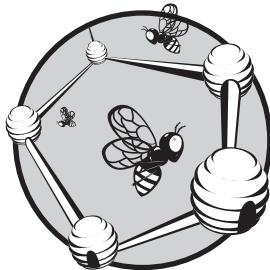


62

DHCP CONFIGURATION AND OPERATION



The big news in DHCP is dynamic address allocation, along with the concept of address leasing. It is this new functionality that makes DHCP significantly more complex than its predecessor, the Boot Protocol (BOOTP). BOOTP is a simple request/reply protocol—a server only needs to look up a client’s hardware address and send back the client’s assigned IP address and other parameters. In contrast, DHCP clients and servers must do much more to carry out both parameter exchange and the many tasks needed to manage IP address leasing.

In this chapter, I delve into the nuts and bolts of how DHCP operates. I begin with two background topics. The first provides an overview of the responsibilities of clients and servers in DHCP, and shows in general terms how they relate to each other. The second discusses DHCP configuration parameters and how they are stored and communicated.

In the rest of the chapter, I illustrate the operation of DHCP in detail. I explain the DHCP client *finite state machine*, which will give you a high-level look at the entire client lease life cycle, including address allocation,

reallocation, renewal, rebinding, and optionally, lease termination. This theoretical description is then used as the basis for several topics that explain the actual processes by which DHCP client lease activities occur. These show the specific actions taken by both client and server and when and how DHCP messages are created and sent. The last part of the chapter describes the special mechanism by which a device not using DHCP for address allocation can request configuration parameters.

DHCP Overview of Client and Server Responsibilities

DHCP is the newest and most current TCP/IP host configuration protocol. However, as you saw in the previous chapter, it wasn't built from scratch—it was designed as an extension of BOOTP. In many ways, DHCP is like BOOTP with more features, and this can be seen in the basic setup of the protocol and how it works.

Both BOOTP and DHCP are designed based on the common TCP/IP model of client/server operation (see Chapter 8). In any interaction, one device plays the role of client and the other server. Each has specific responsibilities and must send and receive messages following the protocol described in the DHCP standard. The difference is that where BOOTP involves relatively little work for servers and clients and uses a simple single-message exchange for communication, DHCP requires that both servers and clients do more, and it uses several types of message exchanges.

DHCP Server Responsibilities

A *DHCP server* is a network device that has been programmed to provide DHCP services to clients. The server plays a central role in DHCP because DHCP's main function is host configuration, and the server configures hosts (clients) that communicate with it. Smaller networks may have only a single server to support many clients, while larger networks may use multiple servers. Regardless of the number of servers, each will usually service many clients.

The following are the key responsibilities of servers in making DHCP work:

Address Storage and Management DHCP servers are the owners of the addresses used by all DHCP clients. The server stores the addresses and manages their use, keeping track of which addresses have been allocated and which are still available.

Configuration Parameter Storage and Management DHCP servers also store and maintain other parameters that are intended to be sent to clients when requested. Many of these are important configuration values that specify in detail how a client is to operate.

Lease Management DHCP servers use leases to dynamically allocate addresses to clients for a limited time. The DHCP server maintains information about each of the leases it has granted to clients, as well as policy information such as lease lengths.

Response to Client Requests DHCP servers respond to different types of requests from clients to implement the DHCP communication protocol. This includes assigning addresses; conveying configuration parameters; and granting, renewing, and terminating leases.

Administration Services To support all of its other responsibilities, the DHCP server includes functionality to allow a human administrator to enter, view, change, and analyze addresses, leases, parameters, and all other information needed to run DHCP.

DHCP Client Responsibilities

A *DHCP client* is any device that sends DHCP requests to a server to obtain an IP address or other configuration information. Due to the advantages of DHCP, most host computers on TCP/IP internetworks today include DHCP client software, making them potential DHCP clients if their administrator chooses to enable the function. There are several main responsibilities of a DHCP client:

Configuration Initiation The client takes the *active* role by initiating the communication exchange that results in it being given an IP address and other parameters. The server, in contrast, is *passive* and will not really do anything for the client until the client makes contact.

Configuration Parameter Management The client maintains parameters that pertain to its configuration, some or all of which may be obtained from a DHCP server.

Lease Management Assuming its address is dynamically allocated, the client keeps track of the status of its own lease. It is responsible for renewing the lease at the appropriate time, rebinding if renewal is not possible, and terminating the lease early if the address is no longer needed.

Message Retransmission Since DHCP uses the unreliable User Datagram Protocol (UDP, see Chapter 44) for messaging, clients are responsible for detecting message loss and retransmitting requests if necessary.

DHCP Client/Server Roles

The DHCP server and client obviously play complementary roles. The server maintains configuration parameters for all clients. Each client maintains its own parameters, as discussed in the next section.

IP address assignment and lease creation, renewal, rebinding, and termination are accomplished through specific exchanges using a set of eight DHCP message types, as discussed in the “DHCP General Operation and Client Finite State Machine” and “DHCP Lease Allocation, Reallocation, and Renewal” sections later in this chapter. To accomplish this messaging, special rules are followed to generate, address, and transport messages, as explained in Chapter 63.

DHCP Relay Agents

Like BOOTP, DHCP also supports a third type of device: the *relay agent*. Relay agents are neither clients nor servers, but rather intermediaries that facilitate cross-network communication between servers and clients. They are described in more detail in Chapter 64 (where you can also find more of the implementation details of servers and clients).

KEY CONCEPT *DHCP servers* are devices programmed to provide DHCP services to clients. They manage address information and other parameters and respond to client configuration requests. *DHCP clients* are TCP/IP devices that have been set to use DHCP to determine their configuration. They send requests and read responses, and are responsible for managing their own leases, including renewing or rebinding a lease when necessary.

DHCP Configuration Parameters, Storage, and Communication

One of the more important oversights in DHCP's predecessor, BOOTP, was that it allowed a server to tell a client only three pieces of information: its IP address, the name of the server it could use to download a boot file, and the name of the boot file to use. This was a result of BOOTP's legacy as a protocol created primarily to let diskless workstations be bootstrapped.

Obviously, the IP address is a very important parameter, but in modern networks it isn't the only one that a client needs to be given for it to function properly. A typical host needs to be given other essential information to allow it to know how it should operate on its local network and interact with other devices. For example, it needs to know the address of a default local router, the subnet mask for the subnet it is on, parameters for creating outgoing IP datagrams, and much more.

Configuration Parameter Management

The inability to specify additional configuration parameters in BOOTP was resolved by using the special BOOTP Vendor-Specific Area for vendor-independent *vendor information fields*, as first defined in RFC 1048. In DHCP, this idea has been extended further, and more important, formalized, as part of the effort to make DHCP a more general-purpose configuration tool. Configuration parameter storage, maintenance, and communication are no longer optional features; they are an essential part of the host configuration process.

Just as DHCP servers are the bosses that own and manage IP addresses, they also act as the repository for other configuration parameters that belong to DHCP clients. This centralization of parameter storage provides many of the same benefits that centralizing IP addresses in DHCP does: Administrators can check and adjust parameters in a single place, rather than needing to go to each client machine.

Each DHCP server is programmed with parameters that are to be communicated to clients in addition to an IP address when an address is assigned. Alternatively, a client that has already been assigned an address using some other mechanism may

still query the DHCP server to get parameter information, using the DHCPINFORM message type. (This was actually added to the protocol in RFC 2131; it was not in the original DHCP standard.)

Parameter Storage

The exact method of storage of client parameters is to some extent implementation-dependent. Typically, there will be some parameters that apply to all clients. For example, on a small network with only one router, that router will probably be the default router for every DHCP client, regardless of address.

The DHCP server will also have certain parameters that are client-specific. The IP address itself is an obvious example, but there are other parameters that may apply to only certain clients on a network. These parameters are stored in some sort of a database and indexed using a particular *client identifier*. The default identifier consists of the client's IP subnet number and its hardware address. Thus, when a server gets a request from a particular subnet, it can use the client's hardware address in the request to look up client-specific parameters and return them. The client identifier can be changed if a different identification scheme is desired.

Clients are also responsible for storing their own parameters. Many of these will be obtained from the DHCP server, although some may be supplied in other ways. The specific implementation of the client determines which parameters it considers important and how they are discovered.

Configuration Parameter Communication

Communication of configuration parameters between DHCP clients and servers is accomplished using *DHCP options*, which replace BOOTP vendor information fields. A number of options were defined when DHCP was first created, and additional new ones have been created over the years.

Today, there are several dozen DHCP options. Obviously, the ability to have so many different parameters automatically delivered to a client provides a great deal of host configuration flexibility to administrators. DHCP options are described further in Chapter 63.

DHCP General Operation and the Client Finite State Machine

Dynamic address allocation is probably the most important new capability introduced by DHCP. In the previous chapter, I discussed the significance of the change from IP address *ownership* to IP address *leasing*. I also provided a high-level look of the activities involved in leasing, by providing an overview of the DHCP lease life cycle.

An overview of this sort is useful to get a general handle on how leases work, but to really understand the mechanics of DHCP address assignment and client/server communication, you need more details on how the devices behave and the messages they send. One tool often employed by networking engineers to describe a protocol is a theoretical model called a *finite state machine (FSM)*. Using this technique, the protocol's specific behavior is illustrated by showing the different *states* a device can be in, what possible *transitions* exist from one state to another, what

events cause transitions to occur, and what *actions* are performed in response to an event. The TCP operational overview contains more general background information on FSMs (see Chapter 47).

The DHCP standard uses an FSM to describe the lease life cycle from the perspective of a DHCP client. The client begins in an initial INIT state where it has no lease and then transitions through various states as it acquires, renews, rebinds, and/or releases its IP address. The FSM also indicates which message exchanges occurs between the server and client at various stages.

NOTE *The DHCP standard does not describe the DHCP server's behavior in the form of a FSM; only the client's is described this way.*

Some people think FSMs are a little dense and hard to understand, and I can see why. You can skip this topic, of course, but I think the FSM provides a useful way of illustrating in a comprehensive way most of the behavior of a DHCP client.

Table 62-1 describes each of the DHCP client states, and summarizes the messages sent and received by the client in each, as well as showing the state transitions that occur in response. The FSM's states, events, and transitions are easier to envision in Figure 62-1, which also incorporates a shading scheme so you can see which states are associated with each of the main DHCP processes.

Table 62-1: DHCP Client Finite State Machine

State	State Description	Event and Transition
INIT	This is the initialization state, where a client begins the process of acquiring a lease. It also returns here when a lease ends or when a lease negotiation fails.	Client sends DHCPDISCOVER. The client creates a DHCPDISCOVER message and broadcasts it to try to find a DHCP server. It transitions to the SELECTING state.
SELECTING	The client is waiting to receive DHCPOFFER messages from one or more DHCP servers, so it can choose one.	Client receives offers, selects preferred offer, and sends DHCPREQUEST. The client chooses one of the offers it has been sent, and broadcasts a DHCPREQUEST message to tell DHCP servers what its choice was. It transitions to the REQUESTING state.
REQUESTING	The client is waiting to hear back from the server to which it sent its request.	Client receives DHCPACK, successfully checks that IP address is free. The client receives a DHCPACK message from its chosen server, confirming that it can have the lease that was offered. It checks to ensure that address is not already used, and assuming it is not, records the parameters the server sent it, sets the lease timers T1 and T2, and transitions to the BOUND state.
		Client receives DHCPACK, but IP address is in use. The client receives a DHCPACK message from its chosen server, confirming that it can have the lease that was offered. However, it checks and finds the address already in use. It sends a DHCPDECLINE message back to the server and returns to the INIT state.
		Client receives DHCPNAK. The client receives a DHCPNAK message from its chosen server, which means the server has withdrawn its offer. The client returns to the INIT state.

(continued)

Table 62-1: DHCP Client Finite State Machine (continued)

State	State Description	Event and Transition
INIT-REBOOT	When a client that already has a valid lease starts up after a power down or reboot, it starts here instead of the INIT state.	Client sends DHCPREQUEST. The client sends a DHCPREQUEST message to attempt to verify its lease and reobtain its configuration parameters. It then transitions to the REBOOTING state to wait for a response.
REBOOTING	A client that has rebooted with an assigned address is waiting for a confirming reply from a server.	Client receives DHCPACK, successfully checks that IP address is free. The client receives a DHCPACK message from the server that has its lease information, confirming that the lease is still valid. To be safe, the client checks anyway to ensure that the address is not already in use by some other device. Assuming it is not, the client records the parameters the server sent it and transitions to the BOUND state.
		Client receives DHCPACK, but IP address is in use. The client receives a DHCPACK message from the server that had its lease, confirming that the lease is still valid. However, the client checks and finds that while the client was offline, some other device has grabbed its leased IP address. The client sends a DHCPDECLINE message back to the server and returns to the INIT state to obtain a new lease.
		Client receives DHCPNAK. The client receives a DHCPNAK message from a server. This tells it that its current lease is no longer valid; for example, the client may have moved to a new network where it can no longer use the address in its present lease. The client returns to the INIT state.
BOUND	A client has a valid lease and is in its normal operating state.	Renewal timer (T1) expires. The client transitions to the RENEWING state.
		Client terminates lease and sends DHCPRELEASE. The client decides to terminate the lease (due to user command, for example). It sends a DHCPRELEASE message and returns to the INIT state.
RENEWING	A client is trying to renew its lease. It regularly sends DHCPREQUEST messages with the server that gave it its current lease specified, and waits for a reply.	Client receives DHCPACK. The client receives a DHCPACK reply to its DHCPREQUEST. Its lease is renewed, it restarts the T1 and T2 timers, and it returns to the BOUND state.
		Client receives DHCPNAK. The server has refused to renew the client's lease. The client goes to the INIT state to get a new lease.
		Rebinding timer (T2) expires. While the client is attempting to renew its lease, the T2 timer expires, indicating that the renewal period has ended. The client transitions to the REBINDING state.
REBINDING	The client has failed to renew its lease with the server that originally granted it and now seeks a lease extension with any server that can hear it. It periodically sends DHCPREQUEST messages with no server specified, until it gets a reply or the lease ends.	Client receives DHCPACK. Some server on the network has renewed the client's lease. The client binds to the new server granting the lease, restarts the T1 and T2 timers, and returns to the BOUND state.
		Client receives DHCPNAK. A server on the network is specifically telling the client it needs to restart the leasing process. This may be the case if a new server is willing to grant the client a lease, but only with terms different from the client's current lease. The client goes to the INIT state.
		Lease expires. The client receives no reply prior to the expiration of the lease. It goes back to the INIT state.

