, the destination host must be one about which the switch has not yet learned and the packet must be forwarded to all the connected networks to ensure delivery. If the destination address is on the same Ethernet as the source, the packet is ignored, since it will be delivered directly. In all other cases, the switch tranmits the packet on the destination host's network, determined from the routing information.

3.4 Make a table similar to Figure 3.5 describing the work done by the software in each protocol layer when Internet applications and the TCP/IP suite are implemented over an Ethernet.

3.4 Ans.

Layer	Description	Examples
Application	Protocols that are designed to meet the communication requirements of specific applications, often defining the interface to a service. network representation that is independent of the representations used in individual computers. Encryption is performed in this layer.	HTTP, FTP, SMTP, CORBA IIOP, Secure Sockets Layer, CORBA Data Rep
Transport	UDP: checksum validation, delivery to process ports. TCP: segmentation, flow control, acknowledgement and reliable delivery.	TCP, UDP
Network	IP addresses translated to Ethernet addresses (using ARP). IP packets segmented into Ether packets.	IP
Data link	Ethernet CSMA CD mechanism.	Ethernet MAC layer
Physical	Various Ethernet transmission standards.	Ethernet base-band signalling

---

3.5 How has the end-to-end argument [Saltzer *et al.* 1984] been applied to the design of the Internet? Consider how the use of a virtual circuit network protocol in place of IP would impact the feasibility of the World Wide Web.

3.5 Ans.

Quote from [www.reed.com]: This design approach has been the bedrock under the Internet's design. The e-mail and web (note they are now lower-case) infrastructure that permeates the world economy would not have been possible if they hadn't been built according to the end-to-end principle. Just remember: underlying a web page that comes up in a fraction of a second are tens or even hundreds of packet exchanges with many unrelated computers. If we had required that each exchange set up a virtual circuit registered with each router on the network, so that the network could track it, the overhead of registering circuits would dominate the cost of delivering the page. Similarly, the decentralized administration of email has allowed the development of list servers and newsgroups which have flourished with little cost or central planning.

3.6 Can we be sure that no two computers in the Internet have the same IP addresses?

3.6 Ans.

This depends upon the allocation of network ids to user organizations by the Network Information Center (NIC). Of course, networks with unauthorized network ids may become connected, in which case the requirement for unique IP addresses is broken.

3.7 Compare connectionless (UDP) and connection-oriented (TCP) communication for the implementation of each of the following application-level or presentation-level protocols:

i) virtual terminal access (for example, Telnet);

ii) file transfer (for example, FTP);

iii) user location (for example, rwho, finger);

iv) information browsing (for example, HTTP);

v) remote procedure call.

3.7 Ans.

i) The long duration of sessions, the need for reliability and the unstructured sequences of characters transmitted make connection-oriented communication most suitable for this application. Performance is not critical in this application, so the overheads are of little consequence.

ii) File calls for the transmission of large volumes of data. Connectionless would be ok if error rates are low and the messages can be large, but in the Internet, these requirements aren't met, so TCP is used.

iii) Connectionless is preferable, since messages are short, and a single message is sufficient for each transaction.

iv) Either mode could be used. The volume of data transferred on each transaction can be quite large, so TCP is used in practice.

v) RPC achieves reliability by means of timeouts and re-trys. so connectionless (UDP) communication is often preferred.

3.8 Explain how it is possible for a sequence of packets transmitted through a wide area network to arrive at their destination in an order that differs from that in which they were sent. Why can't this happen in a local network? Can it happen in an ATM network?

3.8 Ans.

Packets transmitted through a store-and-forward network travels by a route that is determined dynamically for each packet. Some routes will have more hops or slower switches than others. Thus packets may overtake each other. Connection-oriented protocols such as TCP overcome this by adding sequence numbers to the packets and re-ordering them at the receiving host.

It can't happen in local networks because the medium provides only a single channel connecting all of the hosts on the network. Packets are therefore transmitted and received in strict sequence.

It can't happen in ATM networks because they are connection-oriented. Transmission is always through virtual channels, and VCs guarantee to deliver data in the order in which it is transmitted.

3.9 A specific problem that must be solved in remote terminal access protocols such as Telnet is the need to transmit exceptional events such as 'kill signals' from the 'terminal' to the host in advance of previously-transmitted data. Kill signals should reach their destination ahead of any other ongoing transmissions. Discuss the solution of this problem with connection-oriented and connectionless protocols.

3.9 Ans.

The problem is that a kill signal should reach the receiving process quickly even when there is buffer overflow (e.g. caused by an infinite loop in the sender) or other exceptional conditions at the receiving host.

With a connection-oriented, reliable protocol such as TCP, all packets must be received and acknowledged by the sender, in the order in which they are transmitted. Thus a kill signal cannot overtake other data already in the stream. To overcome this, an out-of-band signalling mechanism must be provided. In TCP this is called the URGENT mechanism. Packets containing data that is flagged as URGENT bypass the flow-control mechanisms at the receiver and are read immediately.

With connectionless protocols, the process at the sender simply recognizes the event and sends a message containing a kill signal in the next outgoing packet. The message must be resent until the receiving process acknowledges it.

3.10 What are the disadvantages of using network-level broadcasting to locate resources:

i) in a single Ethernet?

ii) in an intranet?

To what extent is Ethernet multicast an improvement on broadcasting?

3.10 Ans.

i. All broadcast messages in the Ethernet must be handled by the OS, or by a standard daemon process. The overheads of examining the message, parsing it and deciding whether it need be acted upon are incurred by every host on the network, whereas only a small number are likely locations for a given resource. Despite this, note that the Internet ARP does rely on Ethernet braodcasting. The trick is that it doesn't do it very often - just once for each host to locate other hosts on the local net that it needs to communicate with.

ii. Broadcasting is hardly feasible in a large-scale network such as the Internet. It might just be possible in an intranet, but ought to be avoided for the reasons given above.

Ethernet multicast addresses are matched in the Ethernet controller. Multicast message are passed up to the OS only for addresses that match multicast groups the local host is subscribing to. If there are several such, the address can be used to discriminate between several daemon processes to choose one to handle each message.

3.11 Suggest a scheme that improves on MobileIP for providing access to a web server on a mobile device which is sometimes connected to the Internet by mobile phone and at other times has a wired connection to the Internet at one of several locations.

3.11 Ans.

The idea is to exploit the cellular phone system to locate the mobile device and to give the IP address of its current location to the client.

3.12 Show the sequence of changes to the routing tables in Figure 3.8 that would occur (according to the RIP algorithm given in Figure 3.9) after the link labelled 3 in Figure 3.7 is broken.

3.12 Ans.

Routing tables with changes shown in red (grey in monochrome printouts):

Step 1: costs for routes that use Link 3 have been set to ∞ at A, D

Routings from A

To	*Link*	*Cost*
A	local	0
B	1	1
C	1	2
D	3	∞
E	1	2

Routings from B

To	*Link*	*Cost*
A	1	1
B	local	0
C	2	1
D	1	2
E	4	1

Routings from C

To	*Link*	*Cost*
A	2	2
B	2	1
C	local	0
D	5	2
E	5	1

Routings from D

To	*Link*	*Cost*
A	3	∞
B	3	∞
C	6	2
D	local	0
E	6	1

Routings from E

To	*Link*	*Cost*
A	4	2
B	4	1
C	5	1
D	6	1
E	local	0

Step 2: after first exchange of routing tables

Routings from A

To	*Link*	*Cost*
A	local	0
B	1	1
C	1	2
D	3	∞
E	1	2

Routings from B

To	*Link*	*Cost*
A	1	1
B	local	0
C	2	1
D	1	∞
E	4	1

Routings from C

To	*Link*	*Cost*
A	2	2
B	2	1
C	local	0
D	5	2
E	5	1

Routings from D

To	*Link*	*Cost*
A	3	∞
B	3	∞
C	6	2
D	local	0
E	6	1

Routings from E

To	*Link*	*Cost*
A	4	2
B	4	1
C	5	1
D	6	1
E	local	0

Step 3: after second exchange of routing tables

Routings from A		
To	*Link*	*Cost*
A	local	0
B	1	1
C	1	2
D	3	∞
E	1	2

Routings from B		
To	*Link*	*Cost*
A	1	1
B	local	0
C	2	1
D	4	2
E	4	1

Routings from C		
To	*Link*	*Cost*
A	2	2
B	2	1
C	local	0
D	5	2
E	5	1

Routings from D		
To	*Link*	*Cost*
A	6	3
B	6	2
C	6	2
D	local	0
E	6	1

Routings from E		
To	*Link*	*Cost*
A	4	2
B	4	1
C	5	1
D	6	1
E	local	0

Step 4: after third exchange of routing tables.

Routings from A		
To	*Link*	*Cost*
A	local	0
B	1	1
C	1	2
D	1	3
E	1	2

Routings from B		
To	*Link*	*Cost*
A	1	1
B	local	0
C	2	1
D	4	2
E	4	1

Routings from C		
To	*Link*	*Cost*
A	2	2
B	2	1
C	local	0
D	5	2
E	5	1

Routings from D		
To	*Link*	*Cost*
A	6	3
B	6	2
C	6	2
D	local	0
E	6	1

Routings from E		
To	*Link*	*Cost*
A	4	2
B	4	1
C	5	1
D	6	1
E	local	0

3.13 Use the diagram in Figure 3.13 as a basis for an illustration showing the segmentation and encapsulation of an HTTP request to a server and the resulting reply. Assume that request is a short HTTP message, but the reply includes at least 2000 bytes of html.

3.13 Ans.

Left to the reader.

3.14 Consider the use of TCP in a Telnet remote terminal client. How should the keyboard input be buffered at the client? Investigate Nagle's and Clark's algorithms [Nagle 1984, Clark 1982] for flow control and compare them with the simple algorithm described on page 103 when TCP is used by (a) a web server, (b) a Telnet application, (c) a remote graphical application with continuous mouse input.

3.14 Ans.

The basic TCP buffering algorithm described on p. 105 is not very efficient for interactive input. Nagle's algorithm is designed to address this. It requires the sending machine to send any bytes found in the output buffer, then wait for an acknowledgement. Whenever an acknowledgement is received, any additional characters in the buffer are sent. The effects of this are:

a) For a web server: the server will normally write a whole page of HTML into the buffer in a single *write*. When the *write* is completed, Nagle's algorithm will send the data immediately, whereas the basic algorithm would wait 0.5 seconds. While the Nagle's algorithm is waiting for an acknowledgement, the server process can write additional data (e.g. image files) into the buffer. They will be sent as soon as the acknowledgement is received.

b) For a remote shell (Telnet) application: the application will write individual key strokes into the buffer (and in the normal case of full duplex terminal interaction they are echoed by the remote host to the Telnet client for display). With the basic algorithm, full duplex operation would result in a delay of 0.5 seconds before any of the characters typed are displayed on the screen. With Nagle's algorithm, the first character typed is sent immediately and the remote host echoes it with an acknowledgement piggy-backed in the same packet. The acknowledgement triggers the sending of any further characters that have been typed in the intervening period. So if the remote host responds sufficiently rapidly, the display of typed characters appears to be instantaneous. But note that a badly-written remote application that reads data from the TCP buffer one character at a time can still cause problems - each read will result in an acknowledgement indicating that one further character should be sent - resulting in the transmission of an entire IP frame for each character. Clarke [1982] called this the *silly window syndrome*. His solution is to defer the sending of acknowledgements until there is a substantial amount of free space available.

c) For a continuous mouse input (e.g. sending mouse positions to an X-Windows application running on a compute server): this is a difficult form of input to handle remotely. The problem is that the user should see a smooth feedbvack of the path traced by the mouse, with minimal lag. Neither the basic TCP algorithm nor Nagle's nor Clarke's algorithm achieves this very well. A version of the basic algorithm with a short timeout (0.1 seconds) is the best that can be done, and this is effective when the network is lightly loaded and has low end-to-end latency - conditions that can be guaranteed only on local networks with controlled loads.
See Tanenbaum [1996] pp. 534-5 for further discussion of this.

3.15 Construct a network diagram similar to Figure 3.10 for the local network at your institution or company.

3.15 Ans.

Left to the reader.

3.16 Describe how you would configure a firewall to protect the local network at your institution or company. What incoming and outgoing requests should it intercept?

3.16 Ans.

Left to the reader.

3.17 How does a newly-installed personal computer connected to an Ethernet discover the IP addresses of local servers? How does it translate them to Ethernet addresses?

3.17 Ans.

The first part of the question is a little misleading. Neither Ethernet nor the Internet support 'discovery' services as such. A newly-installed computer must be configured with the domain names of any servers that it needs to access. The only exception is the DNS. Services such as BootP and DHCP enable a newly-connected host to acquire its own IP address and to obtain the IP addresses of one ore more local DNS servers. To obtain the IP addresses of other servers (e.g. SMTP, NFS, etc.) it must use their domain names. In Unix, the *nslookup* command can be used to examine the database of domain names in the local DNS servers and a user can select approriate ones for use as servers.The domain names are translated to IP addresses by a simple DNS request.

The Address Resolution Protocol (ARP) provides the answer to the second part of the question. This is described on pages 95-6. Each network type must implement ARP in its own way. The Ethernet and related networks use the combination of broadcasting and caching of the results of previous queries described on page 96.

3.18 Can firewalls prevent denial of service attacks such as the one described on page 96? What other methods are available to deal with such attacks?

3.18 Ans.

Since a firewall is simply another computer system placed in front of some intranet services that require protection, it is unlikely to be able to prevent denial of service (DoS) attacks for two reasons:

- The attacking traffic is likely to closely resemble real service requests or responses.
- Even if they can be recognized as malicious (and they could be in the case described on p. 96), a successful attack is likely to produce malicious messages in such large quantities that the firewall itself is likely to be overwhelemed and become a bottleneck, preventing communication with the services that it protects.

Other methods to deal with DoS attacks: no comprehensive defence has yet been developed. Attacks of the type described on p. 96, which are dependent on IP spoofing (giving a false 'senders address') can be prevented at their source by checking the senders address on all outgoing IP packets. This assumes that all Internet sites are managed in such a manner as to ensure that this check is made - an unlikely circumstance. It is difficult to see how the targets of such attacks (which are usually heavily-used public services) can defend themselves with current network protocols and their security mechanisms. With the advent of quality-of-service mechanisms in IPv6, the situation should improve. It should be possible for a service to allocate only a limited amount of its total bandwidth to each range of IP addresses, and routers thorughout the Internet could be setup to enforce these resource allocations. However, this approach has not yet been fully worked out.

Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001.

Chapter 4 Exercise Solutions

4.1 Is it conceivably useful for a port to have several receivers?

4.1 Ans.

If several processes share a port, then it must be possible for all of the messages that arrive on that port to be received and processed independently by those processes.

Processes do not usually share data, but sharing a port would requires access to common data representing the messages in the queue at the port. In addition, the queue structure would be complicated by the fact that each process has its own idea of the front of the queue and when the queue is empty.

Note that a port group may be used to allow several processes to receive the same message.

4.2 A server creates a port which it uses to receive requests from clients. Discuss the design issues concerning the relationship between the name of this port and the names used by clients.

4.2 Ans.

The main design issues for locating server ports are:

(i) How does a client know what port and IP address to use to reach a service?

The options are:

- use a name server/binder to map the textual name of each service to its port;
- each service uses well-known location-independent port id, which avoids a lookup at a name server.

The operating system still has to look up the whereabouts of the server, but the answer may be cached locally.

(ii) How can different servers offer the service at different times?

Location-independent port identifiers allow the service to have the same port at different locations. If a binder is used, the client needs to reconsult the client to find the new location.

(iii) Efficiency of access to ports and local identifiers.

Sometimes operating systems allow processes to use efficient local names to refer to ports. This becomes an issue when a server creates a non-public port for a particular client to send messages to, because the local name is meaningless to the client and must be translated to a global identifier for use by the client.

4.3 The programs in Figure 4.3 and Figure 4.4 are available on cdk3.net/ipc. Use them to make a test kit to determine the conditions in which datagrams are sometimes dropped. Hint: the client program should be able to vary the number of messages sent and their size; the server should detect when a message from a particular client is missed.

4.3 Ans.

For a test of this type, one process sends and another receives. Modify the program in Figure 4.3 so that the program arguments specify i) the server's hostname ii) the server port, iii) the number, *n* of messages to be sent and iv) the length, *l* of the messages. If the arguments are not suitable, the program should exit immediately. The program should open a datagram socket and then send *n* UDP datagram messages to the server. Message *i* should contain the integer *i* in the first four bytes and the character '*' in the remaining l-4 bytes. It does not attempt to receive any messages.

Take a copy of the program in Figure 4.4 and modify it so that the program argument specifies the server port. The program should open a socket on the given port and then repeatedly receive a datagram message. It should check the number in each message and report whenever there is a gap in the sequence of numbers in the messages received from a particular client.

Run these two programs on a pair of computers and try to find out the conditions in which datagrams are dropped, e.g. size of message, number of clients.

4.4 Use the program in Figure 4.3 to make a client program that repeatedly reads a line of input from the user, sends it to the server in a UDP datagram message, then receives a message from the server. The client sets a timeout on its socket so that it can inform the user when the server does not reply. Test this client program with the server in Figure 4.4.

4.4 Ans.

The program is as Figure 4.4 with the following amendments:

```
DatagramSocket aSocket = new DatagramSocket();
aSocket.setSoTimeout(3000);// in milliseconds
while (// not eof) {
try{
   // get user's input and put in request
  .....
   aSocket.send(request);
   ........
 aSocket.receive(reply);
}catch (InterruptedIOException e){System.out.println("server not responding");}
```

4.5 The programs in Figure 4.5 and Figure 4.6 are available at cdk3.net/ipc. Modify them so that the client repeatedly takes a line of user's input and writes it to the stream and the server reads repeatedly from the stream, printing out the result of each read. Make a comparison between sending data in UDP datagram messages and over a stream.

4.5 Ans.

The changes to the two programs are straightforward. But students should notice that not all sends go immediately and that receives must match the data sent.

For the comparison. In both cases, a sequence of bytes is transmitted from a sender to a receiver. In the case of a message the sender first constructs the sequence of bytes and then transmits it to the receiver which receives it as a whole. In the case of a stream, the sender transmits the bytes whenever they are ready and the receiver collects the bytes from the stream as they arrive.

4.6 Use the programs developed in Exercise 4.5 to test the effect on the sender when the receiver crashes and vice-versa.

4.6 Ans.

Run them both for a while and then kill first one and then the other. When the reader process crashes, the writer gets IOException - broken pipe. When writer process crashes, the reader gets EOF exception.

4.7 Sun XDR marshals data by converting it into a standard big-endian form before transmission. Discuss the advantages and disadvantages of this method when compared with CORBA's CDR.

4.7 Ans.

The XDR method which uses a standard form is inefficient when communication takes place between pairs of similar computers whose byte orderings differ from the standard. It is efficient in networks in which the byte-ordering used by the majority of the computers is the same as the standard form. The conversion by senders and recipients that use the standard form is in effect a null operation.

In CORBA CDR senders include an identifier in each message and recipients to convert the bytes to their own ordering if necessary. This method eliminates all unnecessary data conversions, but adds complexity in that all computers need to deal with both variants.

4.8 Sun XDR aligns each primitive value on a four byte boundary, whereas CORBA CDR aligns a primitive value of size *n* on an *n*-byte boundary. Discuss the trade-offs in choosing the sizes occupied by primitive values.

4.8 Ans.

Marshalling is simpler when the data matches the alignment boundaries of the computers involved. Four bytes is large enough to support most architectures efficiently, but some space is wasted by smaller primitive values. The hybrid method of CDR is more complex to implement, but saves some space in the marshalled form. Although the example in Figure 4.8 shows that space is wasted at the end of each string because the following long is aligned on a 4- byte boundary.

4.9 Why is there no explicit data-typing in CORBA CDR?

4.9 Ans.

The use of data-typing produces costs in space and time. The space costs are due to the extra type information in the marshalled form (see for example the Java serialized form). The performance cost is due to the need to interpret the type information and take appropriate action.

The RMI protocol for which CDR is designed is used in a situation in which the target and the invoker know what type to expect in the messages carrying its arguments and results. Therefore type information is redundant. It is of course possible to build type descriptors on top of CDR, for example by using simple strings.

4.10 Write an algorithm in pseudocode to describe the serialization procedure described in Section 4.3.2. The algorithm should show when handles are defined or substituted for classes and instances. Describe the serialized form that your algorithm would produce when serializing an instance of the following class *Couple*.

```
    class Couple implements Serializable{
    private Person one;
    private Person two;
    public Couple(Person a, Person b) {
        one = a;
        two = b;
    }
}
```

4.10 Ans.

The algorithm must describe serialization of an object as writing its class information followed by the names and types of the instance variables.Then serialize each instance variable recursively.

```
serialize(Object o) {
    c = class(o);
    class_handle = get_handle(c);
    if (class_handle==null) // write class information and define class_handle;
    write class_handle
    write number (n), name and class of each instance variable

    object_handle = get_handle(o);
    if (object_handle==null)  {
        define object_handle;
        for (iv = 0 to n-1)
            if (primitive(iv) ) write iv
            else serialize( iv)
    }
    write object_handle
}
```

To describe the serialized form that your algorithm would produce when serializing an instance of the class *Couple*.

For example declare an instance of Couple as

Couple t1 = new Couple(new Person("Smith", "London", 1934),
new Person("Jones", "Paris", 1945));

The output will be:

Serialized values				*Explanation*
Couple	8 byte version number		h0	*class name, version number, handle*
2	Person one	Person two		*number, type and name of instance variables*
Person	8 byte version number		h1	*serialize instance variable one of Couple*
3	int year	java.lang.String name	java.lang.String place	
1934	5 Smith	6 London	h2	
h1				*serialize instance variable two of Couple*
1945	5 Jones	5 Paris	h3	*values of instance variables*

4.11 Write an algorithm in pseudocode to describe deserialization of the serialized form produced by the algorithm defined in Exercise 4.10. Hint: use reflection to create a class from its name, to create a constructor from its parameter types and to create a new instance of an object from the constructor and the argument values.

4.11 Ans.

Whenever a handle definition is read, i.e. a class_info, handle correspondence or an object, handle correspondence, store the pair by method map. When a handle is read look it up to find the corresponding class or object.

```
Object deserialize(byte [] stream) {
    Constructor aConstructor;
    read class_name and class_handle;
    if (class_information == null) aConstructor = lookup(class_handle);
    else {
        Class cl = Class.forName(class_name);
        read number (n) of instance variables
        Class parameterTypes[]= new Class[n];
        for (int i=0 to n-1) {
            read name and class_name of instance variable i
            parameterTypes[i] = Class.forName(class_name);
        }
        aConstructor = cl.getConstructor(parameterTypes);
        map(aConstructor, class_handle);
    }
    if (next item in stream is object_handle) o = lookup(object_handle);
    else {
        Object args[] = new Object[n];
        for (int i=0 to n-1) {
            if (next item in stream is primitive) args[i] = read value
            else args[i]  = deserialize(//rest of stream)
        }
        Object o = cnew.newInstance(args);
        read object_handle from stream
        map(object, object_handle)
        return o;
    }
}
```

4.12 Define a class whose instances represent remote object references. It should contain information similar to that shown in Figure 4.10 and should provide access methods needed by the request-reply protocol. Explain how each of the access methods will be used by that protocol. Give a justification for the type chosen for the instance variable containing information about the interface of the remote object.

4.12 Ans.

```
class RemoteObjectReference{
    private InetAddress ipAddress;
    private int port;
    private int time;
    private int objectNumber;
    private Class interface;
    public InetAddress getIPaddress() ( return ipAddress;}
    public int getPort() { return port;);
}
```

The server looks up the client port and IP address before sending a reply.

The variable interface is used to recognize the class of a remote object when the reference is passed as an argument or result. Chapter 5 explains that proxies are created for communication with remote objects. A proxy needs to implement the remote interface. If the proxy name is constructed by adding a standard suffix to the interface name and all we need to do is to construct a proxy from a class already available, then its string name is sufficient. However, if we want to use reflection to construct a proxy, an instance of Class would be needed. CORBA uses a third alternative described in Chapter 17.

4.13 Define a class whose instances represent request and reply messages as illustrated in Figure 4.13. The class should provide a pair of constructors, one for request messages and the other for reply messages, showing how the request identifier is assigned. It should also provide a method to marshal itself into an array of bytes and to unmarshal an array of bytes into an instance.

4.13 Ans.

```
          private static int next = 0;
          private int type
          private int requestId;
          private RemoteObjectRef o;
          private int methodId;
          private byte arguments[];
          public RequestMessage( RemoteObjectRef aRef,
                    int aMethod, byte[] args){
                    type=0; ... etc.
                    requestId = next++; // assume it will not run long enough to overflow
          }
          public RequestMessage(int rId, byte[] result){
                    type=1; ... etc.
                    requestId = rid;
                    arguments = result;
          }

          public byte[] marshall() {
                    // converts itself into an array of bytes and returns it
          }
          public RequestMessage unmarshall(byte [] message) {
                    // converts array of bytes into an instance of this class and returns it
          }
          public int length() { // returns length of marshalled state}
          public int getID(){ return requestId;}
          public byte[] getArgs(){ return arguments;}
       }
```

4.14 Program each of the three operations of the request-reply protocol in Figure 4.123, using UDP communication, but without adding any fault-tolerance measures. You should use the classes you defined in Exercise 4.12 and Exercise 4.13.

4.14 Ans.

```
class Client{
   DatagramSocket aSocket ;
   public static messageLength = 1000;
   Client(){
     aSocket = new DatagramSocket();
   }
   public byte [] doOperation(RemoteObjectRef o, int methodId,
     byte [] arguments){
     InetAddress serverIp = o.getIPaddress();
     int serverPort = o.getPort();
     RequestMessage rm = new RequestMessage(0, o, methodId, arguments );
     byte [] message = rm.marshall();
     DatagramPacket request =
       new DatagramPacket(message,message.length(0,serverIp, serverPort);
     try{
          aSocket.send(request);
          byte buffer = new byte[messageLength];
          DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
          aSocket.receive(reply);
          return reply;
     }catch (SocketException e){...}
   }
]
Class Server{
   private int serverPort = 8888;
   public static int messageLength = 1000;
   DatagramSocket mySocket;
   public Server(){
     mySocket = new DatagramSocket(serverPort);
     // repeatedly call GetRequest, execute method and call SendReply
   }
   public byte [] getRequest(){
     byte buffer = new byte[messageLength];
     DatagramPacket request = new DatagramPacket(buffer, buffer.length);
     mySocket.receive(request);
     clientHost = request.getHost();
     clientPort = request.getPort();
     return request.getData();
   }
   public void sendReply(byte[]reply, InetAddress clientHost, int clientPort){
     byte buffer = rm.marshall();
     DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
     mySocket.send(reply);
   }
}
```

4.15 Give an outline of the server implementation showing how the operations *getRequest* and *sendReply* are used by a server that creates a new thread to execute each client request. Indicate how the server will copy the *requestId* from the request message into the reply message and how it will obtain the client IP address and port..

4.15 Ans.

```
class Server{
  private int serverPort = 8888;
  public static int messageLength = 1000;
  DatagramSocket mySocket;

  public Server(){
     mySocket = new DatagramSocket(serverPort);
     while(true){
          byte [] request = getRequest();
          Worker w = new Worker(request);
     }
  }
  public byte [] getRequest(){
     //as above}
  public void sendReply(byte[]reply, InetAddress clientHost, int clientPort){
     // as above}
}
class Worker extends Thread {
     InetAddress clientHost;
     int clientPort;
     int requestId;
     byte [] request;
     public Worker(request){
     // extract fields of message into instance variables
   }
  public void run(){
     try{
          req = request.unmarshal();
          byte [] args = req.getArgs();
          //unmarshall args, execute operation,
          // get results marshalled as array of bytes in result
```

```
            RequestMessage rm = new RequestMessage( requestId, result);
            reply = rm.marshal();
            sendReply(reply, clientHost, clientPort );
        }catch {... }
    }
}
```

4.16 Define a new version of the *doOperation* method that sets a timeout on waiting for the reply message. After a timeout, it retransmits the request message *n* times. If there is still no reply, it informs the caller.

4.16 Ans.

With a timeout set on a socket, a receive operation will block for the given amount of time and then an *InterruptedIOException* will be raised.

In the constructor of *Client*, set a timeout of say, 3 seconds

```
Client(){
    aSocket = new DatagramSocket();
    aSocket.setSoTimeout(3000);// in milliseconds
}
```

In *doOperation*, catch *InterruptedIOException.* Repeatedly send the Request message and try to receive a reply, e.g. 3 times. If there is no reply, return a special value to indicate a failure.

```
public byte [] doOperation(RemoteObjectRef o, int methodId,
    byte [] arguments){
    InetAddress serverIp = o.getIPaddress();
    int serverPort = o.getPort();
    RequestMessage rm = new RequestMessage(0, o, methodId, arguments );
    byte [] message = rm.marshall();
    DatagramPacket request =
      new DatagramPacket(message,message.length(0, serverIp, serverPort);
    for(int i=0; i<3;i++){
      try{
        aSocket.send(request);
        byte buffer = new byte[messageLength];
        DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
        aSocket.receive(reply);
        return reply;
    }catch (SocketException e){);
    }catch (InterruptedIOException e){}
     }
    return null;
}
```

4.17 Describe a scenario in which a client could receive a reply from an earlier call.

4.17 Ans.

Client sends request message, times out and then retransmits the request message, expecting only one reply. The server which is operating under a heavy load, eventually receives both request messages and sends two replies.

