COMPUTER COMMUNICATIONS SOFTWARE

Computer communications software is becoming increasingly important as a result of the increasing deployment and use of computer networks. Stringent requirements of real-time processing, efficiency, reliability, and interoperability make design and development of such software extremely challenging. The development typically starts by identifying the *services* to be provided. A service is defined by its functionality and its interface. The functionality may range from low-level tasks, such as retransmission of lost messages, to high-level applications, such as electronic mail. The interface describes the supported operations and their parameters. The development continues with the design of a *protocol,* which describes the messages that will be exchanged in an implementation of the service; the protocol specifies message format (e.g., message length, division into fields, and data encoding), timing (e.g., minimum and maximum intervals between messages in certain situations), and semantics (i.e., the meaning of each message). Finally, an implementation of the service is constructed. Service definitions are often sufficiently flexible to allow many different implementations of the service using the same protocol. This allows each computer in a network to use an implementation optimized for its particular architecture. Since the implementations all follow the same protocol, they interact correctly to provide the service.

Communication is possible only when all participants follow the same protocol, so standards are essential. A *protocol architecture* is a collection of protocols designed to be used together. The International Organization for Standardization (ISO) issued a standard for an influential—though not widely used—protocol architecture, called the *Open Systems Interconnection (OSI) Reference Model* (1). The Internet Activities Board issues standards for the protocols used on the Internet; collectively these form the *Internet Architecture* or *TCP/IP Architecture.*

Both of the standards just mentioned (and most other protocol standards) incorporate layering, a classic design technique. To help manage the complexity of writing, testing, and maintaining such software, the overall functionality is divided into several services, and the software is divided into layers, each implementing one or more services. Figure 1 illustrates the layered structure. A collection of layers is called a *protocol stack* (or *stack,* for short). The basic principle is that a message m_i sent by layer i in the sender's stack is delivered to layer i in the receiver's stack (2). A layer may modify the body of the message; for example, layer i in the sender's stack encrypts the body for secrecy, and layer i in the receiver's stack decrypts it. A layer may also insert information into the message header; for example, layer i in the sender's stack creates a header field containing a sequence number, and layer i in the receiver's stack uses this information to detect missing messages. Since each layer can add its own header fields to a message, the headers also form a stack. Headers

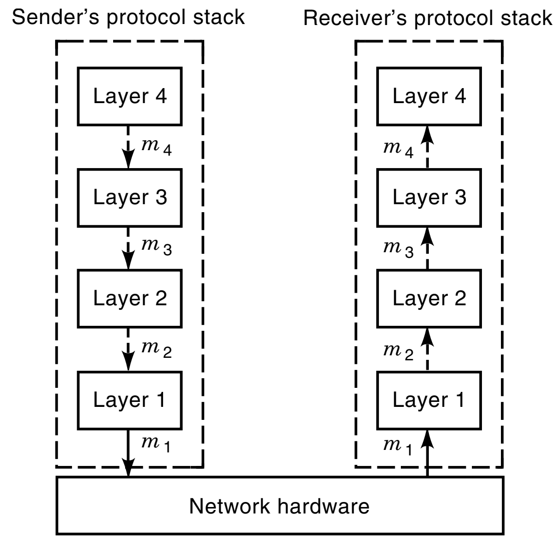

Figure 1. Illustration of message flow in a layered system.

pushed onto the header stack by layer i in the sender's protocol stack are popped off the header stack by layer i in the receiver's protocol stack.

The diversity of network hardware and of the requirements on communication software for different applications has led to the development of a plethora of communication services and protocols, both public and proprietary. It is helpful to classify them according to the following fundamental characteristics. Some of these characteristics apply to individual operations in a service rather than an entire service; different operations in a service may have different characteristics.

Symmetry. *Symmetric* services provide communication between peers. For example, message delivery services are symmetric, that is, they allow any process to send a message to any other process. In *asymmetric* services, the communicating parties have different roles. For example, services that support interaction between a client (such as a user process) and a server (such as a file server) are typically asymmetric. Symmetry of a service is determined primarily by the intrinsic nature of the service.

Synchrony. In a *synchronous* (or *blocking*) service, invoking an operation causes the caller to block until the requested communication (and associated processing) is completed. For example, a remote procedure call (RPC) typically causes the caller to block until a result is received from the remote site; in this case, the RPC operation is synchronous. In an *asynchronous* (or *nonblocking*) service, the caller is able to continue with other tasks processing while the request is actually performed. For example, a request to send a message might allow the sender to continue before the message is actually transmitted on the network. Synchrony is determined partly by the nature of the service but partly by other considerations. For example, although RPC operations are typically synchronous (and might seem inherently so), asynchronous RPC operations are possible: the caller continues immediately with other tasks and is notified later when the result (return value) of the RPC is available. Typically, such notifications are provided via up-calls. An *up-call* is when a service calls a routine in the application; in contrast, a *down-call* is when an application invokes an operation (such as asynchronous RPC) provided by the service. For example, in the down-call invoking an asynchronous RPC, the application supplies the name of a procedure *P*; when the return value *r* of the RPC is available, the service invokes *P* with argument *r*, typically in a new thread. This approach can be used to construct asynchronous versions of most synchronous services. The choice between the synchronous and asynchronous versions is typically based on performance and ease of programming [(3), chapter 2]. The synchronous version avoids the overhead of creating a thread for the up-call but may require more threads in the application to achieve the same degree of concurrency as the asynchronous version.

Reliability. A *reliable* service guarantees that each requested operation (e.g., transmitting a message) is performed successfully, even in the presence of specified numbers or rates of specified types of failures, such as message loss. If failures of the network or other computers prevent a reliable service from performing a requested operation, the service detects the problem and notifies the requester. An *unreliable* service does not include mechanisms for detecting, overcoming, or reporting failures. Reliability is not a Boolean attribute; there is a spectrum of possibilities, characterized by the degree of service degradation resulting from different types and rates of failures. Reliable services have more overhead than unreliable services. For example, an unreliable message service can send a message and then forget about it. A reliable message service that tolerates message loss needs to store a copy of the message at the sending machine until the destination confirms that the message has been received (or the sending application has been notified that delivery is impossible); this may incur overhead from copying, buffer management, and sending and receiving acknowledgments. Whether this cost is worthwhile depends on the application. Many communication packages provide both reliable and unreliable versions of services, leaving the choice to the application.