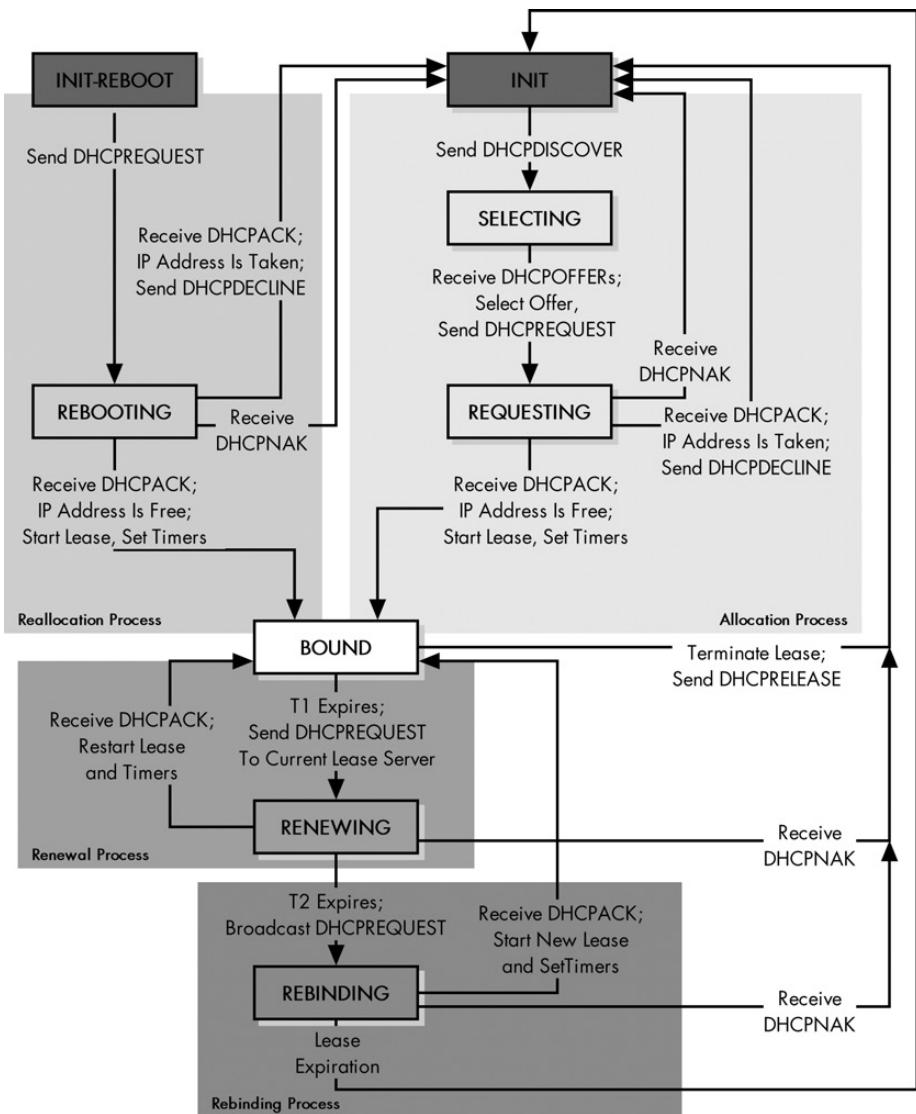


Figure 62-1: DHCP client finite state machine This diagram shows the finite state machine (FSM) used by DHCP clients. The shaded background areas show the transitions taken by a DHCP client as it moves through the four primary DHCP processes: allocation, reallocation, renewal, and rebinding.

This is just a summary of the FSM, and it does not show every possible event and transition, since it is complex enough already. For example, if a client that received two offers in the SELECTING state receives a DHCPNAK from its chosen server in the REQUESTING state, it may choose to send a new DHCPREQUEST to its second choice, instead of starting over from scratch. Also, the client must have logic that lets it time out if it receives no reply to sent messages in various states, such as not receiving any offers in the SELECTING state. The next sections discuss these matters in more detail.

Also note that this FSM applies to dynamically allocated clients—that is, ones with conventional leases. A device configured using automatic allocation will go through the same basic allocation process, but does not need to renew its lease. The process for manual allocation is somewhat different.

DHCP Lease Allocation, Reallocation, and Renewal

To implement DHCP, an administrator must first set up a DHCP server and provide it with configuration parameters and policy information: IP address ranges, lease length specifications, and configuration data that DHCP hosts will need to be delivered to them. Host devices can then have their DHCP client software enabled, but nothing will happen until the client initiates communication with the server. When a DHCP client starts up for the first time, or when it has no current DHCP lease, it will be in an initial state where it doesn't have an address and needs to acquire one. It will do so by initiating the process of *lease allocation*.

Before we examine the steps in the lease allocation, reallocation, and renewal processes, I need to clarify some issues related to DHCP lease communications. First, DHCP assumes that clients will normally broadcast messages, since they don't know the address of servers when they initiate contact, but that servers will send replies back unicast to the client. This can be done even before the client has an IP address, by sending the message at the link layer. Some clients don't support this and require that messages to them be broadcast instead.

DHCP uses many of the same basic fields as BOOTP, but much of the extra information the protocol requires is carried in DHCP *options*. Some of these options aren't really optional, despite the name—they are needed for the basic function of DHCP. An obvious example is the DHCP Message Type option, which is what specifies the message type itself.

The details of how messages are created and addressed, along with a full description of all DHCP fields and options, are presented in Chapter 63.

NOTE I have assumed that no relay agents are in use here. See the discussion of DHCP/BOOTP relay agents in Chapter 60 for more on how they change the allocation process (and other processes).

Initial Lease Allocation Process

The following are the basic steps taken by a DHCP client and server in the initial allocation of an IP address lease, focusing on the most important tasks each device performs (see Figure 62-2).

1. Client Creates DHCPDISCOVER Message

The client begins in the INIT (initialization) state. It has no IP address and doesn't even know whether or where a DHCP server may be on the network. To find one, it creates a DHCPDISCOVER message, including the following information:

- Its own hardware address in the CHAddr field of the message, to identify itself

- A random transaction identifier, put into the XID field (used to identify later messages as being part of the same transaction)

Optionally, the client may request a particular IP address using a Requested IP Address DHCP option, a particular lease length using an IP Address Lease Time option, and/or specific configuration parameters by including a Parameter Request List option in the message

2. Client Sends DHCPDISCOVER Message

The client broadcasts the DHCPDISCOVER message on the local network. The client transitions to the SELECTING state, where it waits for replies to its message.

3. Servers Receive and Process DHCPDISCOVER Message

Each DHCP server on the local network receives the client's DHCPDISCOVER message and examines it. The server looks up the client's hardware address in its database and determines if it is able to offer the client a lease and what the terms of the lease will be. If the client has made requests for a particular IP address, lease length, or other parameters, the server will attempt to satisfy these requests, but it is not required to do so. A server may decide not to offer a lease to a particular client if it has not been programmed to provide service for it, it has no remaining IP addresses, or for other reasons.

4. Servers Create DHCPOFFER Messages

Each server that chooses to respond to the client creates a DHCPOFFER message including the following information:

- The IP address to be assigned to the client, in the YIAddr field (if the server previously had a lease for this client, it will attempt to reuse the IP address it used last time; failing that, it will try to use the client's requested address, if present; otherwise, it will select any available address)
- The length of the lease being offered
- Any client-specific configuration parameters either requested by the client or programmed into the server to be returned to the client
- Any general configuration parameters to be returned to all clients or clients in this client's class
- The server's identifier in the DHCP Server Identifier option
- The same transaction ID (XID) used in the DHCPDISCOVER message

5. Servers Probe and/or Reserve Offered Address (Optional)

The DHCP standard specifies that before sending a DHCPOFFER to a client, the server should check to see that the IP address isn't already in use by sending an ICMP Echo message to that address. It is considered a key part of the DHCP server conflict detection feature (discussed in Chapter 64). This may be disabled by an administrator. Whether or not it probes the address offered, the server may also *reserve* the address so that if the client decides to use it, it will be available. This isn't mandatory, because the protocol handles the case where an offered lease is retracted. It is more efficient if servers do reserve addresses, but if IP addresses are in very short supply, such reservations may not be practical.

6. Servers Send DHCPOFFER Messages

Each server sends its DHCPOFFER message. They may not all be sent at exactly the same time. The messages are sent either unicast or broadcast, as mentioned earlier.

7. Client Collects and Processes DHCPOFFER Messages

The client waits for DHCPOFFER messages to arrive in response to its DHCPDISCOVER message. The exact behavior of the client here is implementation-dependent. The client may decide to simply take the first offer it receives, for expediency. Alternatively, it may choose to shop around by waiting for a period of time. It can then process each offer and take the one with the most favorable terms—for example, the one with the longest lease. If no DHCPOFFER messages are received, the client will enter a retransmission mode and try sending the DHCPDISCOVER again for a period of time.

8. Client Creates DHCPREQUEST Message

The client creates a DHCPREQUEST message for the server offer it has selected. This message serves two purposes: It tells the server whose offer the client has accepted, “Yes, I accept your offer, assuming it is still available,” and also tells the other servers, “Sorry, your offer was rejected.” (Well, except for the “sorry” part; servers are pretty thick-skinned about rejection.) In this message, the client includes the following information:

- The identifier of the chosen server in the DHCP Server Identifier option, so everyone knows who won
- The IP address that the DHCP server assigned the client in the DHCPOFFER message, which the client puts in the Requested IP Address DHCP option as a confirmation
- Any additional configuration parameters it wants in a Parameter Request List option in the message

9. Client Sends DHCPREQUEST Message

The client sends the DHCPREQUEST message. Since it is intended for not just the selected DHCP server, but all servers, it is broadcast. After doing this, the client transitions to the REQUESTING state, where it waits for a reply from the chosen server.

10. Servers Receive and Process DHCPREQUEST Message

Each of the servers receives and processes the client’s request message. The servers not chosen will take the message as a rejection. However, a client may select one offer, attempt to request the lease, and have the transaction not complete successfully. The client may then come back and try its second-choice offer by sending a DHCPREQUEST containing a different Server Identifier. This means that if Server A receives a single DHCPREQUEST with a Server Identifier of Server B, that doesn’t necessarily mean that Server A is finished with the transaction. For this reason, rejected servers will wait for a while before offering a previously offered lease to another client.

11. Server Sends DHCPACK or DHCPNAK Message

The chosen server will see that its lease has been selected. If it did not previously reserve the IP address that was offered to the client, it must check to make sure it is still available. If it is not, the server sends back a DHCPNAK (*negative acknowledgment*) message, which essentially means, “Never mind, that lease is no longer available.” Usually, however, the server will still have that lease. It will create a *binding* for that client, and send back a DHCPACK (*acknowledgment*) message that confirms the lease and contains all the pertinent configuration parameters for the client.

12. Client Receives and Processes DHCPACK or DHCPNAK Message

The client receives either a positive or negative acknowledgment for its request. If the message is a DHCPNAK, the client transitions back to the INIT state and starts over—back to square one (step 1). If it is a DHCPACK, the client reads the IP address from the YIAddr field, and records the lease length and other parameters from the various message fields and DHCP options. If the client receives neither message, it may retransmit the DHCPREQUEST message one or more times. If it continues to hear nothing, it must conclude that the server flaked out and go back to step 1.

13. Client Checks That Address Is Not in Use

The client device should perform a final check to ensure that the new address isn’t already in use before it concludes the leasing process. This is typically done by generating an Address Resolution Protocol (ARP) request on the local network, to see if any other device thinks it already has the IP address this client was just leased. If another device responds, the client sends a DHCPDECLINE message back to the server, which basically means, “Hey server, you messed up. Someone is already using that address.” The client then goes back to step 1 and starts over.

14. Client Finalizes Lease Allocation

Assuming that the address is not already in use, the client finalizes the lease and transitions to the BOUND state. It also sets its two lease timers, T1 and T2. It is now ready for normal operation.

As you can see in this description, there are a number of situations that may occur that require a client to retransmit messages. This is because DHCP uses UDP, which is unreliable and can cause messages to be lost. If retransmissions don’t fix a problem such as not receiving a DHCPOFFER or a DHCPACK from a server, the client may need to start the allocation process over from scratch. The client must include enough intelligence to prevent it from simply trying forever to get a lease when there may not be a point. For example, if there are no DHCP servers on the network, no number of retransmissions will help.

Thus, after a number of retries, the client will give up and the allocation process will fail. If the client is configured to use the Automatic Private IP Addressing (APIPA) feature (see Chapter 64), this is where it would be used to give the client a default address. Otherwise, the client will be, well, dead in the water.

