19.1 Advantages of distributed systems

Introduction

A distributed system is a collection of processors that do not share memory or a clock. Instead, each node has its own local memory. The nodes communicate with one another through various networks, such as high-speed buses. Distributed systems are more relevant than ever, and you have almost certainly used some sort of distributed service. Applications of distributed systems range from providing transparent access to files inside an organization, to large-scale cloud file and photo storage services, to business analysis of trends on large data sets, to parallel processing of scientific data, and more. In fact, the most basic example of a distributed system is one we are all likely very familiar with—the Internet.

In this chapter, we discuss the general structure of distributed systems and the networks that interconnect them. We also contrast the main differences in the types and roles of current distributed system designs. Finally, we investigate some of the basic designs and design challenges of distributed file systems.

Chapter objectives

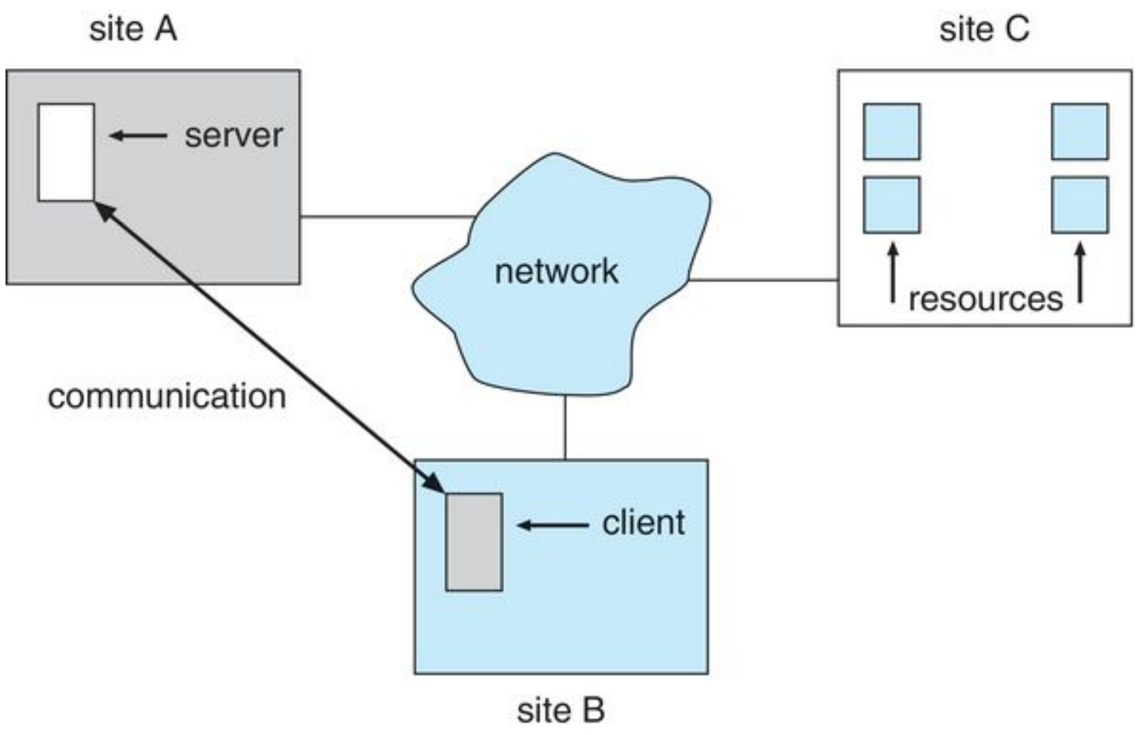
- Explain the advantages of networked and distributed systems.
- Provide a high-level overview of the networks that interconnect distributed systems.
- Define the roles and types of distributed systems in use today.
- Discuss issues concerning the design of distributed file systems.

Advantages of distributed systems

A ***distributed system*** is a collection of loosely coupled nodes interconnected by a communication network. From the point of view of a specific node in a distributed system, the rest of the nodes and their respective resources are remote, whereas its own resources are local.

The nodes in a distributed system may vary in size and function. They may include small microprocessors, personal computers, and large general-purpose computer systems. These processors are referred to by a number of names, such as *processors*, *sites*, *machines*, and *hosts*, depending on the context in which they are mentioned. We mainly use *site* to indicate the location of a machine and *node* to refer to a specific system at a site. Nodes can exist in a *client-server* configuration, a *peer-to-peer* configuration, or a hybrid of these. In the common client-server configuration, one node at one site, the *server*, has a resource that another node, the *client* (or user), would like to use. A general structure of a client-server distributed system is shown in Figure 19.1.1. In a peer-to-peer configuration, there are no servers or clients. Instead, the nodes share equal responsibilities and can act as both clients and servers.

Figure 19.1.1: A client-server distributed system.



When several sites are connected to one another by a communication network, users at the various sites have the opportunity to exchange information. At a low level, ***messages*** are passed between systems, much as messages are passed between processes in the single-computer message system discussed in Section 3.4. Given message passing, all the higher-level functionality found in standalone systems can be expanded to encompass the distributed system. Such functions include file storage, execution of applications, and remote procedure calls (RPCs).

There are three major reasons for building distributed systems: resource sharing, computational speedup, and reliability. In this section, we briefly discuss each of them.

Resource sharing

If a number of different sites (with different capabilities) are connected to one another, then a user at one site may be able to use the resources available at another. For example, a user at site A may query a database located at site B. Meanwhile, a user at site B may access a file that resides at site A. In general, ***resource sharing*** in a distributed system provides mechanisms for sharing files at remote sites, processing information in a distributed database, printing files at remote sites, using remote specialized hardware devices such as a supercomputer or a ***graphics processing unit*** (*GPU*), and performing other operations.

Computation speedup

If a particular computation can be partitioned into subcomputations that can run concurrently, then a distributed system allows us to distribute the subcomputations among the various sites. The subcomputations can be run concurrently and thus provide ***computation speedup***. This is especially relevant when doing large-scale processing of big data sets (such as analyzing large amounts of customer data for trends). In addition, if a particular site is currently overloaded with requests, some of them can be moved or rerouted to other, more lightly loaded sites. This movement of jobs is called ***load balancing*** and is common among distributed system nodes and other services provided on the Internet.

Reliability

If one site fails in a distributed system, the remaining sites can continue operating, giving the system better reliability. If the system is composed of multiple large autonomous installations (that is, general-purpose computers), the failure of one of them should not affect the rest. If, however, the system is composed of diversified machines, each of which is responsible for some crucial system function (such as the web server or the file system), then a single failure may halt the operation of the whole system. In general, with enough redundancy (in both hardware and data), the system can continue operation even if some of its nodes have failed.

The failure of a node or site must be detected by the system, and appropriate action may be needed to recover from the failure. The system must no longer use the services of that site. In addition, if the function of the failed site can be taken over by another site, the system must ensure that the transfer of function occurs correctly. Finally, when the failed site recovers or is repaired, mechanisms must be available to integrate it back into the system smoothly.

PARTICIPATION ACTIVITY | 19.1.1: Section review questions.

1) A distributed system can continue operating after a failure if enough hardware and data redundancy have been provided.

○ True

○ False

2) What is NOT a major reason to build a distributed system?

○ computational speed-up

○ security

○ resource sharing

○ reliability

Section glossary

distributed system: A collection of loosely coupled nodes interconnected by a communication network.

message: In networking, a communication, contained in one or more packets, that includes source and destination information to allow correct delivery. In message-passing communications, a packet of information with metadata about its sender and receiver.

resource sharing: The ability for multiple users, computers, etc., to access computing resources.

graphics processing unit (GPU): A hardware component, sometimes part of the CPU and sometimes a separate device, that provides graphics computation and sometimes graphics output to a display.

computation speedup: An increase in the amount of CPU compute power.

load balancing: The movement of jobs or network packets between various components (say, computers in a network) to distribute the load or route around failures. Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system.

19.2 Network structure

To completely understand the roles and types of distributed systems in use today, we need to understand the networks that interconnect them. This section serves as a network primer to introduce basic networking concepts and challenges as they relate to distributed systems. The rest of the chapter specifically discusses distributed systems.

There are basically two types of networks: ***local-area networks*** (*LAN*) and ***wide-area networks*** (*WAN*). The main difference between the two is the way in which they are geographically distributed. Local-area networks are composed of hosts distributed over small areas (such as a single building or a number of adjacent buildings), whereas wide-area networks are composed of systems distributed over a large area (such as the United States). These differences imply major variations in the speed and reliability of the communications networks, and they are reflected in the distributed system design.

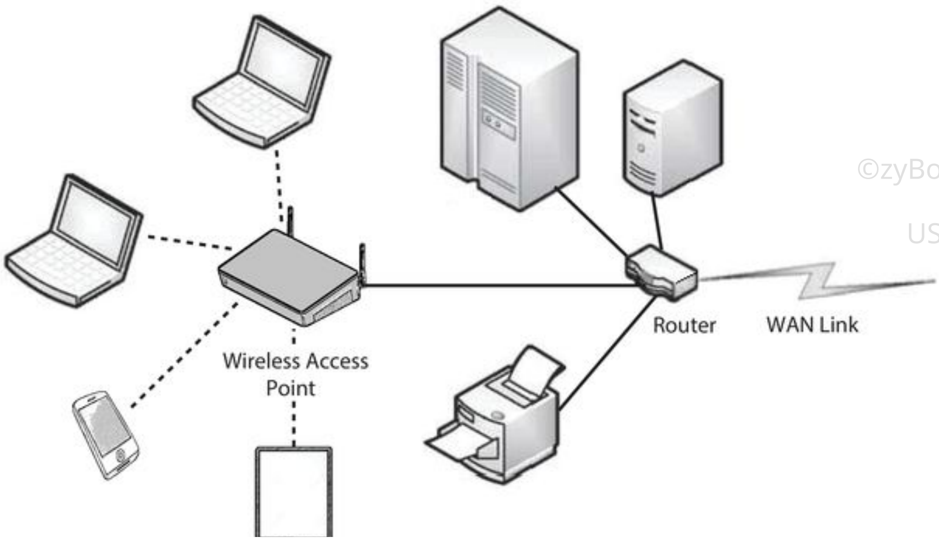
Local-area networks

Local-area networks emerged in the early 1970s as a substitute for large mainframe computer systems. For many enterprises, it is more economical to have a number of small computers, each with its own self-contained applications, than to have a single large system. Because each small computer is likely to need a full complement of peripheral devices (such as disks and printers), and because some form of data sharing is likely to occur in a single enterprise, it was a natural step to connect these small systems into a network.

LANs, as mentioned, are usually designed to cover a small geographical area, and they are generally used in an office or home environment. All the sites in such systems are close to one another, so the communication links tend to have a higher speed and lower error rate than their counterparts in wide-area networks.

A typical LAN may consist of a number of different computers (including workstations, servers, laptops, tablets, and smartphones), various shared peripheral devices (such as printers and storage arrays), and one or more ***routers*** (specialized network communication processors) that provide access to other networks Figure 19.2.1). Ethernet and ***WiFi*** are commonly used to construct LANs. *Wireless access points* connect devices to the LAN wirelessly, and they may or may not be routers themselves.

Figure 19.2.1: Local-area network.



Ethernet networks are generally found in businesses and organizations in which computers and peripherals tend to be nonmobile. These networks use *coaxial*, *twisted pair*, and/or *fiber optic* cables to send signals. An Ethernet network has no central controller, because it is a multiaccess bus, so new hosts can be added easily to the network. The Ethernet protocol is defined by the IEEE 802.3 standard. Typical Ethernet speeds using common twisted-pair cabling can vary from 10 Mbps to over 10 Gbps, with other types of cabling reaching speeds of 100 Gbps.

WiFi is now ubiquitous and either supplements traditional Ethernet networks or exists by itself. Specifically, WiFi allows us to construct a network without using physical cables. Each host has a wireless transmitter and receiver that it uses to participate in the network. WiFi is defined by the IEEE 802.11 standard. Wireless networks are popular in homes and businesses, as well as public areas such as libraries, Internet cafes, sports arenas, and even buses and airplanes. WiFi speeds can vary from 11 Mbps to over 400 Mbps.

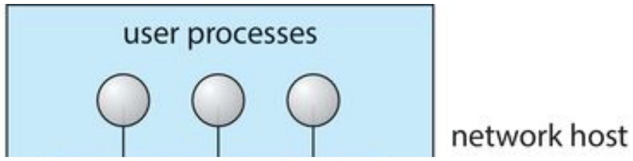
Both the IEEE 802.3 and 802.11 standards are constantly evolving. For the latest information about various standards and speeds, see the references at the end of the chapter.

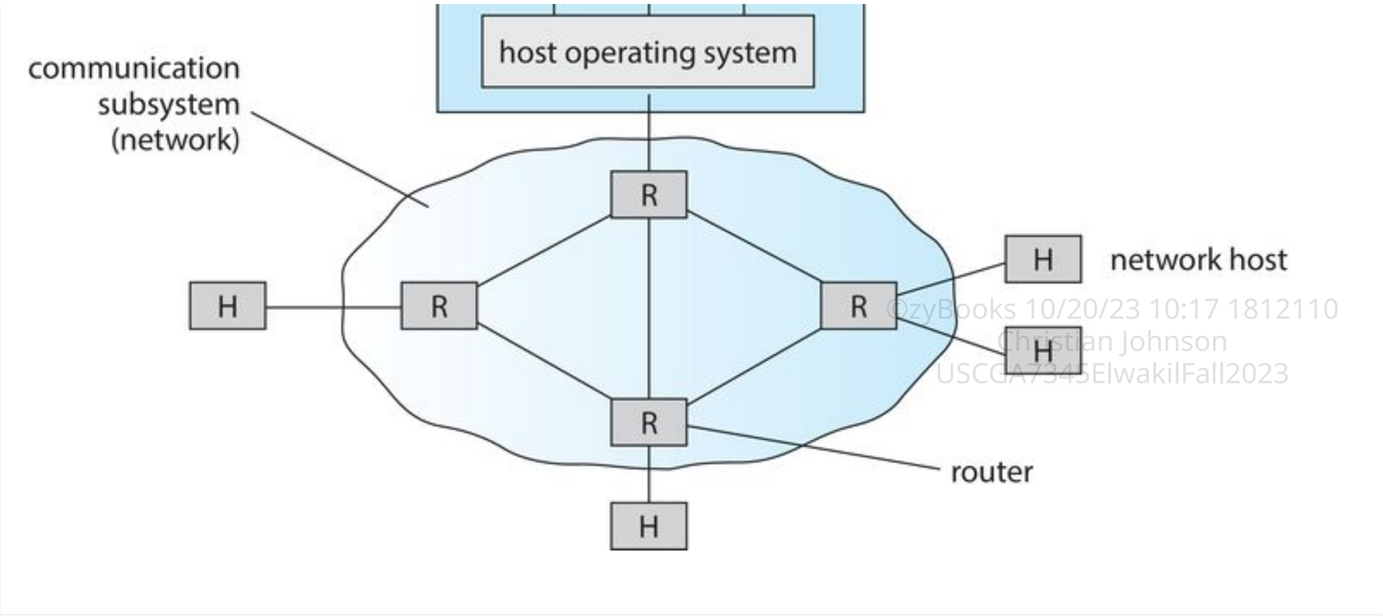
Wide-area networks

Wide-area networks emerged in the late 1960s, mainly as an academic research project to provide efficient communication among sites, allowing hardware and software to be shared conveniently and economically by a wide community of users. The first WAN to be designed and developed was the ARPANET. Begun in 1968, the ARPANET has grown from a four-site experimental network to a worldwide network of networks, the ***Internet*** (also known as the ***World Wide Web***), comprising millions of computer systems.

