Distributed Systems: Concepts and Design

Edition 5

By George Coulouris, Jean Dollimore and Tim Kindberg and Gordon Blair
Addison-Wesley ©Pearson Education 2012

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 Consider the implementation strategies for massively multiplayer online games as discussed in Section 1.2.2. In particular, what advantages do you see in adopting a single server approach for representing the state of the multiplayer game? What problems can you identify and how might they be resolved?

1.3 Ans.

The advantages of having a single server maintain a representation of the game are that: (1) there is a single copy and hence no need to maintain consistency of multiple copies; and (2) clients have a single place to go to discover this state. There may also be advantages to having a global view of the entire systems state.

The potential problems are that this single server may fail and may also become a bottleneck affecting the performance and scalability of the approach. To handle failure, it would be necessary to introduce replication, which in turn would require a solution to maintaining consistency across replicas. There are a number of solutions to improving performance and scalability including running the server on a cluster architecture as described in Section 1.2.2, or again using replication and load balancing within the distributed environment. Alternatively, a peer-to-peer solution can be adopted. These techniques are covered throughout the book.

1.4 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?

1.4 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.

1.5 Compare and contrast cloud computing with more traditional client-server computing? What is novel about cloud computing as a concept?

1.5 Ans.

In this chapter, cloud computing is defined in terms of: (1) supporting Internet-based services (whether application, storage or other computing-based services), where everything is a service; and (2) dispensing with local data storage or application software. (1) is completely consistent with client-server computing and indeed

client-server concepts support the implementation of cloud computing. (2) highlights one of the key elements of cloud computing in moving to a world where you can dispense with local services. This level of ambition may or may not be there in client-server computing. As a final comment, cloud computing promotes a view of computing as a utility and this is linked to often novel business models whereby services can be rented rather than being owned, leading to a more flexible and elastic approach to service provision and acquisition. This is a key distinction in cloud computing and represents the key novelty in cloud computing.

To summarise, cloud computing is partially a technical innovation in terms of the level of ambition, but largely a business innovation in terms of viewing computing services as a utility.

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

1.6 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.

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.

1.7 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.

1.14 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.

1.14 Ans.

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.

1.15 Give an example of an HTTP URL.

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

To what extent is a URL location transparent?

1.15 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.

Distributed Systems: Concepts and Design

Edition 5

By George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair
Addison-Wesley ©Pearson Education 2012

Chapter 2 Exercise Solutions

2.1 Provide three specific and contrasting examples of the increasing levels of heterogeneity experienced in contemporary distributed systems as defined in Section 2.2.

2.1 Ans.

Heterogeneity exists in many areas of a contemporary distributed system including in the areas of hardware, operating systems, networks and programming languages. We look at the first three as examples:

- In terms of hardware, distributed systems are increasingly heterogeneous featuring (typically Intel-based) PCs, smart phones, resource-limited sensor nodes, and resource-rich cluster computers or multi-core processors.
- In terms of operating systems, a distributed system may include computers running Windows, MAC OS, various flavours of Unix, and also more specialist operating systems for smart phones or sensor nodes.
- In terms of networks, the Internet is also increasingly heterogeneous embracing wireless technologies and ad hoc styles of networking.

2.2 What problems do you foresee in the direct coupling between communicating entities that is implicit in remote invocation approaches? Consequently, what advantages do you anticipate from a level of decoupling as offered by space and time uncoupling? Note: you might want to revisit this answer after reading Chapters 5 and 6.

2.2 Ans.

The client is intrinsically bound to the server and vice versa and this is inflexible in terms of dealing with failure, for example if the server fails and a backup server takes over managing requests. More generally, this level of coupling makes it hard to deal with change.