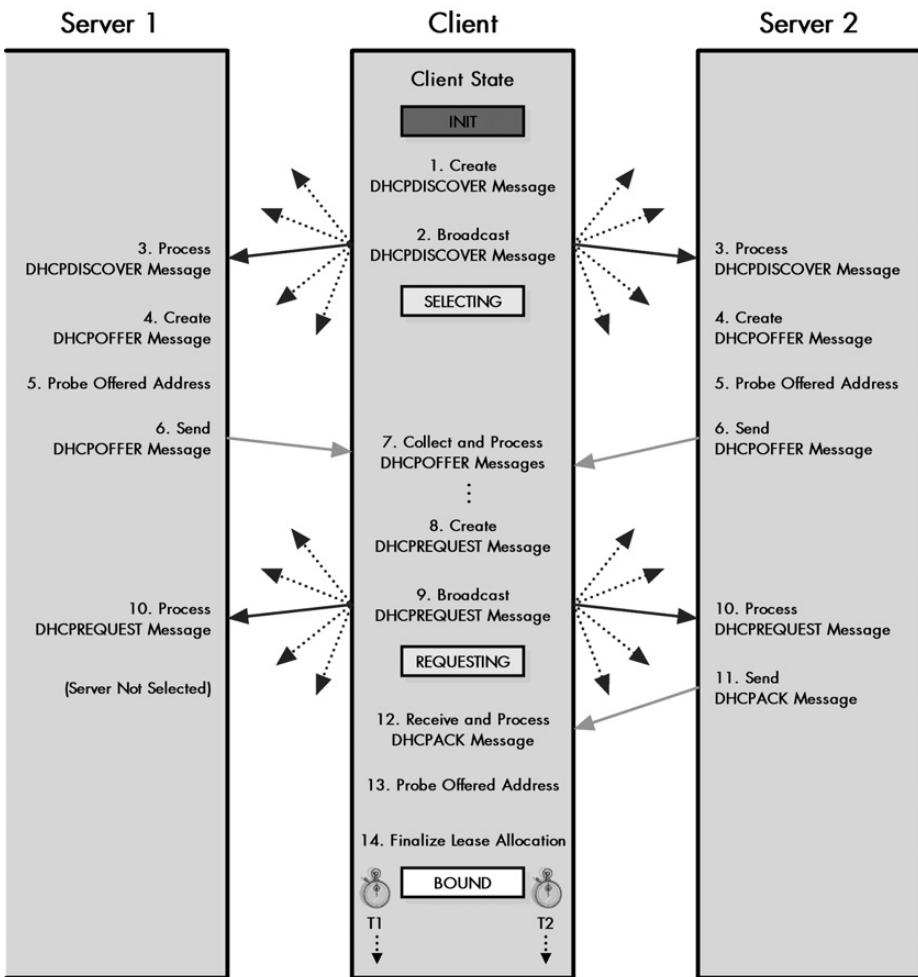


Figure 62-2: DHCP lease allocation process This diagram shows the steps involved in DHCP client lease allocation. This diagram is a bit different from most of the other client/server exchange diagrams in this book, in that I have shown two servers instead of one. This shows how a client handles responses from multiple DHCP servers and how each server reacts differently, depending on whether its lease offer was chosen by the client.

KEY CONCEPT The most important configuration process in DHCP is the *lease allocation* process, used by clients to acquire a lease. The client broadcasts a request to determine if any DHCP servers can hear it. Each DHCP server that is willing to grant the client a lease sends it an offer. The client selects the lease it prefers and sends a response to all servers telling them its choice. The selected server then sends the client its lease information.

DHCP Lease Reallocation Process

When a DHCP client starts up for the first time and has no lease, it begins in the INIT (initialize) state and goes through the allocation process described in the preceding section to acquire a lease. The same process is used when a lease ends, if a lease renewal fails, or if something happens to cause a client to need a new lease.

There are, however, certain situations in which a client starts up while it still has a lease already in place. In this situation, the client does not need to go through the entire process of getting an IP address allocation and a new lease setup. Instead, it simply tries to reestablish its existing lease, through a *reallocation process*.

A client performs reallocation rather than allocation when it restarts with an existing lease. The length of time that a client lease lasts can range from minutes to years; it is entirely a matter of the lease length policy set for the network and client by the administrator. Many, if not most, client machines are not connected to the network 24 hours a day. They are turned on during the day and then shut down at night, and also shut down on weekends. A client with a very short lease that is shut down and then later started again will probably find that its lease has expired, and it will need to get a new one. However, if a lease is longer than a few days, it will still probably be in effect when the client starts up again. Clients are also sometimes rebooted, to install new software or correct a problem. In this case, even when the lease length is very short, the restarting client will still have a valid lease when it starts up.

The reallocation process is essentially an abbreviated version of the allocation process described in the previous section. There is no need for the client to go through the whole “Yoohoo, any servers out there want to give me a lease?” routine. Instead, the client attempts to find the server that gave it the lease in the first place, seeking a confirmation that the lease is still valid and that it may resume using its previously allocated IP address. It also receives confirmation of the parameters it should use.

The following steps summarize the reallocation process (see Figure 62-3).

1. Client Creates DHCPREQUEST Message

The client begins in the INIT-REBOOT state instead of the INIT state. It creates a DHCPREQUEST message to attempt to find a server with information about its current lease. This may or may not be the server that originally granted the lease. The server responsible for a lease could, theoretically, have changed in the time since the client obtained the lease. Thus, unlike the DHCPREQUEST message in step 8 in the allocation process, the client does not include a DHCP Server Identifier option. It does include the following information:

- Its own hardware address in the CHAddr field of the message, to identify itself
- The IP address of its existing lease, in the Requested IP Address DHCP option (this address is not put into the CIAddr field)
- A random transaction identifier, put into the XID field (used to identify later messages as being part of the same transaction)
- Any additional configuration parameters it wants, in a Parameter Request List option in the message

2. Client Sends DHCPREQUEST Message

The client broadcasts the DHCPREQUEST message. It then transitions to the REBOOTING state, where it waits for a reply from a server.

3. Servers Receive and Process DHCPREQUEST Message and Generate Replies

Each server on the network receives and processes the client's request. The server looks up the client in its database, attempting to find information about the lease. Each server then decides how to reply to the client:

- If the server has valid client lease information, it sends a DHCPACK message to confirm the lease. It will also reiterate any parameters the client should be using.
- If the server determines the client lease is invalid, it sends a DHCPNAK message to negate the lease request. Common reasons for this happening are the client trying to confirm a lease after it has moved to a different network or after the lease has already expired.
- If the server has no definitive information about the client lease, it does not respond. A server is also required not to respond unless its information is guaranteed to be accurate. So, for example, if a server has knowledge of an old expired lease, it cannot assume that the lease is no longer valid and send a DHCPNAK, unless it also has certain knowledge that no other server has a newer, valid lease for that client.

4. Servers Send Replies

Servers that are going to respond to the client's DHCPREQUEST send their DHCPACK or DHCPNAK messages.

5. Client Receives and Processes DHCPACK or DHCPNAK Message

The client waits for a period of time to get a reply to its request. Again, there are three possibilities that match the three in step 3:

- The client receives a DHCPACK message, which confirms the validity of the lease. The client will prepare to begin using the lease again, and continue with step 6.
- The client receives a DHCPNAK message, which tells the client that its lease is no longer valid. The client transitions back to the INIT state to get a new lease—step 1 in the allocation process.
- If the client receives no reply at all, it may retransmit the DHCPREQUEST message. If no reply is received after a period of time, it will conclude that no server has information about its lease and will return to the INIT state to try to get a new lease.

6. Client Checks That Address Is Not in Use

Before resuming use of its lease, the client device should perform a final check to ensure that the new address isn't already in use. Even though this should not be the case when a lease already exists, it's done anyway, as a safety measure.

The check is the same as described in step 13 of the allocation process: an ARP request is issued on the local network, to see if any other device thinks it already has the IP address this client was just leased. If another device responds, the

client sends a DHCPDECLINE message back to the server, which tells it that the lease is no good because some other device is using the address. The client then goes back to the INIT state to get a new lease.

7. Client Finalizes Lease Allocation

Assuming that the address is not already in use, the client finalizes the lease and transitions to the BOUND state. It is now ready for normal operation.

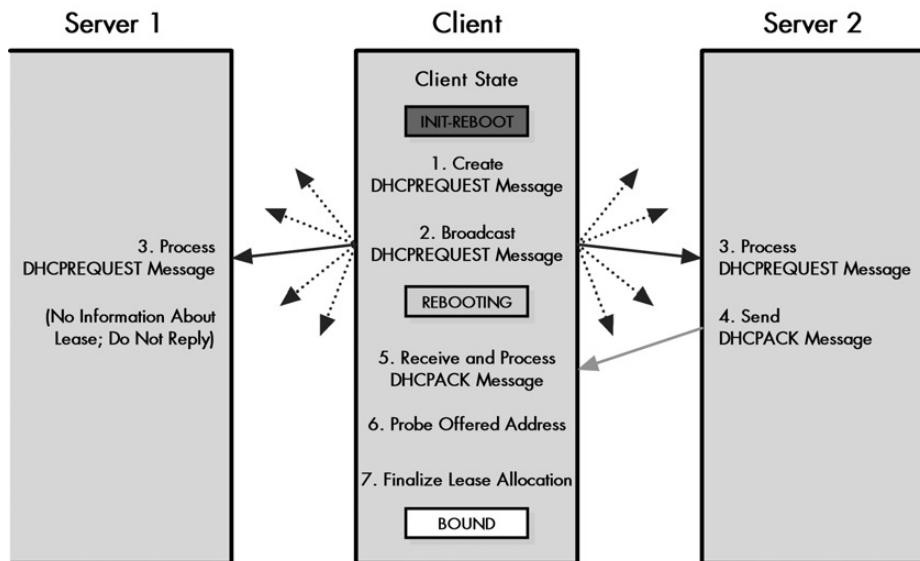


Figure 62-3: DHCP lease reallocation process The lease reallocation process consists of seven steps that correspond approximately to steps 8 through 14 of the full lease allocation process shown in Figure 62-2. In this example, the server that originally granted the lease to the client is Server 2, so it is normally the only one that responds.

KEY CONCEPT If a client starts up and already has a lease, it does not need to go through the full lease allocation process; instead, it can use the shorter *reallocation process*. The client broadcasts a request to find the server that has the current information on its lease. That server responds back to confirm that the client's lease is still valid.

DHCP Lease Renewal and Rebinding Processes

Once a DHCP client completes the allocation or reallocation process, it enters the BOUND state. The client is now in its regular operating mode, with a valid IP address and other configuration parameters it received from the DHCP server, and it can be used like any regular TCP/IP host.

While the client is in the BOUND state, DHCP essentially lies dormant. As long as the client stays on and functioning normally, no real DHCP activity will occur while in this state. The most common occurrence that causes DHCP to wake up and become active again is arrival of the time when the lease is to be *renewed*. Renewal ensures that a lease is perpetuated so it can be used for a prolonged period of time,

and involves its own message-exchange procedure. (The other way that a client can leave the BOUND state is when it terminates the lease early, as described in the next section.)

If DHCP's automatic allocation is used, or if dynamic allocation is used with an infinite lease period, the client's lease will never expire, so it never needs to be renewed. Short of early termination, the device will remain in the BOUND state forever, or at least until it is rebooted. However, most leases are finite in nature. A client must take action to ensure that its lease is extended and normal operation continues.

To manage the lease extension process, two timers are set at the time that a lease is allocated. The *renewal timer* (T_1) goes off to tell the client it is time to try to renew the lease with the server that initially granted it. The *rebinding timer* (T_2) goes off if the client is not successful in renewing with that server, and tells it to try any server to have the lease extended. If the lease is renewed or rebound, the client goes back to normal operation. If it cannot be rebound, it will expire, and the client will need to seek a new lease.

The following steps summarize the renewal/rebinding process (see Figure 62-4). Obviously, the exact sequence of operations taken by a client depends on what happens in its attempts to contact a server. For example, if it is successful with renewal, it will never need to attempt rebinding.

1. **Renewal Timer (T1) Expires**

The renewal timer, T_1 , is set by default to 50 percent of the length of the lease. When the timer goes off, the client transitions from the BOUND state to the RENEWING state. Note that a client may initiate lease renewal prior to T_1 timer expiration, if it desires.

2. **Client Sends DHCPREQUEST Renewal Message**

The client creates a DHCPREQUEST message that identifies itself and its lease. It then transmits the message directly to the server that initially granted the lease, unicast. Note that this is different from the DHCPREQUEST messages used in the allocation/reallocation processes, where the DHCPREQUEST is broadcast. The client may request a particular new lease length, just as it may request a lease length in its requests during allocation, but as always, the server makes the final call on lease length.

3. **Server Receives and Processes DHCPREQUEST Message and Creates Reply**

Assuming the server is reachable, it will receive and process the client's renewal request. There are two possible responses:

- The server decides that the client's lease can be renewed. It prepares to send to the client a DHCPACK message to confirm the lease's renewal, indicating the new lease length and any parameters that may have changed since the lease was created or last renewed.
- The server decides, for whatever reason, not to renew the client's lease. It will create a DHCPNAK message.

4. **Server Sends Reply**

The server sends the DHCPACK or DHCPNAK message back to the client.