Sites in a WAN are physically distributed over a large geographical area. Typical links are telephone lines, leased (dedicated data) lines, optical cable, microwave links, radio waves, and satellite channels. These communication links are controlled by routers (Figure 19.2.2) that are responsible for directing traffic to other routers and networks and transferring information among the various sites.

Figure 19.2.2: Communication processors in a wide-area network.





For example, the Internet WAN enables hosts at geographically separate sites to communicate with one another. The host computers typically differ from one another in speed, CPU type, operating system, and so on. Hosts are generally on LANs, which are, in turn, connected to the Internet via regional networks. The regional networks are interlinked with routers to form the worldwide network. Residences can connect to the Internet by either telephone, cable, or specialized Internet service providers that install routers to connect the residences to central services. Of course, there are other WANs besides the Internet. A company, for example, might create its own private WAN for increased security, performance, or reliability.

WANs are generally slower than LANs, although backbone WAN connections that link major cities may have very fast transfer rates through fiber optic cables. In fact, many backbone providers have fiber optic speeds of 40 Gbps or 100 Gbps. (It is generally the links from local ***Internet Service Providers*** (*ISPs*) to homes or businesses that slow things down.) However, WAN links are being constantly updated to faster technologies as the demand for more speed continues to grow.

Frequently, WANs and LANs interconnect, and it is difficult to tell where one ends and the other starts. Consider the cellular phone data network. Cell phones are used for both voice and data communications. Cell phones in a given area connect via radio waves to a cell tower that contains receivers and transmitters. This part of the network is similar to a LAN except that the cell phones do not communicate with each other (unless two people talking or exchanging data happen to be connected to the same tower). Rather, the towers are connected to other towers and to hubs that connect the tower communications to land lines or other communication media and route the packets toward their destinations. This part of the network is more WAN-like. Once the appropriate tower receives the packets, it uses its transmitters to send them to the correct recipient.

PARTICIPATION ACTIVITY 19.2.1: Section review questions.

1) What is the appropriate network for a

college campus?

○ Local-area network

○ Wide-area network

2) A wireless access point may also serve as a router.

○ True

○ False

Section glossary

local-area network (LAN): A network that connects computers within a room, a building, or a campus.

wide-area network (WAN): A network that links buildings, cities, or countries.

router: A device or software that connects networks to each other (e.g., a home network to the Internet).

WiFi: Wireless networking, consisting of devices and protocols that allow devices to attach to a network via radio waves rather than cables.

Internet: A worldwide system of interconnected computer networks.

World Wide Web (WWW): The Internet; a worldwide system of interconnected computer networks.

Internet Service Providers (ISPs): Companies that provide Internet access.

19.3 Communication structure

Now that we have discussed the physical aspects of networking, we turn to the internal workings.

Naming and name resolution

The first issue in network communication involves the naming of the systems in the network. For a process at site A to exchange information with a process at site B, each must be able to specify the other. Within a computer system, each process has a process identifier, and messages may be addressed with the process identifier. Because networked systems share no memory, however, a host within the system initially has no knowledge about the processes on other hosts.

To solve this problem, processes on remote systems are generally identified by the pair <host name, identifier>, where ***host name*** is a name unique within the network and ***identifier*** is a process identifier or other unique number within that host. A host name is usually an alphanumeric identifier, rather than a number, to make it easier for users to specify. For instance, site A might have hosts named *program, student, faculty*, and *cs*. The host name *program* is certainly easier to remember than the numeric host address *128.148.31.100*.

Names are convenient for humans to use, but computers prefer numbers for speed and simplicity. For this reason, there must be a mechanism to ***resolve*** the host name into a ***host-id*** that describes the destination system to the networking hardware. This mechanism is similar to the name-to-address binding that occurs during program compilation, linking, loading, and execution (Chapter Main Memory). In the case of host names, two possibilities exist. First, every host may have a data file containing the names and numeric addresses of all the other hosts reachable on the network (similar to binding at compile time). The problem with this model is that adding or removing a host from the network requires updating the data files on all the hosts. In fact, in the early days of the ARPANET there was a canonical host file that was copied to every system periodically. As the network grew, however, this method became untenable.

The alternative is to distribute the information among systems on the network. The network must then use a protocol to distribute and retrieve the information. This scheme is like execution-time binding. The Internet uses a ***domain-name system*** (*DNS*) for host-name resolution.

DNS specifies the naming structure of the hosts, as well as name-to-address resolution. Hosts on the Internet are logically addressed with multipart names known as IP addresses. The parts of an IP address progress from the most specific to the most general, with periods separating the fields. For instance, *eric.cs.yale.edu* refers to host *eric* in the Department of Computer Science at Yale University within the top-level domain *edu*. (Other top-level domains include *com* for commercial sites and *org* for organizations, as well as a domain for each country connected to the network for systems specified by country rather than organization type.) Generally, the system resolves addresses by examining the host-name components in reverse order. Each component has a ***name server***—simply a process on a system—that accepts a name and returns the address of the name server responsible for that name. As the final step, the name server for the host in question is contacted, and a host-id is returned. For example, a request made by a process on system A to communicate with *eric.cs.yale.edu* would result in the following steps:

1. The system library or the kernel on system A issues a request to the name server for the edu domain, asking for the address of the name server for `yale.`edu. The name server for the *edu* domain must be at a known address, so that it can be queried.
2. The edu name server returns the address of the host on which the `yale.`edu name server resides.
3. System A then queries the name server at this address and asks about `cs.yale.`edu.

4. An address is returned. Now, finally, a request to that address for `eric.cs.yale.edu` returns an Internet address host-id for that host (for example, `128.148.31.100`).

This protocol may seem inefficient, but individual hosts cache the IP addresses they have already resolved to speed the process. (Of course, the contents of these caches must be refreshed over time in case the name server is moved or its address changes.) In fact, the protocol is so important that it has been optimized many times and has had many safeguards added. Consider what would happen if the primary edu name server crashed. It is possible that no edu hosts would be able to have their addresses resolved, making them all unreachable! The solution is to use secondary, backup name servers that duplicate the contents of the primary servers.

Before the domain-name service was introduced, all hosts on the Internet needed to have copies of a file (mentioned above) that contained the names and addresses of each host on the network. All changes to this file had to be registered at one site (host SRI-NIC), and periodically all hosts had to copy the updated file from SRI-NIC to be able to contact new systems or find hosts whose addresses had changed. Under the domain-name service, each name-server site is responsible for updating the host information for that domain. For instance, any host changes at Yale University are the responsibility of the name server for *yale.edu* and need not be reported anywhere else. DNS lookups will automatically retrieve the updated information because they will contact *yale.edu* directly. Domains may contain autonomous subdomains to further distribute the responsibility for host-name and host-id changes.

Java provides the necessary API to design a program that maps IP names to IP addresses. The program shown in Figure 19.3.1 is passed an IP name (such as *eric.cs.yale.edu*) on the command line and either outputs the IP address of the host or returns a message indicating that the host name could not be resolved. An `InetAddress` is a Java class representing an IP name or address. The static method `getByName()` belonging to the `InetAddress` class is passed a string representation of an IP name, and it returns the corresponding `InetAddress`. The program then invokes the `getHostAddress()` method, which internally uses DNS to look up the IP address of the designated host.

Figure 19.3.1: Java program illustrating a DNS lookup.

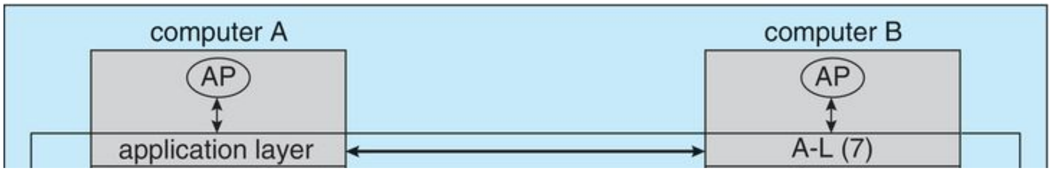
```
/**
 * Usage: java DNSLookUp <IP name>
 * i.e. java DNSLookUp www.wiley.com
 */
public class DNSLookUp {
 public static void main(String[] args) {
  InetAddress hostAddress;

  try {
   hostAddress = InetAddress.getByName(args[0]);
System.out.println(hostAddress.getHostAddress());
  }
  catch (UnknownHostException uhe) {
   System.err.println("Unknown host: " +
args[0]);
 }
}
```

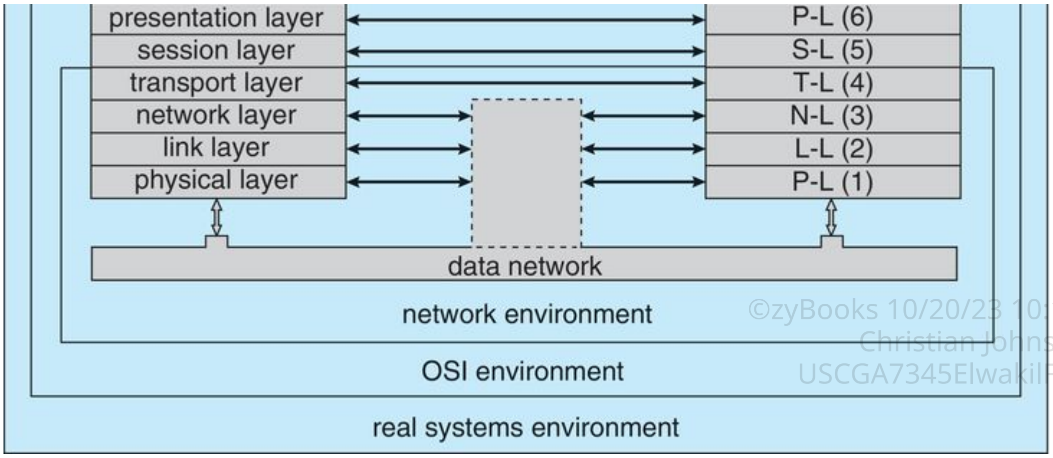
Generally, the operating system is responsible for accepting from its processes a message destined for <host name, identifier> and for transferring that message to the appropriate host. The kernel on the destination host is then responsible for transferring the message to the process named by the identifier. This process is described in Section Transport protocols UDP and TCP.

Communication protocols

When we are designing a communication network, we must deal with the inherent complexity of coordinating asynchronous operations communicating in a potentially slow and error-prone environment. In addition, the systems on the network must agree on a protocol or a set of protocols for determining host names, locating hosts on the network, establishing connections, and so on. We can simplify the design problem (and related implementation) by partitioning the problem into multiple layers. Each layer on one system communicates with the equivalent layer on other systems. Typically, each layer has its own protocols, and communication takes place between peer layers using a specific protocol. The protocols may be implemented in hardware or software. For instance, Figure 19.3.2 shows the logical communications between two computers, with the three lowest-level layers implemented in hardware.

Figure 19.3.2: Two computers communicating via the OSI network model.





The International Standards Organization created the Open Systems Interconnection (OSI) model for describing the various layers of networking. While these layers are not implemented in practice, they are useful for understanding how networking logically works, and we describe them below:

- **Layer 1: physical layer.** The physical layer is responsible for handling both the mechanical and the electrical details of the physical transmission of a bit stream. At the physical layer, the communicating systems must agree on the electrical representation of a binary 0 and 1, so that when data are sent as a stream of electrical signals, the receiver is able to interpret the data properly as binary data. This layer is implemented in the hardware of the networking device. It is responsible for delivering bits.
- **Layer 2: data-link layer.** The data-link layer is responsible for handling *frames*, or fixed-length parts of packets, including any error detection and recovery that occur in the physical layer. It sends frames between physical addresses.
- **Layer 3: network layer.** The network layer is responsible for breaking messages into packets, providing connections between logical addresses, and routing packets in the communication network, including handling the addresses of outgoing packets, decoding the addresses of incoming packets, and maintaining routing information for proper response to changing load levels. Routers work at this layer.
- **Layer 4: transport layer.** The transport layer is responsible for transfer of messages between nodes, maintaining packet order, and controlling flow to avoid congestion.
- **Layer 5: session layer.** The session layer is responsible for implementing sessions, or process-to-process communication protocols.
- **Layer 6: presentation layer.** The presentation layer is responsible for resolving the differences in formats among the various sites in the network, including character conversions and half duplex-full duplex modes (character echoing).
- **Layer 7: application layer.** The application layer is responsible for interacting directly with users. This layer deals with file transfer, remote-login protocols, and electronic mail, as well as with schemas for distributed databases.

Figure 19.3.3 summarizes the ***OSI protocol stack***—a set of cooperating protocols—showing the physical flow of data. As mentioned, logically each layer of a protocol stack communicates with the equivalent layer on other systems. But physically, a message starts at or above the application layer and is passed through each lower level in turn. Each layer may modify the message and include message-header data for the equivalent layer on the receiving side. Ultimately, the message reaches the data-network layer and is transferred as one or more packets Figure 19.3.4). The data-link layer of the target system receives these data, and the message is moved up through the protocol stack. It is analyzed, modified, and stripped of headers as it progresses. It finally reaches the application layer for use by the receiving process.

Figure 19.3.3: The OSI protocol stack.