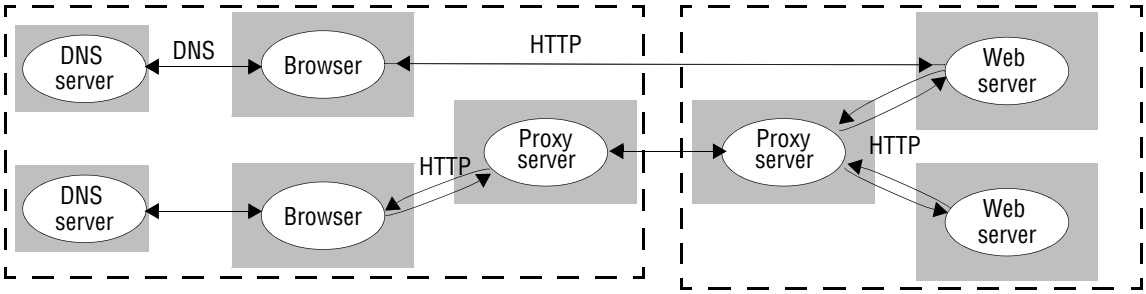
Clients and servers must exist at the same time and hence it is not possible to operate in more volatile environments when either party may be unavailable, for example disconnected in the case of a mobile node.

The benefits of space uncoupling is in providing more degrees of freedom in dealing with change, for example if a new server starts dealing with requests.

The benefit of time uncoupling is in allowing entities to communicate when entities may come and go.

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

Web:

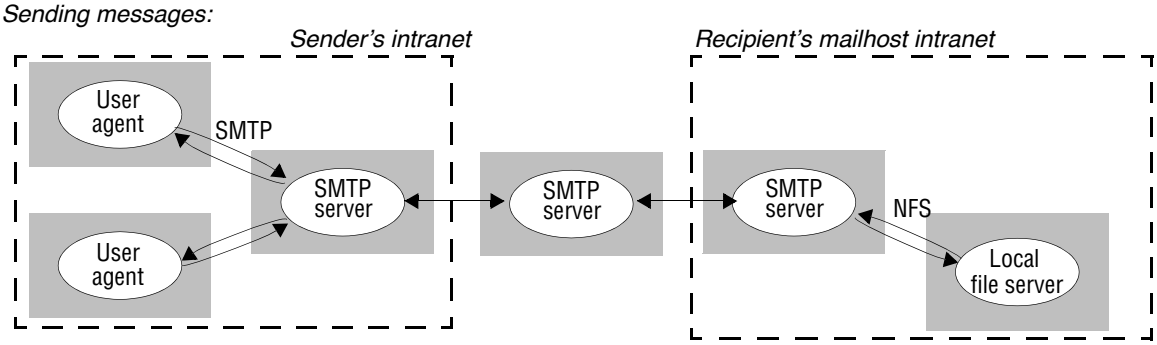


Browsers are clients of Domain Name Servers (DNS) and web servers (HTTP). Some intranets are configured to interpose a Proxy server. Proxy servers fulfil several purposes – when they are located at the same site as the client, they reduce network delays and network traffic. When they are at the same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

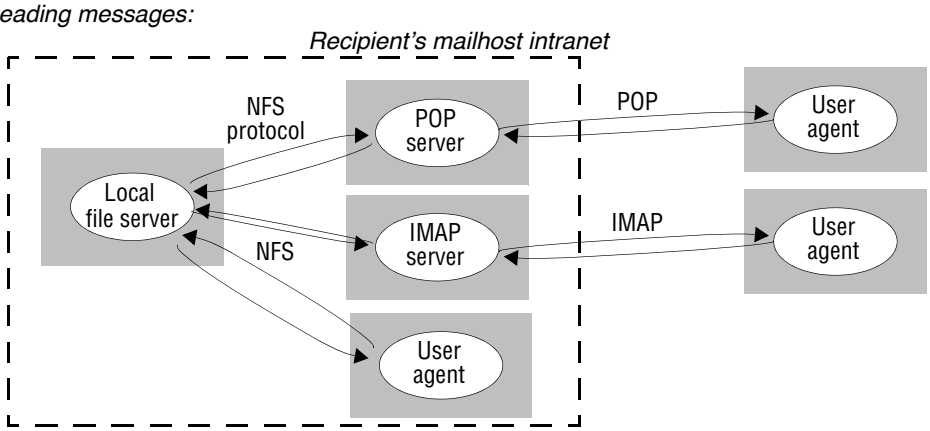
N.B. DNS servers are also involved in all of the application architectures described below, but they ore omitted from the discussion for clarity.