5. Client Receives and Processes Server Reply

The client takes the appropriate action in response to the server's reply:

- If the client receives a DHCPACK message, renewing the lease, it notes the new lease expiration time and any changed parameters sent by the server, resets the T1 and T2 timers, and transitions back to the BOUND state. The client does not need to do an ARP IP address check when it is renewing.
- If the client receives a DHCPNAK message, which tells it its lease renewal request has been denied, it will immediately transition to the INIT state to get a new lease (step 1 in the allocation process).

6. Rebinding Timer (T2) Expires

If the client receives no reply from the server, it will remain in the RENEWING state and will regularly retransmit the unicast DHCPREQUEST to the server. During this period, the client is still operating normally, from the perspective of its user. If no response from the server is received, eventually the rebinding timer (T2) expires. This will cause the client to transition to the REBINDING state. Recall that by default, the T2 timer is set to 87.5 percent (seven-eighths) of the length of the lease.

7. Client Sends DHCPREQUEST Rebinding Message

Having received no response from the server that initially granted the lease, the client gives up on that server and tries to contact any server that may be able to extend its existing lease. It creates a DHCPREQUEST message and puts its IP address in the CIAddr field, indicating clearly that it presently owns that address. It then broadcasts the request on the local network.

8. Servers Receive and Process DHCPREQUEST Message and Send Reply

Each server receives the request and responds according to the information it has for the client (a server that has no information about the lease or may have outdated information does not respond):

- A server may agree to rebind the client's lease. This happens when the server has information about the client's lease and can extend it. It prepares for the client a DHCPACK message to confirm the lease's renewal, indicating any parameters that may have changed since the lease was created or last renewed.
- A server may decide that the client cannot extend its current lease. This occurs when the server determines that, for whatever reason, this client's lease should not be extended. It gets ready to send back to the client a DHCPNAK message.

9. Server Sends Reply

Each server that is responding to the client sends its DHCPACK or DHCPNAK message.

10. Client Receives Server Reply

The client takes the appropriate action in response to the two possibilities in the preceding step:

- The client receives a DHCPACK message, rebinding the lease. The client makes note of the server that is now in charge of this lease, the new lease expiration time, and any changed parameters sent by the server. It resets the T1 and T2 timers, and transitions back to the BOUND state. (It may also probe the new address as it does during regular lease allocation.)
- The client receives a DHCPNAK message, which tells it that some server has determined that the lease should not be extended. The client immediately transitions to the INIT state to get a new lease (step 1 in the allocation process).

11. Lease Expires

If the client receives no response to its broadcast rebinding request, it will, as in the RENEWING state, retransmit the request regularly. If no response is received by the time the lease expires, it transitions to the INIT state to get a new lease.

So, why bother with a two-step process: rebinding and renewal? The reason is that this provides the best blend of efficiency and flexibility. We first try to contact the server that granted the lease using a unicast request, to avoid taking up the time of other DHCP servers and disrupting the network as a whole with broadcast traffic. Usually this will work, because DHCP servers don't change that often and are usually left on continuously. If that fails, we then fall back on the broadcast, giving other servers a chance to take over the client's existing lease.

KEY CONCEPT Each client's lease has associated with it a *renewal timer (T1)*, normally set to 50 percent of the length of the lease, and a *rebinding timer (T2)*, usually set to 87.5 percent of the lease length. When the T1 timer goes off, the client will try to renew its lease by contacting the server that originally granted it. If the client cannot renew the lease by the time the T2 timer expires, it will broadcast a rebinding request to any available server. If the lease is not renewed or rebound by the time the lease expires, the client must start the lease allocation process over again.

DHCP Early Lease Termination (Release) Process

A TCP/IP host can't really do much without an IP address; it's a fundamental component of the Internet Protocol (IP), on which all TCP/IP protocols and applications run. When a host has either a manual IP address assignment or an infinite lease, it obviously never needs to worry about losing its IP address. When a host has a finite DHCP lease, it will use the renewal/rebinding process to try to hang on to its existing IP address as long as possible.

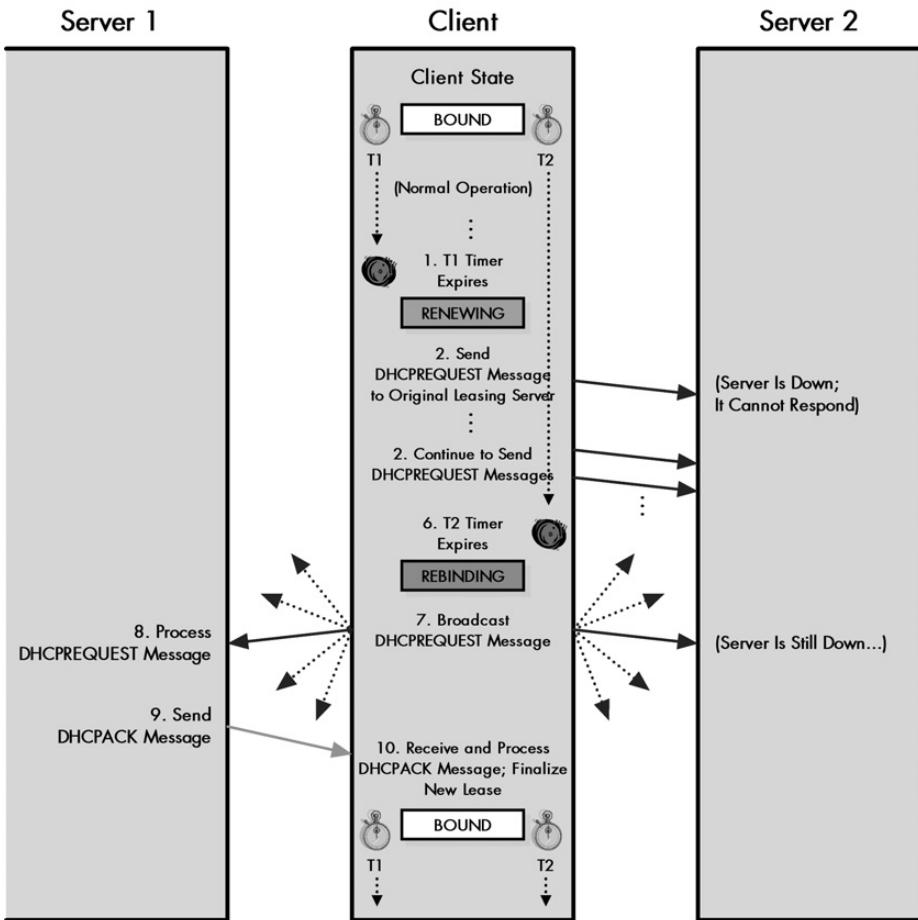


Figure 62-4: DHCP lease renewal and rebinding processes This diagram shows the example of a client presently holding a lease with Server 2 attempting to contact it to renew the lease. However, in this case, Server 2 is down for maintenance. The server is unable to respond, and the client remains stuck at step 2 in the renewal/rebinding process. It keeps sending DHCPREQUEST messages to Server 2 until its T2 timer expires. It then enters the rebinding state and broadcasts a DHCPREQUEST message, which is heard by Server 1, which agrees to extend its current lease.

So, under normal circumstances, a client will continue trying to extend its existing lease indefinitely. In certain cases, however, a host may decide to terminate its lease. This usually will not be something the client just decides to do spontaneously. It will occur in response to a specific request from the user to end the lease. A user may terminate a lease for a number of reasons, including the following:

- The client is being moved to a different network.
- The network is having its IP addresses renumbered.
- The user wants the host to negotiate a new lease with a different server.
- The user wants to reset the lease to fix some sort of problem.

In any of these cases, the user can end the lease through a process called *early lease termination* or *lease release*. This is a very simple, unidirectional communication. The client sends a special DHCPRELEASE message unicast to the server that holds its current lease, to tell it that the lease is no longer required. The server then records the lease as having been ended. It does not need to reply back to the client.

The reason that the client can just assume that the lease termination has been successful is that this is not a mandatory part of the DHCP protocol. Having clients send DHCPRELEASE to end a lease is considered a courtesy, rather than a requirement. It is more efficient to have clients inform servers when they no longer need a lease, and this also allows the IP address in the terminated lease to be reused more quickly. However, DHCP servers are designed to handle the case where a client seemingly disappears without formally ending an existing lease.

DHCP Parameter Configuration Process for Clients with Non-DHCP Addresses

The majority of DHCP clients make use of the protocol to obtain both an IP address and other configuration parameters. This is the reason why so much of DHCP is oriented around address assignment and leasing. A conventional DHCP client obtains all its configuration parameters at the same time it gets an IP address, using the message exchanges and processes described in the preceding sections of this chapter.

There are cases where a device with an IP address assigned using a method other than DHCP still wants to use DHCP servers to obtain other configuration parameters. The main advantage of this is administrative convenience; it allows a device with a static IP address to still be able to automatically get other parameters the same way that regular DHCP clients do.

Ironically, one common case where this capability can be used is in configuring DHCP servers themselves! Administrators normally do not use DHCP to provide an IP address to a DHCP server, but they may want to use it to tell the server other parameters. In this case, the server requesting the parameters actually acts as a client for the purpose of the exchange with another server.

The original DHCP standard did not provide any mechanism for this sort of non-IP configuration to take place. RFC 2131 revised the protocol, adding a new message type (DHCPINFORM) that allows a device to request configuration parameters without going through the full leasing process. This message is used as part of a simple bidirectional communication that is separate from the leasing communications we have looked at so far. Since it doesn't involve IP address assignment, it is not part of the lease life cycle, nor is it part of the DHCP client FSM.

The following steps show how a device with an externally configured address uses DHCP to get other parameters (see Figure 62-5):

1. Client Creates DHCPINFORM Message

The client (which may be a DHCP server acting as a client) creates a DHCP-INFORM message. It fills in its own IP address in the CIAddr field, since that IP address is current and valid. It may request specific parameters using the Parameter Request List option or simply accept the defaults provided by the server.

2. Client Sends DHCPINFORM Message

The client sends the DHCPINFORM message unicast, if it knows the identity and address of a DHCP server; otherwise, it broadcasts it.

3. Server Receives and Processes DHCPINFORM Message

The message is received and processed by the DHCP server or servers (if there are multiple servers and the request was broadcast). Each server checks to see if it has the parameters needed by the client in its database.

4. Server Creates DHCPACK Message

Each server that has the information the client needs creates a DHCPACK message, which includes the needed parameters in the appropriate DHCP option fields. (Often, this will be only a single server.)

5. Server Sends DHCPACK Message

The server sends the message unicast back to the client.

6. Client Receives and Processes DHCPACK Message

The client receives the DHCPACK message sent by the server, processes it, and sets its parameters accordingly.

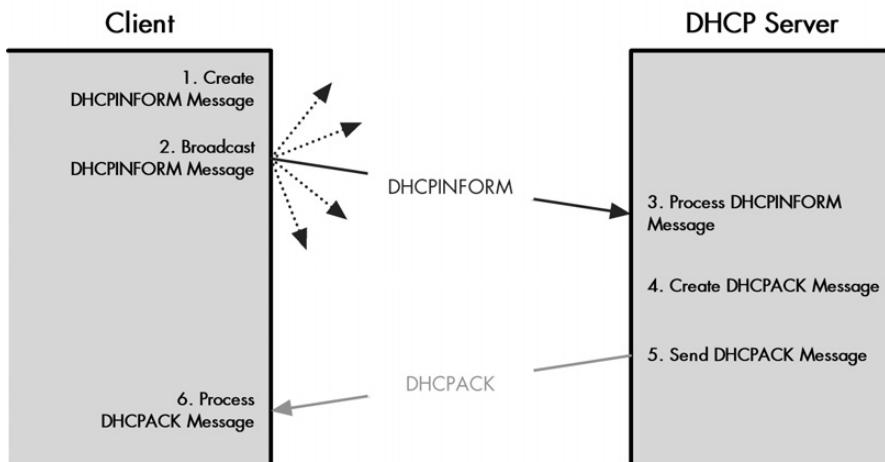


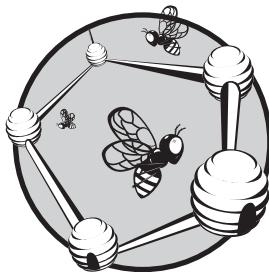
Figure 62-5: DHCP parameter configuration process A device that already has an IP address can use the simple request/reply exchange shown in this figure to get other configuration parameters from a DHCP server. In this case, the client is broadcasting its request.

If a client receives no reply to its DHCPINFORM message, it will retransmit it periodically. After a retry period, it will give up and use default configuration values. It will also typically generate an error report to inform an administrator or user of the problem.

KEY CONCEPT Devices that are not using DHCP to acquire IP addresses can still use its other configuration capabilities. A client can broadcast a DHCPINFORM message to request that any available server send it parameters for how the network is to be used. DHCP servers respond with the requested parameters and/or default parameters, carried in DHCP options of a DHCPACK message.

63

DHCP MESSAGING, MESSAGE TYPES, AND FORMATS



The preceding chapter on DHCP configuration and operation demonstrated how DHCP works by showing the various leasing and information-exchange processes. All of these procedures rely heavily on the exchange of information between the client and server, which is accomplished through DHCP *messages*. Like all protocols, DHCP uses a special message format and a set of rules that govern how messages are created, addressed, and transported.

In this chapter, I provide the details of how DHCP creates and sends messages, and show the formats used for DHCP messages and options. I begin with a description of how DHCP creates, addresses, and transports messages, and how it deals with message retransmission. I then outline the DHCP general message format, showing how it is similar to the BOOTP message format on which it is based and also where it differs. I describe DHCP options, the format used for them, and the special option overloading feature used for efficiency. I conclude the section with a complete list of DHCP options.