When the client sends a subsequent request it will receive the reply from the earlier call as a result. If request identifiers are copied from request to reply messages, the client can reject the reply to the earlier message.

4.18 Describe the ways in which the request-reply protocol masks the heterogeneity of operating systems and of computer networks.

4.18 Ans.

(i) Different operating systems may provide a variety of different interfaces to the communication protocols. These interfaces are concealed by the interfaces of the request-reply protocol.

(ii) Although the Internet protocols are widely available, some computer networks may provide other protocols. The request-reply protocol may equally be implemented over other protocols.

[In addition it may be implemented over either TCP or UDP.]

4.19 Discuss whether the following operations are *idempotent*:

- Pressing a lift (elevator) request button;
- Writing data to a file;
- Appending data to a file.

Is it a necessary condition for idempotence that the operation should not be associated with any state?

4.19 Ans.

The operation to write data to a file can be defined (i) as in Unix where each write is applied at the read-write pointer, in which case the operation is not idempotent; or (ii) as in several file servers where the write operation is applied to a specified sequence of locations, in which case, the operation is idempotent because it can be repeated any number of times with the same effect. The operation to append data to a file is not idempotent, because the file is extended each time this operation is performed.

The question of the relationship between idempotence and server state requires some careful clarification. It is a necessary condition of idempotence that the effect of an operation is independent of previous operations. Effects can be conveyed from one operation to the next by means of a server state such as a read-write pointer or a bank balance. Therefore it is a necessary condition of idempotence that the effects of an operation should not depend on server state. Note however, that the idempotent file write operation does change the state of a file.

4.20 Explain the design choices that are relevant to minimizing the amount of reply data held at a server. Compare the storage requirements when the RR and RRA protocols are used.

4.20 Ans.

To enable reply messages to be re-transmitted without re-executing operations, a server must retain the last reply to each client. When RR is used, it is assumed that a request message is an acknowledgement of the last reply message. Therefore a reply message must be held until a subsequent request message arrives from the same client. The use of storage can be reduced by applying a timeout to the period during which a reply is stored. The storage requirement for RR = average message size x number of clients that have made requests since timeout period. When RRA is used, a reply message is held only until an acknowledgement arrives. When an acknowledgment is lost, the reply message will be held as for the RR protocol.

4.21 Assume the RRA protocol is in use. How long should servers retain unacknowledged reply data? Should servers repeatedly send the reply in an attempt to receive an acknowledgement?

4.21 Ans.

The timeout period for storing a reply message is the maximum time that it is likely for any client to a re-transmit a request message. There is no definite value for this, and there is a trade-off between safety and buffer space. In the case of RRA, reply messages are generally discarded before the timeout period has expired because an acknowledgement is received. Suppose that a server using RRA re-transmits the reply message after a delay and consider the case where the client has sent an acknowledgement which was late or lost. This requires (i) the client to recognise duplicate reply messages and send corresponding extra acknowledgements and (ii) the server to handle delayed acknowledgments after it has re-transmitted reply messages. This possible improvement gives little reduction in storage requirements (corresponding to the occasional lost acknowledgement message) and is not convenient for the single threaded client which may be otherwise occupied and not be in a position to send further acknowledgements.

4.22 Why might the number of messages exchanged in a protocol be more significant to performance than the total amount of data sent? Design a variant of the RRA protocol in which the acknowledgement is piggy-backed on, that is, transmitted in the same message as, the next request where appropriate, and otherwise sent as a separate message. (Hint: use an extra timer in the client.)

4.22 Ans.

The time for the exchange of a message = A + B* length, where A is the fixed processing overhead and B is the rate of transmission. A is large because it represents significant processing at both sender and receiver; the sending of data involves a system call; and the arrival of a message is announced by an interrupt which must be handled and the receiving process is scheduled. Protocols that involve several rounds of messages tend to be expensive because of paying the A cost for every message.

The new version of RRA has:

client	*server*
cancel any outstanding Acknowledgement on a timer	
send Request	
	receive Request
	send Reply
receive Reply	
set timer to send Acknowledgement after delay T	
	receive Acknowledgement

The client always sends an acknowledgement, but it is piggy-backed on the next request if one arises in the next T seconds. It sends a separate acknowledgement if no request arises. Each time the server receives a request or an acknowledgement message from a client, it discards any reply message saved for that client.

4.23 IP multicast provides a service that suffers from omission failures. Make a test kit, possibly based on the program in Figure 4.17, to discover the conditions under which a multicast message is sometimes dropped by one of the members of the multicast group. The test kit should be designed to allow for multiple sending processes.

4.23 Ans.

The program in Figure 4.17 should be altered so that it can run as a sender or just a receiver. A program argument could specify its role. As in Exercise 4.3 the number of messages and their size should be variable and a sequence number should be sent with each one. Each recipient records the last sequence number from each sender (sender IP address can be retrieved from datagrams) and prints out any missing sequence numbers. Test with several senders and receivers and message sizes to discover the load required to cause dropped messages.The test kit should be designed to allow for multiple sending processes.

4.24 Outline the design of a scheme that uses message retransmissions with IP multicast to overcome the problem of dropped messages. Your scheme should take the following points into account:

i) there may be multiple senders;

ii) generally only a small proportion of messages are dropped;

iii) unlike the request-reply protocol, recipients may not necessarily send a message within any particular time limit.

Assume that messages that are not dropped arrive in sender ordering.

4.24 Ans.

To allow for point (i) senders must attach a sequence number to each message. Recipients record last sequence number from each sender and check sequence numbers on each message received.

For point (ii) a negative acknowledgement scheme is preferred (recipient requests missing messages, rather than acknowledging all messages). When they notice a missing message, they send a message to the sender to ask for it. To make this work, the sender must store all recently sent messages for retransmission. The sender re-transmits the messages as a unicast datagram.

Point (iii) - refers to the fact that we can't rely on a reply as an acknowledgement. Without acknowledgements, the sender will be left holding all sent messages in its store indefinitely. Possible solutions: a) senders discards stored messages after a time limit b) occasional acknowledgements from recipients which may be piggy backed on messages that are sent.

Note requests for missing messages and acknowledgments are simple - they just contain the sequence numbers of a range of lost messages.

4.25 Your solution to Exercise 4.24 should have overcome the problem of dropped messages in IP multicast. In what sense does your solution differ from the definition of reliable multicast?

4.25 Ans.

Reliable multicast requires that any message transmitted is received by all members of a group or none of them. If the sender fails before it has sent a message to all of the members (e.g. if it has to retransmit a message) or if a gateway fails, then some members will receive the message when others do not.

4.26 Devise a scenario in which multicasts sent by different clients are delivered in different orders at two group members. Assume that some form of message retransmissions are in use, but that messages that are not dropped arrive in sender ordering. Suggest how recipients might remedy this situation.

4.26 Ans.

Sender1 sends request r1 to members m1 and m2 but the message to m2 is dropped

Sender2 sends request r2 to members m1 and m2 (both arrive safely)

Sender1 re-transmits request r1 to member m2 (it arrives safely).

Member m1 receives the messages in the order r1;r2. However m2 receives them in the order r2;r1.

To remedy the situation. Each recipient delivers messages to its application in sender order. When it receives a message that is ahead of the next one expected, it hold it back until it has received and delivered the earlier re-transmitted messages.

4.27 Define the semantics for and design a protocol for a group form of request-reply interaction, for example using IP multicast.

4.27 Ans.

The group request-reply protocol should not use a request-reply to each member. Instead, the request message is multicast to all the members of the group. But there is a question as to how many replies should be transmitted. As examples: a request to a replicated or a partitioned service requires only one reply. In the first case from any member and in the second case from the server with the information. In contrast a request for a vote requires a majority of replies that agree; and a request for a reading on a sensor requires all replies. The semantics should allow the client to specify the number of replies required.

In a group request-reply protocol, the request message is multicast to all the members of the group, using a retransmission scheme like that of the request-reply protocol to deal with lost messages. This requires that each member returns either an acknowledgement or a reply. The client's communication software collates and filters these replies, returning the desired number of replies to the client.

The protocol must deal with the case when there are less replies that the number specified, either by selective re-transmission of the request messages or by repeating the multicast request.

Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001.

Chapter 5 Exercise Solutions

5.1 The *Election* interface provides two remote methods:

vote: with two parameters through which the client supplies the name of a candidate (a string) and the 'voter's number' (an integer used to ensure each user votes once only). The voter's numbers are allocated sparsely from the range of integers to make them hard to guess.

result: with two parameters through which the server supplies the client with the name of a candidate and the number of votes for that candidate.

Which of the parameters of these two procedures are *input* and which are *output* parameters?

5.1 Ans.

vote: input parameters: name of candidate, voter's number;
result: output parameters: name of candidate, number of votes

---

5.2 Discuss the invocation semantics that can be achieved when the request-reply protocol is implemented over a TCP/IP connection, which guarantees that data is delivered in the order sent, without loss or duplication. Take into account all of the conditions causing a connection to be broken.

5.2 Ans.

A process is informed that a connection is broken:

- when one of the processes exits or closes the connection.
- when the network is congested or fails altogether

Therefore a client process cannot distinguish between network failure and failure of the server.

Provided that the connection continues to exist, no messages are lost, therefore, every request will receive a corresponding reply, in which case the client knows that the method was executed exactly once.

However, if the server process crashes, the client will be informed that the connection is broken and the client will know that the method was executed either once (if the server crashed after executing it) or not at all (if the server crashed before executing it).

But, if the network fails the client will also be informed that the connection is broken. This may have happened either during the transmission of the request message or during the transmission of the reply message. As before the method was executed either once or not at all.

Therefore we have at-most-once call semantics.

5.3 Define the interface to the *Election* service in CORBA IDL and Java RMI. Note that CORBA IDL provides the type *long* for 32 bit integers. Compare the methods in the two languages for specifying *input* and *output* arguments.

5.3 Ans.

CORBA IDL:

```
interface Election {
    void vote(in string name, in long number);
    void result(out string name, out long votes);
};
```

Java RMI

We need to define a class for the result e.g.

```
class Result {
    String name;
    int votes;
}
```

The interface is:

```
import java.rmi.*;
public interface Election extends Remote{
    void vote(String name, int number) throws RemoteException;
    Result result () throws RemoteException;
};
```

This example shows that the specification of input arguments is similar in CORBA IDL and Java RMI.

This example shows that if a method returns more than one result, Java RMI is less convenient than CORBA IDL because all output arguments must be packed together into an instance of a class.

5.4 The *Election* service must ensure that a vote is recorded whenever any user thinks they have cast a vote.

Discuss the effect of maybe call semantics on the *Election* service.

Would at-least-once call semantics be acceptable for the *Election* service or would you recommend at-most-once call semantics?

5.4 Ans.

Maybe call semantics is obviously inadequate for *vote*! Ex 5.1 specifies that the voter's number is used to ensure that the user only votes once. This means that the server keeps a record of who has voted. Therefore at-least-once semantics is alright, because any repeated attempts to vote are foiled by the server.

5.5 A request-reply protocol is implemented over a communication service with omission failures to provide at-least-once RMI invocation semantics. In the first case the implementor assumes an asynchronous distributed system. In the second case the implementor assumes that the maximum time for the communication and the execution of a remote method is T. In what way does the latter assumption simplify the implementation?

5.5 Ans.

In the first case, the implementor assumes that if the client observes an omission failure it cannot tell whether it is due to loss of the request or reply message, to the server having crashed or having taken longer than usual. Therefore when the request is re-transmitted the client may receive late replies to the original request. The implementation must deal with this.

In the second case, an omission failure observed by the client cannot be due to the server taking too long. Therefore when the request is re-transmitted after time T, it is certain that a late reply will not come from the server. There is no need to deal with late replies

---

5.6 Outline an implementation for the *Election* service that ensures that its records remain consistent when it is accessed concurrently by multiple clients.

5.6 Ans.

Suppose that each vote in the form {*String vote*, *int number*} is appended to a data structure such as a Java *Vector*. Before this is done, the voter number in the request message must be checked against every vote recorded in the *Vector*. Note that an array indexed by voter's number is not a practical implementation as the numbers are allocated sparsely.

The operations to access and update a *Vector* are synchronized, making concurrent access safe.

Alternatively use any form of synchronization to ensure that multiple clients' access and update operations do not conflict with one another.

5.7 The *Election* service must ensure that all votes are safely stored even when the server process crashes. Explain how this can be achieved with reference to the implementation outline in your answer to Exercise 5.6.

5.7 Ans.

The state of the server must be recorded in persistent storage so that it can be recovered when the server is restarted. It is essential that every successful vote is recorded in persistent storage before the client request is acknowledged.

A simple method is to serialize the *Vector* of votes to a file after each vote is cast.

A more efficient method would append the serialized votes incrementally to a file.

Recovery will consist of de-serializing the file and recreating a new vector.

5.8 Show how to use Java reflection to construct the client proxy class for the *Election* interface. Give the details of the implementation of one of the methods in this class, which should call the method *doOperation* with the following signature:

byte [] doOperation (RemoteObjectRef o, Method m, byte[] arguments);

Hint: an instance variable of the proxy class should hold a remote object reference (see Exercise 4.12).

5.8 Ans.

Use classes *Class* and *Method*. Use type *RemoteObjectRef* as type of instance variable. The class *Class* has method *getMethod* whose arguments give class name and an array of parameter types. The proxy's *vote* method, should have the same parameters as the vote in the remote interface - that is: two parameters of type *String* and *int*. Get the object representing the *vote* method from the class *Election* and pass it as the second argument of *doOperation*. The two arguments of *vote* are converted to an array of byte and passed as the third argument of *doOperation.*

```
import java.lang.reflect;

class VoteProxy {
      RemoteObjectRef ref;
       private static Method voteMethod;
       private static Method resultMethod;
      static {
        try {
            voteMethod = Election.class.getMethod ("vote", new Class[]
                  {java.lang.String.class,int.class}));
            resultMethod = Election.class.getMethod ("result", new Class[] {}));
        }catch(NoSuchMethodException){}
      }

public void vote  (String arg1, int arg2) throws RemoteException {
            try {
                  byte args [] = // convert arguments arg1 and arg2 to an array of bytes
                  byte result = DoOperation(ref, voteMethod, args);
                  return ;
            } catch (...) {}
}
```

5.9 Show how to generate a client proxy class using a language such as C++ that does not support reflection, for example from the CORBA interface definition given in your answer to Exercise 5.3. Give the details of the implementation of one of the methods in this class, which should call the method *doOperation* defined in Figure 4.12.

5.9 Ans.

Each proxy method is generated from the signature of the method in the IDL interface,e.g.

void vote(in string name, in long number);

An equivalent stub method in the client language e.g. C++ is produced e.g.

*void vote(const char *vote, int number)*

Each method in the interface is given a number e.g. vote = 1, result = 2.

use *char args[length of string + size of int]* and marshall two arguments into this array and call *doOperation* as follows:

*char * result = DoOperation(ref, 1, args);*

we still assume that *ref* is an instance variable of the proxy class. A marshalling method is generated for each argument type used.

5.10 Explain how to use Java reflection to construct a generic dispatcher. Give Java code for a dispatcher whose signature is:

public void dispatch(Object target, Method aMethod, byte[] args)

The arguments supply the target object, the method to be invoked and the arguments for that method in an array of bytes.

5.10 Ans.

Use the class *Method*. To invoke a method supply the object to be invoked and an array of *Object* containing the arguments. The arguments supplied in an array of bytes must be converted to an array of *Object*.

```
public void dispatch(Object target, Method aMethod, byte[] args)
    throws RemoteException {
    Object[] arguments = // extract arguments from array of bytes
    try{
        aMethod.invoke(target, arguments);
    catch(...){}
}
```

5.11 Exercise 5.8 required the client to convert *Object* arguments into an array of bytes before invoking *doOperation* and Exercise 5.10 required the dispatcher to convert an array of bytes into an array of *Object*s before invoking the method. Discuss the implementation of a new version of *doOperation* with the following signature:

Object [] doOperation (RemoteObjectRef o, Method m, Object[] arguments);

which uses the *ObjectOutputStream* and *ObjectInputStream* classes to stream the request and reply messages between client and server over a TCP connection. How would these changes affect the design of the dispatcher?

5.11 Ans.

The method *DoOperation* sends the invocation to the target's remote object reference by setting up a TCP connection (as shown in Figures 4.5 and 4.6) to the host and port specified in *ref*. It opens an *ObjectOutputStream* and uses *writeObject* to marshal *ref*, the method, *m* and the arguments by serializing them to an *ObjectOutputStream*. For the results, it opens an *ObjectIntputStream* and uses *readObject* to get the results from the stream.

At the server end, the dispatcher is given a connection to the client and opens an *ObjectIntputStream* and uses *readObject* to get the arguments sent by the client. Its signature will be:

public void dispatch(Object target, Method aMethod)

5.12 A client makes remote procedure calls to a server. The client takes 5 milliseconds to compute the arguments for each request, and the server takes 10 milliseconds to process each request. The local operating system processing time for each send or receive operation is 0.5 milliseconds, and the network time to transmit each request or reply message is 3 milliseconds. Marshalling or unmarshalling takes 0.5 milliseconds per message.

Calculate the time taken by the client to generate and return from two requests:

(i) if it is single-threaded, and

(ii) if it has two threads that can make requests concurrently on a single processor.

You can ignore context-switching times. Is there a need for asynchronous RPC if client and server processes are threaded?

5.12 Ans.

i) time per call = calc. args + marshal args + OS send time + message transmission +
OS receive time + unmarshall args + execute server procedure
+ marshall results + OS send time + message transmission +
OS receive time + unmarshal args
= 5 + 4*marshal/unmarshal + 4*OS send/receive + 2*message transmission + execute server procedure
= 5+ 4*0.5 + 4*0.5 + +2*3 + 10 ms = 5+2+2+6+10 =25ms.
Time for two calls = 50 ms.

ii) threaded calls:
client does calc. args + marshal args + OS send time (call 1) = 5+.5=.5 = 6
then calc args + marshal args + OS send time (call 2) = 6
= 12 ms then waits for reply from first call

server gets first call after
message transmission + OS receive time + unmarshal args = 6+ 3+.5+.5
= 10 ms, takes 10+1 to execute, marshal, send at 21 ms
server receives 2nd call before this, but works on it after 21 ms taking
10+1, sends it at 32 ms from start
client receives it 3+1 = 4 ms later i.e. at 36 ms
(message transmission + OS receive time + unmarshal args) later
Time for 2 calls = 36 ms.

5.13 Design a remote object table that can support distributed garbage collection as well as translating between local and remote object references. Give an example involving several remote objects and proxies at various sites to illustrate the use of the table. Show what happens when an invocation causes a new proxy to be created. Then show what happens when one of the proxies becomes unreachable.

5.13 Ans..

local reference	*remote reference*	*holders*

The table will have three columns containing the local reference and the remote reference of a remote object and the virtual machines that currently have proxies for that remote object. There will be one row in the table for each remote object exported at the site and one row for each proxy held at the site.

To illustrate its use, suppose that there are 3 sites with the following exported remote objects:
S1: A1, A2, A3 S2: B1, B2; S3: C1;

and that proxies for A1 are held at S2 and S3; a proxy for B1 is held at S3.

Then the tables hold the following information:.

at S1			*at S2*			*at S3*		
local	*remote*	*holders*	*local*	*remote*	*holders*	*local*	*remote*	*holders*
a1	*A1*	*S2, S3*	*b1*	*B1*	*S3*	*c1*	*C1*	
a2	*A2*		*b2*	*B2*		*a1*	*A1proxy*	
a3	*A3*		*a1*	*A1proxy*		*b1*	*B1proxy*	

Now suppose that C1(at S3) invokes a method in B1 causing it to return a reference to B2. The table at S2 adds the holder S3 to the entry for B2 and the table at S3 adds a new entry for the proxy of B2.

Suppose that the proxy for A1 at S3 becomes unreachable. S3 sends a message to S1 and the holder S3 is removed from A1. The proxy for A1 is removed from the table at S3.

5.14 A simpler version of the distributed garbage collection algorithm described in Section 5.2.6 just invokes *addRef* at the site where a remote object lives whenever a proxy is created and *removeRef* whenever a proxy is deleted. Outline all the possible effects of communication and process failures on the algorithm. Suggest how to overcome each of these effects, but without using leases.

5.14 Ans.

AddRef message lost - the owning site doesn't know about the client's proxy and may delete the remote object when it is still needed. (The client does not allow for this failure).

RemoveRef message lost - the owning site doesn't know the remote object has one less user. It may continue to keep the remote object when it is no longer needed.

Process holding a proxy crashes - owning site may continue to keep the remote object when it is no longer needed.

Site owning a remote object crashes. Will not affects garbage collection algorithm

Loss of *addRef* is discussed in the Section 5.2.6.

When a *removeRef* fails, the client can repeat the call until either it succeeds or the owner's failure has been detected.

One solution to a proxy holder crashing is for the owning sites to set failure detectors on holding sites and then remove holders after they are known to have failed.

5.15 Discuss how to use events and notifications as described in the Jini distributed event specification in the context of the shared whiteboard application. The *RemoteEvent* class is defined as follows in Arnold *et al*. [1999].

```
public class RemoteEvent extends java.util.EventObject {
public RemoteEvent(Object source, long eventID,
     long seqNum, MarshalledObject handback)
public Object getSource () {…}
public long getID() {…}
public long getSequenceNumber() {…}
public MarshalledObject getRegistrationObject() {…}
}
```

The first argument of the constructor is a remote object. Notifications inform listeners that an event has occurred but the listeners are responsible for obtaining further details.

5.15 Ans.

Event identifier,. *evID*s. Decided by the *EventGenerator*. Simplest solution is just to have one type of event - the addition of a new *GraphicalObject*. Other event types could for example refer to deletion of a *GraphicalObject*.

Clients need to be notified of the remote object reference of each new *GraphicalObject* that is added to the server. Suppose that an object in the server is the *EventGenerator*. It could implement the *EventGenerator* interface and provide the *register* operation, or it could be done more simply

```
e.g. addListener(RemoteEventListener  listener, long evID)
     // add this listener to a vector of listeners
```

This would be better if *Leases* were used to avoid dealing with lost clients.

The *newShape* method of *shapeListServant* (Figure 5.14) could be the event generator. It will notify all of the *EventListeners* that have registered with it, each time a new *GraphicalObject* is added. e.g.

```
RemoteEvent event = new RemoteEvent(this, ADD_EVENT, version, null)
for all listeners in the vector
     listener.notify(event)
```

Each client creates an *RemoteEventListener* for receiving notifications of events and then registers interest in events with the server, passing the *EventListener* as argument.

```
class MyListener implements RemoteEventListener {
     public MyListener() throws RemoteException[
     }

     public void notify(RemoteEvent event) throws UnknownEventException,
                                  RemoteException {
          Object source = getSource();
          long id = event.getID();
          long version = event.getSequenceNumber();
          // get the newly created GraphicalObject from the server
     }
}
```

Then to become a listener (add the following to the client program shown in Figure 5.15):

```
sList.addListener(new MyListener(), ADD_EVENT);
```

The client getting the *newGraphicalObject* needs to be able to get it directly from the version number, rather than by getting the list of *Shapes* and then getting the *GraphicalObject*. The interface to *ShapeList* could be amended to allow this.

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

5.16 Ans.

The Mailbox service will provide an interface allowing a client to register interest in another object. The client will need to know the *RemoteEventListener* provided by the Mailbox service so that notifications may be passed from event generators to the *RemoteEventListener* and then on to the client. The client will also need a means of interacting with the Mailbox service so as to turn delivery on and off. Therefore define register as follows:

Registration register() ...

The result is a reference to a remote object whose methods enable the client to get a reference to a *RemoteEventListener* and to turn delivery on and off.

To use the Mailbox service, the client registers with it and receives a Registration object, which it saves in a file. It registers the *RemoteEventListener* provided by the Mailbox service with all of the *EventGenerators* whose events it wants to have notification of. If the client crashes, it can restore the *Registration* object when it restarts. Whenever it wants to receive events it turns delivery on and when it does not want them it turns delivery off.

The design should make it possible to specify a lease for each subscriber.

5.17 Explain how a forwarding observer may be used to enhance the reliability and performance of objects of interest in an event service.

5.17 Ans.

Reliability:

The forwarding observer can retry notifications that fail at intervals of time.

If the forwarding observer is on the same computer as the object of interest, then the two could not fail independently.

Performance:

The forwarding observer can optimize multicast protocols to subscribers.
In Jini it could deal with renewing leases.

5.18 Suggest ways in which observers can be used to improve the reliability or performance of your solution to Exercise 5.13.

5.18 Ans.

The server can be relieved of saving information about all of the clients' interests by creating a forwarding agent on the same computer. the forwarding agent could use a multicast protocol to send notifications to the clients. IP multicast would do since it is not crucial that every notification be received. A missed version number can be rectified as soon as another one is received.

Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001

Chapter 6 Exercise Solutions

6.1 Discuss each of the tasks of encapsulation, concurrent processing, protection, name resolution, communication of parameters and results, and scheduling in the case of the UNIX file service (or that of another kernel that is familiar to you).

6.1 Ans.

We discuss the case of a single computer running Unix.

Encapsulation: a process may only access file data and attributes through the system call interface.

Concurrent processing: several processes may access the same or different files concurrently; a process that has made a system call executes in supervisor mode in the kernel; processes share all file-system-related data, including the block cache.

Protection: users set access permissions using the familiar *user/group/other*, *rwx* format. Address space protection and processor privilege settings are used to restrict access to file data and file system data in memory and prevent direct access to storage devices. Processes bear user and group identifiers in protected kernel tables, so the problem of authentication does not arise.

Name resolution: pathnames (for example, */usr/fred*) are resolved by looking up each component in turn. Each component is looked up in a directory, which is a table containing path name components and the corresponding inodes. If the inode is that of a directory then this is retrieved; and the process continues until the final component has been resolved or an error has occurred. Special cases occur when a symbolic link or mount point is encountered.

Parameter and result communication: parameters and results can be communicated by a) passing them in machine registers, b) copying them between the user and kernel address spaces, or c) by mapping data blocks simultaneously in the two address spaces.

Scheduling: there are no separate file system threads; when a user process makes a file system call, it continues execution in the kernel.

NFS is discussed in Section 8.2.

6.2 Why are some system interfaces implemented by dedicated system calls (to the kernel), and others on top of message-based system calls?

6.2 Ans.

Dedicated system calls are more efficient for simple calls than message-based calls (in which a system action is initiated by sending a message to the kernel, involving message construction, dispatch etc.).

However, the advantage of implementing a system call as an RPC is that then a process can perform operations transparently on either remote or local resources.

6.3 Smith decides that every thread in his processes ought to have its own *protected* stack – all other regions in a process would be fully shared. Does this make sense?

6.3 Ans.

If every thread has its own *protected* stack, then each must have its own address space. Smith's idea is better described as a set of single-threaded processes, most of whose regions are shared. The advantage of sharing an address space has thus been lost.

6.4 Should signal (software interrupt) handlers belong to a process or to a thread?

6.4 Ans.

When a process experiences a signal, should any currently running thread handle it, or should a pre-arranged thread handle it? And do threads have their own tables of signal handlers? There is no ’correct’ answer to these questions. For example, it might be convenient for threads within a process to use their own SIGALARM signal handlers. On the other hand, a process only needs one SIGINT handler.

6.5 Discuss the issue of naming applied to shared memory regions.

6.5 Ans.

Memory regions may be given textual names. Two or more processes may then share a region by supplying its name and requesting it to be mapped into their address spaces. Mapped files are examples of this idea, in which the contents of the shared region are maintained on persistent store.

The second naming issue is that of the addresses used to access a shared region. In principle, a shared region may be mapped as different address ranges in different processes. However, it is more convenient if the same address ranges are used, for the region may then contain pointers into itself, without the need for translation. Unfortunately, in the absence of global address space management, some addresses belonging to a particular region might already be occupied in some processes.

6.6 Suggest a scheme for balancing the load on a set of computers. You should discuss:

i) what user or system requirements are met by such a scheme;

ii) to what categories of applications it is suited;

iii) how to measure load and with what accuracy; and

iv) how to monitor load and choose the location for a new process. Assume that processes may not be migrated.

How would your design be affected if processes could be migrated between computers? Would you expect process migration to have a significant cost?

6.6 Ans.

The following brief comments are given by way of suggestion, and are not intended to be comprehensive:

i) Examples of requirements, which an implementation might or might not be designed to meet: good interactive response times despite load level; rapid turnaround of individual compute-intensive jobs; simultaneous scheduling of a set of jobs belonging to a parallel application; limit on load difference between least- and most-loaded computers; jobs may be run on otherwise idle or under-utilized workstations; high throughput, in terms of number of jobs run per second; prioritization of jobs.

ii) A load-balancing scheme should be designed according to a job profile. For example, job behaviour (total execution time, resource requirements) might be known or unknown in advance; jobs might be typically interactive or typically compute-intensive or a mixture of the two; jobs may be parts of single parallel programs. Their total run-time may on average be one second or ten minutes. The efficacy of load balancing is doubtful for very light jobs; for short jobs, the system overheads for a complex algorithm may outweigh the advantages.

iii) A simple and effective way to measure a computer’s load is to find the length of its run queue. Measuring load using a crude set of categories such as LIGHT and HEAVY is often sufficient, given the overheads of collecting finer-grained information, and the tendency for loads to change over short periods.

iv) A useful approach is for computers whose load is LIGHT to advertise this fact to others, so that new jobs are started on these computers.