Email:

Sending messages:



Reading messages:



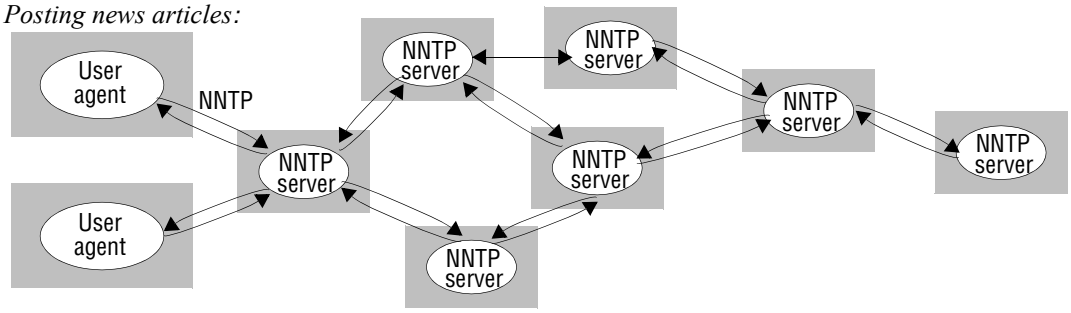
Sending messages: User Agent (the user's mail composing program) is a client of a local SMTP server and passes each outgoing message to the SMTP server for delivery. The local SMTP server uses mail routing tables to determine a route for each message and then forwards the message to the next SMTP server 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. (Examples 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

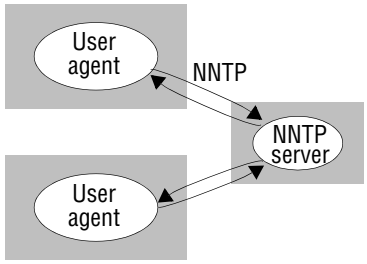
from those servers for presentation to the user. POP and IMAP are protocols specifically designed to support mail access over wide areas and slow network connections, so a user can continue to access her home mailbox while travelling.

Netnews:

Posting news articles:



Browsing/reading articles:



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.

2.4 For the applications discussed in Exercise 2.1 what placement strategies are employed in implementing the associated services?.

2.4 Ans.

We first consider the mapping to multiple machines (covering partitioning and replication strategies):

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 15). 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.

In terms of caching (and proxies), 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.

The web also features significant use of mobile code, for example through applets where code is downloaded and run on the browser machine.

2.5 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?

2.5 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.

2.6 The host computers used in peer-to-peer systems are often simply desktop computers in users' offices or homes. What are the implications of this for the availability and security of any shared data objects that they hold and to what extent can any weaknesses be overcome through the use of replication?

2.6 Ans.

Problems:

- people often turn their desktop computers off when not using them. Even if on most of the time, they will be off when user is away for an extended time or the computer is being moved.
- the owners of participating computers are unlikely to be known to other participants, so their trustworthiness is unknown. With current hardware and operating systems the owner of a computer has total control over the data on it and may change it or delete it at will.
- network connections to the peer computers are exposed to attack (including denial of service).

The importance of these problems depends on the application. For the music downloading that was the original driving force for peer-to-peer it isn't very important. Users can wait until the relevant host is running to access a particular piece of music. There is little motivation for users to tamper with the music. But for more conventional applications such as file storage availability and integrity are all-important.