RELATED INFORMATION DHCP is most closely related to BOOTP in the area of messaging. DHCP options are based closely on BOOTP vendor extensions (see Chapter 60), and many of the specific DHCP option types are the same as BOOTP vendor information fields. To avoid duplication, the summary table in this chapter lists the options/extensions for both protocols, indicating which ones are used by both BOOTP and DHCP, and which are used only by DHCP.

DHCP Message Generation, Addressing, Transport, and Retransmission

As you've learned, nearly every aspect of DHCP's operation is oriented around the notion of a client device exchanging information with a server. You can also see this reflected in all of the major characteristics of DHCP messaging. This includes the format of DHCP messages, as well as the specifics of how DHCP messages are created, addressed, and transmitted, and when necessary, retransmitted.

Message Generation and General Formatting

DHCP messaging is similar in many ways to that of BOOTP, the protocol on which DHCP was based. BOOTP defined only two message types: a request and a reply. DHCP is much more complex. It uses eight different types of messages, but these are still categorized as either request or reply messages, depending on who sends them and why. DHCP uses a special DHCP Message Type option to indicate the exact DHCP message type, but still treats a message from a client seeking information as a request, and a response from a server containing information as a reply.

A client generates a message using the general DHCP message format, which is very similar to the BOOTP message format. When a server replies to a client message, it does not generate the reply as a completely new message, but rather copies the client request, changes fields as appropriate, and sends the reply back to the client. A special transaction identifier (XID) is placed in the request and maintained in the reply, which allows a client to know which reply goes with a particular request.

Message Transport

DHCP uses the User Datagram Protocol (UDP) for transport, just as BOOTP does, and for the same reasons: simplicity and support for broadcasts. It also has many of the same addressing concerns as BOOTP, as discussed in Chapter 60. Clients usually will send requests by broadcast on the local network, to allow them to contact any available DHCP server. The exception to this is when a client is trying to renew a lease with a server that it already knows. For compatibility with BOOTP, DHCP uses the same well-known (reserved) UDP port number, 67, for client requests to servers.

Some DHCP message exchanges require a server to respond back to a client that has a valid and active IP address. An example is a DHCPACK message sent in reply to a DHCPINFORM request. In this situation, the server can always send a reply unicast back to the client. Other message exchanges, however, present the same chicken-and-egg conundrum that we saw with BOOTP: If a client is using DHCP to obtain an IP address, we can't assume that IP address is available for us to use to send a reply.

In BOOTP, there were two possible solutions to this situation: The server could send back its reply using broadcast addressing as well, or the server could send back a reply directly to the host at layer 2. Due to the performance problems associated with broadcasts, DHCP tries to make the latter method the default for server replies. It assumes that a client's TCP/IP software will be capable of accepting and processing an IP datagram delivered at layer 2, even before the IP stack is initialized.

As the standard itself puts it, "DHCP requires creative use of the client's TCP/IP software and liberal interpretation of RFC 1122." RFC 1122 is a key standard describing the detailed implementation requirements of TCP/IP hosts. The DHCP standard, however, acknowledges the fact that not all devices may support this behavior. It allows a client to force servers to send back replies using broadcasts instead. This is done by the client setting the special Broadcast (B) flag to 1 in its request.

Since DHCP, like BOOTP, must use either layer 2 delivery or layer 3 broadcasts for server replies, it requires a separate well-known port number for servers to send to. Again, for compatibility with BOOTP, the same port number is used, 68. This port number is used whether a server reply is sent unicast or broadcast.

KEY CONCEPT Requests from BOOTP clients are normally sent broadcast, to reach any available DHCP server. However, there are certain exceptions, such as in lease renewal, when a request is sent directly to a known server. DHCP servers can send their replies either broadcast to the special port number reserved for DHCP clients or unicast using layer 2. The DHCP standards specify that layer 2 delivery should be used when possible to avoid unnecessary broadcast traffic.

Retransmission of Lost Messages

Using UDP provides benefits such as simplicity and efficiency to DHCP, but since UDP is unreliable, there is no guarantee that messages will get to their destination. This can lead to potential confusion on the part of a client. Consider, for example, a client sending a DHCPDISCOVER message and waiting for DHCPOFFER messages in reply. If it gets no response, does this mean that there is no DHCP server willing to offer it service or simply that its DHCPDISCOVER got lost somewhere on the network? The same applies to most other request/reply sequences, such as a client waiting for a DHCPACK or DHCPNAK message in reply to a DHCPREQUEST or DHCPINFORM message.

The fact that messages can be lost means that DHCP itself must keep track of messages sent, and if there is no response, retransmit them. Since there are so many message exchanges in DHCP, there is much that can go wrong. As in BOOTP, DHCP puts responsibility for this squarely on the shoulders of the client. This makes sense, since the client initiates contact and can most easily keep track of messages sent and retransmit them when needed. A server can't know when a client's request is lost, but a client can react when it doesn't receive a reply from the server.

In any request/reply message exchange, the client uses a retransmission timer that is set to a period of time that represents how long it is reasonable for it to wait for a response. If no reply is received by the time the timer expires, the client assumes that either its request or the response coming back was lost. The client then retransmits the request. If this request again elicits no reply, the client will continue retransmitting for a period of time.

To prevent large numbers of DHCP clients from retransmitting requests simultaneously (which would potentially clog the network), the client must use a randomized exponential backoff algorithm to determine when exactly a retransmission is made. As in BOOTP, this is similar to the technique used to recover from collisions in Ethernet. The DHCP standard specifies that the delay should be based on the speed of the underlying network between the client and the server. More specifically, it says that in a standard Ethernet network, the first retransmission should be delayed 4 seconds plus or minus a random value from 0 to 1 second; in other words, some value is chosen between 3 and 5 seconds. The delay is then doubled with each subsequent transmission (7 to 9 seconds, then 15 to 17 seconds, and so forth) up to a maximum of $64 +/- 1$ second.

To prevent it from retrying endlessly, the client normally has logic that limits the number of retries. The amount of time that retransmissions go on depends on the type of request being sent; that is, what process is being undertaken. If a client is forced to give up due to too many retries, it will generally either take some sort of default action or generate an error message.

KEY CONCEPT Like BOOTP, DHCP uses UDP for transport, which does not provide any reliability features. DHCP clients must detect when requests are sent and no response is received, and retransmit requests periodically. Special logic is used to prevent clients from sending excessive numbers of requests during difficult network conditions.

DHCP Message Format

When DHCP was created, its developers had a bit of an issue related to how exactly they should structure DHCP messages. BOOTP was already widely used, and maintaining compatibility between DHCP and BOOTP was an important goal. This meant that DHCP's designers needed to continue using the existing BOOTP message format. However, DHCP has more functionality than BOOTP, and this means that it can hold more information than can easily fit in the limited BOOTP message format.

This apparent contradiction was resolved in two ways:

- The existing BOOTP message format was maintained for basic functionality, but DHCP clients and servers were programmed to use the BOOTP message fields in slightly different ways.
- The BOOTP vendor extensions were formalized and became DHCP *options*. Despite the name *options*, some of these additional fields are for basic DHCP functionality, and they are quite mandatory!

With this dual approach, DHCP devices have access to the extra information they need. Meanwhile, the basic field format is unchanged, allowing DHCP servers to communicate with older BOOTP clients, which ignore the extra DHCP information that doesn't relate to them. See the discussion of BOOTP/DHCP interoperability (in Chapter 64) for more information.

The DHCP message format is illustrated in Figure 63-1 and described fully in Tables 63-1 and 63-2. In the table, I have specifically indicated which fields are used in DHCP in a manner similar to how they are used in BOOTP, and which are significantly different.

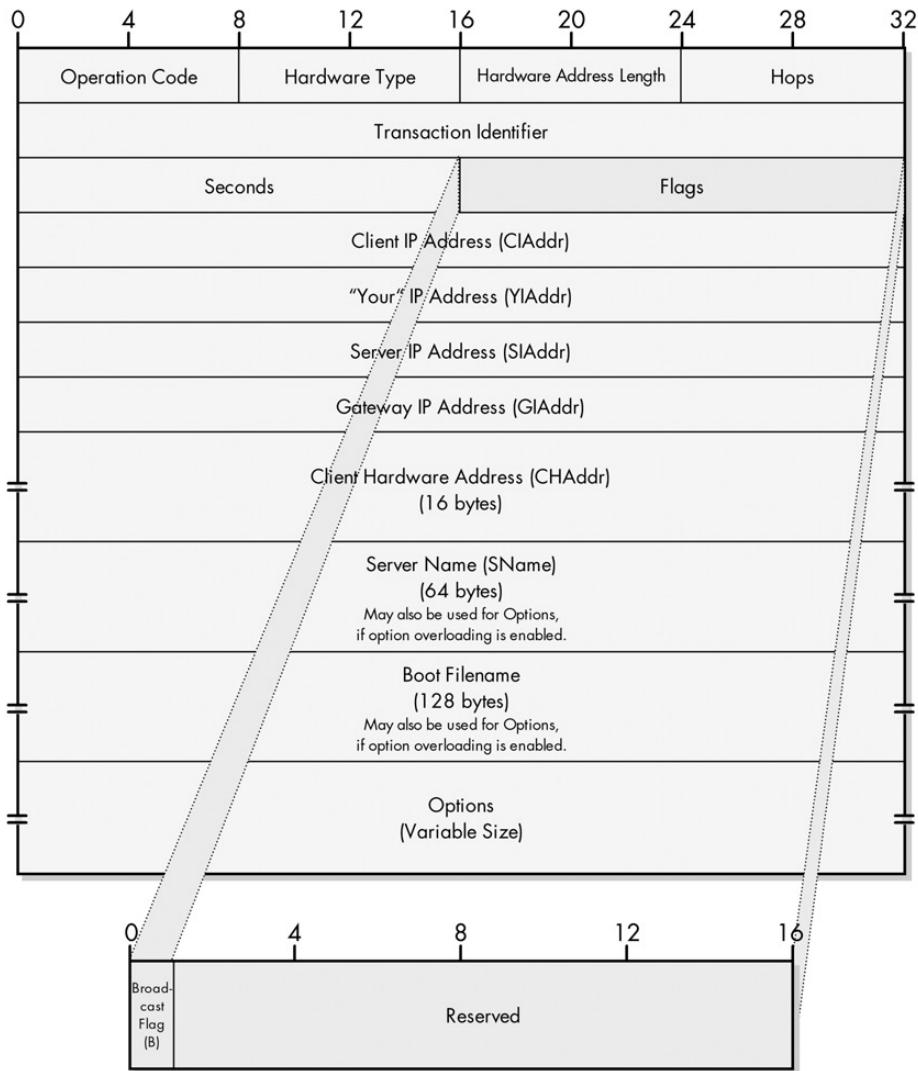


Figure 63-1: DHCP message format

Table 63-1: DHCP Message Format

Field Name	Size (Bytes)	Description
Op	1	Operation Code: This code represents the general category of the DHCP message. A client sending a request to a server uses an opcode of 1; a server replying uses a code of 2. So, for example, a DHCPREQUEST would be a request, and a DHCPACK or DHCPNAK is a reply. The actual specific type of DHCP message is encoded using the DHCP Message Type option.
HType	1	Hardware Type: This field specifies the type of hardware used for the local network, and it is used in exactly the same way as the equivalent field (HRD) in the Address Resolution Protocol (ARP) message format. Some of the most common values for this field are shown in Table 63-2.

(continued)

Table 63-1: DHCP Message Format (continued)

Field Name	Size (Bytes)	Description
Hlen	1	Hardware Address Length: Specifies how long hardware addresses are in this message. For Ethernet or other networks using IEEE 802 MAC addresses, the value is 6. This is also the same as a field in the ARP field format, HLN.
Hops	1	Hops: Set to 0 by a client before transmitting a request and used by relay agents to control the forwarding of BOOTP and/or DHCP messages.
XID	4	Transaction Identifier: A 32-bit identification field generated by the client, to allow it to match up the request with replies received from DHCP servers.
Secs	2	Seconds: In BOOTP, this field was vaguely defined and not always used. For DHCP, it is defined as the number of seconds elapsed since a client began an attempt to acquire or renew a lease. This may be used by a busy DHCP server to prioritize replies when multiple client requests are outstanding.
Flags	2	Flags: This corresponds to the formerly empty 2-byte field in the BOOTP message format defined by RFC 951, which was redefined as a Flags field in RFC 1542. The field presently contains just one flag subfield. This is the B (Broadcast) flag subfield, 1 bit in size, which is set to 1 if the client doesn't know its own IP address at the time it sends its request. This serves as an immediate indicator to the DHCP server or relay agent that receives the request that it should send its reply back by broadcast. The other subfield, which is 15 bits, is reserved, set to 0, and not used.
CIAddr	4	Client IP Address: The client puts its own current IP address in this field if and only if it has a valid IP address while in the <i>BOUND</i> , <i>RENEWING</i> , or <i>REBINDING</i> states; otherwise, it sets the field to 0. The client can only use this field when its address is actually valid and usable, not during the process of acquiring an address. Specifically, the client does not use this field to request a particular IP address in a lease; it uses the Requested IP Address DHCP option.
YIAddr	4	Your IP Address: The IP address that the server is assigning to the client.
SIAddr	4	Server IP Address: The meaning of this field is slightly changed in DHCP. In BOOTP, it is the IP address of the BOOTP server sending a BOOTREPLY message. In DHCP, it is the address of the server that the client should use for the next step in the bootstrap process, which may or may not be the server sending this reply. The sending server always includes its own IP address in the Server Identifier DHCP option.
GIAddr	4	Gateway IP Address: This field is used just as it is in BOOTP, to route BOOTP messages when BOOTP relay agents are involved to facilitate the communication of BOOTP requests and replies between a client and a server on different subnets or networks. See the description of DHCP relaying. As with BOOTP, this field is not used by clients and does not represent the server giving the client the address of a default router (that's done using the Router DHCP option).
CHAddr	16	Client Hardware Address: The hardware (layer 2) address of the client, which is used for identification and communication.
SName	64	Server Name: The server sending a DHCPOFFER or DHCPACK message may optionally put its name in this field. This can be a simple text nickname or a fully qualified DNS domain name (such as myserver.organization.org). This field may also be used to carry DHCP options, using the option overload feature, indicated by the value of the DHCP Option Overload option.
File	128	Boot Filename: Optionally used by a client to request a particular type of boot file in a DHCPDISCOVER message. Used by a server in a DHCPOFFER to fully specify a boot file directory path and filename. This field may also be used to carry DHCP options, using the option overload feature, indicated by the value of the DHCP Option Overload option.
Options	Variable	Options: Holds DHCP options, including several parameters required for basic DHCP operation. Note that this field was fixed at 64 bytes in length in BOOTP but is variable in length in DHCP. See the next section for more information. This field may be used by both the client and server.