Number of Destinations. A *one-to-one* communication service provides communication from a single source to a single destination in a single operation. A *one-to-many* communication service provides communication from a single source to multiple destinations in a single operation. Sending a single message to all machines on a certain network is called *broadcasting*. Sending a single message to a selected set of destinations is called *multicasting*. For example, multicast is useful when a group of processes on different computers maintain replicas of files or other data; replication enhances availability and allows concurrent processing of read-only operations. A multicast may differ in two important ways from a sequence of one-to-one send operations; the same applies to broadcast. First, a multicast can often be implemented more efficiently, especially if the underlying network hardware supports broadcast. Second, a multicast may provide stronger reliability guarantees. For example, a multicast might guarantee that if any destination receives a message, then all destinations that do not crash also receive that message; this is achieved by having the destinations relay the message to each other. A sequence of one-to-one sends (even reliable ones) does not guarantee this, because the sender might crash after some of the sends.

Quality of Service. *Quality of service* (QoS) refers to the performance guarantees provided by a communication service. Naturally, performance of a communication service depends on both the communication software and the underlying network. A *QoS contract* specifies the load to be offered by the application and the performance to be supplied by the service. The load to be offered is characterized, for example, by the minimum and average intervals between requests and the size of requests (e.g., the size of messages being sent). Typical performance metrics for communication services include *throughput,* the rate (e.g., in megabits/s) at which data are conveyed, and *delay,* the amount of time from when a message is sent until it is received. For example, for a specified application load, a messaging service might guarantee an average delay of 2 ms and a maximum delay of 10 ms. Reliability metrics, such as the maximum fraction of sent messages that are lost, are sometimes included in QoS contracts.

Connections. A *connection-oriented* service works like the telephone system: before two processes on different computers can use the service to communicate, an initialization step is needed to construct a logical connection between those processes. When those processes finish communicating, the connection between them is released, analogous to what happens when someone hangs up a telephone. In a *connectionless* service, each communication request is handled independently of other requests: any two processes can communicate at any time, without an initialization step. One benefit of connection-oriented communication is that successful connection establishment assures each party that the other party is alive and reachable over the network. More importantly, connection establishment provides an opportunity for the application processes and the communication service to negotiate a QoS contract and for the communication service to reserve resources so that the connection will provide the agreed QoS. In some protocol architectures, such as the asynchronous transfer mode (ATM) protocol architecture, connection establishment involves determining and fixing a path through the network connecting the two communicating parties. That path may involve any number of intermediate switches or computers and will be used for all messages sent along the connection. When the connection is established, the intermediate nodes on the path can also reserve resources for the connection; thus such systems are better suited to providing QoS guarantees. Also, repeated use of this path can provide a considerable performance benefit, compared with recomputing the path for each message. Reuse of paths is facilitated by use of *connection identifiers.* A connection identifier is selected when the connection is established and is included in the header of each message sent along the connection. This identifier is used by intermediate nodes as an index for efficient table lookup of the next node in the path for that connection. Another benefit of a connection identifier is that it indicates a message's destination and typically is shorter than the destination's globally unique address; with connectionless communication, each message contains the destination's globally unique address.

CORE FUNCTIONALITY AND IMPLEMENTATION TECHNIQUES

This section describes the core functionality that is present in almost all general-purpose communication software and sketches common implementation techniques.

Addressing and Routing

The three most important questions to ask about an addressing scheme are: (1) What kind of entity is identified by an address? (2) How are addresses assigned? (3) Given an address, how is the entity with that address located (in order to send a message to it)? A single protocol architecture may involve multiple kinds of addresses. It is common for different kinds of addresses to be used at different levels. Thus some layers accept requests containing one kind of address and produce requests containing a different kind of address.

The lowest layer of a protocol architecture must produce requests containing *hardware addresses,* that is, addresses understood by the underlying network hardware. A *network interface* is hardware (usually located on a card in a computer) that implements a connection between a computer and a network. For example, in IEEE 802.3 local-area networks (Ethernets), each network interface stores a unique identifier assigned by the manufacturer; this identifier is used as an address by the lowest layer of the software. So, for Ethernet, the answers to the above questions are: (1) A hardware address identifies a network interface; (2) Hardware addresses are assigned by the equipment manufacturer, under the control of IEEE to ensure that addresses are unique; (3) Within an Ethernet, a message can be sent to a given hardware address simply by transmitting the message, with a header containing that destination address, on the Ethernet.

Using hardware addresses in higher layers of the software would be problematic. There are two fundamental (and related) reasons for introducing higher-level kinds of addresses, which are sometimes called *protocol addresses* or *virtual addresses.* One is to provide the ability to address entities (such as processes or user accounts) that do not correspond directly to hardware devices. The other reason is to achieve *independence,* that is, to make an entity's address independent of details of the system configuration. This ensures that changes to those configuration details do not affect an entity's address. This is an example of the general principle of modularity, namely, that the interface to an object (entity) should not reveal implementation details. To make these points more concrete, the different kinds of addresses in the Internet Architecture will be briefly discussed.

The *IP address* is the lowest-level kind of protocol address in the Internet Architecture. IP addresses are independent of the type of underlying network hardware (Ethernet, token ring, ATM, etc.). This is essential for constructing heterogeneous networks like the Internet. Also, hardware addresses in some types of networks (such as token ring) are not globally unique; IP addresses are globally unique. IP addresses can be characterized by answering the three questions above. (1) An IP address identifies a connection between a computer and a network (note that the IP address can remain the same even if the network interface implementing that connection is changed). (2) An IP address has two parts: a *prefix* and a *suffix.* A prefix is assigned by a central authority (e.g., the Internet Assigned Number Authority) to each local network in the Internet; the administrators of that local network assign suffixes to particular connections to that network. (3) An IP address is translated into a hardware address; this is called *address resolution.* A simple and widely applicable approach to address resolution is table lookup, using direct indexing or hashing. The table lookup may be done by the sender itself or by a designated server. In networks that allow hardware addresses to be assigned by the local administrator, hardware addresses can be computed and assigned as some function of the protocol address. On broadcast networks, another possibility is to broadcast a query containing the protocol address in question; if that address belongs to a machine on the local network, that machine sends a reply containing its hardware address. For efficiency, the results of such queries are cached. This last approach (broadcasting plus caching) is commonly used for IP address resolution in Ethernets.