. Solutions:

Replication:

- if data replicas are sufficiently widespread and numerous, the probability that all are unavailable simultaneously can be reduced the a negligible level.
- one method for ensuring the integrity of data objects stored at multiple hosts (against tampering or accidental error) is to perform an algorithm to establish a consensus about the value of the data (e.g. by exchanging hashes of the object's value and comparing them). This is discussed in Chapter 15. But there is a simpler solution for objects whose value doesn't change (e.g. media files such as music, photographs, radio broadcasts or films).

Secure hash identifiers:

- The object's identifier is derived from its hash code. The identifier is used to address the object. When the object is received by a client, the hash code can be checked for correspondence with the identifier. The hash algorithms used must obey the properties required of a secure hash algorithm as described in Chapter 7.

2.7 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.

2.7 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

---

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

2.8 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)

2.9 Consider a hypothetical car hire company and sketch out a three-tier solution to the provision of their underlying distributed car hire service. Use this to illustrate the benefits and drawbacks of a three-tier solution considering issues such as performance, scalability, dealing with failure and also maintaining the software over time.

2.9 Ans.

A three-tier solution might consist of:

- a web-based front-end offering a user interface for the car hire service (the presentation logic);
- a middle tier supporting the core operations associated with the car hire business including locating a particular make and model, checking availability and pricing, getting a quote and purchasing a particular car (the application logic);
- a database which stores all the persistent data associated with the stock (the data logic).

In terms of performance, this approach introduces extra latency in that requests must go from the web-based interface to the middle tier and then to the database (and back). However, processing load is also spread over three machines (especially over the middle tier and the database) and this may help with performance. For this latter reason, the three-tier solution may scale better. This may be enhanced though by other, complementary placement strategies including replication.

In terms of failure, there is an extra element involved and this increases the probability of a failure occurring in the system. Equally, failures are more difficult to deal with, for example if the middle tier is available and the database fails.

The three-tier approach is much better for evolution because of the intrinsic separation of concerns. For example, the middle tier only contains application logic and this should therefore be easier to update and maintain.

2.10 Provide a concrete example of the dilemma offered by Saltzer's end-to-end argument in the context of the provision of middleware support for distributed applications (you may want to focus on one aspect of providing dependable distributed systems, for example related to fault tolerance or security).

Saltzer's end-to-end argument states that communication-related functions can only be completely and reliably implemented with the knowledge and help of the application and therefore providing that function as a feature of the communication system itself (or middleware) is not always sensible. One concrete example is in secure communication. Assume that in a given system, the communication subsystem provides encrypted communication. This is helpful but insufficient. For example, the pathway from the network to the application software may be compromised and this is unprotected. This solution also does not deal with malicious participants in the exchange of data.

Consider also the reliable exchange of data implemented by introducing checksum protection on individual hops in the network. Again, this is insufficient as data may be corrupted by intermediary nodes, for example, gateways, or indeed in the end systems.

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 5

By George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair
Addison-Wesley ©Pearson Education 2012

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?

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.

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 5

By George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair
Addison-Wesley ©Pearson Education 2012

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 cdk5.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 cdk5.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 Why can't binary data be represented directly in XML, for example, by representing it as Unicode byte values? XML elements can carry strings represented as *base64*. Discuss the advantages or disadvantages of using this method to represent binary data.

4.12 Ans.

Binary data can't be represented directly in XML, because somewhere the embedded binary data will include the representation of a special character such as '<' or '>'.

Even if binary is represented as CDATA, it might include the terminator for CDATA: ']]'.

Binary data can be encoded in base64 and represented in XML as a *string*.

Base64 encoding takes three bytes, each consisting of eight bits, and represents them as four printable characters in the ASCII standard (padding when necessary).

Thus a disadvantage in using base64 is that the quantity of data is increased by a factor of 4/3 and the translation process at both ends can take time. In addition, the encoded value isn't really a string and in some cases may be passed on to an application as a string.

The only advantage of base64 is that it can be used when necessary. It is in fact used in XML Security to represent digital signatures and encrypted data. In both cases, the data is enclosed in elements that specify, for example, *signature* or *encrypted dat*a.