Table 63-2: DHCP Message HTYPE Values

HTYPE Value	Hardware Type
1	Ethernet (10 Mb)
6	IEEE 802 Networks
7	ARCNet
11	LocalTalk
12	LocalNet (IBM PCNet or SYTEK LocalNET)
14	Switched Multimegabit Data Service (SMDS)
15	Frame Relay
16	Asynchronous Transfer Mode (ATM)
17	High-Level Data Link Control (HDLC)
18	Fibre Channel
19	ATM
20	Serial Line

The DHCP standard does not specify the details of how DHCP messages are encapsulated within UDP. I would assume that due to the other similarities to BOOTP, DHCP maintains BOOTP's optional use of message checksums. It also most likely assumes that messages will not be fragmented (sent with the Do Not Fragment bit set to 1 in the IP datagram). This is to allow BOOTP clients to avoid the complexity of reassembling fragmented messages.

Unlike with BOOTP, which has a fixed message size, DHCP messages are variable in length. This is the result of changing BOOTP's 64-byte Vend field into the variable-length Options field. DHCP relies on options much more than BOOTP does, and a device must be capable of accepting a message with an Options field at least 312 bytes in length. The SName and File fields may also be used to carry options, as described in the next section.

DHCP Options

When BOOTP was first developed, its message format included a 64-byte Vend field, called the Vendor-Specific Area. The idea behind this field was to provide flexibility to the protocol. The BOOTP standard did not define any specific way of using this field. Instead, the field was left open for the creators of different types of hardware to use it to customize BOOTP to meet the needs of their clients and/or servers.

Including this sort of undefined field is a good idea because it makes a protocol easily *extensible*—allowing the protocol to be easily enhanced in the future through the definition of new fields while not disturbing any existing fields. The problem with the BOOTP Vendor-Specific Area, however, is that the extensibility was vendor-specific. It was useful only for special fields that were particular to a single vendor.

What was really needed was a way to define new fields for general-purpose, vendor-independent parameter communication, but there was no field in the BOOTP message format that would let this happen. The solution came in the form of RFC 1048, which defined a technique called BOOTP *vendor information extensions*.

This method redefines the Vendor-Specific Area to allow it to carry general parameters between a client and server. This idea was so successful that it largely replaced the older vendor-specific use of the Vend field.

DHCP maintains, formalizes, and further extends the idea of using the Vend field to carry general-purpose parameters. Instead of being called vendor information extensions or vendor information fields, these fields are now called simply DHCP *options*. Similarly, the Vend field has been renamed the Options field, reflecting its new role as a way of conveying vendor-independent options between a client and server.

Options and Option Format

Keeping with the desire to maintain compatibility between BOOTP and DHCP, the DHCP Options field is, in most ways, the same as the vendor-independent interpretation of the BOOTP Vend field introduced by RFC 1048. The first four bytes of the field still carry the magic cookie value 99.130.83.99 to identify the information as vendor-independent option fields. The rest of the Option field consists of one or more subfields, each of which has a *type*, *length*, *value* (TLV-encoded) substructure, as in BOOTP.

The main differences between BOOTP vendor information fields and DHCP options are the field names and the fact that the DHCP Options field is variable in length (the BOOTP Vend field is fixed at 64 bytes). The structure of the DHCP Options field as a whole is shown in Figure 63-2, and the subfield names of each option are described in Table 63-3.

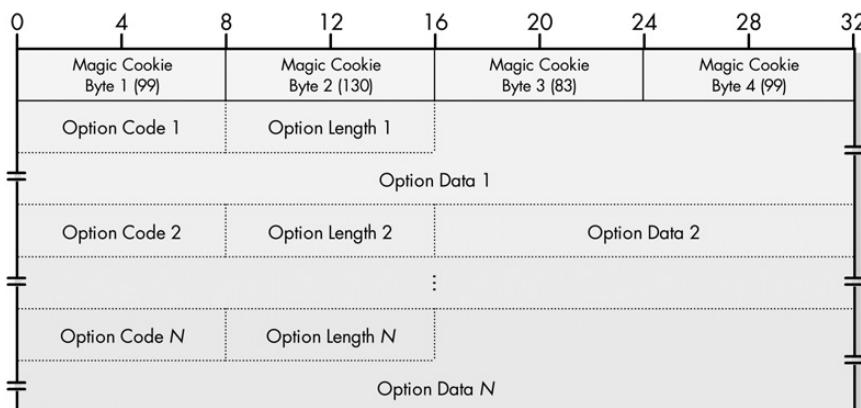


Figure 63-2: DHCP Options field format The format of the DHCP Options field is, unsurprisingly, very similar to that of the BOOTP Vendor-Specific Area, as shown in Figure 60-4 in Chapter 60. The Options field begins with the same four-byte magic cookie and then contains a number of variable-length option fields. Each option has the format described in Table 63-3.

All of the DHCP options follow the format shown in Table 63-3, except for two special cases, again the same as with BOOTP. A Code value of 0 is used as a *pad*, when subfields need to be aligned on word boundaries; it contains no information. The value 255 is used to mark the end of the vendor information fields. Both of these codes contain no actual data, so to save space, when either is used, just the single Code value is included and the Len and Data fields are omitted. A device

seeing a Code value of 0 just skips it as filler. A device seeing a Code value of 255 knows it has reached the end of the fields in this Options field.

Table 63-3: DHCP Option Format

Subfield Name	Size (Bytes)	Description
Code	1	Option Code: A single octet that specifies the option type.
Len	1	Option Length: The number of bytes in this particular option. This does not include the two bytes for the Code and Len subfields.
Data	Variable	Option Data: The data being sent, which has a length indicated by the Len subfield, and which is interpreted based on the Code subfield.

Option Categories

Before DHCP was invented, a series of BOOTP standards was published defining the current list of BOOTP vendor information extensions. When DHCP was developed, a single standard was created that merged both BOOTP vendor information extensions and DHCP options, since again, they are basically the same. The most recent of these is RFC 2132, entitled (ta-da!) “DHCP Options and BOOTP Vendor Extensions.”

RFC 2132 lists several dozen fields that can be used either as DHCP options or BOOTP vendor information fields, grouped into several categories. In addition, there is also a set of fields that are used only in DHCP, not in BOOTP. Despite being called *options*, only some really are optional; others are necessary for the basic operation of DHCP. They are carried as option fields for only one reason: to allow DHCP to keep using the same basic message format as BOOTP for compatibility. Table 63-4 summarizes the categories used for DHCP options.

Table 63-4: DHCP Option Categories

Option Category	Description
RFC 1497 Vendor Extensions	The BOOTP vendor extensions defined in RFC 1497, the last RFC describing vendor extension fields that was BOOTP-specific (before DHCP was created). For easier reference, these were kept in a single group when DHCP options were created, even though some of the functions they represent might fit better in other categories. (See Table 63-5.)
IP Layer Parameters per Host	Parameters that control the operation of the Internet Protocol (IP) on a host, which affect the host as a whole and are not interface-specific. (See Table 63-6.)
IP Layer Parameters per Interface	Parameters that affect the operation of IP for a particular interface of a host. (Some devices have only one interface; others have more.) (See Table 63-7.)
Link Layer Parameters per Interface	Parameters that affect the data link layer operation of a host, on a per-interface basis. (See Table 63-8.)
TCP Parameters	Parameters that impact the operation of the TCP layer; specified on a per-interface basis. (See Table 63-9.)
Application and Service Parameters	Parameters used to configure or control the operation of various miscellaneous applications or services. (See Table 63-10.)
DHCP Extensions	Parameters that are DHCP-specific and used to control the operation of the DHCP protocol itself. (See Table 63-12.)

The tables at the end of this chapter provide a complete list of the DHCP options defined in RFC 2132.

Due to the popularity of DHCP, several other options have been defined since that standard was published. Each time a new option is created, documenting it would have required a new successor to RFC 2132, which would be confusing and time-consuming. Instead, the maintenance of these options and extensions has been moved from the RFC process to a set of files maintained by the Internet Assigned Numbers Authority (IANA), just like so many other parameters. There is also a process by which a developer can request additional standard extensions to be added to DHCP. This is described in section 10 of RFC 2132.

KEY CONCEPT DHCP takes BOOTP's vendor information extensions and formalizes them into an official feature called *DHCP options*. The BOOTP Vendor Specific Area field becomes the DHCP Options field, and it can contain an arbitrary number of parameters to be sent from the server to the client. Some of these include pieces of data that are actually mandatory for the successful operation of DHCP. There are several dozen DHCP options, which are divided into functional categories.

Option Overloading

Since DHCP relies so much more on the use of options than BOOTP did, the size of the Options field could theoretically grow quite large. However, since DHCP is using UDP for transport, the overall size of a message is limited. This theoretically could have led to a situation where a message might run out of room and be unable to carry all its options. Meanwhile, there are two more spacious fields in the message format: SName and File, which are 64 bytes and 128 bytes, respectively. These fields might not even be needed in some cases, because many devices use DHCP for getting a lease and parameters, not to download a boot image. Even if they are needed, they might be carrying much less information than their large fixed size allows.

To make better use of the total space in the message format, DHCP includes a special feature called *option overloading*, which allows the SName and File fields to be used to carry more option fields instead of their conventional information. Use of this option is itself indicated through the use of a DHCP option, Option Overload, which tells a device receiving a message how to interpret the two fields. If option overloading is used, the SName and/or File fields are read and interpreted in the same way as the Options field, after all of the options in the Options field are parsed. If the message actually does need to carry a server name or boot file, these are included as separate options (number 66 and number 67, respectively), which are variable in length and can therefore be made exactly the length needed.

Incidentally, the creators of DHCP did recognize that even though vendor-independent options are important, a vendor might want to be able to send vendor-specific information just as the original BOOTP defined. To this end, they created a DHCP option called Vendor Specific Information. This option allows a vendor to encapsulate a set of vendor-specific option fields within the normal DHCP option structure. In essence, you can think of this as a way of nesting a conventional BOOTP Vend field (of variable length) within a single DHCP option. Other DHCP options can be carried simultaneously, subject to overall message-length limits. Note that this supplements an already existing BOOTP option that allows reference to be made to a file containing vendor-specific information.

KEY CONCEPT Since DHCP messages can contain so many options, a special feature called *option overloading* was created. When enabled, overloading allows options to make use of the large SName and File fields in the DHCP message format for options.

Summary of DHCP Options/BOOTP Vendor Information Fields

BOOTP *vendor information fields* are used to carry additional vendor-independent configuration parameters. These were used as the basis for DHCP *options*, which extend the concept to include parameters used to manage the operation of DHCP as a whole, as described in the previous section. Since BOOTP vendor information fields and DHCP options are essentially the same (except for the DHCP-specific fields), they are described in the same TCP/IP standard, and hence, in this single part of the book.

The following tables list each of the DHCP options/BOOTP vendor information fields. The tables show each option's Code value, the length of the Data subfield for the option, the formal name of the option, and a brief description of how it is used. For simplicity in the tables, where I say *option*, please read it as *option/vendor information field*, since they are the same (except, for the DHCP-specific options).

NOTE There are a lot of options in these tables, and some of them define parameters that are used by somewhat obscure protocols that I do not cover in this book. The brief descriptions may not be enough for you to completely understand how each and every option is used. Note in particular that many of the original BOOTP vendor information fields that are used to communicate the addresses of certain types of servers are now archaic and may no longer be used.

RFC 1497 Vendor Extensions

Table 63-5 shows the DHCP/BOOTP options that were originally defined in RFC 1497.

Table 63-5: DHCP/BOOTP Options: RFC 1497 Vendor Extensions

Code Value	Data Length (Bytes)	Name and Description
0	0	Pad: A single byte used as filler to align a subsequent field on a word (2-byte) boundary. It contains no information. One of two options that is a single byte in length, having no Data subfield (the other is the End option).
1	4	Subnet Mask: A 32-bit subnet mask being supplied for the client to use on the current network. It must appear in the option list before the Router option if both are present.
2	4	Time Offset: Specifies the time offset of the client's subnet in seconds from <i>Coordinated Universal Time (UTC, formerly Greenwich Mean Time or GMT)</i> . Positive values represent areas east of the prime meridian (in the United Kingdom); negative values represent areas west of the prime meridian. Essentially, this is used to indicate the time zone of the subnet.
3	Variable (multiple of 4)	Router: Specifies a list of 32-bit router addresses for the client to use on the local network. Routers are listed in the order of preference for the client to use.