Resolving a protocol address into a hardware address is useful only if the protocol address refers to an entity on the same local network; otherwise, the hardware address is not particularly useful, because a message cannot be addressed directly to it. A message is sent to a nonlocal protocol address by repeatedly forwarding the message along a sequence of machines, each connected to two or more local networks, such that the sequence forms a path from the sender to the final destination. The problem of finding such a path is called *routing.* In networks with irregular topologies, routing is usually done by table lookup; for example, in the Internet, lookup of the prefix part of a destination IP address yields the IP address of the next machine in a path to that destination. Typically, the routing table indicates a *default router,* to which messages are sent when there is no explicit entry for the prefix of the destination address.

For modularity, the lowest layer that introduces protocol addresses should completely hide hardware addresses from higher layers, making those layers more hardware-independent [(2), section 15.15]. In the Internet Architecture, IP addresses are introduced by the layer immediately below the IP layer; that layer is called the *network access* layer, *network interface* layer, or *host-to-network* layer.

Domain names are a higher-level kind of protocol address in the Internet Architecture. There are two main reasons for introducing domain names. One is independence: domain names are more independent of network topology than IP addresses. An IP address is tied to a particular local network; if a machine is moved to a different (e.g., faster) local network, which corresponds to a different IP address prefix, then the machine's IP address must change. In contrast, the domain name of that machine could remain unchanged. The second main reason for introducing domain names is that IP addresses are binary (for efficiency) and thus are hard for users to remember and enter; domain names are easier to remember and enter because they are hierarchical and textual. For example, bone.cs.indiana.edu is a domain name; the dots separate the name into segments that reflect the hierarchical structure. A domain name, like an IP address, identifies a connection between a computer and a network. Assignment of domain names is based on the hierarchical structure of the names. For example, an authority associated with .edu assigns indiana.edu to Indiana University; an authority at Indiana University assigns cs.indiana.edu to the Computer Science Department; and so on. A domain name is resolved (translated) into an IP address by the *Domain Name System* (DNS); DNS is based on table lookups by a hierarchy of servers, corresponding to the hierarchical structure of domain names.

At the application level, the goal of communication is often to access a service provided by a process on a different machine. A domain name is not suitable for identifying a service, because a single machine with a single network connection might run several processes offering different services. This motivates the introduction of a new kind of address. It is desirable for the address of a service to be independent of the machine providing the service; otherwise, if a service is moved between machines for the purpose of fault-tolerance (e.g., because the machine that usually provides the service crashed) or load-balancing, its address must change. The Internet Architecture does not directly support machine-independent addresses, though some experimental architectures, such as Amoeba (a research system), do (3). Consequently, the (machine-dependent) address for a service can be constructed simply by concatenating the domain name (or IP address) of a machine with an identifier—called a *port*—that identifies that service on that machine. For example, on UNIX systems, the DNS server conventionally uses port 53; thus the address of the DNS server on ns.indiana.edu is ns.indiana.edu:53.

Only a few basic services (like DNS) have ports that are fixed by convention. For other services, the port corresonding to a particular service is looked up in a system-specific table. A *directory server* accepts requests containing the textual name of a service (e.g., "time-of-day") and returns the corresponding port and, if appropriate, the domain name (or IP address) of a machine offering that service. The directory service itself is a basic service with a fixed port. In systems with such directory servers, these textual names for services constitute a new machine-independent kind of address, though they are not part of the Internet Architecture per se.

Fragmentation and Reassembly

Each type of network hardware has a maximum transmission unit (MTU), which is the largest amount of data that can be conveyed in a single transmission. A layer in the protocol stack can hide this restriction from higher layers by performing *fragmentation and reassembly,* that is, by splitting large messages into smaller pieces for transmission, and reassembling them into the original message at the receiver.

Flow Control

Differences in hardware speed and operating load between a sender and receiver may cause *data overrun,* in which data arrive at the receiver faster than the receiver can handle, causing the receiver to drop data. The receiver can try to keep up with the sender by simply buffering the incoming data (and processing later), but data overrun will still occur if the receiver runs out of buffer space. *Flow control* is the problem of preventing data overrun. Note that flow control can be performed in one or more layers in a protocol architecture. In the following discussion, "message" refers to the unit of transmission (e.g., packet or frame) at the layer being considered.

The simplest flow-control technique is *stop and wait.* After sending each message, the sender waits for the receiver to send an acknowledgment indicating that it is ready to receive the next message. This technique is easy to implement but greatly reduces the throughput. So, the restriction on the sender is relaxed, allowing it to send multiple messages before checking whether the receiver is ready to receive more. This technique is called *sliding-window flow control.* The *window size* w is the maximum number of messages that can be in transit simultaneously. The sender sends the $(i + w)$th message only after it has received some indication that the receiver has already received the ith message. The name "sliding window" comes from the mental image of a window of width w sliding forward along the stream of messages to be sent. The window size is determined mainly by the amount of buffer space available at the receiver. In connection-oriented communication, the window size is typically determined as part of connection establishment.

The implementation of flow control in a particular layer of a protocol architecture is affected by whether the message service provided by the lower layers is reliable. Implementations of reliable delivery and flow control both involve acknowledgments, so their implementations are combined in some protocol architectures, such as TCP/IP. Combining their implementations has another benefit, discussed in the next subsection. (In short, the window size provides a bound on the number of messages stored for possible retransmission.)

Reliable Delivery

In reliable services, different techniques are used to cope with different kinds of errors. Message corruption is usually handled using *error-detecting codes* (EDCs), which enable the recipient to determine with high probability whether a message has been corrupted by random errors during transmission. For efficiency, error-detecting codes are usually implemented in hardware. If an error is detected, the error-detecting hardware simply reports the problem to the communication software. Typically, the net effect is the same as if the corrupted message had been lost. Error-correcting codes can also be used, but for most communication media (except perhaps wireless) the error rate is sufficiently low that the additional overhead of error-correcting codes is not worthwhile.