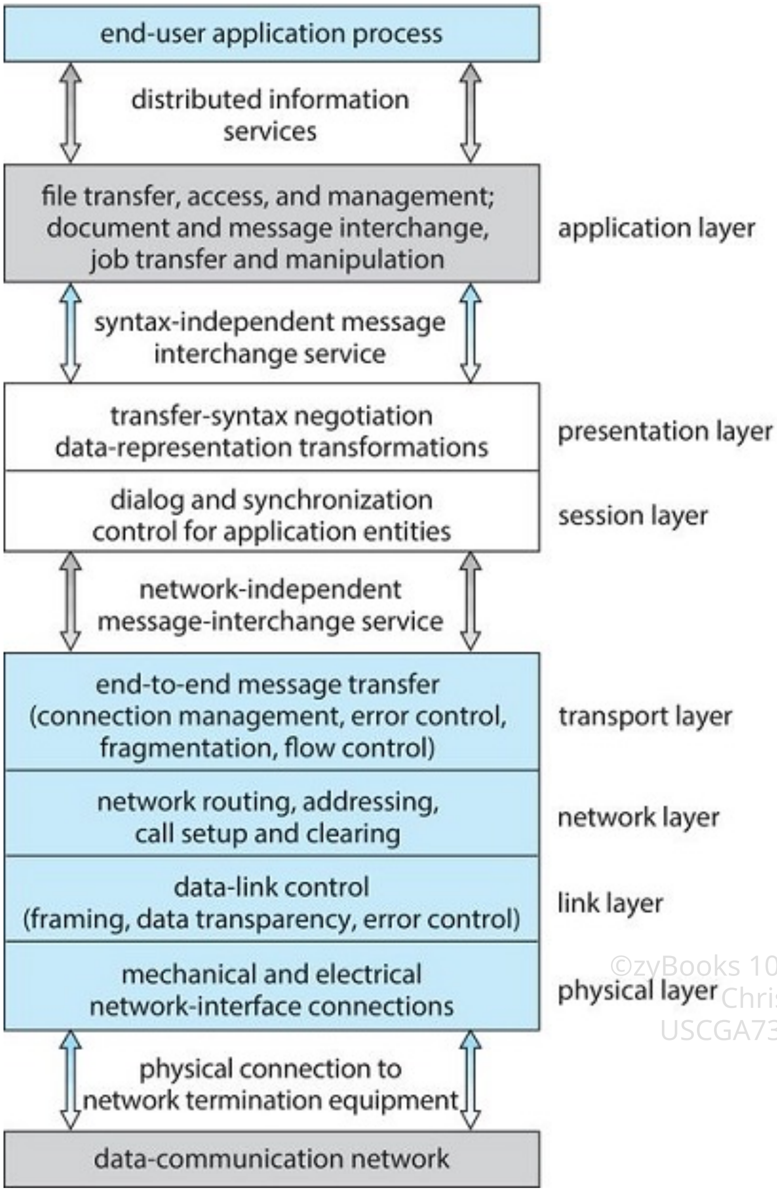
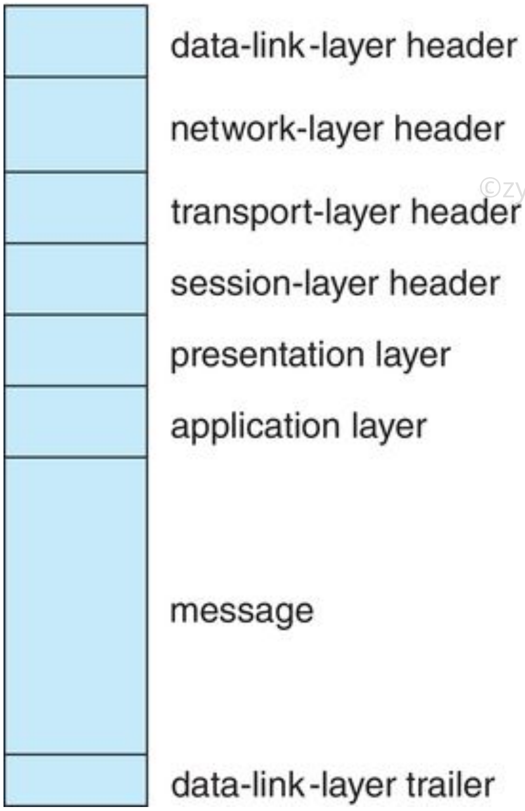
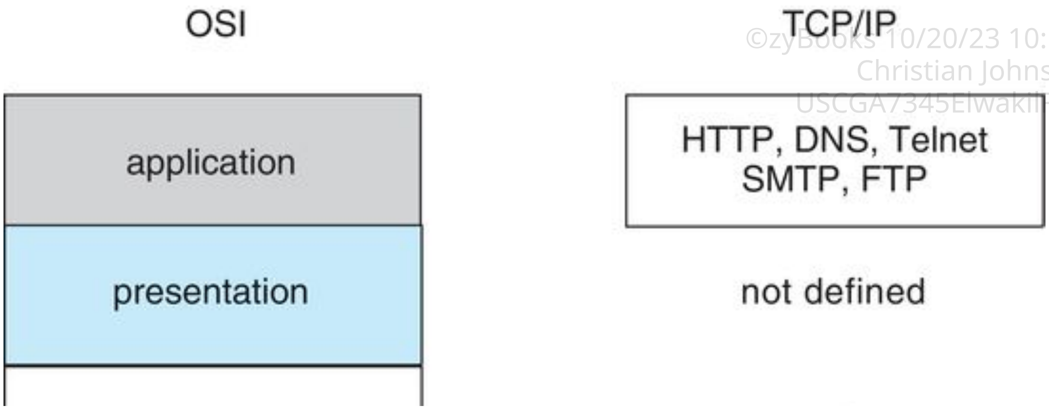

end-user application process
distributed information services
file transfer, access, and management; document and message interchange, job transfer and manipulation
application layer
syntax-independent message interchange service
transfer-syntax negotiation data-representation transformations
presentation layer
dialog and synchronization control for application entities
session layer
network-independent message-interchange service
end-to-end message transfer (connection management, error control, fragmentation, flow control)
transport layer
network routing, addressing, call setup and clearing
network layer
data-link control (framing, data transparency, error control)
link layer
mechanical and electrical network-interface connections
physical layer
physical connection to network termination equipment
data-communication network

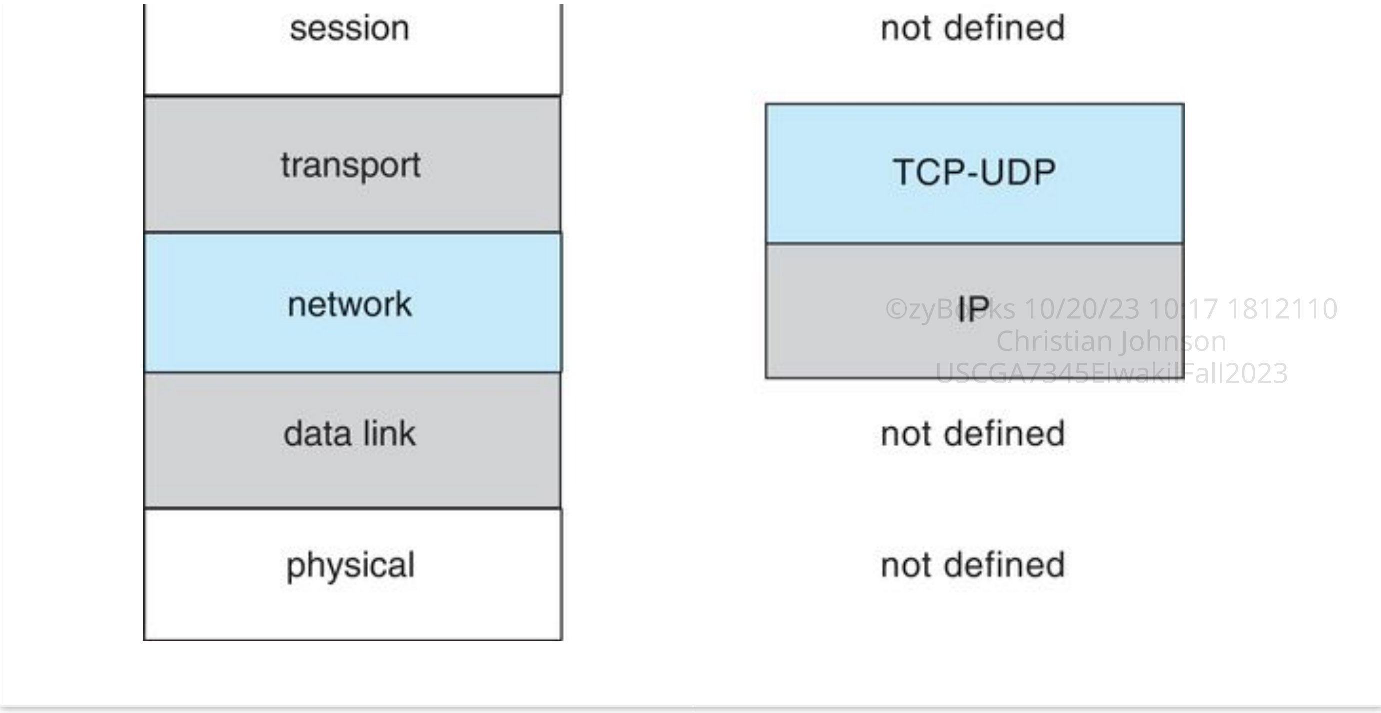
Figure 19.3.4: An OSI network message.



The OSI model formalizes some of the earlier work done in network protocols but was developed in the late 1970s and is currently not in widespread use. Perhaps the most widely adopted protocol stack is the TCP/IP model (sometimes called the *Internet model*), which has been adopted by virtually all Internet sites. The TCP/IP protocol stack has fewer layers than the OSI model. Theoretically, because it combines several functions in each layer, it is more difficult to implement but more efficient than OSI networking. The relationship between the OSI and TCP/IP models is shown in Figure 19.3.5.

Figure 19.3.5: The OSI and TCP/IP protocol stacks.





The TCP/IP application layer identifies several protocols in widespread use in the Internet, including HTTP, FTP, SSH, DNS, and SMTP. The transport layer identifies the unreliable, connectionless ***user datagram protocol*** (*UDP*) and the reliable, connection-oriented ***transmission control protocol*** (*TCP*). The ***Internet protocol*** (*IP*) is responsible for routing IP ***datagrams***, or packets, through the Internet. The TCP/IP model does not formally identify a link or physical layer, allowing TCP/IP traffic to run across any physical network. In Section TCP/IP example, we consider the TCP/IP model running over an Ethernet network.

Security should be a concern in the design and implementation of any modern communication protocol. Both strong authentication and encryption are needed for secure communication. Strong authentication ensures that the sender and receiver of a communication are who or what they are supposed to be. Encryption protects the contents of the communication from eavesdropping. Weak authentication and clear-text communication are still very common, however, for a variety of reasons. When most of the common protocols were designed, security was frequently less important than performance, simplicity, and efficiency. This legacy is still showing itself today, as adding security to existing infrastructure is proving to be difficult and complex.

Strong authentication requires a multistep handshake protocol or authentication devices, adding complexity to a protocol. As to the encryption requirement, modern CPUs can efficiently perform encryption, frequently including cryptographic acceleration instructions so system performance is not compromised. Long-distance communication can be made secure by authenticating the endpoints and encrypting the stream of packets in a virtual private network, as discussed in Section Implementation of cryptography. LAN communication remains unencrypted at most sites, but protocols such as NFS Version 4, which includes strong native authentication and encryption, should help improve even LAN security.

TCP/IP example

Next, we address name resolution and examine its operation with respect to the TCP/IP protocol stack on the Internet. Then we consider the processing needed to transfer a packet between hosts on different Ethernet networks. We base our description on the IPV4 protocols, which are the type most commonly used today.

In a TCP/IP network, every host has a name and an associated IP address (or host-id). Both of these strings must be unique; and so that the name space can be managed, they are segmented. As described earlier, the name is hierarchical, describing the host name and then the organization with which the host is associated. The host-id is split into a network number and a host number. The proportion of the split varies, depending on the size of the network. Once the Internet administrators assign a network number, the site with that number is free to assign host-ids.

The sending system checks its routing tables to locate a router to send the frame on its way. This routing table is either configured manually by the system administrator or is populated by one of several routing protocols, such as the ***Border Gateway Protocol*** (*BGP*). The routers use the network part of the host-id to transfer the packet from its source network to the destination network. The destination system then receives the packet. The packet may be a complete message, or it may just be a component of a message, with more packets needed before the message can be reassembled and passed to the TCP/UDP (transport) layer for transmission to the destination process.

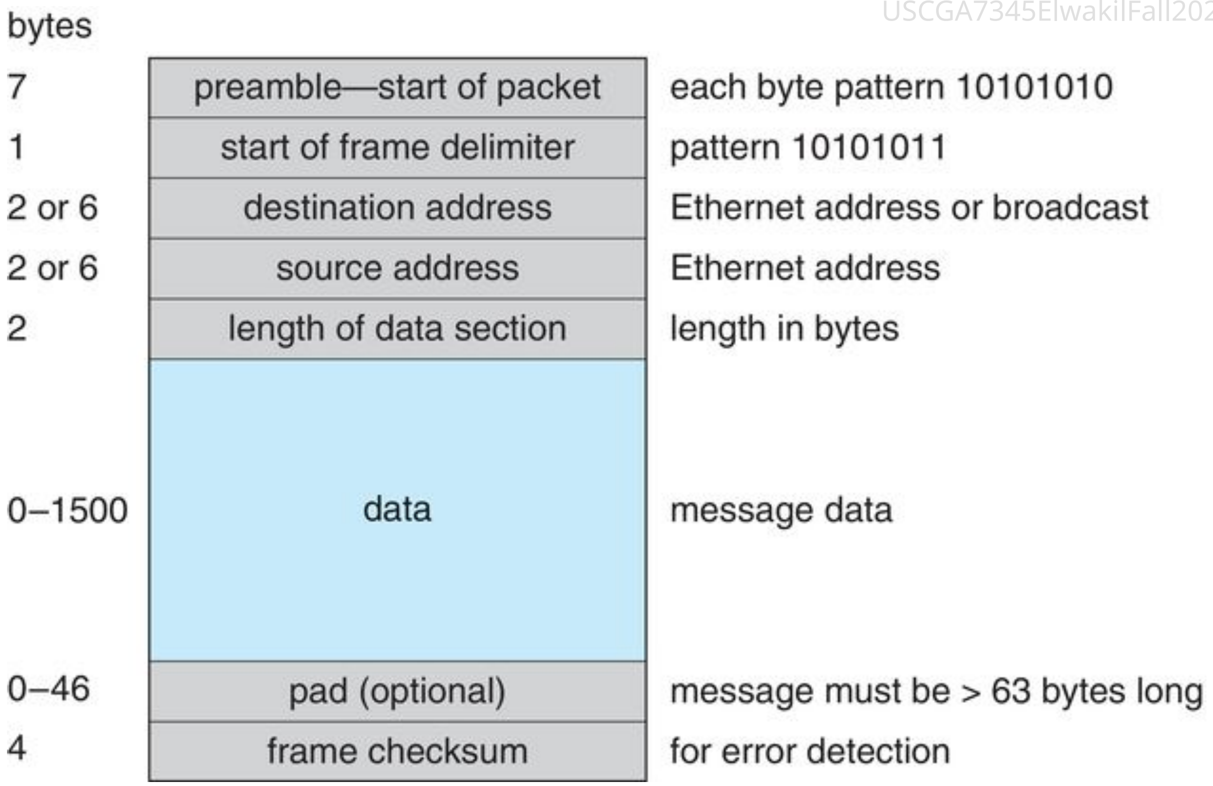
Within a network, how does a packet move from sender (host or router) to receiver? Every Ethernet device has a unique byte number, called the ***medium access control*** (*MAC*) *address*, assigned to it for addressing. Two devices on a LAN communicate with each other only with this number. If a system needs to send data to another system, the networking software generates an ***address resolution protocol*** (***ARP***) packet containing the IP address of the destination system. This packet is ***broadcast*** to all other systems on that Ethernet network.

A broadcast uses a special network address (usually, the maximum address) to signal that all hosts should receive and process the packet. The broadcast is not re-sent by routers in between different networks, so only systems on the local network receive it. Only the system whose IP address matches the IP address of the ARP request responds and sends back its MAC address to the system that initiated the query. For efficiency, the host caches the IP-MAC address pair in an internal table. The cache entries are aged, so that an entry is eventually removed from the cache if an access to that system is not required within a given time. In this way, hosts that are removed from a network are eventually forgotten. For added performance, ARP entries for heavily used hosts may be pinned in the ARP cache.