4.13 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.

4.13 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.

4.14 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.

4.14 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.

4.15 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.

4.15 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.

4.16 Your solution to Exercise 4.15 should have overcome the problem of dropped messages in IP multicast. In what sense does your solution differ from the definition of reliable multicast?

4.16 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.

4.17 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.

4.17 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.

4.18 Revisit the Internet architecture as introduced in Chapter 3 (see Figures 3.12 and 3.14). What impact does the introduction of overlay networks have on this architecture, and in particular on the programmer's conceptual view of the Internet?

4.18 Ans.

The initial Internet architecture was devised for a relatively small number of applications (such as electronic mail and file transfer) to run over relatively homogeneous styles of network. The definition and standardization of the (relatively simple) Internet architecture has been very successful in supporting these applications and other emerging applications. As the Internet grows though in terms of scale and diversity, there is a need to support richer application styles and also to operate more effectively over an increasing variety of network types. The introduction of overlay networks introduces an element of extra complexity into the Internet architecture but also an increasing degree of sophistication to meet these two requirements. For example, overlay networks can be introduced to support new styles of Internet applications, for example related to streaming of continuous media content. The Internet can also be extended to operate more optimally over new network types, for example ad hoc networks.

The key elements in terms of the conceptual view offered by the Internet is that programmers now see the architecture as extensible and also see the potential to have more application-specific network services. They also see a move from rigid standardization to a more experimental environment.

4.19 What are the main arguments for adopting a super node approach in Skype?

4.19 Ans.

The main reason for having super nodes is to implement efficient and reliable search, a key enabling function in Skype. Through this approach, search is supported only by nodes that have the required capabilities in terms of bandwidth, reachability and availability. The alternative is to adopt a pure peer-to-peer approach where all nodes cooperate to provide this function. The downside of this pure approach is that certain nodes involved in the implementation of search may be disconnected, or have weak connectivity in terms of bandwidth, leading to a more erratic service. The trade-off is that super nodes will encounter more traffic due to search, but they are selected on the basis they can absorb this.

4.20 As discussed in Section 4.6, MPI offers a number of variants of *send* including the *MPI_Rsend* operation, which assumes the receiver is ready to receive at the time of sending. What optimizations in implementation are possible if this assumption is correct and what are the repercussions of this assumption being false?

4.20 Ans.

The key optimization is that the sender does not have to check that the receiver is ready to receive a message and this therefore avoids a handshake between sender and receiver, increasing the performance of the implementation. If the receiver is not ready, the outcome of the send operation is unpredictable in terms of the buffer being ready and the MPI documentation deems the outcome undefined. It may be for example that values will be over-written before being consumed.

Distributed Systems: Concepts and Design

Edition 5

By George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair

Chapter 5 Exercise Solutions

5.1 Define a class whose instances represent request and reply messages as illustrated in Figure 5.4. 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.

5.1 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;}
}
```

5.2 Program each of the three operations of the request-reply protocol in Figure 5.3, using UDP communication, but without adding any fault-tolerance measures. You should use the classes you defined in Exercise 4.13 and Exercise 5.1.

5.2 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);
    }
}
```

5.3 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..

5.3 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 {... }
    }
}
```

5.4 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.

5.4 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;
    }
```

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

5.5 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.

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

5.6 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.]

5.7 Discuss whether the following operations are *idempotent*:

i) Pressing a lift (elevator) request button;

ii) Writing data to a file;

iii) Appending data to a file.

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

5.7 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.

5.8 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.

5.8 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.

5.9 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?

5.9 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.

5.10 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.)

5.10 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.

5.11 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?

5.11 Ans.

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

---

5.12 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.

5.12 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.

5.13 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.

5.13 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.

5.14 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?

5.14 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.

5.15 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?

5.15 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

---

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

5.16 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.

5.17 Assume the *Election* service isimplemented in RMI and 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 5.16.

5.17 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.