Message loss is handled by detecting that a message has been lost and then retransmitting it. There are two basic approaches to detecting message loss: (1) *positive acknowledgment* and (2) *negative acknowledgment.* In the positive acknowledgment approach, on receiving a message, the recipient sends an acknowledgment. If the sender does not receive an acknowledgment within the expected time interval, it times out and resends the message. Note that a message might be resent merely because the acknowledgment is lost; thus, on receiving a message that it received before, the recipient just resends the acknowledgment. Including a sequence number (modulo some fixed quantity) in each message allows efficient detection of duplicates. The negative acknowledgment approach also uses sequence numbers (modulo some fixed quantity). If the recipient observes a gap in the sequence numbers on received messages—for example, if it receives a message numbered 7 immediately after receiving a message numbered 5—then it sends a negative acknowledgment to the sender, requesting retransmission of the missing message(s). When the sender finishes transmitting, no gap will be detected even if the last few messages are lost. Similarly, a pause in transmission can delay detection of message loss. To overcome these problems, if the receiver does not receive a message from a sender for some period of time, it times out and sends a message to the sender, specifying the sequence number of the last message received; if any messages were lost, the sender retransmits them. Negative acknowledgments are typically more efficient than positive acknowledgments, though also more complicated to implement.

A potential problem with negative acknowledgment schemes is that, if a continuous stream of messages are sent and no messages are lost, the sender will not receive any feedback from the receiver, so it will not know when to discard copies of old messages. Combining the implementation of reliable delivery with sliding-window flow control, which forces an acknowledgment to be sent at least after every wth message received, overcomes this problem: the sender needs to store copies of at most the last w messages, where w is the window size.

In situations where message delay is predictable (i.e., has low variance)—for example, communication within a local-area network—it is reasonable to use fixed values for the time-outs that control retransmission. In situations where message delay is less predictable—for example, communication over the Internet—*adaptive time-outs* are much more effective. A sender maintains an estimate of the current round-trip delay to the receiver, by recording the time at which it sends each message to which it expects a reply, and, when the reply arrives, computing the round-trip delay for that message/reply and incorporating it into a weighted average. To allow the time-out value to adapt quickly to changes in the round-trip delay, the sender can also maintain an estimate of the variance in the round-trip time and compute the retransmission time-out as a linear combination of the weighted average and the estimated variance (4). This approach is used in most implementations of TCP.

Retransmission is effective against transient problems, but additional mechanisms are needed to cope with longer-term network problems or computer crashes. If an operation has not succeeded after a certain number of retries, a reliable service typically aborts the operation and reports this to the application. If the service is connection-oriented, this typically has the effect of closing the relevant connection.

Where should the layers that provide reliability (using EDCs and retransmission) be located in a protocol architecture? A particularly important issue is whether to place them above or below the layer that performs routing. If they are placed below the routing layer, then reliability is implemented on a "hop-by-hop" (link-by-link) basis; if they are placed above it, then reliability is implemented on an "end-to-end" basis. First consider retransmission. If retransmission is done hop-by-hop, then there is still a small chance that messages get lost, for example, if a software bug causes an intermediate node to lose a message after sending an acknowledgment for it. (In a wide-area network such as the Internet, the two communicating parties might know nothing about the operating systems and protocol implementations being run in the intermediate nodes, so the possibility of bugs should not be dismissed lightly.) Thus, performing retransmission on an end-to-end basis provides a stronger guarantee. This is a classic example of an *end-to-end* argument (5). Now consider EDCs. An end-to-end argument implies that EDCs should be used above the routing layer. This indeed provides the desired reliability. However, in many systems, it is desirable to use EDCs on a hop-by-hop basis as well, to improve performance. If a message gets corrupted, the corruption is detected immediately, and the previous node in the path retransmits the message. If EDCs were not used on a hop-by-hop basis, then the corrupted message would be forwarded to the final destination before the corruption is detected, and then the message would have to be retransmitted along the entire path from source to destination. A similar argument can be made for performing retransmission on a hop-by-hop basis as well. However, for most systems that argument does not hold up quantitatively, because the frequency of message loss is so low relative to the overhead of a hop-by-hop retransmission mechanism that the savings would be outweighed by the overhead.

Congestion Control

Congestion occurs when an intermediate node in a route receives data faster than it can forward the data to the next node in the route. Congestion can occur even if all of the computers and links operate at the same speed. For example, if a node is receiving packets with the same destination from two different senders on two different links, then the maximum rate at which the node can forward those packets to the destination is only half of the maximum rate at which the node can receive those packets. When the node's buffers are full, it will be forced to drop packets. Even if the node has large buffers and does not drop packets, the packets will experience increasing delays, as they remain buffered for increasingly long times. If reliable message delivery is involved, then the delays or message loss due to congestion provoke retransmissions, which can increase the rate at which packets are being sent and thereby cause worse congestion. Furthermore, if a congested node is dropping packets instead of storing and acknowledging them, then the node sending those packets cannot release the buffers containing them, and this might force that node to drop incoming packets, thereby causing congestion to spread. Thus, it is important for a network to detect and react to congestion quickly, or better, to prevent congestion. This is the problem of *congestion control.*

The likelihood of congestion can be reduced by careful design of the entire protocol architecture, including retransmission time-outs, window size, routing algorithm, and so forth. Limiting the rate at which packets are injected into the network can also help prevent congestion. Two techniques for this are admission control and traffic shaping [(6), section 5.3]. *Admission control* is used with connection-oriented communication; if the network is heavily loaded, the admission control mechanism will refuse requests to establish new connections. *Traffic shaping* is based on the observation that bursty communication can cause congestion, even if smooth communication with the same average throughput would not. When an application sends a burst of messages, a traffic shaping algorithm may buffer some of the messages at the sender and inject them gradually into the network.

The above techniques do not completely eliminate congestion, so techniques for detecting and reducing congestion are also needed. One approach to detecting congestion is for each intermediate node to keep track of the number of packets dropped due to lack of buffer space. However, there is a remaining problem of how to inform the appropriate senders of the congestion, so they will reduce their transmission rate. This is nontrivial because, once congestion has started, it is difficult to ensure that any information gets through the network in a timely fashion. A second approach, which has the benefit of circumventing this problem, is for senders to estimate congestion by detecting packet loss. This is reasonable because modern network hardware (except wireless) is sufficiently reliable that most packet loss is due to congestion. With this approach, when a sender detects message delay or loss, it immediately reduces its transmission rate, then gradually increases the rate as long as no further problems occur. If sliding-window flow control is used, the transmission rate can be adjusted by changing the window size.