(continued)

Table 63-5: DHCP/BOOTP Options: RFC 1497 Vendor Extensions (continued)

Code Value	Data Length (Bytes)	Name and Description
4	Variable (multiple of 4)	Time Server: Specifies a list of time server addresses (per RFC 868, see Chapter 88) for the client to use on the local network. Servers are listed in the order of preference for the client to use.
5	Variable (multiple of 4)	IEN-116 Name Server: Specifies a list of IEN-116 name server addresses for the client to use on the local network. Servers are listed in the order of preference for the client to use. Note that this option is not used for DNS name servers.
6	Variable (multiple of 4)	DNS Name Server: Specifies a list of DNS (see Chapter 52) name server addresses for the client to use on the local network. Servers are listed in the order of preference for the client to use.
7	Variable (multiple of 4)	Log Server: Specifies a list of MIT-LCS UDP log server addresses for the client to use on the local network. Servers are listed in the order of preference for the client to use.
8	Variable (multiple of 4)	Cookie Server: Specifies a list of RFC 865 cookie server addresses for the client to use on the local network. Servers are listed in the order of preference for the client to use.
9	Variable (multiple of 4)	LPR Server: Specifies a list of RFC 1179 line printer server addresses for the client to use on the local network. Servers are listed in the order of preference for the client to use.
10	Variable (multiple of 4)	Impress Server: Specifies a list of Imagen Impress server addresses for the client to use on the local network. Servers are listed in the order of preference for the client to use.
11	Variable (multiple of 4)	Resource Location Server: Specifies a list of RFC 887 resource location server addresses for the client to use on the local network. Servers are listed in the order of preference for the client to use.
12	Variable	Host Name: Specifies a host name for the client. This may or may not be a DNS host name; see option 15.
13	2	Boot File Size: Specifies the size of the default boot image file for the client, expressed in units of 512 bytes.
14	Variable	Merit Dump File: Specifies the path and filename of the file to which the client should dump its core image in the event that it crashes.
15	Variable	Domain Name: Specifies the DNS domain name for the client. Compare this with option 12.
16	4	Swap Server: Specifies the address of the client's swap server.
17	Variable	Root Path: Specifies the path name of the client's root disk. This allows the client to access files it may need, using a protocol such as the Network File System (NFS; see Chapter 58).
18	Variable	Extensions Path: Specifies the name of a file that contains vendor-specific fields that the client can interpret in the same way as the Options or Vend field in a DHCP/BOOTP message. This was defined to allow a client and server to still exchange vendor-specific information even though the Option/Vend field is now used for the general-purpose fields described in this chapter. Also see option 43.
255	0	End: Placed after all other options to mark the end of the option list. One of two options that is a single byte in length, having no Data subfield (the other is the Pad option).

IP Layer Parameters per Host

Table 63-6 shows the parameters that control the operation of IP on a host as a whole. They are not interface-specific.

Table 63-6: DHCP/BOOTP Options: IP Layer Parameters per Host

Code Value	Data Length (Bytes)	Name and Description
19	1	IP Forwarding Enable/Disable: A value of 1 turns on IP forwarding (that is, routing) on a client that is capable of that function; a value of 0 turns it off.
20	1	Non-Local Source Routing Enable/Disable Option: A value of 1 tells a client capable of routing to allow forwarding of IP datagrams with nonlocal source routes. A value of 0 tells the client not to allow this. See the source routing IP datagram option (see Chapter 21) for a bit more information on this and option 21.
21	Variable (multiple of 8)	Policy Filter: A set of address/mask pairs used to filter nonlocal source-routed datagrams.
22	2	Maximum Datagram Reassembly Size: Tells the client the size of the largest datagram that the client should be prepared to reassemble. The minimum value is 576 bytes.
23	1	Default IP Time to Live: Specifies the default value that the client should use for the Time to Live field in creating IP datagrams.
24	4	Path MTU Aging Timeout: Specifies the number of seconds the client should use in aging path maximum transmission unit (MTU) values determined using path MTU discovery.
25	Variable (multiple of 2)	Path MTU Plateau Table: Specifies a table of values to be used in performing path MTU discovery.

IP Layer Parameters per Interface

Table 63-7 shows the parameters that are specific to a particular host interface at the IP level.

Table 63-7: DHCP/BOOTP Options: IP Layer Parameters per Interface

Code Value	Data Length (Bytes)	Name and Description
26	2	Interface MTU: Specifies the MTU to be used for IP datagrams on this interface. The minimum value is 68.
27	1	All Subnets Are Local: When set to 1, tells the client that it may assume that all subnets of the IP network it is on have the same MTU as its own subnet. When 0, the client must assume that some subnets may have smaller MTUs than the client's subnet.
28	4	Broadcast Address: Tells the client what address it should use for broadcasts on this interface.
29	1	Perform Mask Discovery: A value of 1 tells the client that it should use Internet Control Message Protocol (ICMP; see Chapter 31) to discover a subnet mask on the local subnet. A value of 0 tells the client not to perform this discovery.
30	1	Mask Supplier: Set to 1 to tell the client that it should respond to ICMP subnet mask requests on this interface.
31	1	Perform Router Discovery: A value of 1 tells the client to use the ICMP router discovery process to solicit a local router. A value of 0 tells the client to not do so. Note that DHCP itself can be used to specify one or more local routers using option 3.

(continued)

Table 63-7: DHCP/BOOTP Options: IP Layer Parameters per Interface (continued)

Code Value	Data Length (Bytes)	Name and Description
32	4	Router Solicitation Address: Tells the client the address to use as the destination for router solicitations.
33	Variable (multiple of 8)	Static Route: Provides the client with a list of static routes it can put into its routing cache. The list consists of a set of IP address pairs; each pair defines a destination and a router to be used to reach the destination.

Link Layer Parameters per Interface

Table 63-8 lists the DHCP/BOOTP options that are specific to a particular link layer (layer 2) interface.

Table 63-8: DHCP/BOOTP Options: Link Layer Parameters per Interface

Code Value	Data Length (Bytes)	Name and Description
34	1	Trailer Encapsulation: When set to 1, tells the client to negotiate the use of trailers, as defined in RFC 893. A value of 0 tells the client not to use this feature.
35	4	ARP Cache Timeout: Specifies how long, in seconds, the client should hold entries in its ARP cache (see Chapter 13).
36	1	Ethernet Encapsulation: Tells the client what type of encapsulation to use when transmitting over Ethernet at layer 2. If the option value is 0, it specifies that Ethernet II encapsulation should be used, per RFC 894; when the value is 1, it tells the client to use IEEE 802.3 encapsulation, per RFC 1042.

TCP Parameters

The options impacting the operation of TCP are shown in Table 63-9.

Table 63-9: DHCP/BOOTP Options: TCP Parameters

Code Value	Data Length (Bytes)	Name and Description
37	1	Default TTL: Specifies the default TTL the client should use when sending TCP segments.
38	4	TCP Keepalive Interval: Specifies how long (in seconds) the client should wait on an idle TCP connection before sending a keepalive message. A value of 0 instructs the client not to send such messages unless specifically instructed to do so by an application.
39	1	TCP Keepalive Garbage: When set to 1, tells a client it should send TCP keepalive messages that include an octet of “garbage” for compatibility with implementations that require this.

Application and Service Parameters

Table 63-10 shows the miscellaneous options that control the operation of various applications and services.

Table 63-10: DHCP/BOOTP Options: Application and Service Parameters

Code Value	Data Length (Bytes)	Name and Description
40	Variable	Network Information Service Domain: Specifies the client's Network Information Service (NIS) domain. Contrast this with option 64.
41	Variable (multiple of 4)	Network Information Servers: Specifies a list of IP addresses of NIS servers the client may use. Servers are listed in the order of preference for the client to use. Contrast this with option 65.
42	Variable (multiple of 4)	Network Time Protocol Servers: Specifies a list of IP addresses of Network Time Protocol (NTP) servers the client may use. Servers are listed in the order of preference for the client to use.
43	Variable	Vendor Specific Information: Allows an arbitrary set of vendor-specific information items to be included as a single option within a DHCP or BOOTP message. This information is structured using the same format as the Options or Vend field itself, except that it does not start with a magic cookie. See the "DHCP Options" section earlier in this chapter for more details.
44	Variable (multiple of 4)	NetBIOS over TCP/IP Name Servers: Specifies a list of IP addresses of NetBIOS name servers (per RFC 1001/1002) that the client may use. Servers are listed in the order of preference for the client to use.
45	Variable (multiple of 4)	NetBIOS over TCP/IP Datagram Distribution Servers: Specifies a list of IP addresses of NetBIOS datagram distribution servers (per RFC 1001/1002) that the client may use. Servers are listed in the order of preference for the client to use.
46	1	NetBIOS over TCP/IP Node Type: Tells the client what sort of NetBIOS node type it should use. Four different bit values are used to define the possible node type combinations, as listed in Table 63-11.
47	Variable	NetBIOS over TCP/IP Scope: Specifies the NetBIOS over TCP/IP scope parameter for the client.
48	Variable (multiple of 4)	X Window System Font Servers: Specifies a list of IP addresses of X Window System Font servers that the client may use. Servers are listed in the order of preference for the client to use.
49	Variable (multiple of 4)	X Window System Display Manager: Specifies a list of IP addresses of systems running the X Window System Display Manager that the client may use. Addresses are listed in the order of preference for the client to use.
64	Variable	Network Information Service+ Domain: Specifies the client's NIS+ domain. Contrast this with option 40.
65	Variable (multiple of 4)	Network Information Service+ Servers: Specifies a list of IP addresses of NIS+ servers the client may use. Servers are listed in the order of preference for the client to use. Contrast this with option 41.
68	Variable (multiple of 4)	Mobile IP Home Agent: Specifies a list of IP addresses of home agents that the client can use in Mobile IP (see Chapter 30). Agents are listed in the order of preference for the client to use; normally a single agent is specified.
69	Variable (multiple of 4)	Simple Mail Transport Protocol (SMTP) Servers: Specifies a list of IP addresses of SMTP servers the client may use. Servers are listed in the order of preference for the client to use. See Chapter 77 for more on SMTP.
70	Variable (multiple of 4)	Post Office Protocol (POP3) Servers: Specifies a list of IP addresses of POP3 servers the client may use. Servers are listed in the order of preference for the client to use. See Chapter 78.
71	Variable (multiple of 4)	Network News Transfer Protocol (NNTP) Servers: Specifies a list of IP addresses of NNTP servers the client may use. Servers are listed in the order of preference for the client to use. See Chapter 85.
72	Variable (multiple of 4)	Default World Wide Web (WWW) Servers: Specifies a list of IP addresses of World Wide Web (HTTP) servers the client may use. Servers are listed in the order of preference for the client to use. See Chapter 79.

(continued)

Table 63-10: DHCP/BOOTP Options: Application and Service Parameters (continued)

Code Value	Data Length (Bytes)	Name and Description
73	Variable (multiple of 4)	Default Finger Servers: Specifies a list of IP addresses of Finger servers the client may use. Servers are listed in the order of preference for the client to use.
74	Variable (multiple of 4)	Default Internet Relay Chat (IRC) Servers: Specifies a list of IP addresses of Internet Relay Chat (IRC) servers the client may use. Servers are listed in the order of preference for the client to use.
75	Variable (multiple of 4)	StreetTalk Servers: Specifies a list of IP addresses of StreetTalk servers the client may use. Servers are listed in the order of preference for the client to use.
76	Variable (multiple of 4)	StreetTalk Directory Assistance (STDA) Servers: Specifies a list of IP addresses of STDA servers the client may use. Servers are listed in the order of preference for the client to use.

Table 63-11: NetBIOS Over TCP/IP Node Type (Option 46) Values

Option 46 Subfield Name	Size (Bits)	Description
Reserved	4	Reserved: Not used.
H-Node	1	H-Node: Set to 1 to tell the client to act as a NetBIOS H-node.
M-Node	1	M-Node: Set to 1 to tell the client to act as a NetBIOS M-node.
P-Node	1	P-Node: Set to 1 to tell the client to act as a NetBIOS P-node.
B-Node	1	B-Node: Set to 1 to tell the client to act as a NetBIOS B-node.

DHCP Extensions

Last, but certainly not least, Table 63-12 describes the DHCP-only options that control the operation of the DHCP protocol.

Table 63-12: DHCP Options: DHCP Extensions

Code Value	Data Length (Bytes)	Name and Description
50	4	Requested IP Address: Used in a client's DHCPDISCOVER message to request a particular IP address assignment.
51	4	IP Address Lease Time: Used in a client request to ask a server for a particular DHCP lease duration, or in a server reply to tell the client the offered lease time. It is specified in units of seconds.
52	1	Option Overload: Used to tell the recipient of a DHCP message that the message's SName and/or File fields are being used to carry options, instead having their normal meanings. This option implements the option overload feature. There are three possible values for this single-byte option: 1 means the File field is carrying the option data, 2 means the SName field has the option data, and 3 means both fields have the option data.
53	1	DHCP Message Type: Indicates the specific type of DHCP message, as listed in Table 6-13.