Once an Ethernet device has announced its host-id and address, communication can begin. A process may specify the name of a host with which to communicate. Networking software takes that name and determines the IP address of the target, using a DNS lookup or an entry in a local `hosts` file where translations can be manually stored. The message is passed from the application layer, through the software layers, and to the hardware layer. At the hardware layer, the packet has the Ethernet address at its start; a trailer indicates the end of the packet and contains a ***checksum*** for detection of packet damage (Figure 19.3.6). The packet is placed on the network by the

Ethernet device. The data section of the packet may contain some or all of the data of the original message, but it may also contain some of the upper-level headers that compose the message. In other words, all parts of the original message must be sent from source to destination, and all headers above the 802.3 layer (data-link layer) are included as data in the Ethernet packets.

Figure 19.3.6: An Ethernet packet.



bytes		
7	preamble—start of packet	each byte pattern 10101010
1	start of frame delimiter	pattern 10101011
2 or 6	destination address	Ethernet address or broadcast
2 or 6	source address	Ethernet address
2	length of data section	length in bytes
0–1500	data	message data
0–46	pad (optional)	message must be > 63 bytes long
4	frame checksum	for error detection

If the destination is on the same local network as the source, the system can look in its ARP cache, find the Ethernet address of the host, and place the packet on the wire. The destination Ethernet device then sees its address in the packet and reads in the packet, passing it up the protocol stack.

If the destination system is on a network different from that of the source, the source system finds an appropriate router on its network and sends the packet there. Routers then pass the packet along the WAN until it reaches its destination network. The router that connects the destination network checks its ARP cache, finds the Ethernet number of the destination, and sends the packet to that host. Through all of these transfers, the data-link-layer header may change as the Ethernet address of the next router in the chain is used, but the other headers of the packet remain the same until the packet is received and processed by the protocol stack and finally passed to the receiving process by the kernel.

Transport protocols UDP and TCP

Once a host with a specific IP address receives a packet, it must somehow pass it to the correct waiting process. The transport protocols TCP and UDP identify the receiving (and sending) processes through the use of a ***port number***. Thus, a host with a single IP address can have multiple server processes running and waiting for packets as long as each server process specifies a different port number. By default, many common services use *well-known* port numbers. Some examples include FTP (21), SSH (22), SMTP (25), and HTTP (80). For example, if you wish to connect to an "http" website through your web browser, your browser will automatically attempt to connect to port 80 on the server by using the number 80 as the port number in the TCP transport header. For an extensive list of well-known ports, log into your favorite Linux or UNIX machine and take a look at the file `/etc/services`.

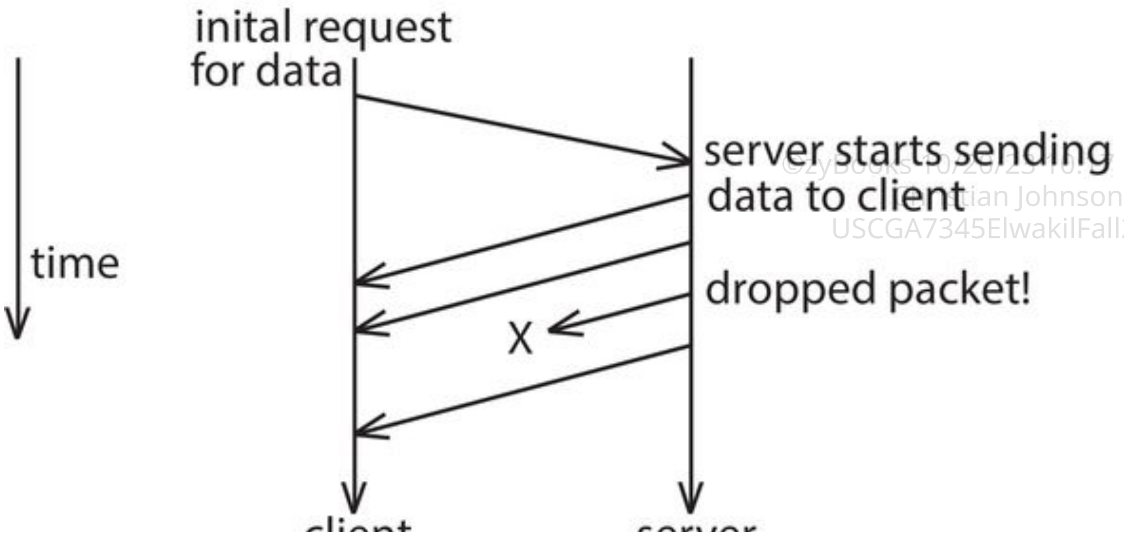
The transport layer can accomplish more than just connecting a network packet to a running process. It can also, if desired, add reliability to a network packet stream. To explain how, we next outline some general behavior of the transport protocols UDP and TCP.

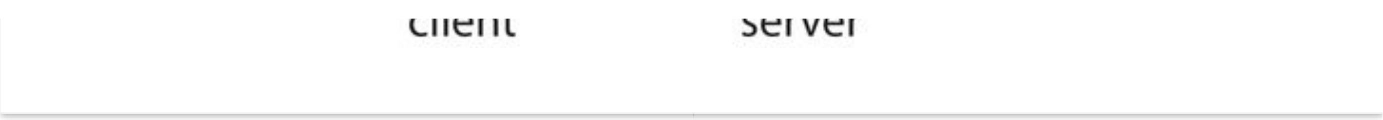
User datagram protocol

The transport protocol UDP is *unreliable* in that it is a bare-bones extension to IP with the addition of a port number. In fact, the UDP header is very simple and contains only four fields: source port number, destination port number, length, and checksum. Packets may be sent quickly to a destination using UDP. However, since there are no guarantees of delivery in the lower layers of the network stack, packets may become lost. Packets can also arrive at the receiver out of order. It is up to the application to figure out these error cases and to adjust (or not adjust).

Figure 19.3.7 illustrates a common scenario involving loss of a packet between a client and a server using the UDP protocol. Note that this protocol is known as a *connectionless* protocol because there is no connection setup at the beginning of the transmission to set up state—the client just starts sending data. Similarly, there is no connection teardown.

Figure 19.3.7: Example of a UDP data transfer with dropped packet.





The client begins by sending some sort of request for information to the server. The server then responds by sending four datagrams, or packets, to the client. Unfortunately, one of the packets is dropped by an overwhelmed router. The client must either make do with only three packets or use logic programmed into the application to request the missing packet. Thus, we need to use a different transport protocol if we want any additional reliability guarantees to be handled by the network.

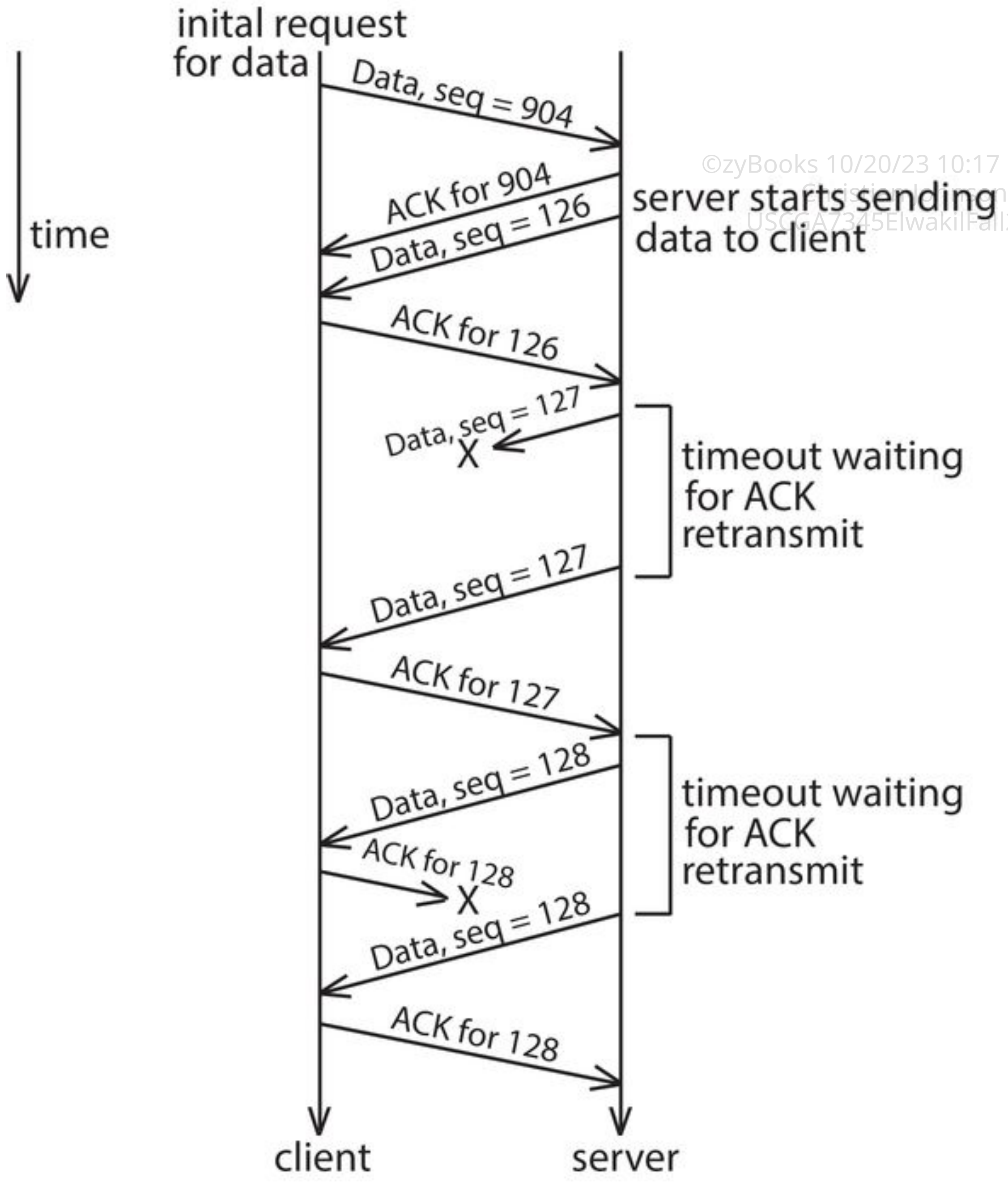
Transmission control protocol

TCP is a transport protocol that is both *reliable* and *connection-oriented*. In addition to specifying port numbers to identify sending and receiving processes on different hosts, TCP provides an abstraction that allows a sending process on one host to send an in-order, uninterrupted *byte stream* across the network to a receiving process on another host. It accomplishes these things through the following mechanisms:

- Whenever a host sends a packet, the receiver must send an ***acknowledgment packet***, or ACK, to notify the sender that the packet was received. If the ACK is not received before a timer expires, the sender will send that packet again.
- TCP introduces ***sequence numbers*** into the TCP header of every packet. These numbers allow the receiver to (1) put packets in order before sending data up to the requesting process and (2) be aware of packets missing from the byte stream.
- TCP connections are initiated with a series of control packets between the sender and the receiver (often called a *three-way handshake*) and closed gracefully with control packets responsible for tearing down the connection. These control packets allow both the sender and the receiver to set up and remove state.

Figure 19.3.8 demonstrates a possible exchange using TCP (with connection setup and tear-down omitted). After the connection has been established, the client sends a request packet to the server with the sequence number 904. Unlike the server in the UDP example, the server must then send an ACK packet back to the client. Next, the server starts sending its own stream of data packets starting with a different sequence number. The client sends an ACK packet for each data packet it receives. Unfortunately, the data packet with the sequence number 127 is lost, and no ACK packet is sent by the client. The sender times out waiting for the ACK packet, so it must resend data packet 127. Later in the connection, the server sends the data packet with the sequence number 128, but the ACK is lost. Since the server does not receive the ACK, it must resend data packet 128. The client then receives a duplicate packet. Because the client knows that it previously received a packet with that sequence number, it throws the duplicate away. However, it must send another ACK back to the server to allow the server to continue.

Figure 19.3.8: Example of a TCP data transfer with dropped packets.



In the actual TCP specification, an ACK isn't required for each and every packet. Instead, the receiver can send a ***cumulative ACK*** to ACK a series of packets. The server can also send numerous data packets sequentially before waiting for ACKs, to take advantage of network throughput.

TCP also helps regulate the flow of packets through mechanisms called *flow control* and *congestion control*. ***Flow control*** involves preventing the sender from overrunning the capacity of the receiver. For example, the receiver may have a slower connection or may have slower hardware components (like a slower network card or processor). Flow-control state can be returned in the

ACK packets of the receiver to alert the sender to slow down or speed up. ***Congestion control*** attempts to approximate the state of the networks (and generally the routers) between the sender and the receiver. If a router becomes overwhelmed with packets, it will tend to drop them. Dropping packets results in ACK timeouts, which results in more packets saturating the network. To prevent this condition, the sender monitors the connection for dropped packets by noticing how many packets are not acknowledged. If there are too many dropped packets, the sender will slow down the rate at which it sends them. This helps ensure that the TCP connection is being fair to other connections happening at the same time.

By utilizing a reliable transport protocol like TCP, a distributed system does not need extra logic to deal with lost or out-of-order packets. However, TCP is slower than UDP.

PARTICIPATION ACTIVITY | 19.3.1: Section review questions.

1) A _______ may be reserved to for a protocol supporting a service running on a host.

○ port number

○ sequence number

○ checksum

2) Both TCP and UDP are reliable transport layer protocols.

○ True

○ False

Section glossary

host name: A human-readable name for a computer.

identifier: Generally, a numeric tag for a device or object. In networking, the unique host-id number identifying a system on a network.

resolve: Generally, to translate from a symbolic representation to a numeric one. In networking, to translate from a host name to a host-id. With files, to follow a link and find the target file.

host-id: In networking, the unique number identifying a system on a network.

domain-name system (DNS): The Internet system for resolving host names to host-ids, in which a distributed information system provides host-name-to-network-address translations for the Internet.

name server: In the domain-name system, a host or software that provides resolving services.

OSI protocol stack: A set of cooperating network protocols that form a fully functional network. The OSI model formalized some of the earlier work done in network protocols but is currently not in widespread use.

user datagram protocol (UDP): A communications protocol layered on IP that is connectionless, is low latency, and does not guarantee delivery.

transmission control protocol/Internet protocol (TCP/IP): The most common network protocol; it provides the fundamental architecture of the Internet.

Internet protocol (IP): The low-level networking protocol on which TCP and UDP are layered; responsible for routing IP datagrams through networks such as the Internet.

datagram: A basic transfer unit on a packet-switched network like the Internet protocol (i.e., a network packet).

Border Gateway Protocol: A network protocol for determining network routes so that packets can move from source to destination across a WAN.

medium access control (MAC) address: A unique byte number assigned to every Ethernet device allowing it to be located by packets sent across a LAN.

broadcast: In networking, the sending of one or more packets to all devices on the local network.

checksum: The general term for an error detection and correction code.

port number: In TCP/IP and UDP/IP networking, an address of a service on a system.

acknowledgment packet: In networking, a packet sent in response to the successful receipt of a message or packet.

sequence number: In networking, a counter assigned to packets to order their assembly after delivery.

flow control: Generally, a method to pause a sender of I/O. In networking, a technique to limit the rate of data flow (e.g., to avoid buffer overflow and packet loss on a router).

congestion control: In networking, the attempt to approximate the state of networks between senders and receivers to avoid packet loss.