The sliding-window technique is remarkable for its utility in so many aspects of communication: flow control, reliability, and congestion control. Many implementations of TCP use a single sliding-window mechanism to deal efficiently with these three issues. One consequence is a lack of modularity in those implementations. A separate layer could be used to deal with each of these issues; the resulting system would be more modular but probably less efficient. This example illustrates that in layered software, the division into layers needs to be carefully chosen, so that it does not unduly constrain the possible implementations.

Ordered Delivery

Many applications expect messages to be delivered in first-in, first-out (FIFO) order, that is, in the order that the messages were sent. Typically, in local networks, communication is intrinsically FIFO. However, in wide-area networks, it is possible (with some routing algorithms) for different messages to follow different paths from the sender to the receiver; if one path is slower than another, messages might arrive out-of-order.

The most straightforward approach to ensuring FIFO delivery is to tag each message with a sequence number. The receiver stores the sequence number i of the last message delivered. If a message with a number other than $i + 1$, arrives, the receiver stores it for later delivery and then continues waiting for message $i + 1$. Unbounded sequence numbers are relatively inefficient, so it is desirable to replace with fixed-size numbers, specifically, with sequence numbers modulo a small fixed value. Justifying this replacement requires additional information about the system, such as an upper bound on message delay or, if messages contain timestamps, an upper bound on the difference between the sender's and receiver's clocks.

Connection Management

Connection management is the problem of establishing and terminating connections between pairs of parties in a connection-oriented communication service. As mentioned in the beginning, in some protocol architectures, such as the ATM protocol architecture, connections are used throughout the architecture; in such systems, connection establishment involves determining and fixing a path through the network that will be used for all messages sent along the connection.

In other protocol architectures, connections are used only at higher levels—in particular, above the routing layer. In such systems, only the sender and receiver (not intermediate nodes) are aware of the connection. This is the case in the Internet Architecture, where TCP, a connection-oriented protocol, is layered over the IP protocol, which is connectionless. If the layer responsible for connection management is above layers that provide reliable FIFO delivery, then the protocols are reasonably straightforward; otherwise, the connection management protocol will itself need to implement time-outs and retransmission to cope with message loss [(7), section 17.2].

Managing connections used for multicasts among groups of arbitrary size is part of *group management,* which is discussed below.

Configuration and Initialization

Communication software must be configured (initialized) before it can be used. Typical configuration parameters for an IP protocol stack include the IP address of the computer it is running on, the IP address of the default router, and the IP address of a DNS server. A simple way to provide values for configuration parameters is to manually create a disk file containing them. This approach is brittle and inconvenient: a change in the network configuration requires changing the configuration file on each affected computer. This approach is especially inconvenient for portable computers, which may be attached to several different networks in a day.

The Internet architecture includes several protocols that help automate configuration of a protocol stack; two of them will be discussed. The Bootstrap Protocol (BOOTP) enables a booting machine to automatically obtain values of several parameters, including the addresses mentioned above, by requesting them from a server, which maintains a database of the necessary information. BOOTP is used on broadcast networks (like Ethernet), so the request is broadcast to all machines on the local network. The BOOTP server replies; other machines simply ignore the request. Thus, the booting machine does not need to know the BOOTP server's IP address or hardware address. However, the request message cannot contain the sender's IP address, since the sender does not know it yet, so how does the BOOTP server determine the destination address for its reply? One option is for the BOOTP server to broadcast the reply. If the sender is able to include its hardware address in the request message, then a more efficient option is for the server to send the reply directly to that hardware address. The latter option is interesting because it violates a modularity principle stated above, namely, that the network access layer hides hardware addresses from the layers and applications above it. The BOOTP server runs above that layer (above the UDP layer, in fact), so according to that modularity principle, it should deal with IP addresses, not hardware addresses. This illustrates how difficult achieving modularity can be in complex communication software.

The Dynamic Host Configuration Protocol (DHCP) is an extension to BOOTP that allows the server to dynamically allocate IP addresses (in BOOTP, the server only looks up preassigned IP addresses in a table). When a portable computer is plugged into a local network, the DHCP server automatically assigns it an IP address, which it uses for the duration of its connection to that local network.

THE INTERNET ARCHITECTURE

As an example of how the core functionality described above can be organized, the layered structure of the Internet Architecture will be sketched. No standard explicitly defines this structure, but it is reasonable to consider the Internet Architecture as having five layers, which will be discussed from bottom to top.

The *physical layer* provides the ability to transmit an unstructured bit stream over a physical link. This layer is often implemented in hardware or firmware in the network interface.

The *network access layer* deals with the organization of data into blocks called *frames* and with the synchronization, error control (e.g., checksums), and flow control needed to transmit frames over a physical link. The format of a frame is dependent on the type of network hardware. This layer also deals with resolution of IP addresses into hardware addresses.

The *Internet layer* deals with the organization of data into blocks called *packets* and with routing of packets. The format of a packet is hardware-independent. This layer performs fragmentation and reassembly when a packet is routed through a local network whose frame size is smaller than the size of the packet. In summary, this layer provides unreliable, unordered (i.e., not necessarily FIFO) transmission of packets between any two hosts in an *internetwork* (i.e., a collection of interconnected local-area networks).

There are two standard *transport layers.* Both extend addresses to contain a port number as well as an IP address. That is essentially all the User Datagram Protocol (UDP) does. UDP is used for applications for which unreliable unordered message delivery suffices. The Transmission Control Protocol (TCP) provides connection-oriented reliable transmission of streams of data. Thus, implementations of TCP must provide connection management, reliability, and ordered delivery. For efficiency, most implementations of TCP are based on a sliding-window mechanism and also deal with flow control and congestion control.

Many different protocols can appear in the *application layer,* including BOOTP and DHCP, which run over UDP, and protocols that support applications like file transfer or electronic mail.

HIGH-LEVEL COMMUNICATION SERVICES

Communication services that provide the ability to send sequences of messages or streams of data are natural from a bottom-up perspective, since they correspond relatively closely to the operations provided by the network interface. From a top-down perspective, there are many applications for which other "higher-level" communication services are more natural and more convenient. The classic examples of such services are remote procedure call and distributed shared memory. More recently, distributed objects and group communication have been receiving increasing attention.