Because of unpredictable job run times, any algorithm might lead to unbalanced computers, with a temporary dearth of new jobs to place on the LIGHT computers. If process migration is available, however, then jobs can be relocated from HEAVY computers to LIGHT computers at such times. The main cost of process migration is address space transfer, although techniques exist to minimise this [Kindberg 1990]. It can take in the order of seconds to migrate a process, and this time must be short in relation to the remaining run times of the processes concerned.

6.7 Explain the advantage of copy-on-write region copying for UNIX, where a call to *fork* is typically followed by a call to *exec*. What should happen if a region that has been copied using copy-on-write is itself copied?

6.7 Ans.

It would be wasteful to copy the forked process's address space contents, since they are immediately replaced. With copy-on-write, only those few pages that are actually needed before calling *exec* will be copied.

Assume now that *exec* is not called. When a process forks, its child may in turn fork a child. Call these the parent, child and grandchild. Pages in the grandchild are logically copied from the child, whose pages are in turn logically copied from the parent. Initially, the grandchild's pages may share a frame with the parent's page. If, say, the child modifies a page, however, then the grandchild's page must be made to share the child's frame. A way must be found to manage chains of page dependencies, which have to be altered when pages are modified.

6.8 A file server uses caching, and achieves a hit rate of 80%. File operations in the server cost 5 ms of CPU time when the server finds the requested block in the cache, and take an additional 15 ms of disk I/O time otherwise. Explaining any assumptions you make, estimate the server's throughput capacity (average requests/sec) if it is:

i) single-threaded;

ii) two-threaded, running on a single processor;

iii) two-threaded, running on a two-processor computer.

6.8 Ans.

80% of accesses cost 5 ms; 20% of accesses cost 20 ms.

average request time is 0.8*5+.2*20 = 4+4=8ms.

i) single-threaded: rate is 1000/8 = 125 reqs/sec

ii) two-threaded: serving 4 cached and 1 uncached requests takes 25 ms. (overlap I/O with computation). Therefore throughput becomes 1 request in 5 ms. on average, = 200 reqs/sec

iii) two-threaded, 2 CPUs: Processors can serve 2 rqsts in 5 ms => 400 reqs/sec. But disk can serve the 20% of requests at only 1000/15 reqs/sec (assume disk rqsts serialised). This implies a total rate of 5*1000/15 = 333 requests/sec (which the two CPUs can service).

6.9 Compare the worker pool multi-threading architecture with the thread-per-request architecture.

6.9 Ans.

The worker pool architecture saves on thread creation and destruction costs compared to the thread-per-request architecture but (a) the pool may contain too few threads to maximise performance under high workloads or too many threads for practical purposes and (b) threads contend for the shared work queue.

6.10 What thread operations are the most significant in cost?

6.10 Ans.

Thread switching tends to occur many times in the lifetime of threads and is therefore the most significant cost. Next come thread creation/destruction operations, which occur often in dynamic threading architectures (such as the thread-per-request architecture).

6.11 A spin lock (see Bacon [1998]) is a boolean variable accessed via an atomic *test-and-set* instruction, which is used to obtain mutual exclusion. Would you use a spin lock to obtain mutual exclusion between threads on a single-processor computer?

6.11 Ans.

The problem that might arise is the situation in which a thread spinning on a lock uses up its timeslice, when meanwhile the thread that is about to free the lock lies idle on the READY queue. We can try to avoid this problem by integrating lock management with the scheduling mechanism, but it is doubtful whether this would have any advantages over a mutual exclusion mechanism without busy-waiting.

6.12 Explain what the kernel must provide for a user-level implementation of threads, such as Java on UNIX.

6.12 Ans.

A thread that makes a blocking system call provides no opportunity for the user-level scheduler to intervene, and so all threads become blocked even though some may be in the READY state. A user-level implementation requires (a) non-blocking (asynchronous) I/O operations provided by the kernel, that only initiate I/O; and (b) a way of determining when the I/O has completed -- for example, the UNIX *select* system call. The threads programmer should not use native blocking system calls but calls in the threading API which make an asynchronous call and then invoke the scheduler.

6.13 Do page faults present a problem for user-level threads implementations?

6.13 Ans.

If a process with a user-level threads implementation takes a page fault, then by default the kernel will deschedule the entire process. In principle, the kernel could instead generate a software interrupt in the process, notifying it of the page fault and allowing it to schedule another thread while the page is fetched.

6.14 Explain the factors that motivate the hybrid scheduling approach of the 'scheduler activations' design (instead of pure user-level or kernel-level scheduling).

6.14 Ans.

A hybrid scheduling scheme combines the advantages of user-level scheduling with the degree of control of allocation of processors that comes from kernel-level implementations. Efficient, custom scheduling takes place inside processes, but the allocation of a multiprocessor's processors to processes can be globally controlled.

6.15 Why should a threads package be interested in the events of a thread's becoming blocked or unblocked? Why should it be interested in the event of a virtual processor's impending preemption? (Hint: other virtual processors may continue to be allocated.)

6.15 Ans.

If a thread becomes blocked, the user-level scheduler may have a READY thread to schedule. If a thread becomes unblocked, it may become the highest-priority thread and so should be run.

If a virtual processor is to be preempted, then the user-level scheduler may re-assign user-level threads to virtual processors, so that the highest-priority threads will continue to run.

6.16 Network transmission time accounts for 20% of a null RPC and 80% of an RPC that transmits 1024 user bytes (less than the size of a network packet). By what percentage will the times for these two operations improve if the network is upgraded from 10 megabits/second to 100 megabits/second?

6.16 Ans.

T_{null} = null RPC time = $f + w_{null}$, where f = fixed OS costs, w_{null} = time on wire at 10 megabits-per-second.

Similarly, T_{1024} = time for RPC transferring 1024 bytes = $f + w_{1024}$.

Let T'_{null} and T'_{1024} be the corresponding figures at 100 megabits per second. Then

$T'_{null} = f + 0.1w_{null}$, and $T'_{1024} = f + 0.1w_{1024}$.

Percentage change for the null RPC = $100(T_{null} - T'_{null})/T_{null} = 100*0.9w_{null} /T_{null} = 90*0.2 = 18\%$.

Similarly, percentage change for 1024-byte RPC = $100*0.9*0.8 = 72\%$.

6.17 A 'null' RMI that takes no parameters, calls an empty procedure and returns no values delays the caller for 2.0 milliseconds. Explain what contributes to this time.

In the same RMI system, each 1K of user data adds an extra 1.5 milliseconds. A client wishes to fetch 32K of data from a file server. Should it use one 32K RMI or 32 1K RMIs?

6.17 Ans.

Page 236 details the costs that make up the delay of a null RMI.

one 32K RMI: total delay is 2 + 32*1.5 = 50 ms.

32 1K RMIs: total delay is 32(2+1.5) = 112 ms -- one RMI is much cheaper.

6.18 Which factors identified in the cost of a remote invocation also feature in message passing?

6.18 Ans.

Most remote invocation costs also feature in message passing. However, if a sender uses asynchronous message passing then it is not delayed by scheduling, data copying at the receiver or waiting for acknowledgements

6.19 Explain how a shared region could be used for a process to read data written by the kernel. Include in your explanation what would be necessary for synchronization.

6.19 Ans.

The shared region is mapped read-only into the process's address space, but is writable by the kernel.The process reads data from the region using standard LOAD instructions (avoiding a TRAP). The process may poll the data in the region from time to time to see if it has changed. However, we may require a way for the kernel to notify the process when it has written new data. A software interrupt can be used for this purpose.

6.20 i) Can a server invoked by lightweight procedure calls control the degree of concurrency within it?

ii) Explain why and how a client is prevented from calling arbitrary code within a server under lightweight RPC.

iii) Does LRPC expose clients and servers to greater risks of mutual interference than conventional RPC (given the sharing of memory)?

6.20 Ans.

i) Although a server using LRPC does not explicitly create and manage threads, it can control the degree of concurrency within it by using semaphores within the operations that it exports.

ii) A client must not be allowed to call arbitrary code within the server, since it could corrupt the server's data. The kernel ensures that only valid procedures are called when it mediates the thread's upcall into the server, as explained in Section 6.5.

iii) In principle, a client thread could modify a call's arguments on the A-stack, while another of the client's threads, executing within the server, reads these arguments. Threads within servers should therefore copy all arguments into a private region before attempting to validate and use them. Otherwise, a server's data is entirely protected by the LRPC invocation mechanism.

6.21 A client makes RMIs to a server. The client takes 5 ms to compute the arguments for each request, and the server takes 10ms to process each request. The local OS processing time for each *send* or *receive* operation is 0.5 ms, and the network time to transmit each request or reply message is 3 ms. Marshalling or unmarshalling takes 0.5 ms per message.

Estimate the time taken by the client to generate and return from 2 requests (i) if it is single-threaded, and (ii) if it has two threads which can make requests concurrently on a single processor. Is there a need for asynchronous RMI if processes are multi-threaded?

6.21 Ans.

(i) Single-threaded time: 2(5 (prepare) + 4(0.5 (marsh/unmarsh) + 0.5 (local OS)) + 2*3 (net)) + 10 (serv))
= 50 ms.

(ii) Two-threaded time: (see figure 6.14) because of the overlap, the total is that of the time for the first operation's request message to reach the server, for the server to perform all processing of both request and reply messages without interruption, and for the second operation's reply message to reach the client.
This is: 5 + (0.5+0.5+3) + (0.5+0.5+10+0.5+0.5) + (0.5+0.5+10+0.5+0.5) + (3 + 0.5+0.5)
= 37ms.

6.22 Explain what is security policy and what are the corresponding mechanisms in the case of a multi-user operating system such as UNIX.

6.22 Ans.

Mechanisms: see answer to 6.1.

Policy concerns the application of these mechanisms by a particular user or in a particular working environment. For example, the default access permissions on new files might be "rw-------" in the case of an environment in which security is a high priority, and "rw-r--r--" where sharing is encouraged.

6.23 Explain the program linkage requirements that must be met if a server is to be dynamically loaded into the kernel's address space, and how these differ from the case of executing a server at user level.

6.23 Ans.

Portions of the kernel's address space must be allocated for the new code and data. Symbols within the new code and data must be resolved to items in the kernel's address space. For example, it would use the kernel's message-handling functions.

By contrast, if the server was to execute as a separate process then it would run from a standard address in its own address space and, apart from references to shared libraries, its linked image would be self-contained.

6.24 How could an interrupt be communicated to a user-level server?

6.24 Ans.

The interrupt handler creates a message and sends it, using a special non-blocking primitive, to a predetermined port which the user-level server owns.

6.25 On a certain computer we estimate that, regardless of the OS it runs, thread scheduling costs about 50 μs, a null procedure call 1 μs, a context switch to the kernel 20 μs and a domain transition 40 μs. For each of Mach and SPIN, estimate the cost to a client of calling a dynamically loaded null procedure.

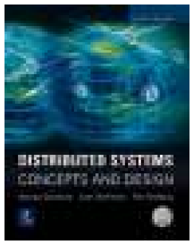
6.25 Ans.

Mach, by default, runs dynamically loaded code in a separate address space. So invoking the code involves control transfer to a thread in a separate address space. This involves four (context switch + domain transitions) to and from the kernel as well as two schedulings (client to server thread and server thread to client thread) -- in addition to the null procedure itself.

Estimated cost: 4(20 + 40) + 2*50 + 1 = 341 μs

In SPIN, the call involve two (context switch + domain transitions) and no thread scheduling.

Estimated cost: 2(20 + 40) + 1 = 121 μs.



Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001

Chapter 7 Exercise Solutions

7.1 Describe some of the physical security policies in your organization. Express them in terms that could be implemented in a computerized door locking system.

7.1 Ans.

For QMW Computer Science Department:

- staff have access to all areas of the department except the offices of others, at all hours;
- students of the department have access to teaching laboratories and classrooms at all times except 0.00 to 0.400 and to other areas of the department, except private offices, during office hours;
- students of other departments taking courses in Computer Science have access to classrooms at all times and to teaching laboratories at designated times according tothe the courses taken;
- visitors have access to the Departmental Office during office hours;
- a master key holder has access to all offices.

Note:

- Access rights should be withdrawn when a user ceases to be a member of staff or a student.
- Changes in policy should be immediately effective.

7.2 Describe some of the ways in which conventional email is vulnerable to eavesdropping, masquerading, tampering, replay, denial of service. Suggest methods by which email could be protected against each of these forms of attack.

7.2 Ans.

Possible weaknesses for a typical mail system with SMTP delivery and client pickup from POP or IMAP mail host on a local network:

Weakness	*Types of attack*	*remedy*
Sender is unauthenticated.	Masquerading, denial of service.	End-to-end authentication with digital signatures (e.g. using PGP)
Message contents not authenticated.	Tampering, masquerading.	End-to-end authentication with digital signatures (e.g. using PGP).
Message contents in the clear.	Eavesdropping.	End-to-end encryption (e.g. using PGP).
Delivery and deletion from POP/IMAP server is authenticated only by a login with password.	Masquerading.	Kerberos or SSL authentication of clients.
Sender's clock is not guranteed.	False dating of messages.	Include time certificates from a trusted time service.

7.3 Initial exchanges of public keys are vulnerable to the man-in-the-middle attack. Describe as many defences against it as you can.

7.3 Ans.

1. Use a private channel for the delivery of initial keys, such as a CDROM delivered by hand or by some other rellable method.
2. Include the Domain Name in the certificate and deal only with the correct corresponding IP address.
3. If certificates are delivered through the network, validate them with a 'key fingerprint' – a character string that is derived from the key with a standard one-way function - that was delivered by a separate channel (e.g. on a business card).

7.4 PGP is widely used to secure email communication. Describe the steps that a pair of users using PGP must take before they can exchange email messages with privacy and authnticity guarantees. What scope is there to make the preliminary negotiations invisible to the users?

7.4 Ans.

PGP is based on a hybrid protocol like those described on pages 264 and 281. Its primary use is for secure email communication. It provides digital signatures for the authentication of messages string encryption for their secrecy and integrity. The signatures are made using the SHA-1 algorithm to make a digest of the message and RSA or DSS for signing with the sender's private key.

The message is (optionally) encrypted with 3DES or IDEA, using a one-time session key generated by the sender, which is encrypted using RSA with the recipient's public key and sent with the message.

PGP is required to generate public/private key pairs for each user and the one-time session keys used to encrypt messages. Users' public/private keys should be changed from time-to-time. (No keys should be used indefinitely in a secure system because of the danger thst they may be compromised through inadvertent disclosure or as a result of an attack.) To achieve the rotation of public/private key pairs, PGP must generate and store multiple key pairs and give each pair a label or identifer.

Key management is based on a *key ring* held by each user and a collection of PGP key servers accessible on the Internet that hold only the public keys of registered PGP users. The key ring is simply a small database holding keys in data structures that are secure. They are secured using secret key encryption with a *pass phrase* that the use must type in order to allow applications to access the keys in the keyring.

If PGP is thoroughly integrated into an email or other application the necessary actions to generate keys, access the key ring and perform signing and encryption on email messages can all be triggered automatically. The only user action required is the input of the *pass phrase* to decrypt the keyring entries. If users are equipped with smart cards or other physical access keys, the pass phrase could be supplied from the card.

7.5 How could email be sent to a list of 100 recipients using PGP or a similar scheme? Suggest a scheme that is simpler and faster when the list is used frequently.

7.5 Ans.

The idea of this exercise is to contrast the need for PGP to encrypt the session key n times (once in the public key of each user) with a scheme where the members of a group would share a single session key. The management and renewal of the shared key can be more easily achieved if the mailing list members are represented as members of a multicast group (pp. 436 *et seq*.).

7.6 The implementation of the TEA symmetric encryption algorithm given in Figure 7.8–7.10 is not portable between all machine architectures. Explain why. How could a message encrypted using the TEA implementation be transmitted to decrypt it correctly on all other architectures?

7.6 Ans.

Byte ordering is an issue. The algorithm as presented assumes that the 4 bytes in a 32-bit word are ordered the same at the sender (encrypter) and the receiver (decrypter). To make it work for all architectures, we would need to transmit messages in a network-standard byte order, and to re-order the bytes to suite the local architecture on receipt.

7.7 Modify the TEA application program in Figure 7.10 to use cipher block chaining (CBC).

7.7 Ans.

Left for the reader.

7.8 Construct a stream cipher application based on the program in Figure 7.10.

7.8 Ans.

Left for the reader.

7.9 Estimate the time required to crack a 56-bit DES key by a brute-force attack using a 500 MIPS (million instruction per second) workstation, assuming that the inner loop for a brute-force attack program involves around 10 instructions per key value, plus the time to encrypt an 8-byte plaintext (see Figure 7.14). Perform the same calculation for a 128-bit IDEA key. Extrapolate your calculations to obtain the time for a 50,000 MIPS parallel processor (or an Internet consortium with similar processing power).

7.9 Ans.

Suppose we have a computer with a 64-bit, 500 MIP cpu, and a short sample (8 bytes or one 64-bit word) of plain text with the corresponding encrypted text. A program can be constructed with an inner loop of N instructions, to generate all possible key values in the range 0 – (2^{56} - 1) and apply the an encryption algorithm to the plain text with each key value in turn. If it takes T_{enc} seconds to apply the encryption algorithm to an 8-byte plain text, then we have the following estimate of the average time t to crack a key of length L by brute force:

$$t = \frac{2^L}{2}(N/(5 \times 10^8) + T_{enc}) \text{ seconds}$$

If N= 10 (i.e. we require an inner loop of 10 instructions) and T_{enc} = 8/(7.746 x 10^6) seconds for the DES algorithm (i.e. the fastest time to encrypt 8 bytes given in Figure 7.14), we have:

$$t = 2^{55}(10/(500 \cdot 10^6) + 8/(7.764 \cdot 10^6)) \approx 3.8 \times 10^{10} \text{ seconds}$$

i.e. about 1200 years. A 50,000 MIPs parallel processor is 100 times faster, so assuming an efficient parallel algorithm, the cracking time would be ~12 years.

For IDEA, the equation is:

$$t = 2^{128}(10/(500 \cdot 10^6) + 8/(7.764 \cdot 10^6)) \approx 3.6 \times 10^{32} \text{ seconds}$$

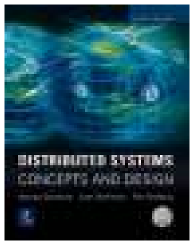
or about 10^{25} years!

7.10 In the Needham and Shroeder authentication protocol with secret keys, explain why the following version of message 5 is not secure:

$$A \rightarrow B: \quad \{N_B\}_{K_{AB}}$$

7.10 Ans.

The purpose of message 5 is for A to convince B that K_{AB} is fresh. B will be convinced if it knows that A has K_{AB} (because it will know that A cannot be merely re-playing overheard messages). The suggested version of message 5 is not secure because A would not need to know K_{AB} in order to send it, it could be sent by copying message 4.



Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001

Chapter 8 Exercise Solutions

8.1 Why is there no open or close operation in the interface to the flat file service or the directory service. What are the differences between our directory service Lookup operation and the UNIX open?

8.1 Ans.

Because both services are stateless. The interface to the flat file service is designed to make *open* unnecessary. The *Lookup* operation performs a single-level lookup, returning the UFID corresponding to a given simple name in a specified directory. To look up a pathname, a sequence of *Lookups* must be used. Unix *open* takes a pathname and returns a file descriptor for the named file or directory.

8.2 Outline methods by which a client module could emulate the UNIX file system interface using our model file service.

8.2 Ans.

Left to the reader.

8.3 Write a procedure PathLookup(Pathname, Dir) → UFID that implements Lookup for UNIX-like pathnames based on our model directory service.

8.3 Ans.

Left to the reader.

8.4 Why should UFIDs be unique across all possible file systems? How is uniqueness for UFIDs ensured?

8.4 Ans.

Uniqueness is important because servers that may be attached to different networks may eventually be connected, e.g. by an internetwork, or because a file group is moved from one server to another. UFIDs can be made unique by including the address of the host that creates them and a logical clock that is increased whenever a UFID is created. Note that the host address is included only for uniqueness, not for addressing purposes (although it might subsequently be used as a hint to the location of a file).

8.5 To what extent does Sun NFS deviate from one-copy file update semantics? Construct a scenario in which two user-level processes sharing a file would operate correctly in a single UNIX host but would observe inconsistencies when running in different hosts.

8.5 Ans.

After a *write* operation, the corresponding data cached in clients other than the one performing the *write* is invalid (since it does not reflect the current contents of the file on the NFS server), but the other clients will not discover the discrepancy and may use the stale data for a period of up to 3 seconds (the time for which cached blocks are assumed to be fresh). For directories, the corresponding period is 30 seconds, but the consequences are less serious because the only operations on directories are to insert and delete file names.

Scenario: any programs that depend on the use of file data for synchronization would have problems. For example, program A checks and sets two locks in a set of lock bits stored at the beginning of a file, protecting records within the file. Then program A updates the two locked records. One second later program B reads the same locks from its cache, finds them unset, sets them and updates the same records. The resulting values of the two records are undefined.

8.6 Sun NFS aims to support heterogeneous distributed systems by the provision of an operating system-independent file service. What are the key decisions that the implementer of an NFS server for an operating system other than UNIX would have to take? What constraints should an underlying filing system obey to be suitable for the implementation of NFS servers?

8.6 Ans.

The Virtual file system interface provides an operating-system independent interface to UNIX and other file systems. The implementor of an NFS server for a non-Unix operating system must decide on a representation for file handles. The last 64 bits of a file handle must uniquely define a file within a file system. The Unix representation is defined (as shown on page ??) but it is not defined for other operating systems. If the operating system does not provide a means to identify files in less than 64 bits, then the server would have to generate identifiers in response to *lookup*, *create* and *mkdir* operations and maintain a table of identifiers against file names.

Any filing system that is used to support an NFS server must provide:

- efficient block-level access to files;
- file attributes must include write timestamps to maintain consistency of client caches;
- other attributes are desirable, such owner identity and access permission bits.

8.7 What data must the NFS client module hold on behalf of each user-level process?

8.7 Ans.

A list of open files, with the corresponding v-node number. The client module also has a v-node table with one entry per open file. Each v-node holds the file handle for the remote file and the current read-write pointer.

8.8 Outline client module implementations for the UNIX open() and read() system calls, using the NFS RPC calls of Figure 8.3, (i) without, and (ii) with a client cache.

8.8 Ans.

Left to the reader.

8.9 Explain why the RPC interface to early implementations of NFS is potentially insecure. The security loophole has been closed in NFS 3 by the use of encryption. How is the encryption key kept secret? Is the security of the key adequate?

8.9 Ans.

The user id for the client process was passed in the RPCs to the server in unencrypted form. Any program could simulate the NFS client module and transmit RPC calls to an NFS server with the user id of any user, thus gaining unauthorized access to their files. DES encryption is used in NFS version 3. The encryption key is established at *mount* time. The mount protocol is therefore a potential target for a security attack. Any workstation could simulate the mount protocol, and once a target filesystem has been mounted, could impersonate any user using the encryption agreed at mount time..

8.10 After the timeout of an RPC call to access a file on a hard-mounted file system the NFS client module does not return control to the user-level process that originated the call. Why?

8.10 Ans.

Many Unix programs (tools and applications) are not designed to detect and recover form error conditions returned by file operations. It was considered preferable to avoid error conditions wherever possible, even at the cost of suspending programs indefinitely.

8.11 How does the NFS Automounter help to improve the performance and scalability of NFS?

8.11 Ans.

The NFS *mount service* operates at system boot time or user login time at each workstation, mounting filesystems wholesale in case they will be used during the login session. This was found too cumbersome for some applications and produces large numbers of unused entries in mount tables. With the Automounter filesystems need not be mounted until they are accessed. This reduces the size of mount tables (and hence the time to search them). A simple form of filesystem replication for read-only filesystems can also be achieved with the Automounter, enabling the load of accesses to frequently-used system files to be shared between several NFS servers.

8.12 How many lookup calls are needed to resolve a 5-part pathname (for example, /usr/users/jim/code/xyz.c) for a file that is stored on an NFS server? What is the reason for performing the translation step-by-step?

8.12 Ans.

Five *lookups*, one for each part of the name.

Here are several reasons why pathnames are looked up one part at a time, only the first of those listed is mentioned in the book (on p. 229):

i) pathnames may cross mount points;

ii) pathnames may contain symbolic links;

iii) pathnames may contain '..';

iv) the syntax of pathnames could be client-specific.

Case (i) requires reference to the client mount tables, and a change in the server to which subsequent *lookups* are dispatched. Case (ii) cannot be determined by the client. Case (iii) requires a check in the client against the 'pseudo-root' of the client process making the request to make sure that the path doesn't go above the pseudo-root. Case (iv) requires parsing of the name at the client (not in itself a bar to multi-part lookups, since the parsed name could be passed to the server, but the NFSD protocol doesn't provide for that).

8.13 What condition must be fulfilled by the configuration of the mount tables at the client computers for access transparency to be achieved in an NFS-based filing system.

8.13 Ans.

All clients must mount the same filesystems, and the names used for them must be the same at all clients. This will ensure that the file system looks the same from all client machines.

8.14 How does AFS gain control when an open or close system call referring to a file in the shared file space is issued by a client?

8.14 Ans.

The UNIX kernel is modified to intercept local *open* and *close* calls that refer to non-local files and pass them to the local Venus process.

8.15 Compare the update semantics of UNIX when accessing local files with those of NFS and AFS. Under what circumstances might clients become aware of the differences?

8.15 Ans.

UNIX:strict one-copy update semantics;

NFS:approximation to one-copy update semantics with a delay (~3 seconds) in achieving consistency;

AFS:consistency is achieved only on *close*. Thus concurrent updates at different clients will result in lost updates – the last client to close the file wins.

See the solution to Exercise 8.1 for a scenario in which the difference between NFS and UNIX update semantics would matter. The difference between AFS and UNIX is much more visible. Lost updates will occur whenever two processes at different clients have a file open for writing concurrently.

8.16 How does AFS deal with the risk that callback messages may be lost?

8.16 Ans.

Callbacks are renewed by Venus before an *open* if a time T has elapsed since the file was cached, without communication from the server. T is of the order of a few minutes.

8.17 Which features of the AFS design make it more scalable than NFS? What are the limits on its scalability, assuming that servers can be added as required? Which recent developments offer greater scalbility?

8.17 Ans.

The load of RPC calls on AFS servers is much less than NFS servers for the same client workload. This is achieved by the elimination of all remote calls except those associated with *open* and *close* operations, and the use of the *callback* mechanism to maintain the consistency of client caches (compared to the use of *getattributes* calls by the clients in NFS). The scalability of AFS is limited by the performance of the single server that holds the most-frequently accessed file volume (e.g. the volume containing */etc/passwd*, */etc/hosts*, or some similar system file). Since read-write files cannot be replicated in AFS, there is no way to distribute the load of access to frequently-used files.

Designs such as xFS and Frangipani offer greater scalability by separating the management and metadata operations from the data handling, and they reduce network traffic by locating files based on usage patterns.

Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001.

Chapter 9 Exercise Solutions

9.1 Describe the names (including identifiers) and attributes used in a distributed file service such as NFS (see Chapter 8).

9.1 Ans.

Names: hierarchical, textual file names; local identifiers of open files (file descriptors); file handles; file system identifiers; inode numbers; textual hostnames; IP addresses; port numbers; physical network addresses; user and group identifiers; disk block numbers; physical disk addresses.

Attributes: file metadata, including physical storage information and user-visible file attributes.

See Section 8.2 for more information about NFS.

9.2 Discuss the problems raised by the use of aliases in a name service, and indicate how, if at all, these may be overcome.

9.2 Ans.

Firstly, if aliases are private and not publicly defined, then there is a risk of misunderstanding through a user referring to an object using an alias. The alias might refer to an (incorrect) object in the other user's name space.

The second problem with aliases is that they may introduce cycles into the naming graph. For example, a name */users/fred* can in principle be made an alias for */users*. A resolver will potentially cycle infinitely in attempting to resolve this name. A solution is to place a limit on the number of aliases that a resolver is prepared to encounter when it resolves a name.

9.3 Explain why iterative navigation is necessary in a name service in which different name spaces are partially integrated, such as the file naming scheme provided by NFS.

9.3 Ans.

The reason why iterative navigation is necessary is that when a client encounters a symbolic link, then this symbolic link should be resolved with respect to the client's name space, even when the server stores the link. For example, suppose that the server's directory */jewel* is mounted on the client's directory */ruby/red*. Suppose that */ruby/red/stone* (stored in the server's name space as */jewel/stone*) is a symbolic link to */ruby/stone*. The server passes back this link to the client, which then must continue to resolve it. The pathname */ruby/stone* might refer to a file stored at the client, or it might be a mount point to another server.

9.4 Describe the problem of unbound names in multicast navigation. What is implied by the installation of a server for responding to lookups of unbound names?

9.4 Ans.

In multicast navigation, a client multicasts a name to a group of servers for resolution. If a server can resolve the name, it replies to the client. To minimise messages, a server that cannot resolve the name does not respond. However if no server can resolve the name – the name is unbound – then the client will be greeted with silence. It must re-send the request, in case it was dropped. The client cannot distinguish this case from that of the failure of a server that can resolve the name.

A solution to this problem is to install a member of the group which keeps track of all bound names, but does not need to store the corresponding attributes. When a request is multicast to the group, this server looks for the name in its list of bound names. If the name appears in the list, it does nothing. If, however, the name is not in its list, then it sends a 'name unbound' response message to the client. The implication is that this special server must be notified whenever a client binds or unbinds a name, increasing the overheads for these operations.