5.18 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).

5.18 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 (...) {}
}
```

5.19 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 5.13. Give the details of the implementation of one of the methods in this class, which should call the method *doOperation* defined in Figure 5.3.

5.19 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.

5.20 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.

5.20 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(...){}
}
```

5.21 Exercise 5.18 required the client to convert *Object* arguments into an array of bytes before invoking *doOperation* and Exercise 5.20 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?

5.21 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)

5.22 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?

5.22 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.

5.23 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.

5.23 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.

5.24 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.

5.24 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.

Distributed Systems: Concepts and Design

Edition 5

By George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair

Chapter 16 Exercise Solutions

16.1 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.

16.1 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*.

16.2 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*.

16.2 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.

16.3 Give serially equivalent interleaving of *T* and *U* in Exercise 16.2 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.

16.3 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 13.6):

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 13.2.

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 13.2 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.

16.4 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*?

16.4 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*.

16.5 A newly created data item like *Z* in Exercise 16.4 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.

16.5 Ans.

The point in Exercise 13.4 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 13.10) that these interleavings are not serially equivalent.

16.6 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.

16.6 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.

16.7 Consider the recovery aspects of the nested transactions defined in Exercise 16.6. 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?

16.7 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 13.6 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.

16.8 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.

16.8 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.

16.9 The transactions *T* and *U* at the server in Exercise 16.8 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);	

16.9 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.

16.10 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 16.8:

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.

16.10 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.

16.11 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 16.16 ensure strict executions.

16.11 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).

16.12 Describe how a non-recoverable situation could arise if write locks are released after the last operation of a transaction but before its commitment.

16.12 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.

16.13 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 16.16.

16.13 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.

16.14 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 16.8.

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.

16.14 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 $V \rightarrow T$

16.15 Consider hierarchic locks as illustrated in Figure 16.26. 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?

16.15 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 13.27).

16.16 Consider optimistic concurrency control as applied to the transactions T and U defined in Exercise 16.9. 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.

16.16 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);	

16.17 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.

16.17 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 (Ti = startTn + 1; Ti++; Ti <= finishTn) {
    let S = set of members of read set of Tj before commit Ti
    IF S intersects write set of T1
        THEN valid := false
}
```

16.18 Make a comparison of the sequences of operations of the transactions *T* and *U* of Exercise 16.8 that are possible under two-phase locking (Exercise 16.9) and under optimistic concurrency control (Exercise 16.16).

16.18 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 13.9.

The ordering of 13.9b for two-phase locking is the same as 13.16 (i) optimistic concurrency control.

The ordering of 13.9c for two-phase locking is the same as 13.16 (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.

.

16.19 Consider the use of timestamp ordering with each of the example interleavings of transactions *T* and *U* in Exercise 16.9. 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?

16.19 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 13.31a 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 13.31 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 13.31 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

16.20 Repeat Exercise 16.19 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);	

16.20 Ans.

The difference between this exercise and the interleavings for Exercise 13.9(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 13.9 (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 13.31 d)

16.21 Repeat Exercise 16.20 using multiversion timestamp ordering.
16.21 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 13.20, we show that the outcome is the same. Timestamps are *T*: t_1 and *U*:t_2. Initial state as in Exercise 13.9 a. Call these committed versions V_{t0}.

U: *write(i, 55)*;

write rule: read timestamp of $D_{\text{maxEarlier}}$ 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 $D_{\text{maxEarlier}}$ 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 $D_{\text{maxEarlier}}$ 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 13.20, 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 $D_{maxEarlier}$ 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

16.22 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.

16.22 Ans.

The answer to Exercise 13.21 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 13.19 where a *read* operation was delayed (i.e. (c) or (d)). Consider the diagram for (c) in Exercise 13.2. 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.

16.23 What are the advantages and drawbacks of multiversion timestamp ordering in comparison with ordinary timestamp ordering?

16.23 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 13.22 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.