19.4 Network and distributed operating systems

In this section, we describe the two general categories of network-oriented operating systems: network operating systems and distributed operating systems. Network operating systems are simpler to implement but generally more difficult for users to access and use than are distributed operating systems, which provide more features.

Network operating systems

A ***network operating system*** provides an environment in which users can access remote resources (implementing resource sharing) by either logging in to the appropriate remote machine or transferring data from the remote machine to their own machines. Currently, all general-purpose operating systems, and even mobile operating systems such as Android and IOS, are network operating systems.

Remote login

An important function of a network operating system is to allow users to log in remotely. The Internet provides the `ssh` facility for this purpose. To illustrate, suppose that a user at Westminster College wishes to compute on `kristen.cs.yale.edu`, a computer located at Yale University. To do so, the user must have a valid account on that machine. To log in remotely, the user issues the command

```
ssh kristen.cs.yale.edu
```

This command results in the formation of an encrypted socket connection between the local machine at Westminster College and the `kristen.cs.yale.edu` computer. After this connection has been established, the networking software creates a transparent, bidirectional link so that all characters entered by the user are sent to a process on `kristen.cs.yale.edu` and all the output from that process is sent back to the user. The process on the remote machine asks the user for a login name and a password. Once the correct information has been received, the process acts as a proxy for the user, who can compute on the remote machine just as any local user can.

Remote file transfer

Another major function of a network operating system is to provide a mechanism for ***remote file transfer*** from one machine to another. In such an environment, each computer maintains its own local file system. If a user at one site (say, Kurt at `albion.edu`) wants to access a file owned by

Becca located on another computer (say, at `colby.edu`), then the file must be copied explicitly from the computer at Colby in Maine to the computer at Albion in Michigan. The communication is one-directional and individual, such that other users at those sites wishing to transfer a file, say Sean at `colby.edu` to Karen at `albion.edu`, must likewise issue a set of commands.

The Internet provides a mechanism for such a transfer with the file transfer protocol (FTP) and the more private secure file transfer protocol (SFTP). Suppose that user Carla at `wesleyan.edu` wants to copy a file that is owned by Owen at `kzoo.edu.` The user must first invoke the `sftp` program by executing

```
sftp owen@kzoo.edu
```

The program then asks the user for a login name and a password. Once the correct information has been received, the user can use a series of commands to upload files, download files, and navigate the remote file system structure. Some of these commands are:

- `get`—Transfer a file from the remote machine to the local machine.
- `put`—Transfer a file from the local machine to the remote machine.
- `ls` or `dir`—List files in the current directory on the remote machine.
- `cd`—Change the current directory on the remote machine.

There are also various commands to change transfer modes (for binary or ASCII files) and to determine connection status.

Cloud storage

Basic cloud-based storage applications allow users to transfer files much as with FTP. Users can upload files to a cloud server, download files to the local computer, and share files with other cloud-service users via a web link or other sharing mechanism through a graphical interface. Common examples include Dropbox and Google Drive.

An important point about SSH, FTP, and cloud-based storage applications is that they require the user to change paradigms. FTP, for example, requires the user to know a command set entirely different from the normal operating-system commands. With SSH, the user must know appropriate commands on the remote system. For instance, a user on a Windows machine who connects remotely to a UNIX machine must switch to UNIX commands for the duration of the SSH session. (In networking, a ***session*** is a complete round of communication, frequently beginning with a login to authenticate and ending with a logoff to terminate the communication.) With cloud-based storage applications, users may have to log into the cloud service (usually through a web browser) or native application and then use a series of graphical commands to upload, download, or share files. Obviously, users would find it more convenient not to be required to use a different set of commands. Distributed operating systems are designed to address this problem.

Distributed operating systems

In a distributed operating system, users access remote resources in the same way they access local resources. Data and process migration from one site to another is under the control of the distributed operating system. Depending on the goals of the system, it can implement data migration, computation migration, process migration, or any combination thereof.

Data migration

Suppose a user on site A wants to access data (such as a file) that reside at site B. The system can transfer the data by one of two basic methods. One approach to ***data migration*** is to transfer the entire file to site A. From that point on, all access to the file is local. When the user no longer needs access to the file, a copy of the file (if it has been modified) is sent back to site B. Even if only a modest change has been made to a large file, all the data must be transferred. This mechanism can be thought of as an automated FTP system. This approach was used in the Andrew file system, but it was found to be too inefficient.

The other approach is to transfer to site A only those portions of the file that are actually *necessary* for the immediate task. If another portion is required later, another transfer will take place. When the user no longer wants to access the file, any part of it that has been modified must be sent back to site B. (Note the similarity to demand paging.) Most modern distributed systems use this approach.

Whichever method is used, data migration includes more than the mere transfer of data from one site to another. The system must also perform various data translations if the two sites involved are not directly compatible (for instance, if they use different character-code representations or represent integers with a different number or order of bits).

Computation migration

In some circumstances, we may want to transfer the computation, rather than the data, across the system; this process is called ***computation migration***. For example, consider a job that needs to access various large files that reside at different sites, to obtain a summary of those files. It would be more efficient to access the files at the sites where they reside and return the desired results to the site that initiated the computation. Generally, if the time to transfer the data is longer than the time to execute the remote command, the remote command should be used.

Such a computation can be carried out in different ways. Suppose that process P wants to access a file at site A. Access to the file is carried out at site A and could be initiated by an RPC. An RPC uses network protocols to execute a routine on a remote system (Section Remote procedure calls). Process P invokes a predefined procedure at site A. The procedure executes appropriately and then returns the results to P.

Alternatively, process P can send a message to site A. The operating system at site A then creates a new process Q whose function is to carry out the designated task. When process Q completes its execution, it sends the needed result back to P via the message system. In this scheme, process P may execute concurrently with process Q. In fact, it may have several processes running concurrently on several sites.

Either method could be used to access several files (or chunks of files) residing at various sites. One RPC might result in the invocation of another RPC or even in the transfer of messages to another site. Similarly, process Q could, during the course of its execution, send a message to another site, which in turn would create another process. This process might either send a message back to Q or repeat the cycle.

Process migration

A logical extension of computation migration is ***process migration***. When a process is submitted for execution, it is not always executed at the site at which it is initiated. The entire process, or parts of it, may be executed at different sites. This scheme may be used for several reasons:

- **Load balancing.** The processes (or subprocesses) may be distributed across the sites to even the workload.
- **Computation speedup.** If a single process can be divided into a number of subprocesses that can run concurrently on different sites or nodes, then the total process turnaround time can be reduced.
- **Hardware preference.** The process may have characteristics that make it more suitable for execution on some specialized processor (such as matrix inversion on a GPU) than on a microprocessor.
- **Software preference.** The process may require software that is available at only a particular site, and either the software cannot be moved, or it is less expensive to move the process.
- **Data access.** Just as in computation migration, if the data being used in the computation are numerous, it may be more efficient to have a process run remotely (say, on a server that hosts a large database) than to transfer all the data and run the process locally.

We use two complementary techniques to move processes in a computer network. In the first, the system can attempt to hide the fact that the process has migrated from the client. The client then need not code her program explicitly to accomplish the migration. This method is usually employed for achieving load balancing and computation speedup among homogeneous systems, as they do not need user input to help them execute programs remotely.

The other approach is to allow (or require) the user to specify explicitly how the process should migrate. This method is usually employed when the process must be moved to satisfy a hardware or software preference.

You have probably realized that the World Wide Web has many aspects of a distributed computing environment. Certainly it provides data migration (between a web server and a web client). It also provides computation migration. For instance, a web client could trigger a database operation on a web server. Finally, with Java, Javascript, and similar languages, it provides a form of process migration: Java applets and Javascript scripts are sent from the server to the client, where they are executed. A network operating system provides most of these features, but a distributed operating system makes them seamless and easily accessible. The result is a powerful and easy-to-use facility—one of the reasons for the huge growth of the World Wide Web.

PARTICIPATION ACTIVITY | 19.4.1: Section review questions.

1) In a ______ operating system, users access remote resources no differently than way they access local resources.

○ network

○ distributed

2) _____ allows load balancing.

○ Data

○ Computation

○ Process

Section glossary

network operating system: A type of operating system that provides features such as file sharing across a network, along with a communication scheme that allows different processes on different computers to exchange messages. It provides an environment in which users can access remote resources by remote login or data transfer between remote and local systems.

remote file transfer: A function of a network operating system providing a means to transfer files between network-attached computers.

session: In networking, a complete round of communication, frequently beginning with a login and ending with a logoff to terminate communications.

data migration: The entire transfer of one or more files from one site to another.

computation migration: The use of a network to allow a task to remotely request resources to speed a computation.

process migration: The movement of a process between computers.

19.5 Design issues in distributed systems

The designers of a distributed system must take a number of design challenges into account. The system should be robust so that it can withstand failures. The system should also be transparent to users in terms of both file location and user mobility. Finally, the system should be scalable to allow the addition of more computation power, more storage, or more users. We briefly introduce these issues here. In the next section, we put them in context when we describe the designs of specific distributed file systems.

Robustness

A distributed system may suffer from various types of hardware failure. The failure of a link, a host, or a site and the loss of a message are the most common types. To ensure that the system is robust, we must detect any of these failures, reconfigure the system so that computation can continue, and recover when the failure is repaired.

A system can be ***fault tolerant*** in that it can tolerate a certain level of failure and continue to function normally. The degree of fault tolerance depends on the design of the distributed system and the specific fault. Obviously, more fault tolerance is better.

We use the term ***fault tolerance*** in a broad sense. Communication faults, certain machine failures, storage-device crashes, and decays of storage media should all be tolerated to some extent. A ***fault-tolerant system*** should continue to function, perhaps in a degraded form, when faced with such failures. The degradation can affect performance, functionality, or both. It should be proportional, however, to the failures that caused it. A system that grinds to a halt when only one of its components fails is certainly not fault tolerant.

Unfortunately, fault tolerance can be difficult and expensive to implement. At the network layer, multiple redundant communication paths and network devices such as switches and routers are needed to avoid a communication failure. A storage failure can cause loss of the operating system, applications, or data. Storage units can include redundant hardware components that automatically take over from each other in case of failure. In addition, RAID systems can ensure continued access to the data even in the event of one or more storage device failures (Section 11.8).

Failure detection

In an environment with no shared memory, we generally cannot differentiate among link failure, site failure, host failure, and message loss. We can usually detect only that one of these failures has occurred. Once a failure has been detected, appropriate action must be taken. What action is appropriate depends on the particular application.

To detect link and site failure, we use a ***heartbeat*** procedure. Suppose that sites A and B have a direct physical link between them. At fixed intervals, the sites send each other an *I-am-up* message. If site A does not receive this message within a predetermined time period, it can assume that site B has failed, that the link between A and B has failed, or that the message from B has been lost. At this point, site A has two choices. It can wait for another time period to receive an *I-am-up* message from B, or it can send an *Are-you-up?* message to B.

If time goes by and site A still has not received an *I-am-up* message, or if site A has sent an *Are-you-up?* message and has not received a reply, the procedure can be repeated. Again, the only conclusion that site A can draw safely is that some type of failure has occurred.

Site A can try to differentiate between link failure and site failure by sending an *Are-you-up?* message to B by another route (if one exists). If and when B receives this message, it immediately replies positively. This positive reply tells A that B is up and that the failure is in the direct link between them. Since we do not know in advance how long it will take the message to travel from A to B and back, we must use a time-out scheme. At the time A sends the *Are-you-up?* message, it specifies a time interval during which it is willing to wait for the reply from B. If A receives the reply message within that time interval, then it can safely conclude that B is up. If not, however (that is, if a time-out occurs), then A may conclude only that one or more of the following situations has occurred:

- Site B is down.
- The direct link (if one exists) from A to B is down.
- The alternative path from A to B is down.
- The message has been lost. (Although the use of a reliable transport protocol such as TCP should eliminate this concern.)

Site A cannot, however, determine which of these events has occurred.

Reconfiguration

Suppose that site A has discovered, through the mechanism just described, that a failure has occurred. It must then initiate a procedure that will allow the system to reconfigure and to continue its normal mode of operation.

- If a direct link from A to B has failed, this information must be broadcast to every site in the system, so that the various routing tables can be updated accordingly.
- If the system believes that a site has failed (because that site can no longer be reached), then all sites in the system must be notified, so that they will no longer attempt to use the services of the failed site. The failure of a site that serves as a central coordinator for some activity (such as deadlock detection) requires the election of a new coordinator. Note that, if the site has not failed (that is, if it is up but cannot be reached), then we may have the undesirable situation in which two sites serve as the coordinator. When the network is partitioned, the two coordinators (each for its own partition) may initiate conflicting actions. For example, if the coordinators are responsible for implementing mutual exclusion, we may have a situation in

which two processes are executing simultaneously in their critical sections.

Recovery from failure

When a failed link or site is repaired, it must be integrated into the system gracefully and smoothly.

- Suppose that a link between A and B has failed. When it is repaired, both A and B must be notified. We can accomplish this notification by continuously repeating the heartbeat procedure described in Section Failure detection.

- Suppose that site B has failed. When it recovers, it must notify all other sites that it is up again. Site B then may have to receive information from the other sites to update its local tables. For example, it may need routing-table information, a list of sites that are down, undelivered messages, a transaction log of unexecuted transactions, and mail. If the site has not failed but simply cannot be reached, then it still needs this information.

Transparency

Making the multiple processors and storage devices in a distributed system ***transparent*** to the users has been a key challenge to many designers. Ideally, a distributed system should look to its users like a conventional, centralized system. The user interface of a transparent distributed system should not distinguish between local and remote resources. That is, users should be able to access remote resources as though these resources were local, and the distributed system should be responsible for locating the resources and for arranging for the appropriate interaction.

Another aspect of transparency is user mobility. It would be convenient to allow users to log into any machine in the system rather than forcing them to use a specific machine. A transparent distributed system facilitates user mobility by bringing over a user's environment (for example, home directory) to wherever he logs in. Protocols like LDAP provide an authentication system for local, remote, and mobile users. Once the authentication is complete, facilities like desktop virtualization allow users to see their desktop sessions at remote facilities.

Scalability

Still another issue is ***scalability***—the capability of a system to adapt to increased service load. Systems have bounded resources and can become completely saturated under increased load. For example, with respect to a file system, saturation occurs either when a server's CPU runs at a high utilization rate or when disks' I/O requests overwhelm the I/O subsystem. Scalability is a relative property, but it can be measured accurately. A scalable system reacts more gracefully to increased load than does a nonscalable one. First, its performance degrades more moderately; and second, its resources reach a saturated state later. Even perfect design however cannot accommodate an ever-growing load. Adding new resources might solve the problem, but it might generate additional indirect load on other resources (for example, adding machines to a distributed system can clog the network and increase service loads). Even worse, expanding the system can call for expensive design modifications. A scalable system should have the potential to grow without these problems. In a distributed system, the ability to scale up gracefully is of special importance, since expanding a network by adding new machines or interconnecting two networks is commonplace. In short, a scalable design should withstand high service load, accommodate growth of the user community, and allow simple integration of added resources.