Each of these communication services is "higher-level" than messaging by virtue of some form of *transparency.* Transparency means that the communication service hides (makes transparent) some aspect of communication or distribution. Thus, the application can be written more like a centralized program; this is typically easier for the programmer.

Group Communication

Group communication allows a collection of processes—called a *group*—to be treated as a single entity. The basic functions of a group communication system are group management and multicast. Group management supports addition and removal of members, allowing a group's membership to change dynamically. Multicast sends a message to all members of a group. Group communication is especially useful for constructing fault-tolerant systems (8). Support for fault-tolerance can be integrated into group management and multicast. This greatly reduces the burden on the application programmer. In such systems, group management includes a mechanism that monitors all members of a group and automatically removes members that are crashed or unreachable. Also, such systems provide totally ordered atomic multicast. *Total ordering* guarantees that multicast messages are received in the same order by all members (except members that crash and hence do not receive some of the messages). *Atomicity* guarantees that if any member of the target group receives the message, then all members that do not crash also receive the message. A variety of distributed algorithms have been developed to enforce these guarantees (9–11).

To illustrate the benefits of group communication, consider a group of servers that provide a directory service. Each server maintains a copy of the directory; this allows concurrent processing of read-only operations and keeps the directory available even if some servers fail. Updates to the directory are disseminated by multicast to the group. Use of totally ordered atomic multicast conveniently ensures that after each update, all noncrashed servers have identical copies of the directory. Since multicasts are addressed to a group, rather than a specific list of machines, the application does not need to keep track of the group membership; the group management system does that automatically.

The use of group names as addresses is a useful abstraction in many settings. This is the basis of a second class of applications of group communication, namely, those involving *publication / subscription* communication (8). In this style of communication, some processes "publish" information associated with some topic, and all processes that have "subscribed" to that topic receive that information. In group-communication terms, a group is formed for each topic, and information is published by multicasting it to the group. Processes subscribe to a topic by joining the corresponding group. For example, group communication is well suited to financial trading applications, because they typically require fault-tolerance and involve publication/subscription communication, with topics corresponding to market sectors or stocks.

Remote Procedure Call

A *remote procedure call* (RPC) mechanism allows a process to call a procedure that gets executed on a different computer (12,13). The code needed for communication—namely, code for the caller to send the procedure's arguments to the remote computer, code for the remote computer to receive the arguments, invoke the procedure, and send the return value back to the caller, and code for the caller to receive the return value—is implicit in the procedure call. Thus, communication is, to some extent, transparent. Normally, the address of the remote machine is not indicated explicitly; instead, a *binding server* (also called a directory server; see the above discussion of addressing) is queried to obtain the address of a computer on which the procedure can be invoked. The most widely used RPC standard is Open Network Computing RPC (14), which is based on Sun RPC. The *remote method invocation* (RMI) facility of the Java programming language (15) is a form of RPC with some extensions. RPC is especially well suited to client–server communication. For example, communication in the Sun Network File System (NFS) (16) is done by RPC. This, the caller and the remote computer are sometimes referred to as the client and the server, respectively.

RPC hides the tasks of *marshalling* and *unmarshalling* from the application programmer. *Marshalling* is the task of formatting and arranging data values (such as a procedure's arguments or return values) so that they can be sent in a single message; *unmarshalling* is the task of extracting those data values from the message. In the simplest case, marshalling involves determining the size (in bytes) of each data value and copying the data values into the message; even this code is tedious to write by hand when variable-length data, such as character strings, is involved. More generally, to allow RPCs between computers with different architectures, marshalling involves conversion between different data representations. Furthermore, some RPC mechanisms support passing of linked data structures, such as linked lists or graphs; efficient marshalling of such data structures is nontrivial, especially if the data structures may contain cycles.

RPC may be implemented over a connectionless protocol, like UDP, or a connection-oriented protocol, like TCP. UDP has less overhead, because it does not provide reliability, flow control, or congestion control. The primary benefit of using UDP is the decreased overhead—in particular, the decreased load on the server, because in many client–server systems, servers are more heavily loaded than clients (17). For a server with hundreds or thousands of frequent or infrequent clients, the costs of establishing, maintaining, and terminating connections could cause the server to become a bottleneck. The lack of flow control in UDP is partially compensated by the fact that RPC has an intrinsic form of flow control. A caller waits for a reply after sending a single RPC request; if the procedure's arguments are not too large, this is like a sliding-window mechanism with a window size of 1. If reliability is needed, it may still be possible to exploit the asymmetric nature of RPC to provide reliability with little increase in the load on the server. Specifically, if its acceptable for the server to execute an RPC multiple times, then reliability can be achieved by incorporating a retransmission mechanism only in the client. The server sends replies unreliably; if a reply gets lost, the client retransmits the request, causing the server to repeat the RPC and resend the reply. Message loss is infrequent, so the decreased overhead at the server typically outweighs the cost of the repeated procedure calls. In contrast, with a symmetric reliable protocol like TCP, servers never execute an RPC twice, but clients and servers both buffer and retransmit their outgoing messages. To provide congestion control, a longer time-out can be used for each successive retransmission by the client; this helps clear the congestion.

An RPC can be repeated without harm if it is *idempotent,* that is, if executing it multiple times has the same effect as executing it once. All read-only operations are idempotent, and with careful interface design, many services can provide idempotent update operations as well. For example, the file access protocol in Sun NFS includes an operation that writes data at a specified offset within a file; this operation is idempotent. It does not include an operation that appends data to a file, because appending is not idempotent. Idempotent operations have an additional benefit. If a server crashes and recovers, it may be difficult or impossible to determine what operations were performed just before the crash. That information is not needed if operations are idempotent: even if the server crashed after executing the procedure call and before sending the reply, it is safe for the client to retransmit the request and have the recovered server reexecute it. Thus, use of idempotent operations helps make server failures transparent to clients.

RPC has several limitations. Typically, procedures that use global variables cannot be called remotely. Similarly, procedures that perform input or output (to screen, disk, printer, etc.) generally produce different effects if called remotely. In some systems, aliasing among input arguments is not preserved when arguments are marshalled. For example, a procedure's arguments might include an integer x and an integer array a. The procedure's return value might depend on whether x is aliased to some element of a. However, straightforward implementations of marshalling would not necessarily preserve such aliasing. Marshalling entire arrays or linked data structures may be inefficient, especially if the procedure only reads or writes a small fraction of the data. Uninitialized pointer variables may cause problems when marshalling linked data structures. RPC is poorly suited to communication of continuous streams of data, such as video, and to communication involving more than two parties. Finally, an RPC can fail in more ways than a local procedure call; for example, an RPC can fail because the remote computer crashed or is running an incompatible version of the software. A mechanism is needed to report such errors to the caller, for example, by introducing new exceptions.