9.5 How does caching help a name service's availability?

9.5 Ans.

Clients cache both object attributes and the addresses of servers that store directories. This helps the service's availability because the client may still access cached attributes even if the server that stores them crashes (although the attributes may have become stale). And if, for example, the server that stores the director */emerald* has crashed, a client can still look up the object */emerald/green/stone* if it has cached the location of the directory */emerald/green*.

9.6 Discuss the absence of a syntactic distinction (such as use of a final '.') between absolute and relative names in DNS.

9.6 Ans.

DNS servers only accept complete domain names without a final '.', such as *dcs.qmw.ac.uk*. Such names are referred to the DNS root, and in that sense are absolute. However, resolvers are configured with a list of domain names which they append to client-supplied names, called a domain suffix list. For example, when supplied with a name *fred* in the department of Computer Science at Queen Mary and Westfield College, a resolver appends*.dcs.qmw.ac.uk* to get *fred.dcs.qmw.ac.uk*, which it then submits to a server. If this should be unbound, the resolver tries *fred.qmw.ac.uk*. Eventually, if necessary, the resolver will submit the name *fred* to a server. Some resolvers accept a final after a domain name. This signifies *to the resolver* that the name is to be sent directly to the server as it is (but stripped of its final '.'); the final '.' is not acceptable domain name syntax.

In practice the lack of syntactic distinction between relative names (*fred*) and absolute names (*fred.dcs.qmw.ac.uk*) is not a problem because of the conventions governing first-level domain names. No-one uses single-component names referred to the root (such as *gov*, *edu*, *uk*), so a single-component name is always relative to some subdomain. In principle, a multi-component name such as *ac.uk* uttered in the domain *elvis.edu* could refer to a (bound) domain *ac.uk.elvis.edu*, but normally organisations neither need to nor want to install such confusing names in their subdomains.

An advantage to the lack of syntactic distinction between absolute and relative names is that the DNS name space could, in principle, be reconfigured. We could, for example, transform *edu*, *gov*, *com* etc. into *edu.us*, *gov.us*, *com.us* etc. and still correctly resolve names such as *purdue.edu* in the USA by configuring all resolvers in the USA to include*.us* in their domain suffix list.

9.7 Investigate your local configuration of DNS domains and servers. You may find a program such as *nslookup* installed on UNIX systems, which enables you to carry out individual name server queries.

9.7 Ans.

Left to the reader.

9.8 Why do DNS root servers hold entries for two-level names such as ac.uk and purdue.edu, rather than one-level names such as uk, edu and com?

9.8 Ans.

First-level domain names such as *edu* and *com* refer to abstract categories of organizations and administrative units, and do not refer to any actual body. Second-level domain names such as *yhaoo.com* and *purdue.edu* are not so many in number as to need to be divided between separate *com* and *edu* servers. Such a division would bring extra complexity and overheads. Although uk etc. do have separate servers.

9.9 Which other name server addresses do DNS name servers hold by default, and why?

9.9 Ans.

A DNS name server holds the addresses of one or more root servers, so that all parts of the name space can be reached. It stores the addresses of servers storing subdomains (thus a server for *qmw.ac.uk* stores addresses of servers for *dcs.qmw.ac.uk*). Thirdly, it is often convenient for it to store the addresses of servers storing its parent domain (thus a server in *dcs.qmw.ac.uk* knows the addresses of servers storing *qmw.ac.uk*).

9.10 Why might a DNS client choose recursive navigation rather than iterative navigation? What is the relevance of the recursive navigation option to concurrency within a name server?

9.10 Ans.

A DNS client may choose recursive navigation simply because it is too basic to perform iterative navigation. A server that performs recursive navigation must await a reply from another server before replying to the client. It is preferable for a server to deal with several outstanding client requests at one time rather than holding off other requests until each one is completed, so that clients are not unduly held up. The server will, in general, refer resolution to several other servers rather than just one, so client requests will be satisfied in parallel to some extent.

9.11 When might a DNS server provide multiple answers to a single name lookup, and why?

9.11 Ans.

A DNS server provides several answers to a single name lookup whenever it possesses them, assuming that the client has requested multiple answers. For example, the server might know the addresses of several mail servers or DNS servers for a given domain. Handing back all these addresses increase the availability of the mail service and DNS respectively.

9.12 The Jini lookup service matches service offers to client requests based on attributes or on Java typing. Explain with examples the difference between these two methods of matching. What is the advantage of allowing both sorts of matching?

9.12 Ans.

Attributes - attributes describe properties of a service, for example a printer might specify its speed, resolution, whether it prints on one of two sides of the paper, its location in the building.

Types - specify the Java data types implemented by the service. e.g. Printer, ColourPrinter.

Both sorts of matching. e.g. if match only by type (e.g. for ColourPrinter), there may be several such services and the client can specify its selection by means of attributes e.g. to get the nearest one. If match only by attributes, the type may not be exactly correct.

9.13 Explain how the Jini lookup service uses leases to ensure that the list of services registered with a lookup server remains current although services may crash or become inaccessible.

9.13 Ans.

Assume a lease expiry time of *t* minutes. The list of services in a lookup service is fresh within t minutes because any server that is still alive will renew its lease every t minutes. If a server crashes, or becomes disconnected from a lookup service, the latter will delete its entry as soon as the lease expires. Therefore an unavailable server is registered only for a maximum of t minutes after it becomes unavailable.

The servers know of the existence (or reappearance) of local lookup services because they send “I am here” messages from time to time.

9.14 Describe the use of IP multicast and group names in the Jini ‘discovery’ service which allows clients and servers to locate lookup servers.

9.14 Ans.

Each lookup service has a set of groups associated with it. A client or server needing to locate a lookup server specifies the names of a set of groups (they are interested in) in a multicast request. Lookup servers belonging to those groups reply by sending a proxy.

Lookup servers also advertise the groups they are associated with when they send their "I am here" messages.

9.15 GNS does not guarantee that all copies of entries in the naming database are up-to-date. How are clients of GNS likely to become aware that they have been given an out-of-date entry? Under what circumstances might it be harmful?

9.15 Ans.

Clients will become aware of the use of an out-of-date entry if a name that they have obtain is no longer a valid communication identifier (such as for example when a user's email address has changed and no forwarding address exists). This is not normally harmful, since the client can recover gracefully by making a delayed request to GNS. However, it may be harmful if the communication identifier obtained from GNS provides access to some protected resource, and the name used to obtain it should no longer be bound to that resource. For example, when a user ceases to be a member of the organisation, GNS may continue to supply information about his or her organisational role, leading users or applications to accord privileges to the user.

9.16 Discuss the potential advantages and drawbacks in the use of an X.500 directory service in place of DNS and the Internet mail delivery programs. Sketch the design of a mail delivery system for an internetwork in which all mail users and mail hosts are registered in an X.500 database.

9.16 Ans.

For access to conventional email addresses (based on Internet Domain Names), X.500 would provide a similar facilities to the DNS service. X.500 is designed to be scalable. If this is achieved in practice, then it should meet the future needs of large-scale networking better than DNS.

The main advantage of X.500 is that it is an attribute-based directory service. In principle, users could address messages to people by quoting their real names and their organisational affiliations, instead of the Domain Name based addresses currently used. The mail system would make a *search* request of X.500 to find the corresponding DNS or other network address of the user's mailbox. A drawback is that searching with a wide scope is quite slow and costly in computing resources, the scope could be limited by the use of the organisational affiliation. Several alternate mailboxes could be held in the directory server, providing fault-tolerant mail delivery.

9.17 What security issues are liable to be relevant to a directory service such as X500 operating within an organization such as a university?

9.17 Ans.

There are two main security issues of relevance to a name service. The first is the question of who may create and modify an entry for a given name. It is important that malicious users cannot create bogus entries or alter stored attributes without permission. It is equally important that administrative boundaries in the name space are respected.

The second issue is privacy. In general, users may want only privileged principals to read their attributes.

Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001

Chapter 10 Exercise Solutions

10.1 Why is computer clock synchronization necessary? Describe the design requirements for a system to synchronize the clocks in a distributed system.

10.1 Ans.

See Section 10.1 for the necessity for clock synchronization.

Major design requirements:

i) there should be a limit on deviation between clocks or between any clock and UTC;

ii) clocks should only ever advance;

iii) only authorized principals may reset clocks (See Section 7.6.2 on Kerberos)

In practice (i) cannot be achieved unless only benign failures are assumed to occur and the system is synchronous.

10.2 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 has elapsed.

10.2 Ans.

Some applications use the current clock value to stamp events, on the assumption that clocks always advance.

We use E to refer to the 'errant' clock that reads 10:27:54.0 when the real time is 10:27:50. We assume that H advances at a perfect rate, to a first approximation, over the next 8 seconds. We adjust our software clock S to tick at rate chosen so that it will be correct after 8 seconds, as follows:

$S = c(E - Tskew) + Tskew$, where $Tskew$ = 10:27:54 and c is to be found.

But $S = Tskew+4$ (the correct time) when $E = Tskew+8$, so:

$Tskew+ 4 = c(Tskew + 8 - Tskew) + Tskew$, and c is 0.5. Finally:

$S = 0.5(E - Tskew) + Tskew$ (when $Tskew \leq E \leq Tskew+8$).

10.3 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.

(i) 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' ?

(ii) When may a receiver remove a timestamp 175,000 (ms) from its table? (Hint: use the receiver's local clock value.)

(iii) Should the clocks be internally synchronized or externally synchronized?

10.3 Ans.

i) If $T \leq T'$ then the message must be a repeat.

ii) The earliest message timestamp that could still arrive when the receiver's clock is r is r - 100 - 50. If this is to be at least 175,000 (so that we cannot mistakenly receive a duplicate), we need r -150 = 175,000, i.e. r = 175,150.

iii) Internal synchronisation will suffice, since only time *differences* are relevant.

10.4 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*

10.4 Ans.

The client should choose the minimum round-trip time of 20 ms = 0.02 s. It then estimates the current time to be 10:54:28.342 + 0.02/2 = 10:54:28.352. The accuracy is ± 10 ms.

If the minimum message transfer time is known to be 8 ms, then the setting remains the same but the accuracy improves to ± 2 ms.

10.5 In the system of Exercise 10.4 it is required to synchronize a file server's clock to within ±1 millisecond. Discuss this in relation to Cristian's algorithm.

10.5 Ans.

To synchronize a clock within ± 1 ms it is necessary to obtain a round-trip time of no more than 18 ms, given the minimum message transmission time of 8 ms. In principle it is of course possible to obtain such a round-trip time, but it may be improbable that such a time could be found. The file server risks failing to synchronize over a long period, when it could synchronize with a lower accuracy.

10.6 What reconfigurations would you expect to occur in the NTP synchronization subnet?

10.6 Ans.

A server may fail or become unreachable. Servers that synchronize with it will then attempt to synchronize to a different server. As a result, they may move to a different stratum. For example, a stratum 2 peer (server) loses its connection to a stratum 1 peer, and must thenceforth use a stratum 2 peer that has retained its connection to a stratum 1 peer. It becomes a stratum 3 peer.

Also, if a primary server's UTC source fails, then it becomes a secondary server.

10.7 An NTP server B receives server A's message at 16:34:23.480 bearing a timestamp 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.

10.7 Ans.

Let $a = T_{i-2} - T_{i-3}$ = 23.48 - 13.43 = 10.05; $b = T_{i-1} - T_i$ = 25.7- 15.725 =9.975.

Then the estimated offset o_i = $(a+b)/2$ = 10.013s, with estimated accuracy = $\pm d_i/2$ = $\pm (a-b)/2$ = 0.038s (answers expressed to the nearest millisecond).

10.8 Discuss the factors to be taken into account when deciding to which NTP server a client should synchronize its clock.

10.8 Ans.

The main factors to take into account are the intrinsic reliability of the server as a source of time values, and the quality of the time information as it arrives at the destination. Sanity checks are needed, in case servers have bugs or are operated maliciously and emit spurious time values. Assuming that servers emit the best time values known to them, servers with lower stratum numbers are closest to UTC, and therefore liable to be the most accurate. On the other hand, a large network distance from a source can introduce large variations in network delays. The choice involves a trade-off between these two factors, and servers may synchronize with several other servers (peers) to seek the highest quality data.

10.9 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.

10.9 Ans.

If we know that the drift rate is constant, then we need only measure it between synchronization points with an accurate source and compensate for it. For example, if the clock loses a second every hour, then we can add a second every hour, in smooth increments, to the value returned to the user. The difficulty is that the clock's drift rate is liable to be variable – for example, it may be a function of temperature. Therefore we need an adaptive adjustment method, which guesses the drift rate, based on past behaviour, but which compensates when the drift rate is discovered to have changed by the next synchronisation point.

10.10 By considering a chain of zero or more messages connecting events e and e' and using induction, show that $e \rightarrow e' \Rightarrow L(e) < L(e')$.

10.10 Ans.

If e and e' are successive events occurring at the same process, or if there is a message m such that $e = send(m)$ and $e' = rcv(m)$, then the result is immediate from LC1 and LC2. Assume that the result to be proved is true for all pairs of events connected in a sequence of events (in which either *HB1* or *HB2* applies between each neighbouring pair) of length N or less ($N \geq 2$). Now assume that e and e' are connected in a series of events $e_1, e_2, e_3, .., e_{N+1}$ occurring at one or more processes such that $e = e_1$ and $e' = e_{N+1}$. Then $e \rightarrow e_N$ and so C(e) < C(e_N) by the induction hypothesis. But by LC1 and LC2, C(e_N) <C(e'). Therefore C(e) < C(e').

10.11 Show that $V_j[i] \leq V_i[i]$ (NB erratum in this exercise in first printing)

10.11 Ans.

Rule VC2 (p. 399) tells us that p_i is the 'source' of increments to $V_i[i]$, which it makes just before it sends each message; and that p_j increments $V_j[i]$ only as it receives messages containing timestamps with larger entries for p_i. The relationship $V_j[i] \leq V_i[i]$ follows immediately.

10.12 In a similar fashion to Exercise 10.10, show that $e \rightarrow e' \Rightarrow V(e) < V(e')$.

10.12 Ans.

If e and e' are successive events occurring at the same process, or if there is a message m such that $e = send(m)$ and $e' = rcv(m)$, then the result follows from VC2–VC4. In the latter case the sender includes its timestamp value and the recipient increases its own vector clock entry; all of its other entries remain at least as great as those in the sender's timestamp.

Assume that the result to be proved is true for all pairs of events connected in a sequence of events (in which either *HB1* or *HB2* applies between each neighbouring pair) of length N or less ($N \geq 2$). Now assume that e and e' are connected in a series of events $e_1, e_2, e_3, .., e_{N+1}$ occurring at one or more processes such that $e = e_1$ and $e' = e_{N+1}$. Then $e \rightarrow e_N$ and so V(e) < V(e_N) by the induction hypothesis. But by VC2–VC4, V(e_N) < V(e'). Therefore V(e) < V(e').

10.13 Using the result of Exercise 10.11, 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 $e \rightarrow e'$.

10.13 Ans.

Let e and e' be concurrent and let e occur at p_i and e' at p_j. Because the events are concurrent (not related by happened-before) we know that no message sent from p_i at or after event e has propagated its timestamp to p_j by the time e' occurs at p_j, and *vice versa*. By the reasoning for Exercise 10.11, it follows that $V_j[i] < V_i[i]$ and $V_i[j] < V_j[j]$ (strict inequalities) and therefore that neither $V(e) \leq V(e')$ nor $V(e') \leq V(e)$.

Therefore if $V(e) < V(e')$ the two events are not concurrent – they must be related by happened-before. Of the two possibilities, it obviously must be that $e \rightarrow e'$.

10.14 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.

10.14 Ans.

P sends msg m

P records state (101)

P sends marker (see initiation of algorithm described on p. 406)

Q receives m, making its state 102

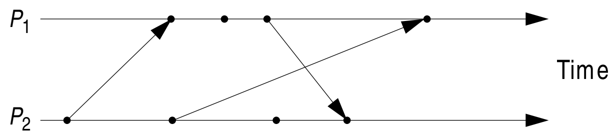
Q receives the marker and by marker-receiving rule, records its state (102) and the state of the channel from P to Q as {}

Q sends marker (marker-sending rule)

(Q sends m again at some point later)

P receives marker

P records the state of the channel from Q to P as set of messages received since it saved its state = {} (marker-receiving rule).

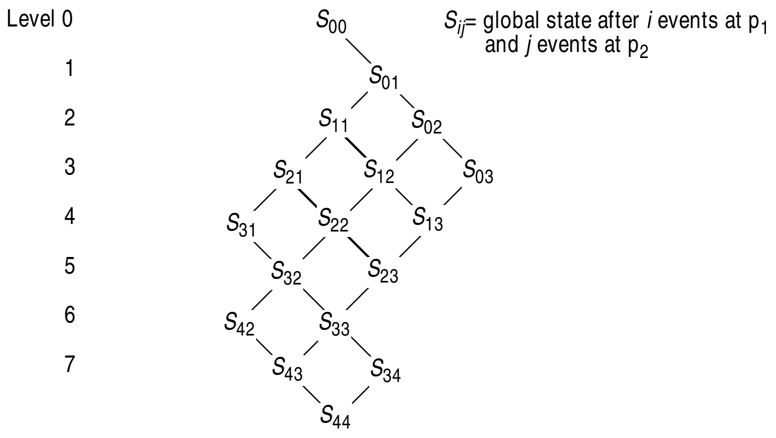


10.15 The figure above shows events occurring for each of two processes, p_1 and p_2. Arrows between processes denote message transmission.

Draw and label the lattice of consistent states (p_1 state, p_2 state), beginning with the initial state (0,0).

10.15 Ans.

The lattice of global states for the execution of Figure of exercise 10.15



10.16 Jones is running a collection of processes $p_1, p_2, \ldots, p_N$. Each process p_i contains a variable v_i. She wishes to determine whether all the variables $v_1, v_2, \ldots, v_N$ were ever equal in the course of the execution.

(i) 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?

(ii) Explain the statement *possibly* ($v_1 = v_2 = \ldots = v_N$). How can Jones determine whether this statement is true of her execution?

10.16 Ans.

(i) communicate new value when local variable v_i changes;

with this value send: current time of day *C(e)* and vector timestamp *V(e)* of the event of the change, *e*.

(ii) *possibly* (...): there is a consistent, potentially simultaneous global state in which the given predicate is true.

Monitor process takes potentially simultaneous events which correspond to a consistent state, and checks predicate $v_1 = v_2 = \ldots = v_N$.

Simultaneous: estimate simultaneity using bound on clock synchronization and upper limit on message propagation time, comparing values of *C* (see p. 415).

Consistent state: check vector timestamps of all pairs of potentially simultaneous events e_i, e_j: check $V(e_i)[i] \geq V(e_j)[i]$.

Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001

Chapter 11 Exercise Solutions

11.1 Is it possible to implement either a reliable or an unreliable (process) failure detector using an unreliable communication channel?

11.1 Ans.

An unreliable failure detector can be constructed from an unreliable channel – all that changes from use of a reliable channel is that dropped messages may increase the number of false suspicions of process failure.

A reliable failure detector requires a synchronous system. It cannot be built on an unreliable channel since a dropped message and a failed process cannot be distinguished – unless the unreliability of the channel can be masked while providing a guaranteed upper bound on message delivery times. A channel that dropped messages with some probability but, say, guaranteed that at least one message in a hundred was not dropped could, in principle, be used to create a reliable failure detector.

11.2 If all client processes are single-threaded, is mutual exclusion condition ME3, which specifies entry in happened-before order, relevant?

11.2 Ans.

ME3 is not relevant if the interface to requesting mutual exclusion is synchronous. For a single-threaded process could not send a message to another process while awaiting entry, and ME3 does not arise.

11.3 Give a formula for the maximum throughput of a mutual exclusion system in terms of the synchronization delay.

11.3 Ans.

If s = synchronization delay and m = minimum time spent in a critical section by any process, then the maximum throughput is $1/(s+m)$ critical-section-entries per second.

11.4 In the central server algorithm for mutual exclusion, describe a situation in which two requests are not processed in happened-before order.

11.4 Ans.

Process A sends a request r_A for entry then sends a message m to B. On receipt of m, B sends request r_B for entry. To satisfy happened-before order, r_A should be granted before r_B. However, due to the vagaries of message propagation delay, r_B arrives at the server before r_A, and they are serviced in the opposite order.

11.5 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?

11.5 Ans.

The server uses the reliable failure detector to determine whether any client has crashed. If the client has been granted the token then the server acts as if the client had returned the token. In case it subsequently receives the token from the client (which may have sent it before crashing), it ignores it.

The resultant system is not fault-tolerant. If a token-holding client crashed then the application-specific data protected by the critical section (whose consistency is at stake) may be in an unknown state at the point when another client starts to access it.

If a client that possesses the token is wrongly suspected to have failed then there is a danger that two processes will be allowed to execute in the critical section concurrently.

11.6 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.

11.6 Ans.

Consider three processes in a ring, in the order P1, P2 and P3. Note that processes may send messages to one another independently of the token-passing protocol.

```
P1 ---------------> P2
|                    |
|                    |
P3 <----------------|
```

The token is initially with P2. P1 requests the token, then sends a message to P3, which also requests the token. The message passes the token at P2. Then P2 sends on the token. P3 gets it, but the token should have been granted to P1 first.

11.7 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?

11.7 Ans.

In Ricart and Agrawala's multicast-based mutual exclusion algorithm, a client issues a multicast request every time it requires entry. This is inefficient in the case described, of a client that repeatedly enters the critical section before another needs entry.

Instead, a client that finishes with a critical section and which has received no outstanding requests could mark the token as *JUST_RELEASED*, meaning that it has not conveyed any information to other processes that it has finished with the critical section. If the client attempts to enter the critical section and finds the token to be *JUST_RELEASED*, it can change the state to *HELD* and re-enter the critical section.

To meet liveness condition ME2, a *JUST_RELEASED* token should become *RELEASED* if a request for entry is received.

11.8 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?

11.8 Ans.

First note that this is an undesirable feature if there is no advantage to using a higher-numbered process: the re-election is wasteful. However, the numbering of processes may reflect their relative advantage (for example, with higher-numbered processes executing at faster machines). In this case, the advantage may be worth the re-election costs. Re-election costs include the message rounds needed to implement the election; they also may include application-specific state transfer from the old coordinator to the new coordinator.

To avoid a re-election, a recovering process could merely send a *requestStatus* message to successive lower-numbered processes to discover whether another process is already elected, and elect itself only if it receives a negative response. Thereafter, the algorithm can operate as before: if the newly-recovered process discovers the coordinator to have failed, or if it receives an *election* message, it sends a *coordinator* message to the remaining processes.

11.9 Suggest how to adapt the Bully algorithm to deal with temporary network partitions (slow communication) and slow processes.

11.9 Ans.

With the operating assumptions stated in the question, we cannot guarantee to elect a unique process at any time. Instead, we may find it satisfactory to form subgroups of processes that agree on their coordinator, and allow several such subgroups to exist at one time. For example, if a network splits into two then we could form two subgroups, each of which elects the process with the highest identifier among its membership. However, if the partition should heal then the two groups should merge back into a single group with a single coordinator.

The algorithm known as the 'invitation algorithm' achieves this. It elects a single coordinator among each subgroup whose members can communicate, but periodically a coordinator polls other members of the entire set of processes in an attempt to merge with other groups. When another coordinator is found, a coordinator sends it an 'invitation' message to invite it to form a merged group. As in the Bully algorithm, when a process suspects the unreachability or failure of its coordinator it calls an election.

For details see Garcia-Molina [1982].

11.10 Devise a protocol for basic multicast over IP multicast.

11.10 Ans.

We can use the algorithm for reliable multicast, except that processes do not retain copies of the messages that they have delivered (and nor do they piggy back acknowledgements on the messages that they multicast). If the sender is correct, a receiver can fetch a missing message from the sender; but if the sender crashes then a message may be delivered to some members of the group but not others.

11.11 How, if at all, should the definitions of integrity, agreement and validity for reliable multicast change for the case of open groups?

11.11 Ans.

For open groups, the definitions of integrity and agreement do not change. The validity property is not appropriately defined for open groups, since senders are not necessarily members of the group.

For open groups we can define validity as follows: 'If a correct process multicasts message *m*, then some member of *group*(*m*) will eventually deliver *m*.'

11.12 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 11.10 makes the algorithm no longer satisfy uniform agreement. Does the reliable multicast algorithm based on IP multicast satisfy uniform agreement?

11.12 Ans.

Reversing the order of those lines means that a process can deliver a message and then crash before sending it to the other group members – which might, in that case, not receive the message at all. This contradicts the uniform agreement property.

The reliable multicast algorithm based on IP multicast does not satisfy uniform agreement. A recipient delivers a message as soon as it receives it; if the sender was to fail during transmission and that same message was not to have reached the other group members then the uniform agreement property would not be met.

11.13 Explain why the algorithm for reliable multicast over IP multicast does not work for open groups. Given any algorithm for closed groups, how, simply, can we derive an algorithm for open groups?

11.13 Ans.

'Does not work' is putting it a little strongly: 'requires more work for correct operation' would be better.

Amongst open groups we include, for example, a group of 'subscribers' with the sender as a publisher. It is possible for one member of the group to receive a message from the publisher just before the latter crashes, but for no other members to receive the same message. Since the group members do not themselves send messages, the other members will remain in ignorance. Our assumption that every process 'multicasts messages' indefinitely is not appropriate in the context of such a group. The fix is for processes to send regular 'heartbeat' messages to one another, telling about which messages they have received (see the next exercise).

We obtain an algorithm for open groups by having senders pick a member of the closed group and sending the message to it, which it then sends to the group.

11.14 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.

11.14 Ans.

A process can delete a retained message when it is known to have been received by all group members. The latter condition can be determined from the acknowledgements that group members piggy back onto the messages they send. (This is one of the main purposes of those acknowledgments; the other is that a process may learn sooner that it has missed a message than if it had to wait for the sender of that message to send another one.)

A group member can send periodic heartbeat messages if it has no application-level messages to send. A heartbeat message records the last sequence number sent and sequence numbers received from each sender, enabling receivers to delete message they might otherwise retain, and detect missing messages.

11.15 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.

11.15 Ans.

Let *p* send a message *m1* with group-specific sequence number 1 to group *g1* and a message *m2* with group-specific sequence number 1 to group *g2*. (The sequence numbers are independent, hence it is possible for two messages to have the same sequence number.) Now consider process *q* in the intersection of *g1* and *g2*. How is *q* to order *m1* and *m2*? It has no information to determine which should be delivered first.

The solution is for the sender *p* to include with its message the latest sequence numbers for each group that it sends to. Thus if *p* sent *m1* before *m2*, *m1* would include <*g1*, 1> and <*g2*, 0> whereas *m2* would include <*g1*, 1> and <*g2*, 1>. Process *q* is in a position to know that *m1* is to be delivered next; it would also know that it had missed a message if it received *m2* first.

11.16 Show that, if the basic multicast that we use in the algorithm of Figure 11.14 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?

11.16 Ans.

We show that causal ordering is achieved for the simplest possible cases of the happened-before relation; the general case follows trivially.

First, suppose *p* TO-multicasts a message *m1* which *q* receives; *q* then TO-multicasts message *m2*. The sequencer must order *m2* after *m1*, so every process will deliver *m1* and *m2* in that order.

Second, suppose *p* TO-multicasts a message *m1* then TO-multicasts message *m2*. Since the basic multicast is FIFO-ordered, the sequencer will receive *m1* and *m2* in that order; so every group member will receive them in that order.

It is clear that the result is generally true, as long as the implementation of total ordering guarantees that the sequence number of any message sent is greater than that of any received by the sending process. See Florin & Toinard [1992].

[Florin & Toinard 1992] Florin, G. and Toinard, C. (1992). A new way to design causally and totally ordered multicast protocols. Operating Systems Review, ACM, Oct. 1992.

11.17 Suggest how to adapt the causally ordered multicast protocol to handle overlapping groups.

11.17 Ans.

A process maintains a different vector timestamp *Vg* for each group *g* to which it belongs and attaches all of its vector timestamps when it sends a message.

When a process *p* receives a message destined for group *g* from member *i* of that group, it checks, as in the single-group case, that $Vg(message)[i] = Vg(p)[i] + 1$; also, all other entries in the vector timestamps contained in the message must be less than or equal to *p*'s vector timestamp entries. Process *p* keeps the message on the hold-back queue if this check fails, since it is temporarily missing some messages that happened-before this one.

11.18 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.

11.18 Ans.

The three groups are $G1 = \{p1, p2\}$; $G2 = \{p2, p3\}$; $G3 = \{p1, p3\}$.

A pairwise total ordering could operate as follows: *m1* sent to *G1* is delivered at *p2* before *m2* sent to *G2*; *m2* is delivered to *p3* before *m3* sent to *G3*. But *m3* is delivered to *p1* before *m1*. Therefore we have the cyclic delivery ordering $m1 \to m2 \to m3 \to m1 \ldots$ We would expect from a global total order that a cycle such as this cannot occur.

11.19 Construct a solution to reliable, totally ordered multicast in a synchronous system, using a reliable multicast and a solution to the consensus problem.

11.19 Ans.

To RTO-multicast (reliable, totally-ordered multicast) a message *m*, a process attaches a totally-ordered, unique identifier to *m* and R-multicasts it.

Each process records the set of message it has R-delivered and the set of messages it has RTO-delivered. Thus it knows which messages have not yet been RTO-delivered.

From time to time it proposes its set of not-yet-RTO-delivered messages as those that should be delivered next. A sequence of runs of the consensus algorithm takes place, where the *k*'th proposals (*k* = 1, 2, 3, ...) of all the processes are collected and a unique decision set of messages is the result.