Scalability is related to fault tolerance, discussed earlier. A heavily loaded component can become paralyzed and behave like a faulty component. In addition, shifting the load from a faulty component to that component's backup can saturate the latter. Generally, having spare resources is essential for ensuring reliability as well as for handling peak loads gracefully. Thus, the multiple resources in a distributed system represent an inherent advantage, giving the system a greater potential for fault tolerance and scalability. However, inappropriate design can obscure this potential. Fault-tolerance and scalability considerations call for a design demonstrating distribution of control and data.

Scalability can also be related to efficient storage schemes. For example, many cloud storage providers use ***compression*** or ***deduplication*** to cut down on the amount of storage used. *Compression* reduces the size of a file. For example, a zip archive file can be generated out of a file (or files) by executing a `zip` command, which runs a lossless compression algorithm over the data specified. (***Lossless compression*** allows original data to be perfectly reconstructed from compressed data.) The result is a file archive that is smaller than the uncompressed file. To restore the file to its original state, a user runs some sort of `unzip` command over the zip archive file. *Deduplication* seeks to lower data storage requirements by removing redundant data. With this technology, only one instance of data is stored across an entire system (even across data owned by multiple users). Both compression and deduplication can be performed at the file level or the block level, and they can be used together. These techniques can be automatically built into a distributed system to compress information without users explicitly issuing commands, thereby saving storage space and possibly cutting down on network communication costs without adding user complexity.

PARTICIPATION ACTIVITY 19.5.1: Section review questions.

1) A fault-tolerant system will have no decline in service in the case of a node failure.

○ True

○ False

2) A heartbeat is used ______.

○ to detect a link or site failure in a distributed system

○ only if a node in a distributed

○ system fails

○ to reconfigure a distributed system when a node fails

3) For what purposes cloud storage systems use deduplication:

○ To recover from failure

○ To support transparency

○ For storage space management

Section glossary

fault-tolerant system: A system that can suffer a failure of any single component and still continue operation.

heartbeat: In computer clustering, a repeating signal to determine the state of the members of the cluster (e.g., to determine if a system is down).

transparent: In distributed computing, a term describing the automatic sharing of resources so that users do not know if a resource is local or remote.

scalability: Generally, the ability of a facility or service to increase capacity as needed by the users (e.g., to add more cores when the load increases).

compression: The use of algorithms to reduce the amount of data stored or sent.

deduplication: The removal of duplicate information (bits, blocks, or files).

19.6 Distributed file systems

Although the World Wide Web is the predominant distributed system in use today, it is not the only one. Another important and popular use of distributed computing is the ***distributed file system***, or *DFS*.

To explain the structure of a DFS, we need to define the terms *service*, *server*, and *client* in the DFS context. A ***service*** is a software entity running on one or more machines and providing a particular type of function to clients. A ***server*** is the service software running on a single machine. A ***client*** is a process that can invoke a service using a set of operations that form its ***client interface***. Sometimes a lower-level interface is defined for the actual cross-machine interaction; it is the ***intermachine interface***.

Using this terminology, we say that a file system provides file services to clients. A client interface for a file service is formed by a set of primitive file operations, such as create a file, delete a file, read from a file, and write to a file. The primary hardware component that a file server controls is a set of local secondary-storage devices (usually, hard disks or solid-state drives) on which files are stored and from which they are retrieved according to the clients' requests.

A DFS is a file system whose clients, servers, and storage devices are dispersed among the machines of a distributed system. Accordingly, service activity has to be carried out across the network. Instead of a single centralized data repository, the system frequently has multiple and independent storage devices. As you will see, the concrete configuration and implementation of a DFS may vary from system to system. In some configurations, servers run on dedicated machines. In others, a machine can be both a server and a client.

The distinctive features of a DFS are the multiplicity and autonomy of clients and servers in the system. Ideally, though, a DFS should appear to its clients to be a conventional, centralized file system. That is, the client interface of a DFS should not distinguish between local and remote files. It is up to the DFS to locate the files and to arrange for the transport of the data. A *transparent* DFS—like the transparent distributed systems mentioned earlier—facilitates user mobility by bringing a user's environment (for example, the user's home directory) to wherever the user logs in.

The most important performance measure of a DFS is the amount of time needed to satisfy service requests. In conventional systems, this time consists of storage-access time and a small amount of CPU-processing time. In a DFS, however, a remote access has the additional overhead associated with the distributed structure. This overhead includes the time to deliver the request to a server, as well as the time to get the response across the network back to the client. For each direction, in addition to the transfer of the information, there is the CPU overhead of running the communication protocol software. The performance of a DFS can be viewed as another dimension of the DFS's transparency. That is, the performance of an ideal DFS would be comparable to that of a conventional file system.

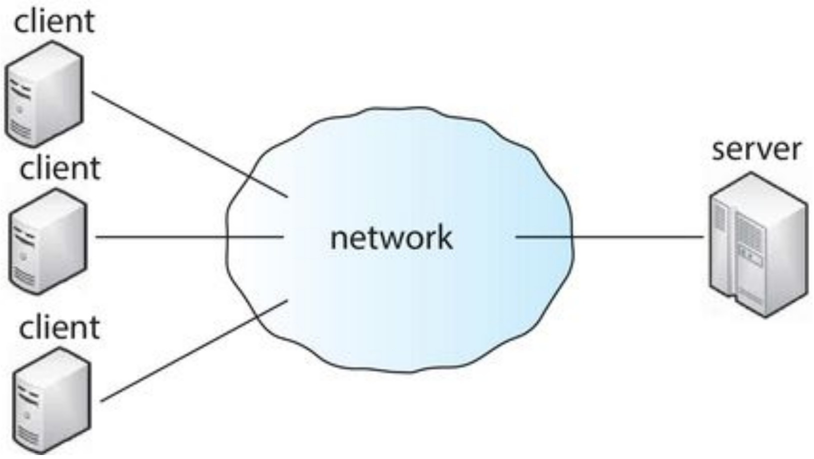
The basic architecture of a DFS depends on its ultimate goals. Two widely used architectural models we discuss here are the ***client—server model*** and the ***cluster-based model***. The main goal of a client-server architecture is to allow transparent file sharing among one or more clients as if the files were stored locally on the individual client machines. The distributed file systems NFS and OpenAFS are prime examples. NFS is the most common UNIX-based DFS. It has several versions, and here we refer to NFS Version 3 unless otherwise noted.

If many applications need to be run in parallel on large data sets with high availability and scalability, the cluster-based model is more appropriate than the client-server model. Two well-known examples are the Google file system and the open-source HDFS, which runs as part of the Hadoop framework.

The client-server DFS model

Figure 19.6.1 illustrates a simple DFS ***client-server model***. The server stores both files and metadata on attached storage. In some systems, more than one server can be used to store different files. Clients are connected to the server through a network and can request access to files in the DFS by contacting the server through a well-known protocol such as NFS Version 3. The server is responsible for carrying out authentication, checking the requested file permissions, and, if warranted, delivering the file to the requesting client. When a client makes changes to the file, the client must somehow deliver those changes to the server (which holds the main copy of the file). The client's and the server's versions of the file should be kept consistent in a way that minimizes network traffic and the server's workload to the extent possible.

Figure 19.6.1: Client-server DFS model.



The ***network file system*** (*NFS*) protocol was originally developed by Sun Microsystems as an open protocol, which encouraged early adoption across different architectures and systems. From the beginning, the focus of NFS was simple and fast crash recovery in the face of server failure. To implement this goal, the NFS server was designed to be stateless; it does not keep track of which client is accessing which file or of things such as open file descriptors and file pointers. This means that, whenever a client issues a file operation (say, to read a file), that operation has to be idempotent in the face of server crashes. ***Idempotent*** describes an operation that can be issued more than once yet return the same result. In the case of a read operation, the client keeps track of the state (such as the file pointer) and can simply reissue the operation if the server has crashed and come back online. You can read more about the NFS implementation in Section 15.8.

The ***Andrew file system*** (*OpenAFS*) was created at Carnegie Mellon University with a focus on scalability. Specifically, the researchers wanted to design a protocol that would allow the server to support as many clients as possible. This meant minimizing requests and traffic to the server.

When a client requests a file, the file's contents are downloaded from the server and stored on the client's local storage. Updates to the file are sent to the server when the file is closed, and new versions of the file are sent to the client when the file is opened. In comparison, NFS is quite chatty and will send block read and write requests to the server as the file is being used by a client.

Both OpenAFS and NFS are meant to be used in addition to local file systems. In other words, you would not format a hard drive partition with the NFS file system. Instead, on the server, you would format the partition with a local file system of your choosing, such as ext4, and export the shared directories via the DFS. In the client, you would simply attach the exported directories to your file-system tree. In this way, the DFS can be separated from responsibility for the local file system and can concentrate on distributed tasks.

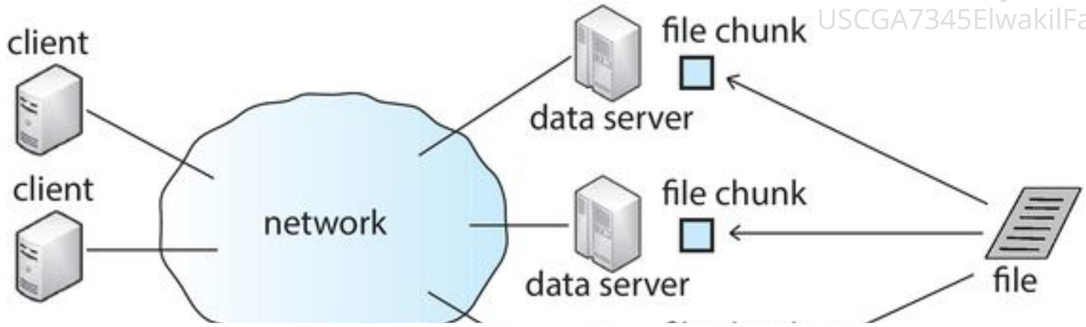
The DFS client-server model, by design, may suffer from a single point of failure if the server crashes. Computer clustering can help resolve this problem by using redundant components and clustering methods such that failures are detected and failing over to working components continues server operations. In addition, the server presents a bottleneck for all requests for both data and metadata, which results in problems of scalability and bandwidth.

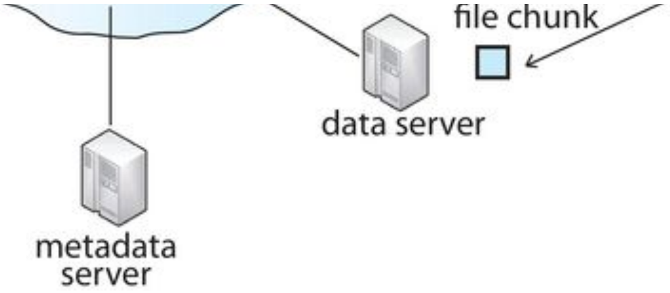
The cluster-based DFS model

As the amount of data, I/O workload, and processing expands, so does the need for a DFS to be fault-tolerant and scalable. Large bottlenecks cannot be tolerated, and system component failures must be expected. Cluster-based architecture was developed in part to meet these needs.

Figure 19.6.2 illustrates a sample cluster-based DFS model. This is the basic model presented by the ***Google file system*** (*GFS*) and the ***Hadoop distributed file system*** (*HDFS*). One or more clients are connected via a network to a main metadata server and several data servers that house "chunks" (or portions) of files. The metadata server keeps a mapping of which data servers hold chunks of which files, as well as a traditional hierarchical mapping of directories and files. Each file chunk is stored on a data server and is replicated a certain number of times (for example, three times) to protect against component failure and for faster access to the data (servers containing the replicated chunks have fast access to those chunks).

Figure 19.6.2: An example of a cluster-based DFS model





To obtain access to a file, a client must first contact the metadata server. The metadata server then returns to the client the identities of the data servers that hold the requested file chunks. The client can then contact the closest data server (or servers) to receive the file information. Different chunks of the file can be read or written to in parallel if they are stored on different data servers, and the metadata server may need to be contacted only once in the entire process. This makes the metadata server less likely to be a performance bottleneck. The metadata server is also responsible for redistributing and balancing the file chunks among the data servers.

GFS was released in 2003 to support large distributed data-intensive applications. The design of GFS was influenced by four main observations:

- Hardware component failures are the norm rather than the exception and should be routinely expected.
- Files stored on such a system are very large.
- Most files are changed by appending new data to the end of the file rather than overwriting existing data.
- Redesigning the applications and file system API increases the system's flexibility.

Consistent with the fourth observation, GFS exports its own API and requires applications to be programmed with this API.

Shortly after developing GFS, Google developed a modularized software layer called ***MapReduce*** to sit on top of GFS. MapReduce allows developers to carry out large-scale parallel computations more easily and utilizes the benefits of the lower-layer file system. Later, HDFS and the Hadoop framework (which includes stackable modules like MapReduce on top of HDFS) were created based on Google's work. Like GFS and MapReduce, Hadoop supports the processing of large data sets in distributed computing environments. As suggested earlier, the drive for such a framework occurred because traditional systems could not scale to the capacity and performance needed by "big data" projects (at least not at reasonable prices). Examples of big data projects include crawling and analyzing social media, customer data, and large amounts of scientific data points for trends.

PARTICIPATION ACTIVITY 19.6.1: Section review questions.

1) The ______ DFS model stores files and

metadata on the same server.

○ client-server

○ cluster-based

2) Which of the following file systems are client-server based?

○ Google file system

○ Andrew file system

○ Hadoop file system

Section glossary

distributed file system (DFS): A file system that works across a network in which remote directories are visible from a local machine.

service: A software entity running on one or more machines and providing a particular type of function to calling clients. In Android, an application component with no user interface; it runs in the background while executing long-running operations or performing work for remote processes.

server: In general, any computer, no matter the size, that provides resources to other computers.

client: A computer that uses services from other computers (such as a web client). The source of a communication.

client interface: In distributed computing, a set of services provided to a caller of services.

intermachine interface: In distributed computing, a set of low-level functions for cross-machine interaction.

client—server model: A mode of computing in which a server provides services to one or more clients. In distributed computing, a model in which a computer acts as a resource server to other computers that are clients of those resources.

cluster-based model: In distributed computing, a model of resource sharing where all systems are equal users and providers of resources.

client-server model: A mode of computing in which a server provides services to one or more clients. In distributed computing, a model in which a computer acts as a resource server to other computers that are clients of those resources.

network file system (NFS): A common network file system used by UNIX, Linux, and other operating systems to share files across a network.

idempotent: Describes a function that, when applied more than once, has the same result every time.

AFS (OpenAFS): A network file system designed at Carnegie Mellon University with the goal of enabling servers to support as many clients as possible.

Google file system (GFS): A cluster-based distributed file system designed and used by Google.

Hadoop distributed file system (HDFS): A cluster-based distributed file system used for big-data projects.

MapReduce: A Google-created big data programming model and implementation for parallel processing across nodes in a distributed cluster. A layer on top of the Google file system (GFS), it allows developers to carry out large-scale parallel computations easily.

19.7 DFS naming and transparency

Naming is a mapping between logical and physical objects. For instance, users deal with logical data objects represented by file names, whereas the system manipulates physical blocks of data stored on disk tracks. Usually, a user refers to a file by a textual name. The latter is mapped to a lower-level numerical identifier that in turn is mapped to disk blocks. This multilevel mapping provides users with an abstraction of a file that hides the details of how and where on the disk the file is stored.