Distributed Shared Memory

RPC takes a specific program construct—namely, procedure call—and extends it to operate remotely. Distributed shared memory (DSM) (18) takes two program constructs—namely, memory read and memory write—and extends them to operate remotely. Shared memory is attractive because it provides a unifying model for programming different types of concurrent systems: multiple threads in one process, multiple processes on one uniprocessor or multiprocessor computer, and—with DSM—multiple processes on different computers. For collections of peer processes that share data, DSM allows a more natural programming style than RPC. Primarily, this is because DSM hides from the application programmer decisions about where data should be stored and when data need to be transmitted. With RPC, procedure declarations and interfaces explicitly indicate what data to send. In contrast, a DSM system automatically transmits and stores data as needed to make it available to all processes.

Two important dimensions for classifying DSM systems are the consistency model and the unit of sharing. The *consistency model* specifies when the effect of an update becomes visible on other computers, that is, when memory reads on other computers should return the newly written value. The behavior of a centralized memory is characterized by *strict consistency:* any read to a memory location a returns the value stored by the most recent write to a [(3), chapter 6]. Implementing strict consistency in a distributed system is prohibitively expensive. A slightly weaker model is *sequential* *consistency:* the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appears in this sequence in the order specified by its program (19). Intuitively, sequential consistency differs from strict consistency by allowing a read to return an "old" value if there is no way for any process to determine that the returned value is old. Implementing sequential consistency can incur significant overhead, so a multitude of weaker models have been proposed; Tanenbaum provides a good overview [(3), chapter 6]. Weaker models incur less overhead but are harder for application programmers to use, because weaker models are farther from providing the illusion of a centralized shared memory.

The *unit of sharing* specifies the chunks of data that are necessarily stored and transmitted together. DSM can be viewed as an extension to a traditional virtual-memory system, in which invalid pages are fetched from other computers instead of from disk. From this perspective, it is natural to use a page of memory as the unit of sharing, as in (18). This allows the DSM implementation to exploit hardware and operating-system support for virtual memory. When a shared page is not available locally, it is marked as invalid in the process's page table, so an access to that page causes a page fault. The page fault handler requests the page from an appropriate computer (as described below) and blocks the process. When the page arrives, the process is unblocked, with the program counter pointing to the instruction that caused the page fault.

Enforcing sequential consistency is easy in implementations where there is always at most one copy of each object. To efficiently support objects that are read concurrently by several computers, most implementations of DSM allow objects to be *replicated,* that is, allow multiple copies to exist. A typical protocol for ensuring sequential consistency in such a system works as follows: Each copy of a sharing unit (SU) is tagged as read-only or read-write. Before writing to a SU, a computer must acquire the SU in read-write mode. When a computer acquires a SU in read-write mode, all other copies of that SU are invalidated. The process with the read-write copy (or, if there is none, the last process to have such a copy) is called the *owner.* The owner maintains a list of the computers having read-only copies of the SU. When a computer wants a copy of a SU, it sends a request to the owner. When the owner receives a request for a read-only copy, the owner makes its copy read-only. When the owner receives a request for a read-write copy, it invalidates its own copy and tells all other machines with read-only copies to invalidate them; when those other machines have replied to the invalidation message, the owner grants a read-write copy (and hence ownership) to the requester. How does a computer find the owner of a SU? A simple approach is to designate for each SU a particular computer called its *manager.* The manager keeps track of the owner of the SU. Thus, to obtain a copy of a SU, a computer sends a request to the manager, which forwards the request to the owner, which replies to the requester.

Synchronization constructs, such as semaphores, require special treatment in DSM implementations, to avoid busy-waiting loops that repeatedly access shared variables; such loops would cause excessive communication.

Page-based DSM suffers from *false sharing:* if two shared variables happen to be on the same page, and one computer repeatedly writes to one of them, and another computer repeatedly reads (or writes) the other, then there will be significant inefficiency as one (or both) copies of the page repeatedly get(s) invalidated. To avoid this problem, some DSM systems take the unit of sharing to be a single shared variable, rather than a page. The page-fault-based implementation described above can still be used if each shared variable is put a separate page. Another benefit of variable-based DSM is that shared variables are explicit in the application program, so hints about typical access patterns for each variable can be obtained from program analysis or from programmer annotations. Based on these hints, the DSM system can increase efficiency by using different implementations for different shared variables. In short, compared to page-based DSM, variable-based DSM is higher-level and provides more opportunity for exploiting high-level information about programs. A logical next step in the same direction is object-based shared memory, or distributed objects.

Shared Objects

In object-oriented programming, an *object* encapsulates both data and *methods,* that is, procedures that access the data in the object. For example, a stack object includes data (the sequence of items on the stack) and some methods (e.g., push, pop, and is-empty?) that access those data. Objects are typed; the types are called *classes.* Objects provide modularity, because (normally) the data in an object can be accessed only by that object's methods. The concept of *shared objects* is a natural generalization of the concept of shared variables. In concurrent programming, a major advantage of shared objects over shared variables is that common patterns of synchronization, such as mutually exclusive access to an object, can be expressed in declaratively in class definitions and implemented by the run-time system of the programming language, thereby reducing the burden on the application programmer.

Just as objects combine data and methods, *shared objects* combine aspects of DSM and RPC. A shared object system, like a DSM system, hides from the application programmer decisions about where to store and when to transmit objects. If a computer does not have a copy of an object when a method is called, the shared object system can either obtain a local copy, as for a shared variable in DSM, or invoke the method remotely, like an RPC. The latter is called *remote method invocation* (RMI). Shared objects can be implemented by combining implementation techniques for RPC and DSM. This approach underlies the shared objects provided by the Orca programming language (20).