When a process receives the *k*'th consensus decision, it takes the intersection of the decision value and its set of not-yet-RTO-delivered messages and delivers them in the order of their identifiers, moving them to the record of messages it has RTO-delivered.

In this way, every process delivers messages in the order of the concatenation of the sequence of consensus results. Since the consensus results given to different correct processes are identical, we have a RTO multicast.

11.20 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, then this would contradict the impossibility result of Fischer *et al.*!) Hint: consider slow/failed processes.

11.20 Ans.

If we used a reliable but not totally ordered multicast, the majority function can only be used meaningfully if it is applied to the same set of values. But, in an asynchronous system, we cannot know how long to wait for the set of all values – the source of a missing message might be slow or it might have crashed. Waiting for the first message delivered by a reliable totally ordered multicast finesses that problem.

11.21 Show that byzantine agreement can be reached for three generals, with one of them faulty, if the generals digitally sign their messages.

11.21 Ans.

Any lieutenant can verify the signature on any message. No lieutenant can forge another signature. The correct lieutenants sign what they each received and send it to one another.

A correct lieutenant decides *x* if it receives messages [*x*](signed commander) and either [[*x*](signed commander)](signed lieutenant) or a message that either has a spoiled lieutenant signature or a spoiled commander signature.

Otherwise, it decides on a default course of action (retreat, say).

A correct lieutenant either sees the proper commander's signature on two different courses of action (in which case both correct lieutenants decide 'retreat'); or, it sees one good signature direct from the commander and one improper commander signature (in which case it decides on whatever the commander signed to do); or it sees no good commander signature (in which case both correct lieutenants decide 'retreat').

In the middle case, either the commander sent an improperly signed statement to the other lieutenant, or the other lieutenant is faulty and is pretending that it received an improper signature. In the former case, both correct lieutenants will do whatever the (albeit faulty) commander told one of them to do in a signed message. In the latter case, the correct lieutenant does what the correct commander told it to do.

Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001.

Chapter 12 Exercise Solutions

12.1 The TaskBag is a service whose functionality is to provide a repository for 'task descriptions'. It enables clients running in several computers to carry out parts of a computation in parallel. A *master* process places descriptions of sub-tasks of a computation in the TaskBag, and *worker* processes select tasks from the TaskBag and carry them out, returning descriptions of results to the TaskBag. The *master* then collects the results and combines them to produce the final result.

The TaskBag service provides the following operations:

setTask	allows clients to add task descriptions to the bag;
takeTask	allows clients to take task descriptions out of the bag.

A client makes the request *takeTask*, when a task is not available but may be available soon. Discuss the advantages and drawbacks of the following alternatives:

(i) the server can reply immediately, telling the client to try again later;

(ii) make the server operation (and therefore the client) wait until a task becomes available.

(iii) use callbacks.

12.1 Ans.

This is a straight-forward application of the ideas on synchronising server operations. One of the projects in the Project Work section is based on the *TaskBag service*.

12.2 A server manages the objects a_1, a_2,... a_n. The server provides two operations for its clients:

read (i) returns the value of a_i;

write(i, Value) assigns *Value* to a_i.

The transactions *T* and *U* are defined as follows:

T: x= read (j); y = read (i); write(j, 44); write(i, 33);
U: x= read(k); write(i, 55); y = read (j); write(k, 66).

Give three serially equivalent interleavings of the transactions *T* and *U*.

12.2 Ans.

The interleavings of *T* and *U* are serially equivalent if they produce the same outputs (in x and y) and have the same effect on the objects as some serial execution of the two transactions. The two possible serial executions and their effects are:

If *T* runs before *U*: $x_T = a_j^0$; $y_T = a_i^0$; $x_U = a_k^0$; $y_U = 44$; $a_i = 55$; $a_j = 44$; $a_k = 66$.

If *U* runs before *T*: $x_T = a_j^0$; $y_T = 55$; $x_U = a_k^0$; $y_U = a_j^0$; $a_i = 33$; $a_j = 44$; $a_k = 66$,

where x_T and y_T are the values of x and y in transaction *T*; x_U and y_U are the values of x and y in transaction *U* and a_i^0, a_j^0 and a_k^0, are the initial values of a_i, a_j and a_k

We show two examples of serially equivalent interleavings:

A

T	*U*
x:=read(j)	
	x:= read(k)
	write(i, 55)
y:=read (i)	
	y:=read (j)
	write(k, 66)
write(j, 44)	
write(i, 33)	

B

T	*U*
x:=read(j)	
y:=read (i)	
	x:= read(k)
write(j, 44)	
write(i, 33)	
	write(i, 55)
	y:=read (j)
	write(k, 66)

A is equivalent to *U* before *T*. y_T gets the value of 55 written by *U* and at the end a_i =33; a_j =44; a_k = 66.

B is equivalent to *T* before *U*. y_U gets the value of 44 written by *T* and at the end a_i =55; a_j =44; a_k = 66.

12.3 Give serially equivalent interleaving of *T* and *U* in Exercise 12.2 with the following properties: (i) that is strict; (ii) that is not strict but could not produce cascading aborts; (iii) that could produce cascading aborts.

12.3 Ans.

i) For strict executions, the *reads* and *writes* of a transaction are delayed until all transactions that have previously written the same objects are either committed or aborted. We therefore indicate the commit of the earlier transaction in our solution (a variation of B in the Answer to 12.6):

B

T	*U*
x:=read(j)	
y:=read (i)	
	x:= read(k)
write(j, 44)	
write(i, 33)	
Commit	
	write(i, 55)
	y:=read (j)
	write(k, 66)

Note that *U*'s *write(i, 55)* and *read(j)* are delayed until after *T*'s commit, because *T writes* a_i and a_j.

ii) For serially equivalent executions that are not strict but cannot produce cascading aborts, there must be no dirty reads, which requires that the *reads* of a transaction are delayed until all transactions that have previously written the same objects are either committed or aborted (we can allow *writes* to overlap). Our answer is based on B in Exercise 12.2.

B

T	*U*
x:=read(j)	
y:=read (i)	
	x:= read(k)
write(j, 44)	
write(i, 33)	
	write(i, 55)
Commit	
	y:=read (j)
	write(k, 66)

Note that *U*'s *write(i, 55)* is allowed to overlap with *T*, whereas *U*'s *read (j)* is delayed until after *T* commits.

iii)For serially equivalent executions that can produce cascading aborts, that is, dirty reads are allowed. Taking A from Exercise 12.2 and adding a *commit* immediately after the last operation of *U*, we get:

A

T	*U*
x:=read(j)	
	x:= read(k)
	write(i, 55)
y:=read (i)	
	y:=read (j)
	write(k, 66)
	commit
write(j, 44)	
write(i, 33)	

Note that *T*'s *read (i)* is a dirty read because *U* might abort before it reaches its commit operation.

12.4 The operation *create* inserts a new bank account at a branch. The transactions *T* and *U* are defined as follows:

T: aBranch.create("Z");
U: z.deposit(10); z.deposit(20).

Assume that *Z* does not yet exist. Assume also that the *deposit* operation does nothing if the account given as argument does not exist. Consider the following interleaving of transactions *T* and *U*:

T	*U*
	z.deposit(10);
aBranch.create(Z);	
	z.deposit(20);

State the balance of *Z* after their execution in this order. Are these consistent with serially equivalent executions of *T* and *U*?

12.4 Ans.

In the example, *Z*'s balance is $20 at the end.

Serial executions of *T* and *U* are:

T then *U*: *aBranch.create("Z")*; *z.deposit(10)*;*z.deposit(20)*. *Z*'s final balance is $30.

U then *T*: *z.deposit(10)*; *z.deposit(20)*; *aBranch.create("Z")*. *Z*'s final balance is $0.

Therefore the example is not a serially equivalent execution of *T* and *U*.

12.5 A newly created data item like *Z* in Exercise 12.4 is sometimes called a *phantom*. From the point of view of transaction *U*, *Z* is not there at first and then appears (like a ghost). Explain with an example, how a phantom could occur when an account is deleted.

12.5 Ans.

The point in Exercise 12.4 is that the insertion of a data item like *Z* should not be interleaved with operations on the same data item by another transaction. Suppose that we define transaction *V* as: *aBranch.delete("Z")* and consider the following interleavings of *V* and *U*:

V	*U*
	z.deposit(10);
aBranch.delete("Z")	
	z.deposit(20);

Then we have a phantom because Z is there at first and then disappears. It can be shown (as in Exercise 12.10) that these interleavings are not serially equivalent.

12.6 The 'Transfer' transactions T and U are defined as:

T: a.withdraw(4); b.deposit(4);

U: c.withdraw(3); b.deposit(3);

Suppose that they are structured as pairs of nested transactions:

T_1: a.withdraw(4); T_2: b.deposit(4);

U_1: c.withdraw(3); U_2: b.deposit(3);

Compare the number of serially equivalent interleavings of T_1, T_2, U_1 and U_2 with the number of serially equivalent interleavings of T and U. Explain why the use of these nested transactions generally permits a larger number of serially equivalent interleavings than non-nested ones.

12.6 Ans.

Considering the non-nested case, a serial execution with T before U is:

	T: a.withdraw(4)*;* b.deposit(4)*);*		*U*: c.withdraw(3); b.deposit(3);
T_1	a.withdraw(4);		
T_2	b.deposit(4)		
		U_1	c.withdraw(3)
		U_2	b.deposit(3)

We can derive some serially equivalent interleavings of the operations, in which T's write on B must be before U's read. Let us consider all the ways that we can place the operations of U between those of T.

All the interleavings must contain T_2; U_2. We consider the number of permutations of U_1 with the operations T_1-T_2 that preserve the order of T and U. This gives us (3!)/(2!*1!) = 3 serially equivalent interleavings.

We can get another 3 interleavings that are serially equivalent to an execution of U before T. They are different because they all contain U_2; T_2. The total is 6.

Now consider the nested transactions. The 4 transactions may be executed in any (serial) order, giving us 4! = 24 orders of execution.

Nested transactions allow more serially equivalent interleavings because: i) there is a larger number of serial executions, ii) there is more potential for overlap between transactions (iii) the scope of the effect of conflicting operations can be narrowed.

12.7 Consider the recovery aspects of the nested transactions defined in Exercise 12.6. Assume that a *withdraw* transaction will abort if the account will be overdrawn and that in this case the parent transaction will also abort. Describe serially equivalent interleavings of T_1, T_2, U_1 and U_2 with the following properties: (i) that is strict; (ii) that is not strict. To what extent does the criterion of strictness reduce the potential concurrency gain of nested transactions?

12.7 Ans.

If a child transaction's abort can cause the parent to abort, with the effect that the other children abort, then strict executions must delay *reads* and *writes* until all the relations (siblings and ancestors) of transactions that have previously written the same objects are either committed or aborted. Our deposit and withdraw operations read and then write the balances.

i) For strict executions serially equivalent to T_1; T_2; U_1; U_2 we note that T_2 has written B. We then delay U_2's *deposit* until after the commit of T_2 and its sibling T_1. The following is an example of such an interleaving:

T_1: a.withdraw(4)	T_2: b.deposit(4);	U_1: c.withdraw(3)	U_2: b.deposit(3)
a.withdraw(3)			
	b.deposit(4)		
		c.withdraw(3)	
	commit		
commit			
			b.deposit(3)

ii) Exercise 12.6 discusses all possible serially equivalent executions. They are non-strict if they do not obey the constraints discussed in part (i).

The criterion of strictness does not in any way reduce the possible concurrency between siblings (e.g. T_1 and T_2). It does make unrelated transactions wait for entire families to commit instead of single members with which it is in conflict over access to a data item.

12.8 Explain why serial equivalence requires that once a transaction has released a lock on an object, it is not allowed to obtain any more locks.

A server manages the objects a_1, a_2, ... a_n. The server provides two operations for its clients:

read (i) returns the value of a_i

write(i, Value) assigns *Value* to a_i

The transactions *T* and *U* are defined as follows:

T: x= read (i); write(j, 44);

U: *write(i, 55);write(j, 66);*

Describe an interleaving of the transactions *T* and *U* in which locks are released early with the effect that the interleaving is not serially equivalent.

12.8 Ans.

Because the ordering of different pairs of conflicting operations of two transactions must be the same.

For an example where locks are released early:

T	*T's locks*	*U*	*U's locks*
	lock i		
x:= read (i);			
	unlock i		
			lock i
		write(i, 55);	
			lock j
		write(j, 66);	
		commit	unlock i, j
	lock j		
write(j, 44);			
	unlock j		
commit			

T conflicts with *U* in access to a_i. Order of access is *T* then *U*.

T conflicts with *U* in access to a_j. Order of access is *U* then *T*. These interleavings are not serially equivalent.

12.9 The transactions *T* and *U* at the server in Exercise 12.8 are defined as follows:

T: x= read (i); write(j, 44);
U: write(i, 55);write(j, 66);

Initial values of a_i and a_j are 10 and 20. Which of the following interleavings are serially equivalent and which could occur with two-phase locking?

(a)

T	*U*
x= read (i);	
	write(i, 55);
write(j, 44);	
	write(j, 66);

(b)

T	*U*
x= read (i);	
write(j, 44);	
	write(i, 55);
	write(j, 66);

(c)

T	*U*
	write(i, 55);
	write(j, 66);
x= read (i);	
write(j, 44);	

(d)

T	*U*
	write(i, 55);
x= read (i);	
	write(j, 66);
write(j, 44);	

12.9 Ans.

a) serially equivalent but not with two-phase locking.

b) serially equivalent and with two-phase locking.

c) serially equivalent and with two-phase locking.

d) serially equivalent but not with two-phase locking.

12.10 Consider a relaxation of two-phase locks in which read only transactions can release read locks early. Would a read only transaction have consistent retrievals? Would the objects become inconsistent? Illustrate your answer with the following transactions *T* and *U* at the server in Exercise 12.8:

T: x = read (i); y= read(j);

U: *write(i, 55);write(j, 66);*

in which initial values of a_i and a_j are 10 and 20.

12.10 Ans.

There is no guarantee of consistent retrievals because overlapping transactions can alter the objects after they are unlocked.

The database does not become inconsistent.

T	*T's locks*	*U*	*U's locks*
	lock i		
x:= read (i);			
	unlock i		
			lock i
		write(i, 55)	
			lock j
		write(j, 66)	
		Commit	unlock i, j
	lock j		
y:= read(j)			
Commit	unlock j		

In the above example *T* is read only and conflicts with *U* in access to a_i and a_j. a_i is accessed by *T* before *U* and a_j by *U* before *T*. The interleavings are not serially equivalent. The values observed by *T* are x=10, y= 66, and the values of the objects at the end are a_i=55, a_j= 66.

Serial executions give either (*T* before *U*) x=10, y=20, a_i=55, a_j=66; or (*U* before *T*) x=55, y=66, a_i=55, a_j=66). This confirms that retrievals are inconsistent but that the database does not become inconsistent.

12.11 The executions of transactions are strict if *read* and *write* operations on an object are delayed until all transactions that previously wrote that object have either committed or aborted. Explain how the locking rules in Figure 12.16 ensure strict executions.

12.11 Ans.

If a previous transaction has written a data item, it holds its locks until after it has committed, therefore no other transaction may either *read* or *write* that data item (which is the requirement for serial executions).

12.12 Describe how a non-recoverable situation could arise if write locks are released after the last operation of a transaction but before its commitment.

12.12 Ans.

An earlier transaction may release its locks but not commit, meanwhile a later transaction uses the objects and commits. Then the earlier transaction may abort. The later transaction has done a dirty read and cannot be recovered because it has already committed.

12.13 Explain why executions are always strict even if read locks are released after the last operation of a transaction but before its commitment. Give an improved statement of Rule 2 in Figure 12.16.

12.13 Ans.

Strict executions require that *read* and *write* operations on a data item are delayed until all transactions that previously wrote that data item have either committed or aborted. The holding of a write lock is sufficient to protect future transactions from non-strictness because we are concerned only with previous *write* operations.

Rule 2: When the client indicates that the last operation has been done (by a request to commit or abort), release read locks. Hold write locks until commit or abort is completed.

12.14 Consider a deadlock detection scheme for a single server. Describe precisely when edges are added to and removed from the wait-for-graph.

Illustrate your answer with respect to the following transactions T, U and V at the server of Exercise 12.8.

T	U	V
	write(i, 66)	
write(i, 55)		
		write(i, 77)
	commit	

When U releases its write lock on a_i, both T and V are waiting to obtain write locks on it. Does your scheme work correctly if T (first come) is granted the lock before V? If your answer is 'No', then modify your description.

12.14 Ans.

Scheme:

When transaction T blocks on waiting for transaction U, add edge $T \rightarrow U$

When transaction T releases a lock, remove all edges leading to T.

Illustration: U has write lock on a_i.

T requests write a_i. Add $T \rightarrow U$

V requests write a_i. Add $V \rightarrow U$

U releases a_i. Delete both of above edges.

No it does not work correctly! When T proceeds, the graph is wrong because V is waiting for T and it should indicate $V \rightarrow$T.

Modification: we could make the algorithm unfair by always releasing the last transaction in the queue.

To make it fair: store both direct and indirect edges when conflicts arise. In our example, when transaction T blocks on waiting for transaction U add edge $T \rightarrow U$ then, when V starts waiting add $V \rightarrow U$ and $V \rightarrow T$

12.15 Consider hierarchic locks as illustrated in Figure 12.26. What locks must be set when an appointment is assigned to a time-slot in week w, day d, at time, t? In what order should these locks be set? Does the order in which they are released matter?

What locks must be set when the time slots for every day in week **w** are viewed?

Can this be done when the locks for assigning an appointment to a time-slot are already set?

12.15 Ans.

Set write lock on the time-slot t, intention-to-write locks on week w and day d in week w. The locks should be set from the top downwards (i.e. week w then day d then time t). The order in which locks are released does matter - they should be released from the bottom up.

When week w is viewed as a whole, a read lock should be set on week w. An intention-to-write lock is already set on week w (for assigning an appointment), the read lock must wait (see Figure 12.27).

12.16 Consider optimistic concurrency control as applied to the transactions T and U defined in Exercise 12.9. Suppose that transactions T and U are active at the same time as one another. Describe the outcome in each of the following cases:

i) T's request to commit comes first and backward validation is used;

ii) U's request to commit comes first and backward validation is used;

iii) T's request to commit comes first and forward validation is used;

iv) U's request to commit comes first and forward validation is used.

In each case describe the sequence in which the operations of T and U are performed, remembering that writes are not carried out until after validation.

12.16 Ans.

i) *T*'s *read(i)* is compared with *write*s of overlapping committed transactions: OK (*U* has not yet committed).

U - no *read* operations: OK.

ii) *U* - no *read* operations: OK.

T's *read(i)* is compared with *write*s of overlapping committed transactions (*U*'s *write(i)):* FAILS.

iii) *T*'s *write(j)* is compared with *reads* of overlapping active transactions (*U*): OK.

U's *write(i)* is compared with *reads* of overlapping active transactions (none): OK (*T* is no longer active).

iv) *U*'s *write(i)* is compared with *reads* of overlapping active transactions (*T*'s *read(i)*): FAILS.

T's *write(j)* is compared with *reads* of overlapping active transactions (none): OK.

(i)

T	*U*
x:= read (i);	
write(j, 44);	
	write(i, 55);
	write(j, 66);

(ii)

T	*U*
	write(i, 55);
x:= read (i);	*write(j, 66);*
Abort	

(iii)

T	*U*
x:= read (i);	
write(j, 44);	
	write(i, 55);
	write(j, 66);

(iv)

T	*U*
x:= read (i);	
	Abort
write(j, 44);	

12.17 Consider the following interleaving of transactions *T* and *U*:

T	*U*
openTransaction	*openTransaction*
y= read(k);	
	write(i, 55);
	write(j, 66);
	commit
x= read(i);	
write(j, 44);	

The outcome of optimistic concurrency control with backward validation is that *T* will be aborted because its read operation conflicts with *U*'s write operation on a_i, although the interleavings are serially equivalent. Suggest a modification to the algorithm that deals with such cases.

12.17 Ans.

Keep ordered read sets for active transactions. When a transaction commits after passing its validation, note the fact in the read sets of all active transactions. For example, when *U* commits, note the fact in *T*'s read set.

Thus, *T*'s read set = {*U* commit, i}

Then the new validate procedure becomes:

```
  boolean valid = true
for ($T_i$ = startTn + 1; $T_i$++; $T_i$ <= finishTn) {
    let S = set of members of read set of $T_j$ before commit $T_i$
    IF S intersects write set of $T_1$
        THEN valid := false
}
```

12.18 Make a comparison of the sequences of operations of the transactions *T* and *U* of Exercise 12.8 that are possible under two-phase locking (Exercise 12.9) and under optimistic concurrency control (Exercise 12.16).

12.18 Ans.

The order of interleavings allowed with two-phase locking depends on the order in which *T* and *U* access a_i. If *T* is first we get (b) and if *U* is first we get (c) in Exercise 12.9.

The ordering of 12.9b for two-phase locking is the same as 12.16 (i) optimistic concurrency control.

The ordering of 12.9c for two-phase locking is the same as 12.16 (ii) optimistic concurrency control if we allow transaction *T* to restart after aborting.

In this example, the sequences of operations are the same for both methods.

.

12.19 Consider the use of timestamp ordering with each of the example interleavings of transactions *T* and *U* in Exercise 12.9. Initial values of a_i and a_j are 10 and 20, respectively, and initial read and write timestamps are t_0. Assume each transaction opens and obtains a timestamp just before its first operation, for example, in (a) *T* and *U* get timestamps t_1 and t_2 respectively where $0 < t_1 < t_2$. Describe in order of increasing time the effects of each operation of *T* and *U*. For each operation, state the following:

i) whether the operation may proceed according to the write or read rule;

ii) timestamps assigned to transactions or objects;

iii) creation of tentative objects and their values.

What are the final values of the objects and their timestamps?

12.19 Ans.

a) Initially:

a_i: value = 10; write timestamp =max read timestamp = t_0
a_j: value = 20; write timestamp =max read timestamp = t_0

T: *x:= read (i)*; *T* timestamp = t_1;

read rule: t_1> write timestamp on committed version (t_0) and D_{selected} is committed:
allows *read* x = 10; max read timestamp(a_i) = t_1. (see Figure 12.31a and read rule page 500)

U: *write(i, 55)*; *U* timestamp = t_2

write rule: t_2 >= max read timestamp (t_1) and t_2> write timestamp on committed version (t_0):
allows *write* on tentative version a_i: value = 55; write timestamp =t. (See write rule page 499)

T: *write(j, 44)*;

write rule: t_1 >= max read timestamp (t_0) and t_1 > write timestamp on committed version (t_0):
allows *write* on tentative version a_j: value = 44; write timestamp(a_j) =t_1

U: *write(j, 66)*;

write rule: t_2 >= max read timestamp (t_0) and t_2 > write timestamp on committed version (t_0)
allows *write* on tentative version a_j: value = 66; write timestamp =t_2

T commits first:

a_j: committed version: value = 44; write timestamp =t_1; read timestamp = t_0

U commits:

a_i: committed version: value = 55; write timestamp =t_2; max read timestamp = t_1
a_j: committed version: value = 66; write timestamp =t_2; max read timestamp = t_0

b) Initially as (a);

T: *x:= read (i)*; *T* timestamp = t_1;

read rule: t_1> write timestamp on committed version (t_0) and D_{selected} is committed:
allows *read,* x = 10; max read timestamp(a_i) = t_1

T: *write(j,44)*

write rule: t_1 >= max read timestamp (t_0) and t_1 > write timestamp on committed version (t_0):
allows *write* on tentative version a_j: value = 44; write timestamp =t_1

U: *write(i, 55*); *U* timestamp = t_2

write rule: t_2 >= max read timestamp (t_1) and t_2> write timestamp on committed version (t_0):
allows *write* on tentative version a_i: value = 55; write timestamp =t_2

U: *write(j, 66)*;

write rule: t_2 >= max read timestamp (t_0) and t_2 > write timestamp on committed version (t_0):
allows *write* on tentative version a_j: value = 66; write timestamp =t_2

T commits first:

a_j: committed version: value = 44; write timestamp =t_1; max read timestamp = t_0

U commits:

a_i: committed version: value = 55; write timestamp =t_2; max read timestamp = t_1
a_j: committed version: value = 66; write timestamp =t_2; max read timestamp = t_0

c) Initially as (a);

U: *write(i, 55*); *U* time stamp = t_1;

write rule: t_1 >= max read timestamp (t_0) and t_2 > write timestamp on committed version (t_0):
allows *write* on tentative version a_i: value = 55; write timestamp =t_1

U: *write(j, 66)*;

write rule: t_2 >= max read timestamp (t_0) and t_2 > write timestamp on committed version (t_0):
allows *write* on tentative version a_j: value = 66; write timestamp =t_1

T: *x:= read (i)*; *T* time stamp = t_2

read rule: t_2> write timestamp on committed version (t_0), write timestamp of D_{selected} = t_1 and D_{selected} is not committed: WAIT for *U* to commit or abort. (See Figure 12.31 c)

U commits:

a_i: committed version: value = 55; write timestamp =t_1; max read timestamp(a_i) = t_0
a_j: committed version: value = 66; write timestamp =t_1; max read timestamp(a_j) = t_0

T continues by reading version made by *U* x = 55; write timestamp(a_i) =t_1; read timestamp(a_i) = t_2t_2

T: *write(j,44)*

write rule: t_2 >= max read timestamp (t_0) and t_2 > write timestamp on committed version (t_1):
allows *write* on tentative version a_j: value = 44; write timestamp =t_2

T commits:

a_i: committed version: value = 55; write timestamp =t_1; max read timestamp = t_2

a_j: committed version: value =44; write timestamp =t_2; max read timestamp = t_0

d) Initially as (a);

U: *write(i, 55*); *U* time stamp = t_1;

write rule: t_1 >= max read timestamp (t_0) and t_2> write timestamp on committed version (t_0)
allows *write* on tentative version a_i: value = 55; write timestamp(a_i) =t_1

T: *x:= read (i)*; *T* time stamp = t_2

read rule: t_2> write timestamp on committed version (t_0), write timestamp of D_{selected} = t1 and D_{selected} is not committed: Wait for *U* to commit or abort. (See Figure 12.31 c)

U: *write(j, 66)*;

write rule: t_2 >= max read timestamp (t_0) and t_2 > write timestamp on committed version(t_0) allows *write* on tentative version a_j: value = 66; write timestamp =t_1

U commits:

a_i: committed version: value = 55; write timestamp =t_1; max read timestamp = t_0
a_j: committed version: value = 66; write timestamp =t_1; max read timestamp = t_0

T continues by reading version made by *U*, x = 55; max read timestamp(a_i) = t_2

T: *write(j,44)*

write rule: t_2 >= max read timestamp (t_0) and t_2 > write timestamp on committed version (t_1) allows *write* on tentative version a_j: value = 44; write timestamp=t_2

T commits:

a_i: committed version: value = 55; write timestamp =t_1; max read timestamp = t_2
a_j: committed version: value = 44; write timestamp =t_2; max read timestamp = t_0

12.20 Repeat Exercise 12.19 for the following interleavings of transactions *T* and *U*:

T	*U*
openTransaction	
	openTransaction
	write(i, 55);
	write(j, 66);
x= read (i);	
write(j, 44);	
	commit

T	*U*
openTransaction	
	openTransaction
	write(i, 55);
	write(j, 66);
	commit
x= read (i);	
write(j, 44);	

12.20 Ans.

The difference between this exercise and the interleavings for Exercise 12.9(c) is that *T* gets its timestamp before *U*. The difference between the two orderings in this exercise is the time of the commit requests. Let t_1 and t_2 be the timestamps of *T* and *U*. The initial situation is as for 12.9 (a).

U: *write(i, 55*);

write rule: t_2 >= max read timestamp (t_0) and t_2> write timestamp on committed version (t_0) allows *write* on tentative version a_i: value = 55; write timestamp =t_2

U: *write(j, 66)*;

write rule: t_2 >= max read timestamp (t_0) and t_2 > write timestamp on committed version (t_0) allows *write* on tentative version a_j: value = 66; write timestamp =t_2

T: *x:= read (i)*;

read rule: t_1> write timestamp t_0 on committed version and D_{selected} is committed: allows *read,* x = 10; max read timestamp(a_i) = t_1

T: *write(j,44)*

write rule: t_1 >= max read timestamp (t_1) and t_1 > write timestamp on committed version (t_0) allows *write* on tentative version a_j: value = 44; write timestamp =t_2

U commits:

a_i: committed version: value = 55; write timestamp =t_2; max read timestamp = t_1

a_j: T has a tentative version with write timestamp = t_1. U's version with value = 66 and write timestamp =t_2 cannot be committed until after T's version.

T commits:

a_j: committed version: value = 44; write timestamp =t_1; max read timestamp = t_0; T's version is replaced with U's version: value = 66; write timestamp =t_2; max read timestamp = t_0

The second ordering proceeds in the same way as the first until U has performed both its *write* operations and commits. At this stage we have

a_i: committed version: value = 55; write timestamp =t_2; max read timestamp = t_0

a_j: committed version: value = 66; write timestamp =t_2; max read timestamp = t_0

T: *x:= read (i)*;

read rule: NOT t_1> write timestamp t_2 on committed version
T is Aborted (see Figure 12.31 d)

12.21 Repeat Exercise 12.20 using multiversion timestamp ordering.

12.21 Ans.

The main difference for multiversion timestamp ordering is that *read* operations can use old committed versions of objects instead of aborting when they are too late (see Page 502). The read rule is:

```
let $D_{selected}$ be the version of D with the maximum write timestamp <= Tj
IF $D_{selected}$ is committed THEN
    perform read operation on the version $D_{selected}$
ELSE Wait until the transaction that made version $D_{selected}$ commits or aborts
```

write operations cannot be too late, but writes are checked against potentially conflicting *read* operations. The write rule is taken from page 402. Recall that each data item has a history of committed versions.