In a transparent DFS, a new dimension is added to the abstraction: that of hiding where in the network the file is located. In a conventional file system, the range of the naming mapping is an address within a disk. In a DFS, this range is expanded to include the specific machine on whose disk the file is stored. Going one step further with the concept of treating files as abstractions leads to the possibility of ***file replication***. Given a file name, the mapping returns a set of the locations of this file's replicas. In this abstraction, both the existence of multiple copies and their locations are hidden.

Naming structures

We need to differentiate two related notions regarding name mappings in a DFS:

1. ***Location transparency***. The name of a file does not reveal any hint of the file's physical storage location.

2. ***Location independence***. The name of a file need not be changed when the file's physical storage location changes.

Both definitions relate to the level of naming discussed previously, since files have different names at different levels (that is, user-level textual names and system-level numerical identifiers). A location-independent naming scheme is a dynamic mapping, since it can map the same file name to different locations at two different times. Therefore, location independence is a stronger property than location transparency.

In practice, most of the current DFSs provide a static, location-transparent mapping for user-level names. Some support ***file migration***—that is, changing the location of a file automatically, providing location independence. OpenAFS supports location independence and file mobility, for example. HDFS includes file migration but does so without following POSIX standards, providing more flexibility in implementation and interface. HDFS keeps track of the location of data but hides this information from clients. This dynamic location transparency allows the underlying mechanism to self-tune. In another example, Amazon's S3 cloud storage facility provides blocks of storage on demand via APIs, placing the storage where it sees fit and moving the data as necessary to meet performance, reliability, and capacity requirements.

A few aspects can further differentiate location independence and static location transparency:

- Divorce of data from location, as exhibited by location independence, provides a better abstraction for files. A file name should denote the file's most significant attributes, which are its contents rather than its location. Location-independent files can be viewed as logical data containers that are not attached to a specific storage location. If only static location transparency is supported, the file name still denotes a specific, although hidden, set of physical disk blocks.
- Static location transparency provides users with a convenient way to share data. Users can share remote files by simply naming the files in a location-transparent manner, as though the files were local. Dropbox and other cloud-based storage solutions work this way. Location independence promotes sharing the storage space itself, as well as the data objects. When files can be mobilized, the overall, system-wide storage space looks like a single virtual resource. A possible benefit is the ability to balance the utilization of storage across the system.
- Location independence separates the naming hierarchy from the storage-devices hierarchy and from the intercomputer structure. By contrast, if static location transparency is used (although names are transparent), we can easily expose the correspondence between component units and machines. The machines are configured in a pattern similar to the naming structure. This configuration may restrict the architecture of the system unnecessarily and conflict with other considerations. A server in charge of a root directory is an example of a structure that is dictated by the naming hierarchy and contradicts decentralization guidelines.

Once the separation of name and location has been completed, clients can access files residing on remote server systems. In fact, these clients may be ***diskless*** and rely on servers to provide all files, including the operating-system kernel. Special protocols are needed for the boot sequence, however. Consider the problem of getting the kernel to a diskless workstation. The diskless workstation has no kernel, so it cannot use the DFS code to retrieve the kernel. Instead, a special boot protocol, stored in read-only memory (ROM) on the client, is invoked. It enables networking and retrieves only one special file (the kernel or boot code) from a fixed location. Once the kernel is copied over the network and loaded, its DFS makes all the other operating-system files available. The advantages of diskless clients are many, including lower cost (because the client machines require no disks) and greater convenience (when an operating-system upgrade occurs, only the server needs to be modified). The disadvantages are the added complexity of the boot protocols and the performance loss resulting from the use of a network rather than a local disk.

Naming schemes

There are three main approaches to naming schemes in a DFS. In the simplest approach, a file is identified by some combination of its host name and local name, which guarantees a unique system-wide name. In Ibis, for instance, a file is identified uniquely by the name *host:local-name*, where *local-name* is a UNIX-like path. The Internet URL system also uses this approach. This naming scheme is neither location transparent nor location independent. The DFS is structured as a collection of isolated component units, each of which is an entire conventional file system. Component units remain isolated, although means are provided to refer to remote files. We do not consider this scheme any further here.

The second approach was popularized by NFS. NFS provides a means to attach remote directories to local directories, thus giving the appearance of a coherent directory tree. Early NFS versions allowed only previously mounted remote directories to be accessed transparently. The advent of the ***automount*** feature allowed mounts to be done on demand based on a table of mount points and file-structure names. Components are integrated to support transparent sharing, but this integration is limited and is not uniform, because each machine may attach different remote directories to its tree. The resulting structure is versatile.

We can achieve total integration of the component file systems by using a third approach. Here, a single global name structure spans all the files in the system. OpenAFS provides a single global namespace for the files and directories it exports, allowing a similar user experience across different client machines. Ideally, the composed file-system structure is the same as the structure of a conventional file system. In practice, however, the many special files (for example, UNIX device files and machine-specific binary directories) make this goal difficult to attain.

To evaluate naming structures, we look at their administrative complexity. The most complex and most difficult-to-maintain structure is the NFS structure. Because any remote directory can be attached anywhere on the local directory tree, the resulting hierarchy can be highly unstructured. If a server becomes unavailable, some arbitrary set of directories on different machines becomes unavailable. In addition, a separate accreditation mechanism controls which machine is allowed to attach which directory to its tree. Thus, a user might be able to access a remote directory tree on one client but be denied access on another client.

Implementation techniques

Implementation of transparent naming requires a provision for the mapping of a file name to the associated location. To keep this mapping manageable, we must aggregate sets of files into component units and provide the mapping on a component-unit basis rather than on a single-file basis. This aggregation serves administrative purposes as well. UNIX-like systems use the hierarchical directory tree to provide name-to-location mapping and to aggregate files recursively into directories.

To enhance the availability of the crucial mapping information, we can use replication, local caching, or both. As we noted, location independence means that the mapping changes over time. Hence, replicating the mapping makes a simple yet consistent update of this information impossible. To overcome this obstacle, we can introduce low-level, ***location-independent file identifiers***. (OpenAFS uses this approach.) Textual file names are mapped to lower-level file identifiers that indicate to which component unit the file belongs. These identifiers are still location independent. They can be replicated and cached freely without being invalidated by migration of component units. The inevitable price is the need for a second level of mapping, which maps component units to locations and needs a simple yet consistent update mechanism. Implementing UNIX-like directory trees using these low-level, location-independent identifiers makes the whole hierarchy invariant under component-unit migration. The only aspect that does change is the component-unit location mapping.

A common way to implement low-level identifiers is to use structured names. These names are bit strings that usually have two parts. The first part identifies the component unit to which the file belongs; the second part identifies the particular file within the unit. Variants with more parts are possible. The invariant of structured names, however, is that individual parts of the name are unique at all times only within the context of the rest of the parts. We can obtain uniqueness at all times by taking care not to reuse a name that is still in use, by adding sufficiently more bits (this method is used in OpenAFS), or by using a timestamp as one part of the name (as was done in Apollo Domain). Another way to view this process is that we are taking a location-transparent system, such as Ibis, and adding another level of abstraction to produce a location-independent naming scheme.

PARTICIPATION ACTIVITY | 19.7.1: Section review questions.

1) Naming is a mapping between logical and physical objects.

○ True

○ False

2) Location ______ means the name of a file does not reveal the physical storage location of a file.

○ independence

○ transparency

Section glossary

naming: In distributed computing, the mapping between logical and physical objects.

location transparency: In distributed computing, a feature in which the name of an object does not reveal its physical location.

location independence: In distributed computing, a feature in which the name of an object does not need to be changed when the object's physical location changes.

file migration: A file system feature in which a file's location is changed automatically by the system.

diskless: A term describing systems that have no local storage.

automount: A network/distributed file system feature in which file systems from remote servers are automatically located and mounted as needed.

19.8 Remote file access

Next, let's consider a user who requests access to a remote file. The server storing the file has been located by the naming scheme, and now the actual data transfer must take place.

One way to achieve this transfer is through a ***remote-service mechanism***, whereby requests for accesses are delivered to the server, the server machine performs the accesses, and their results are forwarded back to the user. One of the most common ways of implementing remote service is the RPC paradigm, which we discussed in chapter Processes. A direct analogy exists between disk-access methods in conventional file systems and the remote-service method in a DFS: using the remote-service method is analogous to performing a disk access for each access request.

To ensure reasonable performance of a remote-service mechanism, we can use a form of caching. In conventional file systems, the rationale for caching is to reduce disk I/O (thereby increasing performance), whereas in DFSs, the goal is to reduce both network traffic and disk I/O. In the following discussion, we describe the implementation of caching in a DFS and contrast it with the basic remote-service paradigm.

Basic caching scheme

The concept of caching is simple. If the data needed to satisfy the access request are not already cached, then a copy of the data is brought from the server to the client system. Accesses are performed on the cached copy. The idea is to retain recently accessed disk blocks in the cache, so that repeated accesses to the same information can be handled locally, without additional network traffic. A replacement policy (for example, the least-recently-used algorithm) keeps the cache size bounded. No direct correspondence exists between accesses and traffic to the server. Files are still identified with one main copy residing at the server machine, but copies (or parts) of the file are scattered in different caches. When a cached copy is modified, the changes need to be reflected on the main copy to preserve the relevant consistency semantics. The problem of keeping the cached copies consistent with the main file is the ***cache-consistency problem***, which we discuss in Section Consistency. DFS caching could just as easily be called ***network virtual memory***. It acts similarly to demand-paged virtual memory, except that the backing store usually is a remote server rather than a local disk. NFS allows the swap space to be mounted remotely, so it actually can implement virtual memory over a network, though with a resulting performance penalty.

The granularity of the cached data in a DFS can vary from blocks of a file to an entire file. Usually, more data are cached than are needed to satisfy a single access, so that many accesses can be served by the cached data. This procedure is much like disk read-ahead (Section Performance). OpenAFS caches files in large chunks (64 KB). The other systems discussed here support caching of individual blocks driven by client demand. Increasing the caching unit increases the hit ratio, but it also increases the miss penalty, because each miss requires more data to be transferred. It increases the potential for consistency problems as well. Selecting the unit of caching involves considering parameters such as the network transfer unit and the RPC protocol service unit (if an RPC protocol is used). The network transfer unit (for Ethernet, a packet) is about 1.5 KB, so larger units of cached data need to be disassembled for delivery and reassembled on reception.

Block size and total cache size are obviously of importance for block-caching schemes. In UNIX-like systems, common block sizes are 4 KB and 8 KB. For large caches (over 1 MB), large block sizes (over 8 KB) are beneficial. For smaller caches, large block sizes are less beneficial because they result in fewer blocks in the cache and a lower hit ratio.

Cache location

Where should the cached data be stored—on disk or in main memory? Disk caches have one clear advantage over main-memory caches: they are reliable. Modifications to cached data are lost in a crash if the cache is kept in volatile memory. Moreover, if the cached data are kept on disk, they are still there during recovery, and there is no need to fetch them again. Main-memory caches have several advantages of their own, however:

- Main-memory caches permit workstations to be diskless.
- Data can be accessed more quickly from a cache in main memory than from one on a disk.
- Technology is moving toward larger and less expensive memory. The resulting performance speedup is predicted to outweigh the advantages of disk caches.
- The server caches (used to speed up disk I/O) will be in main memory regardless of where user caches are located; if we use main-memory caches on the user machine, too, we can build a single caching mechanism for use by both servers and users.

Many remote-access implementations can be thought of as hybrids of caching and remote service. In NFS, for instance, the implementation is based on remote service but is augmented with client- and server-side memory caching for performance. Thus, to evaluate the two methods, we must evaluate the degree to which either method is emphasized. The NFS protocol and most implementations do not provide disk caching (but OpenAFS does).

Cache-update policy

The policy used to write modified data blocks back to the server's main copy has a critical effect on the system's performance and reliability. The simplest policy is to write data through to disk as soon as they are placed in any cache. The advantage of a ***write-through policy*** is reliability: little information is lost when a client system crashes. However, this policy requires each write access to wait until the information is sent to the server, so it causes poor write performance. Caching with write-through is equivalent to using remote service for write accesses and exploiting caching only for read accesses.

An alternative is the ***delayed-write policy***, also known as ***write-back caching***, where we delay updates to the main copy. Modifications are written to the cache and then are written through to the server at a later time. This policy has two advantages over write-through. First, because writes are made to the cache, write accesses complete much more quickly. Second, data may be overwritten before they are written back, in which case only the last update needs to be written at all. Unfortunately, delayed-write schemes introduce reliability problems, since unwritten data are lost whenever a user machine crashes.

Variations of the delayed-write policy differ in when modified data blocks are flushed to the server. One alternative is to flush a block when it is about to be ejected from the client's cache. This option can result in good performance, but some blocks can reside in the client's cache a long time before they are written back to the server. A compromise between this alternative and the write-through policy is to scan the cache at regular intervals and to flush blocks that have been modified since the most recent scan, just as UNIX scans its local cache. NFS uses the policy for file data, but once a write is issued to the server during a cache flush, the write must reach the server's disk before it is considered complete. NFS treats metadata (directory data and file-attribute data) differently. Any metadata changes are issued synchronously to the server. Thus, file-structure loss and directory-structure corruption are avoided when a client or the server crashes.

Yet another variation on delayed write is to write data back to the server when the file is closed. This ***write-on-close policy*** is used in OpenAFS. In the case of files that are open for short periods or are modified rarely, this policy does not significantly reduce network traffic. In addition, the write-on-close policy requires the closing process to delay while the file is written through, which reduces the performance advantages of delayed writes. For files that are open for long periods and are modified frequently, however, the performance advantages of this policy over delayed write with more frequent flushing are apparent.

Consistency

A client machine is sometimes faced with the problem of deciding whether a locally cached copy of data is consistent with the main copy (and hence can be used). If the client machine determines that its cached data are out of date, it must cache an up-to-date copy of the data before allowing further accesses. There are two approaches to verifying the validity of cached data:

1. **Client-initiated approach.** The client initiates a validity check in which it contacts the server and checks whether the local data are consistent with the main copy. The frequency of the validity checking is the crux of this approach and determines the resulting consistency semantics. It can range from a check before every access to a check only on first access to a file (on file open, basically). Every access coupled with a validity check is delayed, compared with an access served immediately by the cache. Alternatively, checks can be initiated at fixed time intervals. Depending on its frequency, the validity check can load both the network and the server.

2. **Server-initiated approach.** The server records, for each client, the files (or parts of files) that it caches. When the server detects a potential inconsistency, it must react. A potential for inconsistency occurs when two different clients in conflicting modes cache a file. If UNIX semantics (Section 15.7) is implemented, we can resolve the potential inconsistency by having the server play an active role. The server must be notified whenever a file is opened, and the intended mode (read or write) must be indicated for every open. The server can then act when it detects that a file has been opened simultaneously in conflicting modes by disabling caching for that particular file. Actually, disabling caching results in switching to a remote-service mode of operation.