Most current implementations of distributed object systems are simpler (hence, for some access patterns, slightly more efficient, but for some access patterns, much less efficient) than the DSM-like shared objects described above. Specifically, most current implementations do not support replication of objects and do not allow the owner of an object to change. Consequently, all invocations of the methods of a particular object are executed on the same computer, regardless of which computer invoked them. For example, this is the case for distributed objects in version 1.1 of the Java programming language (21). (Objects are sometimes copied, but this is fundamentally different from replication. An update to a copy of an object has no effect on the original or other copies.) However, it is expected that future implementations will support replication.

OPTIMIZING COMMUNICATION SOFTWARE

As network hardware continues to improve, software is becoming the bottleneck in many communication-intensive applications. Specialized optimizations can greatly improve the performance of communication software. We consider two important classes of optimizations: copy elimination and integrated layer processing.

Sending a message can involve copying the contents of the message multiple copies. For example, the message might be copied from the address space of the sending user process into a buffer in the operating system kernel (e.g., because the network interface is busy, so the message can't be sent immediately) and then copied to the network interface for transmission. Similarly, receiving a message might involve copying the message from a buffer on the network interface into a kernel buffer and then into the address space of the user process. Copies between kernel buffers and user space can be eliminated by exploiting hardware support for page-based virtual memory [(3), chapter 2]. By manipulating the page table (or a corresponding data structure, depending on the system), a page—and hence the data on that page—can be moved between address spaces. Thus, if each message is put on a separate page, such manipulations can replace one copy operation at the sender and another at the receiver. For short messages, the benefit is negligible; for large messages, the benefit can be significant. Some experimental systems achieve "zero-copy" communication by reprogramming the network interface to directly access buffers in user address spaces (22); an additional benefit of this design is that messages can be sent and received without the participation of the kernel.

Integrated layer processing (ILP) reduces the overhead from modular (layered) implementation of communication software (23,24). Consider a protocol stack containing two or more layers that each access every byte of a message (e.g., layers that compute a checksum or put the data into a standard format for transmission). In a strictly layered implementation, each byte of the message is loaded into a register (from cache or main memory), processed, and then stored (into cache or main memory) in each of these layers. Combining these separate loops into a single loop reduces the number of loads and stores: each byte of the message is loaded once, processed by the operations from all layers, and then stored once. Having the programmer combine the loops manually is tedious and destroys modularity. A more promising approach is to have a program transformation system that automatically integrates the loops (25).

BIBLIOGRAPHY

1. H. Zimmermann, OSI reference model—The ISO model of architecture for open systems interconnection, *IEEE Trans. Commun.,* **COM-28**: 425–432, 1980.
2. D. E. Comer, *Computer Networks and Internets,* Englewood Cliffs, NJ: Prentice-Hall, 1997.
3. A. S. Tanenbaum, *Distributed Operating Systems,* Englewood Cliffs, NJ: Prentice-Hall, 1995.
4. V. Jacobson, Congestion avoidance and control, *Proc. SIGCOMM '88,* Stanford, CA: ACM Press, 1988.
5. J. H. Saltzer, D. P. Reed, and D. D. Clark, End-to-end arguments in system design, *ACM Trans. Comput. Syst.,* **2** (4): 277–288, 1984.
6. A. S. Tanenbaum, *Computer Networks,* 3rd ed., Englewood Cliffs, NJ: Prentice-Hall, 1996.
7. W. Stallings, *Data and Computer Communications,* 5th ed., Englewood Cliffs, NJ: Prentice-Hall, 1997.
8. K. P. Birman, The process group approach to reliable distributed computing, *Commun. ACM,* **36** (12): 36–53, 1993.
9. F. Cristian, Reaching agreement on processor group membership in synchronous distributed systems, *Distrib. Comput.,* **4** (4): 175–187, 1991.
10. A. Ricciardi, Consistent process membership in asynchronous environments, in K. P. Birman and R. van Renesse (eds.), *Reliable Distributed Computing with the Isis Toolkit,* Los Alamitos, CA: IEEE Computer Society Press, 1993, Chap. 13.
11. D. Dolev and D. Malki, The transis approach to high availability cluster communication, *Commun. ACM,* **39** (4): 87–92, 1996.
12. A. D. Birrell and B. J. Nelson, Implementing remote procedure calls, *ACM Trans. Comput. Syst.,* **2**: 39–59, 1984.
13. B. H. Tay and A. L. Ananda, A survey of remote procedure calls, *Oper. Syst. Rev.,* **24**: 68–79, 1990.
14. R. Srinivasan, RPC: Remote Procedure Call specification version 2. Request for Comments 1831, Internet Engineering Task Force, August 1995.
15. K. Arnold and J. Gosling, *The Java Programming Language,* 2nd ed., Reading, MA: Addison-Wesley, 1998.
16. R. Sandberg, *The Sun Network File System: Design, Implementation, and Experience,* Mountain View, CA: Sun Microsystems, Inc., 1987.
17. M. Satyanarayanan, Scalable, secure, and highly available distributed file access, *IEEE Comput.,* **23** (5): 9–21, 1990.
18. K. Li and P. Hudak, Memory coherence in shared virtual memory systems, *ACM Trans. Comput. Syst.,* **7**: 321–359, 1989.
19. L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, *IEEE Trans. Comput.,* **C-28**: 690–691, 1979.
20. H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, Orca: A language for parallel programming of distributed systems, *IEEE Trans. Softw. Eng.,* **18**: 190–205, 1992.
21. C. S. Horstmann and G. Cornell, *Core Java 1.1, Volume II—Advanced Features,* Englewood Cliffs, NJ: Prentice-Hall, 1998.
22. T. von Eicken, et al., U-net: A user-level network interface for parallel and distributed computing, *Proc. 15th ACM Symp. Operating Syst. Principles,* Copper Mountain Resort, CO, 1995, pp. 40–53.
23. D. Clark and D. Tennenhouse, Architectural considerations for a new generation of protocols, *Proc. ACM SIGCOMM '90,* Philadelphia, PA, September 1990, pp. 201–208.
24. M. Abbott and L. Peterson, Increasing network throughput by integrating protocol layers, *IEEE/ACM Trans. Netw.,* **1** (5): 600–610, 1993.
25. T. Proebsting and S. Watterson, Filter fusion, *Proc. 23rd ACM Symp. Principles Programming Languages,* St. Petersburg Beach, FL, 1996, pp. 119–130.

SCOTT D. STOLLER
Indiana University

COMPUTER DISPLAYS. See FLAT PANEL DISPLAYS.