For the ordering on the left of Exercise 12.20, we show that the outcome is the same. Timestamps are T: t_1 and U:t_2. Initial state as in Exercise 12.9 a. Call these committed versions V_{t0}.

U: *write(i, 55)*;

write rule: read timestamp of $D_{maxEarlier}$ t_0 <=t_2
allows *write* on tentative version a_i: value = 55; write timestamp =t_2

U: *write(j, 66)*;

write rule: read timestamp of $D_{maxEarlier}$ t_0 <= t_2
allows *write* on tentative version a_j: value = 66; write timestamp =t_2

T: *x:= read (i)*;

Select version with write timestamp t_0 *read* x = 10; its read timestamp becomes t_1

T: *write(j,44)*

write rule: read timestamp of $D_{maxEarlier}$ t_1 <= t_1
allows *write* on tentative version a_j: value = 44; write timestamp =t_1

U commits:

a_i: committed version V_{t2}: value = 55; write timestamp =t_2;
a_j: committed version V_{t2}: value = 66 and write timestamp =t_2

T commits:

a_j: committed version V_{t1}: value = 44; write timestamp =t_1

For the ordering on the right of Exercise 12.20, we show that the outcome is different in that T does not abort. It proceeds in the same way as the first until U has performed both its *write* operations and commits. At this stage we have:

a_i: committed version V_{t2}: value = 55; write timestamp =t_2

a_j: committed version V_{t2}: value = 66; write timestamp =t_2

T: *x:= read (i)*;

The *read* selects the committed version V_{t0} and gets x=10 and the read timestamp = t_1

T: *write(j,44)*

write rule: read timestamp of $D_{\text{maxEarlier}}$ t_1 <= t1

allows *write* on tentative version. a_j: value = 44; write timestamp =t_2

T commit:

a_j: committed version V_{t1}: value = 44; write timestamp =t1

12.22 In multiversion timestamp ordering, *read* operations can access tentative versions of objects. Give an example to show how cascading aborts can happen if all read operations are allowed to proceed immediately.

12.22 Ans.

The answer to Exercise 12.21 gives the read rule for multiversion timestamp ordering. *read* operations are delayed to ensure recoverability. In fact this delay also prevents dirty reads and cascading aborts.

Suppose now that *read* operations are not delayed, but that commits are delayed as follows to ensure recoverability. If a transaction, *T* has observed one of *U*'s tentative objects, then *T*'s commit is delayed until *U* commits or aborts (see page 503). Cascading aborts can occur when *U* aborts because *T* has done a dirty read and will have to abort as well.

To find an example, look for an answer to Exercise 12.19 where a *read* operation was delayed (i.e. (c) or (d)). Consider the diagram for (c) in Exercise 12.2. With delays, *T*'s *x := read(i)* is delayed until *U* commits or aborts. Now consider allowing *T*'s *x := read(i)* to use *U*'s tentative version immediately. Then consider the situation in which *T* asks to commit and *U* subsequently aborts. Note that *T*'s commit request is delayed until the outcome of *U* is known, so the situation is recoverable, but *T* has performed a 'dirty read' and must be aborted. Cascading aborts can now arise because some other transactions may have observed *T*'s tentative objects.

12.23 What are the advantages and drawbacks of multiversion timestamp ordering in comparison with ordinary timestamp ordering?

12.23 Ans.

The algorithm allows more concurrency than single version timestamp ordering but incurs additional storage costs.

Advantages:

The presence of multiple committed versions allows late *read* operations to succeed.

write operations are allowed to proceed immediately unless they will invalidate earlier reads (a *write* by a transaction with timestamp T_i is rejected if a transaction with timestamp T_j has read a data item with write timestamp T_k and $T_k < T_i < T_j$).

Drawbacks:

The algorithm requires storage for multiple versions of each committed objects and for information about the read and write timestamps of each version to be used in carrying out the read and write rules. In the case that a version is deleted, *read* operations will have to be rejected and transactions aborted.

Exercise 13.15 shows that the algorithm can provide yet more concurrency, at the risk of cascading aborts, by allowing *read* operations to proceed immediately. In this case, to ensure recoverability, requests to commit must be delayed until any the completion (commitment or abortion) of any transaction whose tentative objects have been observed.

12.24 Make a comparison of the sequences of operations of the transactions *T* and *U* of Exercise 12.8 that are possible under two-phase locking (Exercise 12.9) and under optimistic concurrency control (Exercise 12.16).

12.24 Ans.

The order of interleavings allowed with two-phase locking depends on the order in which *T* and *U* access a_i. If *T* is first we get (b) and if *U* is first we get (c) in Exercise 12.9.

The ordering of 12.9b for two-phase locking is the same as 12.9 (i) optimistic concurrency control.

The ordering of 12.9c for two-phase locking is the same as 12.9 (ii) optimistic concurrency control if we allow transaction *T* to restart after aborting.

Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001.

Chapter 13 Exercise Solutions

13.1 In a decentralized variant of the two-phase commit protocol the participants communicate directly with one another instead of indirectly via the coordinator. In Phase 1, the coordinator sends its vote to all the participants. In Phase 2, if the coordinator's vote is *No*, the participants just abort the transaction; if it is *Yes*, each participant sends its vote to the coordinator and the other participants, each of which decides on the outcome according to the vote and carries it out. Calculate the number of messages and the number of rounds it takes. What are its advantages or disadvantages in comparison with the centralized variant?

13.1 Ans.

In both cases, we consider the normal case with no time outs.
In the decentralised version of the two-phase commit protocol:

No of messages:

Phase 1: coordinator sends its vote to *N* workers = *N*

Phase 2: each of *N* workers sends its vote to (*N*-1) other workers + coordinator = *N**(*N* - 1).

Total = *N**N*.

No. of rounds:

coordinator to workers + workers to others = 2 rounds.

Advantages: the number of rounds is less than for normal two-phase commit protocol which requires 3. Disadvantages: the number of messages is far more: *N**N* instead of 3*N*.

13.2 A three-phase commit protocol has the following parts:

Phase 1: is the same as for two-phase commit.

Phase 2: the coordinator collects the votes and makes a decision; if it is *No*, it *aborts* and informs participants that voted *Yes*; if the decision is *Yes*, it sends a *preCommit* request to all the participants. participants that voted *Yes* wait for a *preCommit* or *doAbort* request. They acknowledge *preCommit* requests and carry out *doAbort* requests.

Phase 3: the coordinator collects the acknowledgments. When all are received, it *Commits* and sends *doCommit* to the participants. participants wait for a *doCommit* request. When it arrives they *Commit*.

Explain how this protocol avoids delay to participants during their 'uncertain' period due to the failure of the coordinator or other participants. Assume that communication does not fail.

13.2 Ans.

In the two-phase commit protocol: the 'uncertain' period occurs because a worker has voted *yes* but has not yet been told the outcome. (It can no longer abort unilaterally).

In the three-phase commit protocol: the workers 'uncertain' period lasts from when the worker votes *yes* until it receives the *PreCommit* request. At this stage, no other participant can have committed. Therefore if a group of workers discover that they are all 'uncertain' and the coordinator cannot be contacted, they can decide unilaterally to abort.

13.3 Explain how the two-phase commit protocol for nested transactions ensures that if the top-level transaction commits, all the right descendents are committed or aborted.

13.3 Ans.

Whenever a nested transaction commits, it reports its status and the status of its descendants to its parent. Therefore when a transaction enters the committed state, it has a correct list of its committed descendants. Therefore when the top-level transaction starts the two-phase commit protocol, its list of committed descendants is correct. It checks the descendants and makes sure they can still commit or must abort. There may be nodes that ran unsuccessful descendants which are not included in the two-phase commit protocol. These will discover the outcome by querying the top-level transaction.

13.4 Give an example of the interleavings of two transactions that is serially equivalent at each server but is not serially equivalent globally.

13.4 Ans.

Schedule at server *X*:

T: Read(*A*); Write(*A*); *U*:Read(*A*); Write(*A*); serially equivalent with *T* before *U*

Schedule at Server *Y*:

U: Read(*B*); Write(*B*); *T*: Read(*B*); Write(*B*); serially equivalent with *U* before *T*

This is not serially equivalent globally because there is a cycle $T \rightarrow U \rightarrow T$.

13.5 The *getDecision* procedure defined in Figure 13.4 is provided only by coordinators. Define a new version of *getDecision* to be provided by participants for use by other participants that need to obtain a decision when the coordinator is unavailable.

Assume that any active participant can make a *getDecision* request to any other active participant. Does this solve the problem of delay during the ‘uncertain’ period? Explain your answer.

At what point in the two-phase commit protocol would the coordinator inform the participants of the other participants’ identities (to enable this communication)?

13.5 Ans.

The signature for the new version is:

getDecision (trans) -> Yes/ No/ Uncertain

The worker replies as follows:

If it has already received the *doCommit* or *doAbort* from the coordinator or received the result via another worker, then reply *Yes* or *No;*

if it has not yet voted, reply *No* (the workers can abort because a decision cannot yet have been reached);

if it is uncertain, reply *uncertain.*

This does not solve the problem of delay during the ‘uncertain’ period. If all of the currently active workers are uncertain, they will remain uncertain.

The coordinator can inform the workers of the other workers’ identities when it sends out the *canCommit* request.

13.6 Extend the definition of two-phase locking to apply to distributed transactions. Explain how this is ensured by distributed transactions using strict two-phase locking locally.

13.6 Ans.

Two-phase locking in a distributed transaction requires that it cannot acquire a lock at any server after it has released a lock at any server.

A client transaction will not request *commit* (or *abort*) (at the coordinator) until after it has made all its requests and had replies from the various servers involved, by which time all the locks will have been acquired. After that, the coordinator sends on the *commit* or *abort* to the other servers which release the locks. Thus all locks are acquired first and then they are all released, which is two-phase locking

13.7 Assuming that strict two-phase locking is in use, describe how the actions of the two-phase commit protocol relate to the concurrency control actions of each individual server. How does distributed deadlock detection fit in?

13.7 Ans.

Each individual server sets locks on its own data items according to the requests it receives, making transactions wait when others hold locks.

When the coordinator in the two-phase commit protocol sends the *doCommit* or *doAbort* request for a particular transaction, each individual server (including the coordinator) first carries out the commit or abort action and then releases all the local locks held by that transaction.

(Workers in the uncertain state may hold locks for a very long time if the coordinator fails)

Only transactions that are waiting on locks can be involved in deadlock cycles. When a transaction is waiting on a lock it has not yet reached the client request to commit, so a transaction in a deadlock cycle cannot be involved in the two-phase commit protocol.

When a transaction is aborted to break a cycle, the coordinator is informed by the deadlock detector. The coordinator then sends the *doAbort* to the workers.

13.8 A server uses timestamp ordering for local concurrency control. What changes must be made to adapt it for use with distributed transactions? Under what conditions could it be argued that the two-phase commit protocol is redundant with timestamp ordering?

13.8 Ans.

Timestamps for local concurrency control are just local counters. But for distributed transactions, timestamps at different servers must have an agreed global ordering. For example, they can be generated as (local timestamp, server-id) to make them different. The local timestamps must be roughly synchronized between servers.

With timestamp ordering, a transaction may be aborted early at one of the servers by the read or write rule, in which case the *abort* result is returned to the client. If a server crashes before the client has done all its actions at that server, the client will realise that the transaction has failed. In both of these cases the client should the send an *abortTransaction* to the coordinator.

When the client request to *commit* arrives, the servers should all be able to commit, provided they have not crashed after their last operation in the transaction.

The two-phase commit protocol can be considered redundant under the conditions that (i) servers are assumed to make their changes persistent before replying to the client after each successful action and (ii) the client does not attempt to commit transactions that have failed.

13.9 Consider distributed optimistic concurrency control in which each server performs local backward validation sequentially (that is, with only one transaction in the validate and update phase at one time), in relation to your answer to Exercise 13.4. Describe the possible outcomes when the two transactions attempt to commit. What difference does it make if the servers use parallel validation?

13.9 Ans.

At server *X*, *T* precedes *U*. At server *Y*, *U* precedes *T*. These are not serially equivalent because there are Read/ Write conflicts.

T starts validation at server *X* and passes, but is not yet committed. It requests validation at server *Y*. If *U* has not yet started validation, *Y* can validate *T*. Then *U* validates after *T* (at both). Similarly for *T* after *U*.

T starts validation at server *X* and passes, but is not yet committed. It requests validation at server *Y*. If *U* has started validation, *T* will be blocked. When *U* requests validation at *X*, it will be blocked too. So there is a deadlock.

If parallel validation is used, *T* and *U* can be validated (in different orders) at the two servers, which is wrong.

13.10 A centralized global deadlock detector holds the union of local wait-for graphs. Give an example to explain how a phantom deadlock could be detected if a waiting transaction in a deadlock cycle aborts during the deadlock detection procedure.

13.10 Ans.

A centralized global deadlock detector holds the union of local wait-for graphs. Give an example to explain how a phantom deadlock could be detected if a waiting transaction in a deadlock cycle aborts during the deadlock detection procedure.

Suppose that at servers *X*, *Y* and *Z* we have:

X	*Y*	*Z*	*global detector*
$T \rightarrow U$	U$\rightarrow V$	V$\rightarrow T$	
	U aborts		$T \rightarrow U$U$\rightarrow V$V$\rightarrow T$
$T \rightarrow U$	-	V$\rightarrow T$	
-	-	V$\rightarrow T$	V$\rightarrow T$

when *U* aborts, *Y* knows first, then *X* finds out, eventually the global detector finds out, but by then it may be too late (it will have detected a deadlock).

13.11 Consider the edge chasing algorithm (without priorities). Give examples to show that it could detect phantom deadlocks.

13.11 Ans.

Transaction *U*, *V* and *W* perform operations on a data item at each of the servers *X*, *Y* and *Z* in the following order:

U gets data item at *Y*

V gets data item at *X* and then blocks at *Y*.

W gets data item at *Z*

U blocks at *Y*.

W blocks at *X*.

V aborts at *Y*

The table below shows three servers *X*, *Y* and *Z*, the transactions they coordinate, the holders and requesters of their data items and the corresponding wait-for relationships before *V* aborts:

X (coordinator of: *V*)	*Y* (coordinator of: *U*)	*Z* (coordinator of: *W*)
held by: *V* requested by: *W*	held by: *U* requested by: *V*	held by: *W* requested by: *U*
$W \rightarrow V$ (blocked at *Y*)	$V \rightarrow U$ (blocked at *Z*)	$U \rightarrow W$ (blocked at *X*)

Now consider the probes sent out by the three servers:

At server *X*: $W \rightarrow V$ (which is blocked at *Y*); probe <$W \rightarrow V$> sent to *Y*; then

At *Y*: probe <$W \rightarrow V$> received; observes $V \rightarrow U$ (blocked at *Z*), so adds to probe to get <$W \rightarrow$ V$\rightarrow U$> and sends it to *Z*.

When *V* aborts at *Y*; *Y* tells *X* - the coordinator of *V*, but *V* has not visited *Z*, so *Z* will not be told about the fact that *V* has aborted!

At *Z*: probe <$W \rightarrow$ V$\rightarrow U$> is received; *Z* observes $U \rightarrow W$ and notes the cycle $W \rightarrow$ V$\rightarrow$ U$\rightarrow W$ and as a result detects phantom deadlock. It picks a victim, (presumably not *V*).

13.12 A server manages the objects a_1, a_2,... a_n. The server provides two operations for its clients:

Read (i) returns the value of a_i

Write(i, Value) assigns *Value* to a_i

The transactions *T*, *U* and *V* are defined as follows:

T: x= Read (i); Write(j, 44);

U: *Write(i, 55);Write(j, 66);*

V: *Write(k, 77);Write(k, 88);*

Describe the information written to the log file on behalf of these three transactions if strict two-phase locking is in use and *U* acquires a_i and a_j before *T*. Describe how the recovery manager would use this information to recover the effects of *T*, *U* and *V* when the server is replaced after a crash. What is the significance of the order of the commit entries in the log file?

13.12 Ans.

As transaction *U* acquires a_j first, it commits first and the entries of transaction *T* follow those of *U*. For simplicity we show the entries of transaction *V* after those of transaction *T*.

P_0: ...	P_1: Data: i 55	P_2: Data: j 66	P_3: Trans: *U* prepared <i, P_1> <j, P_2> P_0	P_4: Trans: *U* commit P_3;
P_5: Data: j 44	P_6: Trans: *T* prepared <j, P_5> P_4	P_7: Trans: *T* commit P_6;	P_8: Data: k 88	P_9: Trans: *V* prepared <k,P_8> P_7
P_{10}: Trans: *V* commit P_9				

The diagram is similar to Figure 13.9. It shows the information placed at positions P_0, P_1, ...P_{10} in the log file.

On recovery, the recovery manager sets default values in the data items $a_{1..}.a_n$. It then starts at the end of the log file (at position P_{10}). It sees *V* has committed, finds P_9 and *V*'s intentions list <*k*,P_8> and restores a_k=88. It then goes back to P_7 *(T commit)*, back to P_6 for *T*'s intentions list <*j*,P_5> and restores a_j=44. It then goes back to P_4 (*U commit*), back to P_3 for *U*'s intentions list <*i*,P_1><*j*, P_2>. It ignores the entry for a_j because it has already been recovered, but it gets a_i=55. The values of the other data items are found earlier in the log file or in a checkpoint.

The order of the commit entries in the log file reflect the order in which transactions committed. More recent transactions come after earlier ones. Recovery starts from the end, taking the effects of the most recent transactions first.

13.13 The appending of an entry to the log file is atomic, but append operations from different transactions may be interleaved. How does this affect the answer to Exercise 13.12?

13.13 Ans.

As there are no conflicts between the operations of transaction *V* and those of *T* and *U*, the log entries due to transaction *V* could be interleaved with those due to transactions *T* and *U*. In contrast, the entries due to *T* and *U* cannot be interleaved because the locks on a_i and a_j ensure that *U* precedes *T*.

13.14 The transactions *T*, *U* and *V* of Exercise 13.12 use strict two-phase locking and their requests are interleaved as follows:

T	*U*	*V*
x = Read(i);		
		Write(k, 77);
	Write(i, 55)	
Write(j, 44)		
		Write(k,88)
	Write(j, 66)	

Assuming that the recovery manager appends the data entry corresponding to each *Write* operation to the log file immediately instead of waiting until the end of the transaction, describe the information written to the log file on behalf of the transactions *T*, *U* and *V*. Does early writing affect the correctness of the recovery procedure? What are the advantages and disadvantages of early writing?

13.14 Ans.

As *T* acquires a read lock on a_i, *U*'s *Write(i,55)* waits until *T* has committed and released the lock:

P_0: ...	P_1:Data: k 77	P_2:Data: j 44	P_3:Trans: *T* prepared <j, P_2> P_0	P_4:Data: k 88
P_5: Trans: *V* prepared <k,P_4> P_3	P_6: Trans: *T* commit P_5	P_7: Trans: *V* commit P_6	P_8:Data: i 55;	P_9: Data: j 66
P_{10}: Trans: *U* prepared <i,P_8> <j,P_9> P_7	P11: Trans: *U* commit P_{10}			

We have shown a possible interleaving of *V*'s *Write(k,88)* and *prepared* entries between *T*'s *prepared* and *commit* entries. Early writing does not affect the correctness of the recovery procedure because the *commit* entries reflect the order in which transactions were committed.

Disadvantages of early writing: a transaction may abort after entries have been written, due to deadlock. Also there can be duplicate entries (like *k*=77 and *k*=88) if the same data item is written twice by the same transaction.

Advantages of early writing: commitment of transactions is faster.

13.15 The transactions *T* and *U* are run with timestamp ordering concurrency control. Describe the information written to the log file on behalf of *T* and *U*, allowing for the fact that *U* has a later timestamp than *T* and must wait to commit after *T*. Why is it essential that the commit entries in the log file should be ordered by timestamps? Describe the effect of recovery if the server crashes (i) between the two *Commits* and (ii) after both of them.

T	*U*
x= Read(i);	
	Write(i, 55);
	Write(j, 66);
Write(j, 44);	
	Commit
Commit	

What are the advantages and disadvantages of early writing with timestamp ordering?

13.15 Ans.

The timestamps of *T* and *U* are put in the recovery file. Call them *t*(*T*) and *t*(*U*), where *t*(*T*)<*t*(*U*).

P_0: ...	P_1: Data: i 55	P_2: Data: j 66	P_3: Data: j 44	P_4:Trans: t(*U*) prepared <i,P_1> <j,P_2> P_0
P_5:Trans: t(*U*) waiting to commit P_4	P_6:Trans: t(*T*) prepared <j,P_4> P_5	P_7: Trans: t(*T*) commit P_6	P_8: Trans: t(*U*) commit P_7	

The entry at P_5 shows that *U* has committed, but must be ordered after *T*. If the transaction *T* aborts or the server fails before *T* commits, this entry indicates that *U* has committed.

It essential that the commit entries in the log file should be ordered by timestamps because recovery works through the log file backwards. The effects of later transactions must be over write the effects of earlier ones.

The effect of recovery:

(i) if the server crashes between the two commits, we lose the entry at P_8, but we have P_6: *Trans t(U) waiting to commit*. Transaction *U* can be committed as the transaction it waits for has either committed, or if it has not yet committed, it will be aborted.

(ii)if the server crashes after both the commits, both will be recovered.

The advantage of early writing with timestamp ordering is that commitment is quicker. Transactions can always commit if they get that far (clients don't abort them). There do not appear to be any important disadvantages.

13.16 The transactions *T* and *U* in Exercise 13.15 are run with optimistic concurrency control using backward validation and restarting any transactions that fail. Describe the information written to the log file on their behalf. Why is it essential that the commit entries in the log file should be ordered by transaction numbers? How are the write sets of committed transactions represented in the log file?

13.16 Ans.

P_0: ...	P_1: Data: i 55	P_2: Data: j 66	P_3: Trans: T_U prepared <i,P_1> <j,P_2> P_0	P_4: Trans: T_U commit P_3
P_5: Data: j 44	P_6:Trans T_T prepared <j,P_4> P_5	P_7: Trans T_T commit P_6		

Transaction numbers rather than transaction identifiers (or timestamps) go in the recovery file after a transaction has passed its validation. *U* passes validation because it has no *read* operations. Suppose *U* is given the transaction number T_U. Transaction *T* fails validation because its read set {*i*} overlaps with *U*'s Write set {i,j}. *T* is restarted after *U*. It passes validation with transaction number T_T. where $T_T > T_U$

It essential that the commit entries in the log file should be ordered by transaction number because transaction numbers reflect the order in which transactions are committed at the server. Recovery takes transactions from newest to oldest by reading the log file backwards.

Write sets are represented in the log file in the prepared entries.

13.17 Suppose that the coordinator of a transaction crashes after it has recorded the intentions list entry but before it has recorded the participant list or sent out the *canCommit?* requests. Describe how the participants resolve the situation. What will the coordinator do when it recovers? Would it be any better to record the participant list before the intentions list entry?

13.17 Ans.

As the coordinator is the only server to receive the *closeTransaction* request from the client, the workers will not know the transaction has ended, but they can time out and unilaterally decide to abort the transaction (see pages 521-3). They are allowed to do this because they have not yet voted. When the coordinator recovers it also aborts the transaction.

An apparent advantage of recording a worker list earlier (before the coordinator fails), is that it could be used to notify the workers when a coordinator recovers, with a view to avoiding the need for timeouts in workers. Unfortunately workers cannot avoid the need for timeouts because the coordinator may not recover for a very long time.

Next question was omitted from the third edition, because it would have been alone on an odd numbered page.

13.18 Consider the distributed transaction *T* in Figure 13.3. Describe the information concerning transaction *T* that would be written to the log files at each of the servers if the two-phase commit protocol is completed and *T* is committed.

Suppose that the server *BranchY* crashes when it is 'uncertain': will this affect the progress of *BranchX* and *BranchZ*? Describe the recovery at *BranchY* relating to transaction *T*; what is the responsibility of *BranchX* with respect to the recovery at *BranchY*?

Suppose that *BranchX* crashes after sending out the vote requests, but *BranchY* and *BranchZ* are still active: describe the effect on the three logs and describe the recovery of *BranchX*.

13.18 Ans..

Each server will have its own log file. These are shown in the rows of the table below. We assume that the balances of *A*, *B*, *C* and *D* are initially \$100, \$200, \$300 and \$400.

BranchX:	P_0...	P_1: Data: *A* 96	P_2: Trans: *T* prepared <*A*,P_1> P_0	P_3: Coord'r: *T* workers: BranchY BranchZ P_2	P_4: Trans: *T* committed P_3	P_5: Trans: *T* done P_4	
BranchY:	P_0...	P_1: Data: *B* 197	P_2: Trans: *T* prepared <*B*,P_1> P_0	P_3: Worker: *T* coordinator: BranchX P_2	P_4: Trans: *T* uncertain P_3	P_5:Trans: *T* committed P_4	
BranchZ:	P_0...	P_1: Data: *C* 304	P_2: Data: *D* 403	P_3: Trans: *T* prepared <*C*,P_1> <*D*,P_2> P_0	P_4: Worker: *T* coordinator: BranchX P_3	P_5: Trans: *T* uncertain P_4	P_6: Trans *T*: committed P_5

If the server *BranchY* crashes when it is 'uncertain' this will not affect *BranchX* and *BranchZ* because all servers have voted *yes* and the coordinator can decide to commit the transaction. It sends *DoCommit* to *BranchY* (which does not reply) and *BranchZ* which does.

When *BranchY* recovers, the *T uncertain* entry will be found. A *getDecision* request will be sent to *BranchX* which will inform *BranchY* that the transaction has committed. The responsibility of *BranchX* is to record the outcome of *T* until it gets an acknowledgement from all the servers (including *BranchY*). It will not record *T done* until this is the case, meanwhile it will not remove *T committed* if checkpointing takes place.

If *BranchX* crashes after sending out the vote requests, but *BranchY* and *BranchZ* are still active then *BranchY* and *BranchZ* will reply *yes* and become *uncertain*. When *BranchX* recovers it is *prepared* and will decide to abort the transaction and will inform the other two servers. All three will record *T aborted* in their logs.

Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001

Chapter 14 Exercise Solutions

14.1 Three computers together provide a replicated service. The manufacturers claim that each computer has a mean time between failure of five days; a failure typically takes four hours to fix. What is the availability of the replicated service?

14.1 Ans.

The probability that an individual computer is down is 4/(5*24 + 4) ~ 0.03. Assuming failure-independence of the machines, the availability is therefore $1 – 0.03^3 = 0.999973$.

14.2 Explain why a multi-threaded server might not qualify as a state machine.

14.2 Ans.

The order in which operations are applied within such a server might differ from the order in which they are initiated. This is because operations could be delayed waiting for some other resource, and the resource scheduling policy could, in principle, reverse the order of two operations.

14.3 In a multi-user game, the players move figures around a common scene. The state of the game is replicated at the players' workstations and at a server, which contains services controlling the game overall, such as collision detection. Updates are multicast to all replicas.

(i) The figures may throw projectiles at one another and a hit debilitates the unfortunate recipient for a limited time. What type of update ordering is required here? Hint: consider the 'throw', 'collide' and 'revive' events.

(ii) The game incorporates magic devices which may be picked up by a player to assist them. What type of ordering should be applied to the pick-up-device operation?

14.3 Ans.

i) The event of the collision between the projectile and the figure, and the event of the player being debilitated (which, we may assume, is represented graphically) should occur in causal order. Moreover, changes in the velocity of the figure and the projectile chasing it should be causally ordered. Assume that the workstation at which the projectile was launched regularly announces the projectile's coordinates, and that the workstation of the player corresponding to the figure regularly announces the figure's coordinates and announces the figure's debilitation. These announcements should be processed in causal order. (The reader may care to think of other ways of organising consistent views at the different workstations.)

ii) If two players move to pick up a piece at more-or-less the same time, only one should succeed and the identity of the successful player should be agreed at all workstations. Therefore total ordering is required.

The most promising architecture for a game such as this is a peer group of game processes, one at each player's workstation. This is the architecture most likely to meet the real-time update propagation requirements; it also is robust against the failure of any one workstation (assuming that at least two players are playing at the same time).

14.4 A router separating process *p* from two others, *q* and *r*, fails immediately after *p* initiates the multicasting of message *m*. If the group communication system is view-synchronous, explain what happens to *p* next.

14.4 Ans.

Process *p* must receive a new group view containing only itself, and it must receive the message it sent. The question is: in what order should these events be delivered to *p*?.

If p received the message first, then that would tell p that q and r received the message; but the question implies that they did not receive it. So p must receive the group view first.

14.5 You are given a group communication system with a totally ordered multicast operation, and a failure detector. Is it possible to construct view-synchronous group communication from these components alone?

14.5 Ans.

If the multicast is reliable, yes. Then we can solve consensus. In particular, we can decide, for each message, the view of the group to deliver it to. Since both messages and new group views can be totally ordered, the resultant communication will be view-synchronous. If the multicast is unreliable, then we do not have a way of ensuring the consistency of view delivery to all of the processes involved.