In a cluster-based DFS, the cache-consistency issue is made more complicated by the presence of a metadata server and several replicated file data chunks across several data servers. Using our earlier examples of HDFS and GFS, we can compare some differences. HDFS allows append-only write operations (no random writes) and a single file writer, while GFS does allow random writes with concurrent writers. This greatly complicates write consistency guarantees for GFS while simplifying them for HDFS.

PARTICIPATION ACTIVITY 19.8.1: Section review questions.

1) Because more data are cached than are needed to satisfy a single access, this means ________.

○ caching is inefficient in distributed file systems

○ many accesses can be served by the cached data

○ more network accesses are required

2) A cache update policy, where we delay updates to the main copy is known as:

○ write-through caching

○ write-on-close caching

○ write-back caching

Section glossary

remote-service mechanism: A facility, implemented by a feature such as RPC, in which clients ask a remote system to perform a function for them.

cache-consistency problem: A challenge in caching in which the cached copies of data must be kept consistent with the main copy of the data.

network virtual memory: A distributed computing feature similar to virtual memory but with the backing store on a remote system.

write-through policy: In caching, a policy whereby writes are not cached but are written through the cache to the main copy of the data.

delayed-write policy: In caching, a policy whereby data are first written to a cache; later, the cache writes the change to the main copy of the data.

write-back caching: In caching, a policy whereby data are first written to the cache; later, the cache writes the change to the main copy of the data.

write-on-close policy: In caching, a policy whereby writes reside in the cache until the file is closed and are only then written back to the main copy of the data.

19.9 Final thoughts on distributed file systems

The line between DFS client-server and cluster-based architectures is blurring. The NFS Version 4.1 specification includes a protocol for a parallel version of NFS called pNFS, but as of this writing, adoption is slow.

GFS, HDFS, and other large-scale DFSs export a non-POSIX API, so they cannot transparently map directories to regular user machines as NFS and OpenAFS do. Rather, for systems to access these DFSs, they need client code installed. However, other software layers are rapidly being developed to allow NFS to be mounted on top of such DFSs. This is attractive, as it would take advantage of the scalability and other advantages of cluster-based DFSs while still allowing native operating-system utilities and users to access files directly on the DFS.

As of this writing, the open-source HDFS NFS Gateway supports NFS Version 3 and works as a proxy between HDFS and the NFS server software. Since HDFS currently does not support random writes, the HDFS NFS Gateway also does not support this capability. That means a file must be deleted and recreated from scratch even if only one byte is changed. Commercial organizations and researchers are addressing this problem and building stackable frameworks that allow stacking of a DFS, parallel computing modules (such as MapReduce), distributed databases, and exported file volumes through NFS.

One other type of file system, less complex than a cluster-based DFS but more complex than a client-server DFS, is a ***clustered file system*** (*CFS*) or ***parallel file system*** (*PFS*). A CFS typically runs over a LAN. These systems are important and widely used and thus deserve mention here, though we do not cover them in detail. Common CFSs include ***Lustre*** and ***GPFS***, although there are many others. A CFS essentially treats ***N*** systems storing data and ***Y*** systems accessing that data as a single client-server instance. Whereas NFS, for example, has per-server naming, and two separate NFS servers generally provide two different naming schemes, a CFS knits various storage contents on various storage devices on various servers into a uniform, transparent name space. GPFS has its own file-system structure, but Lustre uses existing file systems such as ZFS for file storage and management. To learn more, see http://lustre.org.

Distributed file systems are in common use today, providing file sharing within LANs, within cluster environments, and across WANs. The complexity of implementing such a system should not be underestimated, especially considering that the DFS must be operating-system independent for widespread adoption and must provide availability and good performance in the presence of long distances, commodity hardware failures, sometimes frail networking, and ever-increasing users and workloads.

PARTICIPATION ACTIVITY | 19.9.1: Section review questions.

1) A clustered file system or parallel file system is a type of a cluster-based DFS.

○ True

○ False

2) The distinction between DFS client–server and cluster-based architectures is very clear in current DFS technology.

○ True

○ False

Section glossary

clustered file system (CFS): A file system that is LAN-based and treats N systems storing data and Y clients accessing the data as a single client-server instance; more complex than a client-server DFS but less complex than a cluster-based DFS. GPFS and Lustre are examples.

parallel file system (PFS): A file system that is LAN-based and treats N systems storing data and Y clients accessing the data as a single client-server instance; more complex than a client-server DFS but less complex than a cluster-based DFS. GPFS and Lustre are examples.

Lustre: A common open-source clustered file system.

GPFS: A common commercial clustered file system by IBM.

19.10 Summary

- A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with one another through various communication lines, such as high-speed buses and the Internet. The processors in a distributed system vary in size and function.
- A distributed system provides the user with access to all system resources. Access to a

shared resource can be provided by data migration, computation migration, or process migration. The access can be specified by the user or implicitly supplied by the operating system and applications.

- Protocol stacks, as specified by network layering models, add information to a message to ensure that it reaches its destination.
- A naming system (such as DNS) must be used to translate from a host name to a network address, and another protocol (such as ARP) may be needed to translate the network number to a network device address (an Ethernet address, for instance).
- If systems are located on separate networks, routers are needed to pass packets from source network to destination network.
- The transport protocols UDP and TCP direct packets to waiting processes through the use of unique system-wide port numbers. In addition, the TCP protocol allows the flow of packets to become a reliable, connection-oriented byte stream.
- There are many challenges to overcome for a distributed system to work correctly. Issues include naming of nodes and processes in the system, fault tolerance, error recovery, and scalability. Scalability issues include handling increased load, being fault tolerant, and using efficient storage schemes, including the possibility of compression and/or deduplication.
- A DFS is a file-service system whose clients, servers, and storage devices are dispersed among the sites of a distributed system. Accordingly, service activity has to be carried out across the network; instead of a single centralized data repository, there are multiple independent storage devices.
- There are two main types of DFS models: the client-server model and the cluster-based model. The client-server model allows transparent file sharing among one or more clients. The cluster-based model distributes the files among one or more data servers and is built for large-scale parallel data processing.
- Ideally, a DFS should look to its clients like a conventional, centralized file system (although it may not conform exactly to traditional file-system interfaces such as POSIX). The multiplicity and dispersion of its servers and storage devices should be transparent. A transparent DFS facilitates client mobility by bringing the client's environment to the site where the client logs in.
- There are several approaches to naming schemes in a DFS. In the simplest approach, files are named by some combination of their host name and local name, which guarantees a unique system-wide name. Another approach, popularized by NFS, provides a means to attach remote directories to local directories, thus giving the appearance of a coherent directory tree.
- Requests to access a remote file are usually handled by two complementary methods. With remote service, requests for accesses are delivered to the server. The server machine performs the accesses, and the results are forwarded back to the client. With caching, if the data needed to satisfy the access request are not already cached, then a copy of the data is brought from the server to the client. Accesses are performed on the cached copy. The problem of keeping the cached copies consistent with the main file is the cache-consistency problem.

19.11 Practice exercises

EXERCISE 19.11.1: (Problem 19.1 in the 10th edition).

(a) Why would it be a bad idea for routers to pass broadcast packets between networks? What would be the advantages of doing so?

EXERCISE 19.11.2: (Problem 19.2 in the 10th edition).

(a) Discuss the advantages and disadvantages of caching name translations for computers located in remote domains.

EXERCISE 19.11.3: (Problem 19.3 in the 10th edition).

(a) What are two formidable problems that designers must solve to implement a network system that has the quality of transparency?

EXERCISE 19.11.4: (Problem 19.4 in the 10th edition).

To build a robust distributed system, you must know what kinds of failures can occur.

(a) List three possible types of failure in a distributed system.

(b) Specify which of the entries in your list also are applicable to a centralized system.

(c) Three common failures in a distributed system are: (1) network link failure, (2) host failure, (3) site failure failure. Both (2) and (3) are failures that could also occur in a centralized system, whereas a network link failure can occur only in a networked distributed system.

EXERCISE 19.11.5: (Problem 19.5 in the 10th edition).

(a) Is it always crucial to know that the message you have sent has arrived at its destination safely? If your answer is "yes," explain why. If your answer is "no," give appropriate examples.

EXERCISE 19.11.6: (Problem 19.6 in the 10th edition).

(a) A distributed system has two sites, A and B. Consider whether site A can distinguish among the following:

a. B goes down.
b. The link between A and B goes down.
c. B is extremely overloaded, and its response time is 100 times longer than normal.

What implications does your answer have for recovery in distributed systems?

19.12 Exercises

EXERCISE 19.12.1: (Problem 19.7 in the 10th edition).

(a) What is the difference between computation migration and process migration? Which is easier to implement, and why?

EXERCISE 19.12.2: (Problem 19.8 in the 10th edition).

(a) Even though the OSI model of networking specifies seven layers of functionality, most computer systems use fewer layers to implement a network. Why do they use fewer layers? What problems could the use of fewer layers cause?

EXERCISE 19.12.3: (Problem 19.9 in the 10th edition).

(a) Explain why doubling the speed of the systems on an Ethernet segment may result in decreased network performance when the UDP transport protocol is used. What changes could help solve this problem?

EXERCISE 19.12.4: (Problem 19.10 in the 10th edition).

(a) What are the advantages of using dedicated hardware devices for routers? What are the disadvantages of using these devices compared with using general-purpose computers?

EXERCISE 19.12.5: (Problem 19.11 in the 10th edition).

(a) In what ways is using a name server better than using static host tables? What problems or complications are associated with name servers? What methods could you use to decrease the amount of traffic name servers generate to satisfy translation requests?

EXERCISE 19.12.6: (Problem 19.12 in the 10th edition).

(a) Name servers are organized in a hierarchical manner. What is the purpose of using a hierarchical organization?

EXERCISE 19.12.7: (Problem 19.13 in the 10th edition).

(a) The lower layers of the OSI network model provide datagram service, with no delivery guarantees for messages. A transport-layer protocol such as TCP is used to provide reliability. Discuss the advantages and disadvantages of supporting reliable message delivery at the lowest possible layer

EXERCISE 19.12.8: (Problem 19.14 in the 10th edition).

(a) Run the program shown in the figure below and determine the IP addresses of the following host names:

```
/**
* Usage: java DNSLookUp <IP name>
* i.e. java DNSLookUp www.wiley.com
*/
public class DNSLookUp {
    public static void main(String[] args) {
      InetAddress hostAddress;

      try {
        hostAddress = InetAddress.getByName(args[0]);
        System.out.println(hostAddress.getHostAddress());
      }
      catch (UnknownHostException uhe) {
        System.err.println("Unknown host: " + args[0]);
    }
}
```

- www.wiley.com
- www.cs.yale.edu
- www.apple.com
- www.westminstercollege.edu
- www.ietf.org

EXERCISE 19.12.9: (Problem 19.15 in the 10th edition).

(a) A DNS name can map to multiple servers, such as www.google.com. However, if we run the program shown in the previous exercise's figure, we get only one IP address. Modify the program to display all the server IP addresses instead of just one.

EXERCISE 19.12.10: (Problem 19.16 in the 10th edition).

(a) The original HTTP protocol used TCP/IP as the underlying network protocol. For each page, graphic, or applet, a separate TCP session was constructed, used, and torn down. Because of the overhead of building and destroying TCP/IP connections, performance problems resulted from this implementation method. Would using UDP rather than TCP be a good alternative? What other changes could you make to improve HTTP performance?

EXERCISE 19.12.11: (Problem 19.17 in the 10th edition).

(a) What are the advantages and the disadvantages of making the computer network transparent to the user?

EXERCISE 19.12.12: (Problem 19.18 in the 10th edition).

(a) What are the benefits of a DFS compared with a file system in a centralized system?

EXERCISE 19.12.13: (Problem 19.19 in the 10th edition).

(a) For each of the following workloads, identify whether a cluster-based or a client–server DFS model would handle the workload best. Explain your answers.

- Hosting student files in a university lab.
- Processing data sent by the Hubble telescope.
- Sharing data with multiple devices from a home server.

EXERCISE 19.12.14: (Problem 19.20 in the 10th edition).

(a) Discuss whether OpenAFS and NFS provide the following: (a) location transparency and (b) location independence.

EXERCISE 19.12.15: (Problem 19.21 in the 10th edition).

(a) Under what circumstances would a client prefer a location-transparent DFS? Under what circumstances would she prefer a location-independent DFS? Discuss the reasons for these preferences.

EXERCISE 19.12.16: (Problem 19.22 in the 10th edition).

(a) What aspects of a distributed system would you select for a system running on a totally reliable network?

EXERCISE 19.12.17: (Problem 19.23 in the 10th edition).

(a) Compare and contrast the techniques of caching disk blocks locally, on a client system, and remotely, on a server.

EXERCISE 19.12.18: (Problem 19.24 in the 10th edition).

(a) Which scheme would likely result in a greater space saving on a multiuser DFS: file-level deduplication or block-level deduplication? Explain your answer.

EXERCISE 19.12.19: (Problem 19.25 in the 10th edition).

(a) What types of extra metadata information would need to be stored in a DFS that uses deduplication?

19.13 Further reading

[Peterson and Davie (2012)] and [Kurose and Ross (2017)] provide general overviews of computer networks. The Internet and its protocols are described in [Comer (2000)]. Coverage of TCP/IP can be found in [Fall and Stevens (2011)] and [Stevens (1995)]. UNIX network programming is described thoroughly in [Steven et al. (2003)].

Ethernet and WiFi standards and speeds are evolving quickly. Current IEEE 802.3 Ethernet standards can be found at http://standards.ieee.org/about/get/802/802.3.html. Current IEEE 802.11 Wireless LAN standards can be found at http://standards.ieee.org/about/get/802/802.11.html.

Sun's network file system (NFS) is described by [Callaghan (2000)]. Information about OpenAFS is available from http://www.openafs.org.

Information on the Google file system can be found in [Ghemawat et al. (2003)]. The Google MapReduce method is described in http://research.google.com/archive/mapreduce.html. The Hadoop distributed file system is discussed in [K. Shvachko and Chansler (2010)], and the Hadoop framework is discussed in http://hadoop.apache.org/.

To learn more about Lustre, see http://lustre.org.

19.14 Bibliography

[Callaghan (2000)] B. Callaghan, *NFS Illustrated*, Addison-Wesley (2000).

[Comer (2000)] D. Comer, *Internetworking with TCP/IP*, Volume I, Fourth Edition, Prentice Hall (2000).

[Fall and Stevens (2011)] K. Fall and R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, Second Edition, John Wiley and Sons (2011).

[Ghemawat et al. (2003)] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System", *Proceedings of the ACM Symposium on Operating Systems Principles* (2003).

[K. Shvachko and Chansler (2010)] S. R. K. Shvachko, H. Kuang and R. Chansler, "*The Hadoop Distributed File System*" (2010).

[Kurose and Ross (2017)] J. Kurose and K. Ross, *Computer Networking—A Top- Down Approach*, Seventh Edition, Addison-Wesley (2017).

[Peterson and Davie (2012)] L. L. Peterson and B. S. Davie, *Computer Networks: A Systems Approacm*, Fifth Edition, Morgan Kaufmann (2012).

[Steven et al. (2003)] R. Steven, B. Fenner, and A. Rudoff, *Unix Network Programming, Volume 1: The Sockets Networking API*, Third Edition, John Wiley and Sons (2003).

[Stevens (1995)] R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley (1995).