14.6 A *sync-ordered* multicast operation is one whose delivery ordering semantics are the same as those for delivering views in a view-synchronous group communication system. In a *thingumajig* service, operations upon thingumajigs are causally ordered. The service supports lists of users able to perform operations on each particular thingumajig. Explain why removing a user from a list should be a sync-ordered operation.

14.6 Ans.

Sync-ordering the remove-user update ensures that all processes handle the same set of operations on a thingumajig before the user is removed. If removal were only causally ordered, there would not be any definite delivery ordering between that operation and any other on the thingumajig. Two processes might receive an operation from that user respectively before and after the user was removed, so that one process would reject the operation and the other would not.

14.7 What is the consistency issue raised by state transfer?

14.7 Ans.

When a process joins a group it acquires state S from one or more members of the group. It may then start receiving messages destined for the group, which it processes. The consistency problem consists of ensuring that no update message that is already reflected in the value S will be applied to it again; and, conversely, that any update message that is not reflected in S will be subsequently received and processed.

14.8 An operation X upon an object o causes o to invoke an operation upon another object o'. It is now proposed to replicate o but not o'. Explain the difficulty that this raises concerning invocations upon o', and suggest a solution.

14.8 Ans.

The danger is that all replicas of o will issue invocations upon o', when only one should take place. This is incorrect unless the invocation upon o' is idempotent and all replicas issue the same invocation.

One solution is for the replicas of o to be provided with smart, replication-aware proxies to o'. The smart proxies run a consensus algorithm to assign a unique identifier to each invocation and to assign one of them to handle the invocation. Only that smart proxy forwards the invocation request; the others wait for it to multicast the response to them, and pass the results back to their replica.

14.9 Explain the difference between linearizability and sequential consistency, and why the latter is more practical to implement, in general.

14.9 Ans.

See pp. 566-567 for the difference. In the absence of clock synchronization of sufficient precision, linearizability can only be achieved by funnelling all requests through a single server – making it a performance bottleneck.

14.10 Explain why allowing backups to process read operations leads to sequentially consistent rather than linearizable executions in a passive replication system.

14.10 Ans.

Due to delays in update propagation, a *read* operation processed at a backup could retrieve results that are older than those at the primary – that is, results that are older than those of an earlier operation requested by another process. So the execution is not linearizable.

The system is sequentially consistent, however: the primary totally orders all updates, and each process sees some consistent interleaving of *reads* between the same series of updates.

14.11 Could the gossip architecture be used for a distributed computer game as described in Exercise 14.3?

14.11 Ans.

As far as ordering is concerned, the answer is ‘yes’ – gossip supports causal and total ordering. However, gossip introduces essentially arbitrary propagation delays, instead of the best-effort propagation of multicast. Long delays would tend to affect the interactivity of the game.

14.12 In the gossip architecture, why does a replica manager need to keep both a ‘replica’ timestamp and a ‘value’ timestamp?

14.12 Ans.

The value timestamp reflects the operations that the replica manager has applied. Replica managers also need to manage operations that they cannot yet apply. In particular, they need to assign identifiers to new operations, and they need to keep track of which updates they have received in gossip messages, whether or not they have applied them yet. The replica timestamp reflects updates that the replica manager has received, whether or not it has applied them all yet.

14.13 In a gossip system, a front end has vector timestamp (3, 5, 7) representing the data it has received from members of a group of three replica managers. The three replica managers have vector timestamps (5, 2, 8), (4, 5, 6) and (4, 5, 8), respectively. Which replica manager(s) could immediately satisfy a query from the front end and what is the resultant time stamp of the front end? Which could incorporate an update from the front end immediately?

14.13 Ans.

The only replica manager that can satisfy a query from this front end is the third, with (value) timestamp (4,5,8). The others have not yet processed at least one update seen by the front end. The resultant time stamp of the front end will be (4,5,8).

Similarly, only the third replica manager could incorporate an update from the front-end immediately.

14.14 Explain why making some replica managers read-only may improve the performance of a gossip system.

14.14 Ans.

First, read operations may be satisfied by local read-only replica managers, while updates are processed by just a few other replica managers. This is an efficient arrangement if on average there are many read operations to every write operation. Second, since read-only replicas do not accept updates, they need no vector timestamp entries. Vector timestamp sizes are therefore reduced.

14.15 Write pseudocode for dependency checks and merge procedures (as used in Bayou) suitable for a simple room-booking application.

14.15 Ans.

```
Operation: room.book(booking).
    let timeSlot = booking.getPreferredTimeSlot();
    Dependency check:
    existingBooking = room.getBooking(timeSlot);
    if (existingBooking != null) return “conflict” else return “no conflict”;
    Merge procedure:
    existingBooking = room.getBooking(timeSlot);
    // Choose the booking that should take precedence over the other
    if (greatestPriority(existingBooking, booking) == booking)
        then { room.setBooking(timeSlot, booking); existingBooking.setStatus(“failed”);}
        else {booking.setStatus(“failed”);}
```

– in a more sophisticated version of this scheme, bookings have alternative time slots. When a booking cannot be made at the preferred time slot, the merge procedure runs through the alternative time slots and only reports failure if none is available. Similarly, alternative rooms could be tried.

14.16 In the Coda file system, why is it sometimes necessary for users to intervene manually in the process of updating the copies of a file at multiple servers?

14.16 Ans.

Conflicts may be detected between the timestamps of versions of files at a Coda file server and a disconnected workstation when the workstation is reintegrated, Conflicts arise because the versions have diverged, that is, the version on the file server may have been updated by one client and the version on the workstation by another. When such conflicts occur, the version of the file from the workstation is placed in a covolume – an off-line version of the file volume that is awaiting manual processing by a user. The user may either reject the new version, install the new version in preference to the one on the server, or merge the two files using a tool appropriate to the format of the file.

14.17 Devise a scheme for integrating two replicas of a file system directory that underwent separate updates during disconnected operation. Use either Bayou's operational transformation approach, or supply a solution for Coda.

14.17 Ans.

The updates possible on a directory are (a) changing protection settings on existing entries or the directory itself, (b) adding entries and (c) deleting entries.

Many updates may be automatically reconciled, e.g. if two entries with different names were added in different partitions then both are added; if an entry was removed in one partition and not updated in the other then the removal is confirmed; if an entry's permissions were updated in one partition and it was not updated (including deletion) in the other, then the permissions-update is confirmed.

Otherwise, two updates, in partitions A and B, respectively, may conflict in such a way that automatic reconciliation is not possible. e.g. an entry was removed in A and the same entry in B had its permissions changed; entries were created with the same name (but referring to a different file) in A and B; an entry was added in A but in B the directory's write permissions were removed.

We leave the details to the reader.

14.18 Available copies replication is applied to data items *A* and *B* with replicas A_x, A_y and B_m, B_n. The transactions *T* and *U* are defined as:

T: *Read*(*A*); *Write*(*B,* 44). *U*: *Read*(*B*); *Write*(*A,* 55).

Show an interleaving of *T* and *U*, assuming that two-phase locks are applied to the replicas. Explain why locks alone cannot ensure one copy serializability if one of the replicas fails during the progress of *T* and *U*. Explain with reference to this example, how local validation ensures one copy serializability.

14.18 Ans.

An interleaving of T and U at the replicas assuming that two-phase locks are applied to the replicas:

T		U	
x:= Read (Ax)	lock Ax		
Write(Bm, 44)	lock Bm		
		x:= Read (Bm)	Wait
Write(Bn, 44)	lock Bn	•	
Commit unlock	Ax,Bm,Bn	•	
		Write(Ax, 55)	lock Ax
		Write(Ay, 55)	lock Ay

Suppose Bm fails before T locks it, then U will not be delayed. (It will get a lost update). The problem arises because Read can use one of the copies before it fails and then Write can use the other copy. Local validation ensures one copy serializability by checking before it commits that any copy that failed has not yet been recovered. In the case of T, which observed the failure of Bm, Bm should not yet have been recovered, but it has, so T is aborted.

14.19 Gifford's quorum consensus replication is in use at servers *X*, *Y* and *Z* which all hold replicas of data items *A* and *B*. The initial values of all replicas of *A* and *B* are 100 and the votes for *A* and *B* are 1 at each of *X*, *Y* and *Z*. Also *R* = *W* = 2 for both *A* and *B*. A client reads the value of *A* and then writes it to *B*.

(i) At the time the client performs these operations, a partition separates servers *X* and *Y* from server Z. Describe the quora obtained and the operations that take place if the client can access servers *X* and *Y*.

(ii) Describe the quora obtained and the operations that take place if the client can access only server *Z*.

(iii) The partition is repaired and then another partition occurs so that *X* and *Z* are separated from *Y*. Describe the quora obtained and the operations that take place if the client can access servers X and Z.

14.19 Ans.

i) Partition separates X and Y from Z when all data items have version v0 say:

X Y Z

A= 100 (vo)	A= 100(vo)	A= 100(vo)
B= 100(vo)	B= 100(vo)	B= 100(vo)

A client reads the value of A and then writes it to B:

read quorum = 1+1 for A and B - client Reads A from X or Y

write quorum = 1+1 for B client Writes B at X and Y

ii) Client can access only server Z: read quorum = 1, so client cannot read, write quorum = 1 so client cannot write, therefore neither operation takes place.

iii) After the partition is repaired, the values of A and B at Z may be out of date, due to clients having written new values at servers X and Y. e.g. versions v1:

X	Y	Z
A= 200(v1)	A= 200(v1)	A= 100(vo)
B= 300(v1)	B= 300(v1)	B= 100(vo)

Then another partition occurs so that X and Z are separated from Y.

The client *Read* request causes an attempt to obtain a read quorum from X and Z. This notes that the versions (v0) at Z are out of date and then Z gets up-to-date versions of A and B from X.

Now the read quorum = 1+1 and the read operation can be done. Similarly the write operation can be done.

Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001.

Chapter 15 Solutions

15.1 Outline a system to support a distributed music rehearsal facility. Suggest suitable QoS requirements and a hardware and software configuration that might be used.

15.1 Ans..

This is a particularly demanding interactive distributed multimedia application. Konstantas *et al.* [1997] report that a round-trip delay of less than 50 ms is required for it. Clearly, video and sound should be tightly synchronized so that the musicians can use visual cues as well as audio ones. Bandwidths should be suitable for the cameras and audio inputs used, e.g. 1.5 Mbps for video streams and 44 kbps for audio streams. Loss rates should be low, but not necessarily zero.

The QoS requirements are much stricter than for conventional videoconferencing – music performance is impossible without strict synchronization. A software environment that includes QoS management with resource contracts is required. The operating systems and networks used should provide QoS guarantees. Few general-purpose OS's provide them at present. Dedicated real-time OS's are available but they are difficult to use for high-level application development.

Current technologies that should be suitable:

- ATM network.
- PC's with hardware for MPEG or MJPEG compression.
- Real-time OS with support for high-level software development, e.g. in CORBA or Java.

15.2 The Internet does not currently offer any resource reservation or quality of service management facilities. How do the existing Internet-based audio and video streaming applications achieve acceptable quality? What limitations do the solutions they adopt place on multimedia applications?

15.2 Ans..

There are two types of Internet-based applications:

a) Media delivery systems such as music streaming, Internet radio and TV applications.

b) Interactive applications such as Internet phone and video conferencing (NetMeeting, CuSeemMe).

For type (a), the main technique used is *traffic shaping*, and more specifically, buffering at the destination.Typically, the data is played out some 5–10 seconds after its delivery at the destination. This masks the uneven latency and delivery rate (jitter) of Internet protocols and masks the delays incurred in the network and transport layers of the Internet due to store-and-forward transmission and TCP's reliability mechanisms.

For type (b), the round trip delay must be kept below 100 ms so the above technique is ruled out. Instead, *stream adaptation* is used. Specifically, video is transmitted with high levels of compression and reduced frame rates. Audio requires less adaptation. UDP is generally used.

Overall, type (a) systems work reasonably well for audio and low-resolution video only. For type (b) the results are usually unsatisfactory unless the network routes and operating system priorities are explicitly managed.

15.3 Explain the distinctions between the three forms of synchronization (synchronous distributed state, media synchronization and external synchronization) that may be required in distributed multimedia applications. Suggest mechanisms by which each of them could be achieved, for example in a videoconferencing application.

15.3 Ans..

synchronous distributed state: All users should see the same application state. For example, the results of operation on controls for a video, such as start and pause should be synchronized, so that all users see the same frame. This can be done by associating the current state (sample number) of the active multimedia streams with each state-change message. This constitutes a form of logical vector timestamp.

media synchronization: Certain streams are closely coupled. E.g. the audio that accompanies a video stream. They should be synchronised using timestamps embedded in the media data.

external synchronization: This is really an instance of synchronous distributed state. The messages that update shared whiteboards and other shared objects should carry vector timestamps giving the states of media streams.

15.4 Outline the design of a QoS manager to enable desktop computers connected by an ATM network to support several concurrent multimedia applications. Define an API for your QoS manager, giving the main operations with their parameters and results.

15.4 Ans..

Each multimedia application requires a resource contract for each of its multimedia streams. Whenever a new stream is to be started, a request is made to the QoS manager specifying CPU resources, memory and network connections with their Flow Specs. The QoS manager performs an analysis similar to Figure 15.6 for each end-to-end stream. If several streams are required for a single application, there is a danger of deadlock – resources are allocated for some of the streams, but the needs of the remaining streams cannot be satisfied. When this happens, the QoS negotiation should abort and restart, but if the application is already running, this is impractical, so a negotiation takes place to reduce the resources of existing streams.

API:

QoSManager.QoSRequest(FlowID, FlowSpec) –> ResourceContract

The above is the basic interface to the QoS Manager. It reserves resources as specified in the *FlowSpec* and returns a corresponding *ResourceContract*.

A *FlowSpec* is a multi-valued object, similar to Figure 15.8.

A *ResourceContract* is a token that can be submitted to each of the resource handlers (CPU scheduler, memory manager, network driver, etc.).
Application.ScaleDownCallback(FlowID, FlowSpec) -> AcceptReject

The above is a callback from the QoS Manager to an application, requesting a change in the *FlowSpec* for a stream. The application can return a value indicating acceptance or rejection.

15.5 In order to specify the resource requirements of software components that process multimedia data, we need estimates of their processing loads. How should this information be obtained?

15.5 Ans..

The main issue is how to measure or otherwise evaluate the resource requirements (CPU, memory, network bandwidth, disk bandwidth) of the components that handle multimedia streams without a lot of manual testing. A test framework is required that will evaluate the resource utilization of a running component. But there is also a need for resource requirement models of the components – so that the requirements can be extrapolated to different application contexts and stream characteristics and different hardware environments (hardware performance parameters).

15.6 How does the Tiger system cope with a large number of clients all requesting the same movie at random times?

15.6 Ans..

If they arrive within a few seconds of each other, then they can be placed sufficiently close together in the schedule to take advantage of caching in the cubs, so a single disk access for a block can service several clients. If they are more widely spaced, then they are placed independently (in an empty slot near the disk holding he first block of the movie at the time each request is received). There will be no conflict for resources because different blocks of the movie are stored on different disks and cubs.

15.7 The Tiger schedule is potentially a large data structure that changes frequently, but each cub needs an up-to-date representation of the portions it is currently handling. Suggest a mechanism for the distribution of the schedule to the cubs.

15.7 Ans..

In the first implementation of Tiger the controller computer was responsible for maintaining an up-to-date version of the schedule and replicating it to all of the cubs. This does not scale well – the processing and communication loads at the controller grow linearly with the number of clients – and is likely to limit the scale of the service that Tiger can support. In a later implementation, the cubs were made collectively responsible for maintaining the schedule. Each cub holds a fragment of the schedule – just those slots that it will be playing processing in the near future. When slots have been processed they are updated to show the current viewer state and then they are passed to the next 'downstream' cub. Cubs retain some extra fragments for fault-tolerance purposes.

When the controller needs to modify the schedule – to delete or suspend an existing entry or to insert a viewer into an empty slot – it sends a request to the cub that is currently responsible for the relevant fragment of the schedule to make the update. The cub then uses the updated schedule fragment to fulfil its responsibilities and passes it to the next downstream cub.

15.8 When Tiger is operating with a failed disk or cub, secondary data blocks are used in place of missing primaries. Secondary blocks are *n* times smaller than primaries (where *n* is the decluster factor), how does the system accommodate this variability in block size?

15.8 Ans..

Whether they are large primary or smaller secondary blocks, they are always identified by a play sequence number. The cubs simply deliver the blocks to the clients in order via the ATM network. It is the clients' responsibility to assemble them in the correct sequence and then to extract the frames from the incoming sequence of blocks and play the frames according to the play schedule.

Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001

Chapter 16 Exercise Solutions

16.1 Explain in which respects DSM is suitable or unsuitable for client-server systems.

16.1 Ans.

DSM is unsuitable for client-server systems in that it is not conducive to heterogeneous working. Furthermore, for security we would need a shared region per client, which would be expensive.

DSM may be suitable for client-server systems in some application domains, e.g. where a set of clients share server responses.

16.2 Discuss whether message passing or DSM is preferable for fault-tolerant applications.

16.2 Ans.

Consider two processes executing at failure-independent computers. In a message passing system, if one process has a bug that leads it to send spurious messages, the other may protect itself to a certain extent by validating the messages it receives. If a process fails part-way through a multi-message operation, then transactional techniques can be used to ensure that data are left in a consistent state.

Now consider that the processes share memory, whether it is physically shared memory or page-based DSM. Then one of them may adversely affect the other if it fails, because now one process may update a shared variable without the knowledge of the other. For example, it could incorrectly update shared variables due to a bug. It could fail after starting but not completing an update to several variables.

If processes use middleware-based DSM, then it may have some protection against aberrant processes. For example, processes using the Linda programming primitives must explicitly request items (tuples) from the shared memory. They can validate these, just as a process may validate messages.

16.3 How would you deal with the problem of differing data representations for a middleware-based implementation of DSM on heterogeneous computers? How would you tackle the problem in a page-based implementation? Does your solution extend to pointers?

16.3 Ans.

The middleware calls can include marshalling and unmarshalling procedures. In a page-based implementation, pages would have to be marshalled and unmarshalled by the kernels that send and receive them. This implies maintaining a description of the layout and types of the data, in the DSM segment, which can be converted to and from the local representation.

A machine that takes a page fault needs to describe which page it needs in a way that is independent of the machine architecture. Different page sizes will create problems here, as will data items that straddle page boundaries, or items that straddle page boundaries when unmarshalled.

A solution would be to use a 'virtual page' as the unit of transfer, whose size is the maximum of the page sizes of all the architectures supported. Data items would be laid out so that the same set of items occurs in each virtual page for all architectures. Pointers can also be marshalled, as long as the kernels know the layout of data, and can express pointers as pointing to an object with a description of the form "Offset o in data item i", where o and i are expressed symbolically, rather than physically.

This activity implies huge overheads.

16.4 Why should we want to implement page-based DSM largely at user-level, and what is required to achieve this?

16.4 Ans.

A user-level DSM implementation facilitates application-specific memory (consistency) models and protocol options.

We require the kernel to export interfaces for (a) handling page faults from user level (in UNIX, as a signal) and (b) setting page protections from user level (see the UNIX *mmap* system call).

16.5 How would you implement a semaphore using a tuple space?

16.5 Ans.

We implement a semaphore with standard *wait* and *signal* operations; the style of semaphore that counts the number of blocked processes if its count is negative;

The implementation uses a tuple <"count", int> to maintain the semaphore's integer value; and tuples <"blocked", int> and <"unblocked", int> for each process that is blocked on the semaphore.

The *wait* operation is implemented as follows:

```
<"count", count> = ts.take(<"count", int>);
if (count > 0)
    count := count - 1;
else
{
    // Use ts.read(<"blocked", int>) to find the smallest b such that <"blocked", b> is not in ts
    ts.write(<"blocked", b>);
}
ts.write(<"count", count>);
ts.take(<"unblocked", b>;      // blocks until a corresponding tuple enters ts
```

The *signal* operation is implemented as follows:

```
<"count", count> = ts.take(<"count", int>);
if (there exists any b such that <"blocked", b> is in ts)
{
    ts.take(<"blocked", b>);
    ts.write("unblocked", b>); // unblocks a process
}
else
    count := count + 1;
ts.write("count", count>);
```

16.6 Is the memory underlying the following execution of two processes sequentially consistent (assuming that, initially, all variables are set to zero)?

P_1: $R(x)1$; $R(x)2$; $W(y)1$

P_2: $W(x)1$; $R(y)1$; $W(x)2$

16.6 Ans.

P1 reads the value of *x* to be 2 before setting *y* to be 1. But *P2* sets *x* to be 2 only after it has read *y* to be 1 (*y* was previously zero). Therefore these two executions are incompatible, and the memory is not sequentially consistent.

16.7 Using the *R*(), *W*() notation, give an example of an execution on a memory that is coherent but not sequentially consistent. Can a memory be sequentially consistent but not coherent?

16.7 Ans.

The execution of Exercise 16.6 is coherent – the reads and writes for each variable are consistent with both program orders – but it is not sequentially consistent, as we showed.

A sequentially consistent memory is consistent with program order; therefore the sub-sequences of operations on each individual variable are consistent with program order – the memory is coherent.

16.8 In write-update, show that sequential consistency could be broken if each update were to be made locally before asynchronously multicasting it to other replica managers, even though the multicast is totally ordered. Discuss whether an asynchronous multicast can be used to achieve sequential consistency. (Hint: consider whether to block subsequent operations.)

16.8 Ans.

P1: *W*(*x*)1; // *x* is updated immediately at *P1*, and multicast elsewhere

R(*x*)1;

R(*y*)0; // the multicast from *P2* has not arrived yet

P2: *W*(*y*)1;

R(*x*)0; // the multicast from *P1* has not arrived yet

R(*y*)1; // *y* was updated immediately and multicast to *P1*

P1 would say that *x* was updated before *y*; *P2* would say that *y* was updated before *x*. So the memory is not sequentially consistent when we use an asynchronous totally ordered multicast, and if we update the local value immediately.

If the multicast was synchronous (that is, a writer is blocked until the update has been delivered everywhere) and totally ordered, then it is easy to see that all processes would agree on a serialization of their updates (and therefore of all memory operations).

We could allow the totally-ordered multicast to be asynchronous, if we block any subsequent read operation until all outstanding updates made by the process have been assigned their total order (that is, the corresponding multicast messages have been delivered locally). This would allow writes to be pipelined, up to the next read.

16.9 Sequentially consistent memory can be implemented using a write-update protocol employing a synchronous, totally ordered multicast. Discuss what multicast ordering requirements would be necessary to implement coherent memory.

16.9 Ans.

One could implement a coherent memory by using a multicast that totally ordered writes to each individual location, but which did not order writes to different locations. For example, one could use different sequencers for different location. Updates for a location are sequenced (totally ordered) by the corresponding sequencer; but updates to different locations could arrive in different orders at different locations.

16.10 Explain why, under a write-update protocol, care is needed to propagate only those words within a data item that have been updated locally.

Devise an algorithm for representing the differences between a page and an updated version of it. Discuss the performance of this algorithm.

16.10 Ans.

Assume that two processes update different words within a shared page, and that the whole page is sent when delivering the update to other processes (that is, they falsely share the page). There is a danger that the unmodified words in one process's updated page will overwrite another's modifications to the falsely shared words.

To get around this problem, each process sends only the differences it has made to the page (see the discussion of Munin's write-shared data items on page 540). That way, updates to falsely shared words will be applied only to the affected words, and will not conflict. To implement this, it is necessary for each process to keep a copy of the page before it updates it.

A simple encoding of the differences between the page before and after it was modified is to create a series of tuples:

<*pageOffset*, *size*, *changedBytes*>

– which store in *changedBytes* a run of *size* bytes to change, starting at *pageOffset*. The list of tuples is created by comparing the pages byte-for-byte. Starting at the beginning, when a difference is encountered, we create a new tuple and record the byte's offset. We then copy the bytes from the modified page into *changedBytes*, until we reach a run of bytes that are the same in both pages and whose length is greater than a certain minimum value *M*. The encoding procedure then continues until the whole page has been encoded.

In judging such an algorithm, we are mindful of the processing time and the storage space taken up by the encoded differences. The minimum length *M* is chosen so that the processing time and storage space taken up by creating a new tuple is justified against the overhead of copying bytes that have not been modified. It is likely, for example, that it is cheaper to store and copy a run of four unmodified bytes than to create an extra tuple.

The reader may care to improve further upon this algorithm.

16.11 Explain why granularity is an important issue in DSM systems. Compare the issue of granularity between object-oriented and byte-oriented DSM systems, bearing in mind their implementations.

Why is granularity relevant to tuple spaces, which contain immutable data?

What is false sharing? Can it lead to incorrect executions?

16.11 Ans.

We refer the reader to the discussion of granularity on pages 648-649.

In a page-based implementation, the minimum granularity is fixed by the hardware. In Exercise 16.10 we have seen a way of updating smaller quantities of data than a page, but the expense of this technique makes it not generally applicable.

In middleware-based DSM, the granularity is up to the implementation, and may be as small as one byte. If a process updates one field in a data structure, then the implementation may choose to send just the field or the whole data structure in its update.

Consider a DSM such as Linda's Tuple Space, which consists of immutable data items. Suppose a process needs to update one element in a tuple containing a million-element array. Since tuples are immutable the process must extract the tuple, copying the whole array into its local variables; then it modifies the element; then it writes the new tuple back into Tuple Space. Far more data has been transferred than was needed for the modification. If the data had been stored as separate tuples, each containing one array element, then the update would have been much cheaper. On the other hand, a process wishing to access the whole array would have to make a million accesses to tuple space. Because of latency, this would be far more expensive than accessing the whole array in one tuple.

False sharing is described on pages 648-649. It does not of itself lead to incorrect executions, but it lowers the efficiency of a DSM system.

16.12 What are the implications of DSM for page replacement policies (that is, the choice of which page to purge from main memory in order to bring a new page in)?

16.12 Ans.

When a kernel wishes to replace a page belonging to a DSM segment, it can choose between pages that are read-only, pages that are read-only but which the kernel owns, and pages that the kernel has write access to (and has modified). Of these options, the least cost is associated with deleting the unowned read-only page (which the kernel can always obtain again if necessary); if the kernel deletes a read-only page that it owns, then it has lost a potential advantage if write access is soon required; and if it deletes the modified page then it must first transfer it elsewhere over the network or onto a local disk. So the kernel would prefer to delete pages in the order given. Of course it can discriminate between pages with equal status by choosing, for example, the least recently accessed.

16.13 Prove that Ivy's write-invalidate protocol guarantees sequential consistency.

16.13 Ans.

A memory is not sequentially consistent if (and only if) there is an execution in which two processes disagree about the order in which two or more updates were made. In a write-invalidate protocol, updates to any particular page are self-evidently serialised (only one process may update it at a time, and readers are meanwhile excluded). So we may assume that the updates are to different variables, residing in different pages.

Suppose the variables are *x* and *y*, with initial values (without loss of generality) 6 and 7. Let us suppose, further that *x* is incremented to 16, and *y* is incremented to 17. We suppose again, without loss of generality, that two processes' histories contain the following evidence of disorder:

P1: *R*/*W*(*x*)16; ...; *R*(*y*)7; .. // *x* was incremented first

P2: *R*/*W*(*y*)17; ...; *R*(*x*)6; ..// *y* was incremented first

Since the write-invalidate protocol was used, *P1* obtained access to *x*'s page, where it either read *x* or wrote *x* as 16. Subsequently, *P1* obtained access to *y*'s page, where it read *y* to be 7. For *P2* to have read or written *y* with the value 17, it must have obtained access to *y*'s page after *P1* finished with it. Later still, it obtained access to x's page and found that x had the value 6. By *reductio ad absurdum*, we may assume that our hypothesis was false: no such executions can exist, and the memory is sequentially consistent.

Lamport [1979] gives a more general argument, that a memory is sequentially consistent if the following two conditions apply:

R1: Each processor (process) issues memory requests in the order specified by its program.

R2: Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request on this queue.

Write-invalidation satisfies these conditions, where we substitute 'page' for 'memory module'.

The reader is invited to generalise the argument we have given, to obtain Lamport's result. You will need to construct a partial order between memory requests, based upon the order of issue by a single process, and the order of request servicing at the pages.

16.14 In Ivy's dynamic distributed manager algorithm, what steps are taken to minimize the number of lookups necessary to find a page?

16.14 Ans.

We refer the reader to the discussion on page 655.

16.15 Why is thrashing an important issue in DSM systems and what methods are available for dealing with it?

16.15 Ans.

Thrashing and the Mirage approach to it are discussed on page 657. The discussion on Munin (pages 661-663) describes how sharing annotations may be used to aid the DSM run-time in preventing thrashing. For example, migratory data items are always given read-and-write access, even if the first access is a read access. This is done in the expectation that a read is typically followed closely by a write. The producer-consumer annotation causes the run-time to use a write-update protocol, instead of an invalidation protocol. This is more appropriate, in that it avoids continually transferring access to the data item between the writer (producer) and readers (consumers).

16.16 Discuss how condition RC2 for release consistency could be relaxed. Hence distinguish between eager and lazy release consistency.

16.16 Ans.

Consider a process P that updates variables within a critical section. It may not be strictly necessary for another process to observe P's updates until it enters the critical section, whereas RC2 stipulates that the updates should occur on P's *release* operation. (Neither of these semantics is absolutely 'right' or 'wrong'; but the programmer has to be made aware of which is used.)

Implementations of eager release consistency propagate updates or invalidations upon the *release* operation; lazy ones propagate them when another process enters the critical section (issues an *acquire* operation).

16.17 A sensor process writes the current temperature into a variable t stored in a release-consistent DSM. Periodically, a monitor process reads t. Explain the need for synchronization to propagate the updates to t, even though none is otherwise needed at the application level. Which of these processes needs to perform synchronization operations?

16.17 Ans.

A release-consistent DSM implementation is free never to propagate an update in the absence of synchronisation. To guarantee that the monitor process sees new values of the variable t, synchronisation operations must be used. Using the definition of release consistency on p. 660, only the sensor process needs to issue *acquire* and *release* operations.

16.18 Show that the following history is not causally consistent:

P_1:	$W(a)0; W(a)1$
P_2:	$R(a)1; W(b)2$
P_3:	$R(b)2; R(a)0$

16.18 Ans.

In the following, we use subscripts to denote which process issued an operation.

$W_2(b)2$ writes-into $R_3(b)2$

Taking this together with P_3's execution order implies the following order:

$W_2(b)2$; $R_3(b)2$; $R_3(a)0$; $W_1(a)1$;

But $W_1(a)1$ is causally before $W_2(b)2$ – so no causally consistent serialisation exists.

16.19 What advantage can a DSM implementation obtain from knowing the association between data items and synchronization objects? What is the disadvantage of making the association explicit?

16.19 Ans.

The DSM implementation can use the association to determine which variables' updates/invalidations need to be propagated with a lock (there may be multiple critical sections, used for different sets of variables).

The disadvantage of making the association explicit is the work that this represents to the programmer (who, moreover, may make inaccurate associations).

Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001.

Chapter 17 Exercise Solutions

17.1 The Task Bag is an object that stores pairs of (key and value). A key is a string and a value is a sequence of bytes. Its interface provides the following remote methods:

pairOut: with two parameters through which the client specifies a *key* and a *value* to be stored.

pairIn: whose first parameter allows the client to specify the *key* of a pair to be removed from the Task Bag. The *value* in the pair is supplied to the client via a second parameter. If no matching pair is available, an exception is thrown.

readPair: is the same as *pairIn* except that the pair remains in the Task Bag.

Use CORBA IDL to define the interface of the Task Bag. Define an exception that can be thrown whenever any one of the operations cannot be carried out. Your exception should return an integer indicating the problem number and a string describing the problem. The Task Bag interface should define a single attribute giving the number of tasks in the bag.

17.1 Ans.

Note that sequences must be defined as *typedefs*. *Key* is also a *typedef* for convenience.

```
typedef string Key;
typedef sequence<octet> Value;
interface TaskBag {
readonly attribute long numberOfTasks;
exception TaskBagException { long no; string reason; };
void pairOut (in Key key, in Value value) raises (TaskBagException);
void pairIn (in Key key, out Value value) raises (TaskBagException);
void readPair (in Key key, out Value value) raises (TaskBagException);
};
```

17.2 Define an alternative signature for the methods *pairIn* and *readPair*, whose return value indicates when no matching pair is available. The return value should be defined as an enumerated type whose values can be *ok* and *wait*. Discuss the relative merits of the two alternative approaches. Which approach would you use to indicate an error such as a key that contains illegal characters?

17.2 Ans.

```
enum status { ok, wait};
status pairIn (in Key key, out Value value);
status readPair (in Key key, out Value value);
```

It is generally more complex for a programmer to deal with an exception that an ordinary return because exceptions break the normal flow of control. In this example, it is quite normal for the client calling *pairIn* to find that the server hasn't yet got a matching pair. Therefore an exception is not a good solution.

For the key containing illegal characters it is better to use an exception: it is an error (and presumably an unusual occurrence) and in addition, the exception can supply additional information about the error. The client must be supplied with sufficient information to recognise the problem.

17.3 Which of the methods in the Task Bag interface could have been defined as a *oneway* operation? Give a general rule regarding the parameters and exceptions of *oneway* methods. In what way does the meaning of the *oneway* keyword differ from the remainder of IDL?

17.3 Ans.

A *oneway* operation cannot have *out* or *inout* parameters, nor must it raise an exception because there is no reply message. No information can be sent back to the client. This rules out all of the Task Bag operations. The *pairOut* method might be made one way if its exception was not needed, for example if the server could guarantee to store every pair sent to it. However, *oneway* operations are generally implemented with *maybe* semantics, which is unacceptable in this case. Therefore the answer is *none*.

General rule. Return value *void*. No *out* or *inout* parameters, no user-defined exceptions.

The rest of IDL is for defining the interface of a remote object. But *oneway* is used to specify the required quality of delivery.

17.4 The IDL *union* type can be used for a parameter that will need to pass one of a small number of types. Use it to define the type of a parameter that is sometimes empty and sometimes has the type *Value*.

17.4 Ans.

```
union ValueOption switch (boolean){
case TRUE: Value value;
};
```

When the value of the tag is TRUE, a *Value* is passed. When it is FALSE, only the tag is passed.

17.5 In Figure 17.1 the type *All* was defined as a sequence of a fixed length. Redefine this as an array of the same length. Give some recommendations as to the choice between arrays and sequences in an IDL interface.

17.5 Ans.

```
typedef Shape All[100];
```

Recommendations

- if a fixed length structure then use an array.
- if variable length structure then use a sequence
- if you need to embed data within data, then use a sequence because one may be embedded in another
- if your data is sparse over its index, it may be better to use a sequence of <index, value> pairs.

17.6 The Task Bag is intended to be used by cooperating clients, some of which add pairs (describing tasks) and others remove them (and carry out the tasks described). When a client is informed that no matching pair is available, it cannot continue with its work until a pair becomes available. Define an appropriate callback interface for use in this situation.

17.6 Ans.

This callback can send the value required by a *readPair* or *pairIn* operation. Its method should not be a *oneway* as the client depends on receiving it to continue its work.

```
interface TaskBagCallback{
void data(in Value value);
}
```

17.7 Describe the necessary modifications to the Task Bag interface to allow callbacks to be used.

17.7 Ans.

The server must allow each client to register its interest in receiving callbacks and possibly also deregister. These operations may be added to the TaskBag interface or to a separate interface implemented by the server.

```
int register (in TaskBagCallback callback);
void deregister (in int callbackId);
```

See the discussion on callbacks in Chapter 5. (page 200)

17.8 Which of the parameters of the methods in the TaskBag interface are passed by value and which are passed by reference?

17.8 Ans.

In the original interface, all of the parameters are passed by value.
The parameter of *register* is passed by reference. (and that of *deregister* by value)

17.9 Use the Java IDL compiler to process the interface you defined in Exercise 17.1. Inspect the definition of the signatures for the methods *pairIn* and *readPair* in the generated Java equivalent of the IDL interface. Look also at the generated definition of the holder method for the value argument for the methods *pairIn* and *readPair*. Now give an example showing how the client will invoke the *pairIn* method, explaining how it will acquire the value returned via the second argument.

17.9 Ans.

The Java interface is:

```
public interface TaskBag extends org.omg.CORBA.Object {
    void pairOut(in Key p, in Value value);
    void pairIn(in Key p, out ValueHolder value);
    void readPair(in Key p, out ValueHolder value);
    int numberOfTasks();
}
```

The class *ValueHolder* has an instance variable

```
public Value value;
```

and a constructor

```
    public ValueHolder(Value __arg) {
value = __arg;
    }
```

The client must first get a remote object reference to the TaskBag (see Figure 17.5). probably via the naming service.

```
  ...
TaskBag taskBagRef = TaskBagHelper.narrow(taskBagRef.resolve(path));
Value aValue;
taskbagRef.pairIn("Some key", new ValueHolder(aValue));
```

The required value will be in the variable *aValue*.

17.10 Give an example to show how a Java client will access the attribute giving the number of tasks in the Task bag object. In what respects does an attribute differ from an instance variable of an object?

17.10 Ans.

Assume the IDL attribute was called *numberOfTasks* as in the answer to Exercise 17.1. Then the client uses the method *numberOfTasks* to access this attribute. e.g.

```
taskbagRef.numberOfTasks();
```

Attributes indicate methods that a client can invoke in a CORBA object. They do not allow the client to make any assumption about the storage used in the CORBA object, whereas an instance variable declares the type of a variable. An attribute may be implemented as a variable or it may be a method that calculates the result. Either way, the client invokes a method and the server implements it.

17.11 Explain why the interfaces to remote objects in general and CORBA objects in particular do not provide constructors. Explain how CORBA objects can be created in the absence of constructors.

17.11 Ans.

If clients were allowed to request a server to create instances of a given interface, the server would need to provide its implementation. It is more effective for a server to provide an implementation and then offer its interface.

CORBA objects can be created within the server:

1. the server (e.g. in the *main* method) creates an instance of the implementation class and then exports a remote object reference for accessing its methods.
2. A factory method (in another CORBA object) creates an instance of the implementation class and then exports a remote object reference for accessing its methods.

17.12 Redefine the Task Bag interface from Exercise 17.1 in IDL so that it makes use of a *struct* to represent a *Pair*, which consists of a *Key* and a *Value*. Note that there is no need to use a *typedef* to define a *struct*.

17.12 Ans.

```
typedef string Key;
typedef sequence<octet> Value;
struct Pair {
      Key key;
      Value value;
  };
interface TaskBag {
readonly attribute int numberOfTasks;
exception TaskBagException { int no; string reason; };
void pairOut (in Pair) raises (TaskBagException);
      // pairIn and readPair might use the pair, or could be left unchanged
};
```

17.13 Discuss the functions of the implementation repository from the point of view of scalability and fault tolerance.

17.13 Ans.

The implementation repository is used by clients to locate objects and activate implementations. A remote object reference contains the address of the IR that manages its implementation.

An IR is shared by clients and servers within some location domain.

To improve scalability, several IRs can be deployed, with the implementations partitioned between them (the object references locate the appropriate IR). Clients can avoid unnecessary requests to an IR if they parse the remote object reference to discover whether they are addressing a request to an object implementation already located.

From the fault tolerance point of view, an IR is a single point of failure. Information in IRs can be replicated - note that a remote object reference can contain the addresses of several IRs. Clients can try them in turn if one is unobtainable. The same scheme can be used to improve availability.

17.14 To what extent may CORBA objects be migrated from one server to another?

17.14 Ans.

CORBA persistent IORs contain the address of the IR used by a group of servers. That IR can locate and activate CORBA objects within any one of those servers. Therefore, it will still be able deal with CORBA objects that migrate from one server in the group to another. But the object adapter name is the key for the implementation in the IR. Therefore all of the objects in one server must move together to another server. This could be modified by allowing groups of objects within each server to have separate object adapters and to be listed under different object adapter names in the IR.

Also, CORBA objects cannot move to a server that uses a different IR. It would be possible for servers to move and to register with a new IR, but then there are issues related to finding it from the old location domain, which would need to have forwarding information.

17.15 Discuss the benefits and drawbacks of the two-part names or *NameComponents* in the CORBA naming service.

17.15 Ans.

A *NameComponent* contains a *name* and a *kind*. The *name* field is the name by which the component is labelled (like a name component in a file or DNS name).

The *kind* field is intended for describing the name. It is not clear that this has any useful function. The use of two parts for each name complicates the programming interface. Without it, each name component could be a simple string.

17.16 Give an algorithm that describes how a multipart name is resolved in the CORBA naming service. A client program needs to resolve a multipart name with components "A", "B" and "C", relative to an initial naming context. How would it specify the arguments for the *resolve* operation in the naming service?

17.16 Ans.

Given a multipart name, *mn*[] with components *mn*[0], *mn*[1], *mn*[2], ...*mn*[N], starting in context C. Note *mn* is of type *Name* (a sequence of *NameComponents*).

The method *resolve* returns the *RemoteObjectRef* of the object (or context) in the context matched by the name, *mn*[]. There are several ways in which *resolve* may fail:

i) If the name *mn* is longer than the path in the naming graph, then when *mn[i]* (*i*<*n*) is looked up, we get an object and throw a *NameNotFound* exception with reason *NonContext*. Assume that *length*(*mn*) > 0 to start with.

ii) If the part *nm[i*] does not match a name in the current context, throw a *NameNotFound* exception with reason *missingName*.

iii) if a *RemoteObjectRef* turns out not to refer to an object or context at all throw a *NameNotFound* exception with reason *invalidRef*.

A more sophisticated answer might return the remainder of the path with the exception. The algorithm can be defined as follows:

```
RemoteObjectRef resolve(C, Name mn[]) {
  RemoteObjectRef ref = lookInContext(C, first(mn));
  if(ref==null) throw NameNotFoundException (missingName);
  else if(length(mn) == 1) return ref;
  else if(type(ref) ==  object) throw NameNotFoundException(NonContext)
  else  resolve( C‘, tail(mn));
}
```

where *lookInContext(C, name)* looks up the name component, *name* in context *C* and returns a *RemoteObjectRef* of an object or a context (or null if the name is not found).

```
Name name;

NamingContext C =  resolve_initial_references("NameService");
name[0] = new NameComponent("A","");
name[1] = new NameComponent("B","");
name[2] = new NameComponent("C","");
C.resolve(name);
```

17.17 A virtual enterprise consists of a collection of companies who are cooperating with one another to carry out a particular project. Each company wishes to provide the others with access to only those of its CORBA objects relevant to the project. Describe an appropriate way for the group to federate their CORBA Naming Services.

17.17 Ans.

The solution should suggest that each company manages its own naming graph and decides which portion of it should be made available for sharing by the other companies in the virtual enterprise. Each company provides the others with a remote object reference to the shared portion of their naming graph (a remote object reference may be converted to a string and passed to the others e.g. by email). Each the companies provides names that link the remote object reference to the set of CORBA objects held by the others via its naming graph. The companies might agree on a common naming scheme.

17.18 Discuss how to use directly connected suppliers and consumers of the CORBA event service in the context of the shared whiteboard application. The *PushConsumer* and *PushSupplier* interfaces are defined in IDL as follows:

```
interface PushConsumer {
void push(in any data) raises (Disconnected);
void disconnect_push_consumer();
}

interface PushSupplier {
void disconnect_push_supplier();
}
```

Either the supplier or the consumer may decide to terminate the event communication by calling *disconnect_push_supplier()* or *disconnect_push_consumer()* respectively.

17.18 Ans.

The shared whiteboard application is described on pages 195-6.

Choose whether to use *push* or *pull* model. We use the *push* model in which the supplier (object of interest) initiates the transfer of notifications to subscribers (consumers). Since we have only one type of event in this application (a new *GraphicalObject* has been added at the server), we use generic events.

In the push model, the consumer (the whiteboard client) must implement the *PushConsumer* interface. It could do this by providing an implementation of a servant with the following interface (including the *push* method).

```
interface WhiteboardConsumer: PushConsumer{
    push(in any data) raises (Disconnected);
};
```

The implementation will make *push* do whatever is needed - to get the latest graphical objects from the server. The client creates an instance of *WhiteboardConsumer* and informs the server about it.

The supplier (our server) provides an operation allowing clients to inform it that they are consumers. For example:

```
e.g. addConsumer(WhiteboardConsumer consumer)
    // add this consumer to a vector of WhiteboardConsumers
```

Whenever a new graphical object is created, the server calls the push operation in order to send an event to the client, passing the event data (version number) as argument. For example (*Any* is Java's representation of CORBA any):

```
  Any a;
int version;
  // assign the int version to the Any a
  // repeat for all consumers in the vector
consumer.push(a);
```

17.19 Describe how to interpose an Event Channel between the supplier and the consumers in your solution to 17.13. An event channel has the following IDL interface:

```
interface EventChannel {
ConsumerAdmin for_consumers();
SupplierAdmin for_suppliers();
};
```

where the interfaces *SupplierAdmin* and *ConsumerAdmin,* which allow the supplier and the consumer to get proxies are defined in IDL as follows:

```
interface SupplierAdmin {
ProxyPushConsumer obtain_push_consumer();
---
};

interface ConsumerAdmin {
ProxyPushSupplier obtain_push_supplier();
---
};
```

The interface for the proxy consumer and proxy supplier are defined in IDL as follows:

```
interface ProxyPushConsumer : PushConsumer{
void connect_push_supplier (in PushSupplier supplier)
    raises (AlreadyConnected);
};

interface ProxyPushSupplier : PushSupplier{
void connect_push_consumer (in PushConsumer consumer)
    raises (AlreadyConnected);
};
```

What advantage is gained by the use of the event channel?

17.19 Ans.

We need to assume something to create an event channel. (E.g. the specification of the Event service has an example in the Appendix showing the creation of an event channel from a factory.

```
EventChannelFactory ecf = ...
ecf.create_eventchannel();
```

The event channel can be created by the whiteboard server and registered with the Naming Service so that clients can get a remote reference to it. More simply, the clients can get a remote reference to the event channel by means of an RMI method in the server interface.

The event channel will carry out the work of sending each event to all of the consumers.

We require one proxy push consumer that receives all notifications from the whiteboard server. The event channel forwards it to all of the proxy push suppliers - one for each client.

The whiteboard server (as an instance of *PushSupplier*) gets a proxy consumer from the event channel and the connects to it by providing its own object reference.

```
SupplierAdmin sadmin = ec.for_suppliers();
ProxyPushConsumer ppc = sadmin.obtain_push_consumer();
```

It connects to the proxy consumer by providing its own object reference

```
ppc.connect_push_supplier(this)
```

The supplier pushes data to the event channel, which pushes it on to the consumers. It may supply events to one or more consumers.

The *newShape* method of *shapeListServant* (Figure 17.3) will use the *push* operation to inform the event channel, each time a new *GraphicalObject* is added. e.g.

```
ppc.push(version);
```

As before, each client implements a CORBA object with a *PushConsumer* interface. It then gets a proxy supplier from the event channel and then connects to it by providing its own object reference.

ConsumerAdmin cadmin = ec.for_consumers();
ProxyPushSupplier pps = cadmin.*obtain_push_supplier();*

It should connect to the proxy supplier by providing its own object reference
pps. connect_push_supplier(this);

As before, the client receives notifications via the *push* method in its *PushConsumer* interface.

The advantage of using an event channel is that the whiteboard server does not need to know how many clients are connected nor to ensure that notifications are sent to all of them, using the appropriate reliability characteristics.

Distributed Systems: Concepts and Design

Edition 3

By George Coulouris, Jean Dollimore and Tim Kindberg
Addison-Wesley, ©Pearson Education 2001

Chapter 18 Exercise Solutions

18.1 How does a kernel designed for multiprocessor operation differ from one intended to operate only on single-processor computers?

18.1 Ans.

Issues arise due to the need to coordinate and synchronize the kernels' accesses to shared data structures. The kernel of a uniprocessor computer achieves mutual exclusion between threads by manipulating the hardware interrupt mask. Kernels on a multiprocessor must instead use a shared-memory-based mechanism such as spinlocks. Moreover, care must be taken in enforcing caching protocols so as to produce a memory model with the required consistency guarantees.

18.2 Define (binary-level) operating system emulation. Why is it desirable and, given that it is desirable, why is it not routinely done?

18.2 Ans.

A binary-level OS emulation executes binary files that run on the native OS, without adaptation and with exactly the same results (functions and error behaviour). By contrast, a source-level emulation requires recompilation of the program and may behave differently because some 'system' functions are actually emulated in libraries.

Binary-level emulation is desirable so that users can retain their software investment while migrating to a system with superior functionality. Technically, achieving binary-level emulation is complex. The producers of the emulated OS may change that native OS's functionality, leaving the emulation behind.

18.3 Explain why the contents of Mach messages are typed.

18.3 Ans.

The contents of Mach messages are type-tagged because:

a) The designers wanted to support marshalling of simple data types as a system service.

b) The kernel monitors the movement of port rights, in order to monitor communication connectivity, to transmit port location hints for send rights and to implement port migration for receive rights. So the rights must be tagged in messages.

c) Messages may contain pointers to out-of-line data, which the kernel must be aware of.

18.4 Discuss whether the Mach kernel's ability to monitor the number of send rights for a particular port should be extended to the network.

18.4 Ans.

The Mach network server does in fact implement monitoring of the number of send rights to a port over the network. Clients do not always terminate gracefully, and so this feature is useful for servers that need to garbage-collect ports when no clients exist that can send to them. However, this convenience is gained at the expense of considerable management overhead. It would seem preferable to be able to switch off this feature if there is no strong need for it – perhaps on a port-by-port basis.

18.5 Why does Mach provide port sets, when it also provides threads?

18.5 Ans.

Kernel-level Mach threads are an expensive and limited resource. Some tasks acquire thousands of ports. Instead of utilising a thousand threads, a small number of threads can read messages from a port set containing all the ports, and dispatch message processing according to which port each message arrives at.

18.6 Why does Mach provide only a single communication system call, *mach_msg*? How is it used by clients and by servers?

18.6 Ans.

The single call replaces two system-calls per client-server interaction with one (saving two domain transitions). The client call sends a message and receives a reply; the server call sends the reply to the previous request and receives the next request.

18.7 What is the difference between a network port and a (local) port?

18.7 Ans.

A 'network port' is an abstraction of a global port – one to which threads on multiple machines can send messages.

The network port is implemented by a local port and a globally-unique identifier, maintained by a network server at each machine that forwards network messages to the local port and participates in locating the port.

18.8 A server in Mach manages many *thingumajig* resources.

(i) Discuss the advantages and disadvantages of associating:

a) a single port with all the thingumajigs;

b) a single port per thingumajig;

c) a port per client.

(ii) A client supplies a thingumajig identifier to the server, which replies with a port right. What type of port right should the server send back to the client? Explain why the server's identifier for the port right and that of the client may differ.

(iii) A thingumajig client resides at a different computer from the server. Explain in detail how the client comes to possess a port right that enables it to communicate with the server, even though the Mach kernel can only transmit port rights between local tasks.

(iv) Explain the sequence of communication events that take place under Mach when the client sends a message requesting an operation upon a thingumajig, assuming again that client and server reside at different computers.

18.8 Ans.

i) Associating a single port with all the thingumajigs minimises the number of ports. But it makes it difficult to relocate some but not all the thingumajigs with another server at run-time. Individual resource relocation can be desirable for performance reasons.

Associating a single port with each thingumajig requires potentially large numbers of ports, but it enables any resource to be relocated with another server at run time, independently of the others. It also makes it easy to vary the number and priority of threads associated with each resource.

Associating a port with each client would make it difficult to move resources to other servers. The only advantage of this scheme is that a server can take for granted the identity of the principal involved when it processes the requests arriving at a particular port (assuming that Mach ensures the integrity of the port).

ii) The server should send a send right for the server's port. The identifiers may differ, because each uses a local name, which is allocated from its own local name space and invalid beyond it.

iii) Assume that the server at computer *S* sends a send right back to a client at computer *C*. The message first arrives at the local network server at *S*, which examines the message and notices the send right *s* within it. If the network server has not received this right before, it generates a network port number *n* and sets up a table entry relating *n* to *s*. It forwards the message to *C*'s network server, enclosing *n* and the location of the network port, namely *S*. When *C*'s network server receives this message, it examines *n* and finds that it has no entry for it. It creates a port with receive rights *r* and send rights *t*, and creates a table entry relating *r* to *n* and *S*. It forwards the message to the local client, enclosing the send right *t*.

iv) When the network server at *C* later receives a message using r, it knows to forward the message quoting the network port *n* to the network server at *S*. The network server at *S*, in turn, knows that messages addressed to *n* should be forwarded using its stored send right *s*, which refers to the server's port.

18.9 A Mach task on machine *A* sends a message to a task on a different machine *B*. How many domain transitions occur, and how many times are the message contents copied if the message is page-aligned?

18.9 Ans.

Domain transistions are: task A → kernel A → Network server A, Network server B → kernel B → task B (4).

Using copy-on-write with no page faults (i.e. if the sending task does not write on the message before tranmission is complete), the message is copied four times: once from the Network server's address space to kernel buffers, once from kernel buffers to the network interface, and *vice versa* at the receiving end.

18.10 Design a protocol to achieve migration transparency when ports are migrated.

18.10 Ans.

Page 712 outlines some of the ingredients of such a protocol. The mechanisms available to us are (a) forwarding hints at nodes from which the port has migrated; (b) multicast, used to locate a port using its unique identifier. Sub-protocols are:

(1) a port-location and sender-rebinding protocol that operates when a sender attempts to send to a port that has migrated; this protocol must include provision to garbage-collect forwarding hints and it must cope with inaccurate hints and hints that point to crashed machines;

(2) a protocol that moves the message queue before other messages are appended at the new site.

We leave the details to the reader. Note that an implementation of such a protocol based on TCP/UDP sockets rather than Mach ports could be used in an implementation of object migration.

18.11 How can a device driver such as a network driver operate at user level?

18.11 Ans.

If the device registers are memory-mapped, the process must be allowed to map the registers into its user-level address space. If the registers are accessible only by special instructions, then the process needs to be allowed to run with the processor in supervisor mode.

18.12 Explain two types of region sharing that Mach uses when emulating the UNIX *fork*() system call, assuming that the child executes at the same computer. A child process may again call *fork*(). Explain how this gives rise to an implementation issue, and suggest how to solve it.

18.12 Ans.

Two types of region sharing that Mach uses when emulating the UNIX *fork*() system call: (1) Physical sharing of pages in read-only shared regions, e.g. program text, libraries. (2) Copy-on-write sharing of copied regions, e.g. stack and heap.

If the child again calls *fork*(), then its copy-on-write page table entries need to refer to the correct page: that of its parent or its grandparent? If the grandparent modifies a shared page, then both parent and child's page table entries should be updated. Keeping track of dependencies despite arbitrary recursive calls to *fork*() is an implementation problem.

A doubly-linked tree structure of page-nodes representing the 'logical copy' relationship can be used to keep track of dependencies. If a process modifies a page, then the page is physically copied and the (bidirectional) link with the parent node is broken. Links from the descendant nodes are replaced by links to the parent node.

18.13 (i) Is it necessary that a received message's address range is chosen by the kernel when copy-on-write is used?

(ii) Is copy-on-write of use for sending messages to remote destinations in Mach?

(iii) A task sends a 16 kilobyte message asynchronously to a local task on a 10 MIPS, 32-bit machine with an 8 kilobyte page size. Compare the costs of (1) simply copying the message data (without using copy-on-write) (2) best-case copy-on-write and (3) worst-case copy-on-write. You can assume that:

- creating an empty region of size 16 kilobytes takes 1000 instructions;
- handling a page fault and allocating a new page in the region takes 100 instructions.

18.13 Ans.

(i) In general, no. The copy-on-write mechanism is independent of the logical addresses in use. A process could therefore receive a message into a pre-specified region.

(ii) No: it is useful only for local memory copying. If a message is transferred remotely, then it must be copied to the remote machine anyway.

(iii) (1) Copying: 2 instructions/loop, 4 bytes at a time => (10/2) * 4 = 20 Mbyte/sec = 20K/ms. Sent asynchronously => receiver not waiting => 2 copies => 2 * 16/20 = 1.6 ms.

(2) In the best case, we create a new region but there are no page faults: 1000/10MIPS = 0.1 ms (1000 instructions).

3) In the worst case, we create a new region and there are two page faults and two page copies: 0.1 ms + 0.02ms + 16/20 (copy) = 0.92 ms.

Thus copy-on-write always wins in this example.

18.14 Summarize the arguments for providing external pagers.

18.14 Ans.

The central argument is to allow the kernel to support a variety of (distributed shared) memory abstractions, including files mapped from remote servers. See pp. 716-719.

18.15 A file is opened and mapped at the same time by two tasks residing at machines without shared physical memory. Discuss the problem of consistency this raises. Design a protocol using Mach external pager messages which ensures sequential consistency for the file contents (see Chapter 16).

18.15 Ans.

Suppose that several tasks residing at different machines map a common file. If the file is mapped read-only in every region used to access it, then there is no consistency problem and requests for file pages can be satisfied immediately. If, however, at least one task maps the file for writing, then the external pager (that is, the file server) has to implement a protocol to ensure that tasks do not read inconsistent versions of the same page. If no special action is taken, a page can be modified at one computer while stale versions of the page exist in memory cache objects at other computers. Note that no consistency problem arises between tasks at a single kernel sharing a mapped memory object. The kernel keeps only one memory cache object in this case, and the associated frames are physically shared between the tasks.

In order to maintain consistency between a number of memory cache objects (managed by different kernels), the external pager controls, for each page of the shared file, whether or not the page is physically present at client machines, and if it is present, whether tasks there have read-only or read-write access to the page. It uses the single-writer/multiple-reader sharing scheme described in Chapter 16. To control access, the *memory_object_lock_request* message can be used to set permissions (as well as or instead of directing the kernel to write modified data). The *memory_object_data_provided* message contains a parameter to specify read-only or read-write permissions.

Enabling access to a page. A task takes a page-fault either when a) a non-resident page is required, or b) the page is resident but read-only, and an upgrade to write access is required. To handle the page-fault, the kernel respectively sends to the external pager (file server) either a *memory_object_data_request* message to request a non-resident page, or a *memory_object_data_unlock* message to request an upgrade to write access. To satisfy the request, the external pager may first downgrade other kernels' access to the page as described in the next paragraph. The external pager responds to the faulting machine with a *memory_object_data_provided* message to give the page data and set the access to it, or a *memory_object_lock_request* message to upgrade the access on the page to write access, as appropriate.

Downgrading access to a page. The external pager can issue a *memory_object_lock_request* message with appropriate parameters to fetch a page from a kernel if it has been modified, and at the same time to downgrade its access to read-only or none. Access becomes read-only if a page is required elsewhere for reading; it becomes null if the page is required elsewhere for writing.