(continued)

Table 63-12: DHCP Options: DHCP Extensions (continued)

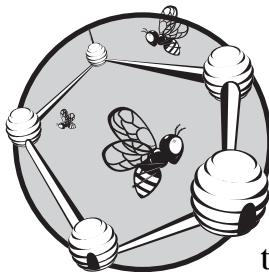
Code Value	Data Length (Bytes)	Name and Description
54	4	Server Identifier: The IP address of a particular DHCP server. This option is included in messages sent by DHCP servers to identify themselves as the source of the message. It is also used by a client in a DHCPREQUEST message to specify which server's lease it is accepting.
55	Variable	Parameter Request List: Used by a DHCP client to request a list of particular configuration parameter values from a DHCP server.
56	Variable	Message: Used by a server or client to indicate an error or other message.
57	2	Maximum DHCP Message Size: Used by a DHCP client or server to specify the maximum size of DHCP message it is willing to accept. The minimum legal value is 576 bytes.
58	4	Renewal (T1) Time Value: Tells the client the value to use for its renewal timer.
59	4	Rebinding (T2) Time Value: Tells the client the value to use for its rebinding timer.
60	Variable	Vendor Class Identifier: Included in a message sent by a DHCP client to specify its vendor and configuration. This may be used to prompt a server to send the correct vendor-specific information using option 43.
61	Variable	Client Identifier: Used optionally by a client to specify a unique client identification for itself that differs from the DHCP default. This identifier is expected by servers to be unique among all DHCP clients and is used to index the DHCP server's configuration parameter database.
66	Variable	TFTP Server Name: When the DHCP message's SName field has been used for options using the option overload feature, this option may be included to specify the Trivial File Transfer Protocol (TFTP) server name that would normally appear in the SName field.
67	Variable	Bootfile Name: When the DHCP message's File field has been used for options using the option overload feature, this option may be included to specify the boot filename that would normally appear in the File field.

Table 63-13: DHCP Message Type (Option 53) Values

Option 53 Value	DHCP Message Type
1	DHCPDISCOVER
2	DHCPOFFER
3	DHCPREQUEST
4	DHCPDECLINE
5	DHCPACK
6	DHCPONAK
7	DHCPRELEASE
8	DHCPIINFORM

64

DHCP CLIENT/SERVER IMPLEMENTATION, FEATURES, AND IPV6 SUPPORT



The preceding chapters in this part describe the fundamentals of the operation of DHCP: the address leasing system, configuration processes, and messaging. With this foundation in place, we can now proceed to look into some of the more interesting details of how DHCP is implemented. We can also delve into some of the extra capabilities and special features that change the basic DHCP mechanisms we have already studied.

In this chapter, I discuss DHCP client/server implementation issues, special features that enhance the protocol, and some of the problems and issues related to making DHCP work. I begin with a discussion of DHCP server and client implementation and management issues. I discuss DHCP message relaying and how it is related to the relaying feature used for the Boot Protocol (BOOTP). I describe the DHCP feature for providing automatic default addressing when a client cannot contact a server, and the conflict detection feature for multiple servers. I then cover some of the issues related

to interoperability of DHCP and BOOTP, and provide an outline of some of the more important problems and issues related to DHCP security. I conclude with an overview of DHCP for IP version 6 (DHCPv6).

DHCP Server and Client Implementation and Management Issues

DHCP is a client/server protocol, relying on both the server and client to fulfill certain responsibilities. Of the two device roles, the DHCP server is arguably the more important, because it is in the server that most of the functionality of DHCP is actually implemented.

DHCP Server Implementations

The server maintains the configuration database, keeps track of address ranges, and manages leases. For this reason, DHCP servers are typically much more complex than DHCP clients. In essence, without a DHCP server, there really is no DHCP. Thus, deciding how to implement DHCP servers is a large part of implementing the protocol.

A classic DHCP server consists of DHCP server software running on a server hardware platform of one sort or another. A DHCP server usually will not be a dedicated computer, except on very large networks. It is more common for a hardware server to provide DHCP services along with performing other functions, such as acting as an application server, serving as a general database server, providing DNS services, and so forth. So, a DHCP server does not need to be a special computer; any device that can run a DHCP server implementation can act as a server.

In fact, the DHCP server may not even need to be a host computer at all. Today, many routers include DHCP functionality. Programming a router to act as a DHCP server allows clients that connect to the router to be automatically assigned IP addresses. This provides numerous potential advantages in an environment where a limited number of public IP addresses is shared among multiple clients, or where IP Network Address Translation (NAT; see Chapter 28) is used to dynamically share a small number of addresses. Since DHCP requires a database, a router that acts as a DHCP server requires some form of permanent storage. This is often implemented using flash memory on routers; “true” servers use hard disk storage.

Virtually all modern operating systems include support for DHCP, including most variants of UNIX, Linux, newer versions of Microsoft Windows, Novell NetWare, and others. In some cases, you may need to run the server version of the operating system to have a host act as a DHCP server. For example, while Microsoft Windows XP supports DHCP, I don’t believe that a DHCP server comes in the Windows XP Home Edition, though you could install one yourself.

DHCP Server Software Features

In most networks, you will choose the operating system based on a large number of factors. The choice of operating system will then dictate what options you have for selecting DHCP server software. Most common operating systems have a number of

options available for software. While all will implement the core DHCP protocol, they will differ in terms of the usual software attributes: cost, performance, ease of use, and so on. They may also differ in terms of their features, such as the following:

- How they allow address ranges (scopes) to be defined
- How clients can be grouped and managed
- The level of control an administrator has over parameters returned to a client
- The level of control an administrator has over general operation of the protocol, such as specification of the T1 and T2 timers and other variables, and how leases are allocated and renewals handled
- Security features
- Ability to interact with DNS to support dynamic device naming
- Optional features such as BOOTP support, conflict detection, and Automatic Private IP Addressing (all discussed later in this chapter)

Choosing the Number of Servers

In setting up DHCP for a network, there are a number of important factors to consider and decisions to be made. One of the most critical is the number of servers you want to have. In theory, each network requires only one DHCP server; in practice, this is often not a great idea. Servers sometimes experience hardware or software failures, or they must be taken down for maintenance. If there is only one server and clients can't reach it, no DHCP clients will be able to get addresses. For this reason, two or more servers are often used.

If you do use more than one server, you need to carefully plan how you will configure each one. One of the first decisions you will need to make is which servers will be responsible for which addresses and clients. You need to determine whether you want the servers to have distinct or overlapping address pools, as discussed in the explanation of DHCP address ranges in Chapter 61. Distinct pools ensure that addresses remain unique, but result in unallocatable addresses if a server fails. Overlapping addresses are more flexible, but risk address conflicts unless a feature like conflict detection (described later in this chapter) is used.

Server Placement, Setup, and Maintenance

Once you know how many servers you want, you need to determine on which part of the network you want to place them. If you have many physical networks, you may also need to use DHCP relaying to allow all clients to reach a server. Since the structure of the network may affect the number of servers you use, many of these decisions are interrelated.

You must make policy decisions related to all the DHCP operating parameters discussed in the previous chapters. The two big decisions are the size and structure of the address pool, and making lease policy decisions such as the lease length and the settings for the T1 and T2 timers. You also must decide which clients will be dynamically allocated addresses and how manually configured clients will be handled.

Finally, it's essential for the administrator to remember that an organization's DHCP server is a database server and must be treated accordingly. Like any database server, it must be maintained and managed carefully. Administrative policies must be put into place to ensure the security and efficient operation of the server. Also, unlike certain other types of database systems, the DHCP database is not automatically replicated; the server database should therefore be routinely backed up, and using RAID storage is also a good idea.

DHCP Client Implementations

Just as a DHCP server consists of server software running on a server platform or hardware acting as a server, a DHCP client is simply DHCP client software running on a client device. Most often, a client device is a host computer connected to a TCP/IP internetwork. DHCP is so widely accepted today that virtually all hosts include DHCP client software. The DHCP client is usually integrated into graphical operating systems like Windows, or is implemented using a specific client daemon like *dhclient* or *dhcpcd* on UNIX/Linux.

Since the entire idea behind DHCP is to put the server in charge of parameter storage, configuration, and address management, DHCP clients are relatively simple. The client implements the messaging protocol and communicates parameters received from the DHCP server to the rest of the TCP/IP software components as needed. It doesn't do a whole lot else.

In fact, there's not really much for an administrator to do to set up a client to use DHCP. In some operating systems, it's as simple as "throwing a switch," by enabling DHCP support within the client itself. This prompts the client to then stop using any manually configured parameters and start searching for a DHCP server instead. The server then becomes responsible for the client's configuration and address assignment.

Since the client doesn't do a great deal in DHCP other than communicate with the server, not much is required in the way of user software for a DHCP client. In most cases, control over the DHCP client software is accomplished using a TCP/IP configuration utility, as described in Chapter 88. Windows clients use the programs *ipconfig* or *winipcfg* to display the status of their current DHCP leases. These programs also allow the client to manually release the current lease or renew it.

Releasing the lease means early lease termination using the DHCPRELEASE message. This is usually the only way that a lease is terminated. Renewing the lease is a manual version of the automated renewal process. Releasing and renewing the lease may be done in sequence to reset a client that is in a confused state or is having some other type of DHCP or connectivity problem.

DHCP Message Relaying and BOOTP Relay Agents

DHCP is the third-generation host configuration protocol for TCP/IP. We've already discussed extensively how it was based directly on BOOTP, which was, in turn, an enhancement of the earlier Reverse Address Resolution Protocol (RARP). Even though each new protocol has made significant improvements over its predecessor, each iteration has retained certain limitations that are actually common to all host configuration protocols.

One of the most important limitations with host configuration protocols is the reliance on broadcasts for communication. Whenever we are dealing with a situation where a client needs to communicate but doesn't know its IP address and doesn't know the address of a server that will provide it, the client needs to use broadcast addressing. However, for performance reasons, broadcasts are normally propagated only on the local network. This means that the client and server would always need to be on the same physical network for host configuration to occur. Of course, we don't want this to be the case. It would require that a large internetwork have a different server on every network, greatly reducing the benefits of centralized configuration information and creating numerous administrative hassles.

RARP didn't have any solution to this problem, which is one reason why it was so limited in usefulness. BOOTP's solution is to allow a client and server to be on different networks through the use of *BOOTP relay agents*.

BOOTP Relay Agents for DHCP

A *relay agent* is a device that is not a BOOTP server, but which runs a special software module that allows it to act in the place of a server. A relay agent can be placed on networks where there are BOOTP clients but no BOOTP servers. The relay agent intercepts requests from clients and relays them to the server. The server then responds back to the agent, which forwards the response to the client. A full rationale and description of operation of BOOTP relay agents can be found in Chapter 60.

The designers of DHCP were satisfied with the basic concepts and operation behind BOOTP relay agents, which had already been in use for many years. For this reason, they made the specific decision to continue using BOOTP relay agent functionality in DHCP. In fact, this is one of the reasons why the decision was made to retain the BOOTP message format in DHCP, and also the basic two-message request/reply communication protocol. This allows BOOTP relay agents to handle DHCP messages as if they were BOOTP messages. This is also why the mention of BOOTP in the title of this topic is not a typo—DHCP uses BOOTP relay agents. Even the DHCP standard says that a “BOOTP relay agent is an Internet host or router that passes DHCP messages between DHCP clients and DHCP servers.”

In practice, the agents are indeed sometimes called *DHCP relay agents*. You may also see the terms *BOOTP/DHCP relay agent* and *DHCP/BOOTP relay agent*.

DHCP Relaying Process

Since DHCP was designed specifically to support BOOTP relay agents, the agents behave in DHCP much as they do in BOOTP. Although DHCP has much more complex message exchanges, they are all still designed around the notion of a client request and server response. There are just more requests and responses.

The BOOTP agent looks for broadcasts sent by the client and then forwards them to the server (as described in the BOOTP relay agent behavior discussion in Chapter 60), and then returns replies from the server. The additional information in the DHCP protocol is implemented using additions to the BOOTP message format in the form of DHCP options, which the relay agent doesn't look at. It just treats them as it does BOOTP requests and replies.

KEY CONCEPT To permit DHCP clients and DHCP servers to reside on different physical networks, an intermediary device is required to facilitate message exchange between networks. DHCP uses the same mechanism for this as BOOTP: the deployment of *BOOTP relay agents*. The relay agent captures client requests, forwards them to the server, and then returns the server's responses back to the client.

In summary, when a relay agent is used, here's what the various client requests and server replies in the DHCP operation section become:

Client Request When a client broadcasts a request, the relay agent intercepts it on UDP port 67. It checks the Hops field and discards the request if the value is greater than 16; otherwise, it increments the field. The agent puts its own address into the GIAddr field unless another relay agent has already put its address in the field. It then forwards the client request to a DHCP server, either unicast or broadcast on another network.

Server Reply The server sees a nonzero value in the GIAddr field and sends the reply to the relay agent whose IP address is in that field. The relay agent then sends the reply back to the client, using either unicast or broadcast (as explained in the discussion of DHCP addressing in Chapter 61).

One difference between BOOTP and DHCP is that certain communications from the client to the server are unicast. The most noticeable instance of this is when a client tries to renew its lease with a specific DHCP server. Since it sends this request unicast, it can go to a DHCP server on a different network using conventional IP routing, and the relay agent does not need to be involved.

DHCP Autoconfiguration/Automatic Private IP Addressing (APIPA)

The IP address of a TCP/IP host is, in many ways, its identity. Every TCP/IP network requires that all hosts have unique addresses to facilitate communication. When a network is manually configured with a distinct IP address for each host, the hosts permanently know who they are. When hosts are made DHCP clients, they no longer have a permanent identity; they rely on a DHCP server to tell them who they are.

This dependency is not a problem as long as DHCP is functioning normally and a host can get a lease, and, in fact, has many benefits that we have explored. Unfortunately, a number of circumstances can arise that result in a client failing to get a lease. The client may not be able to obtain a lease, reacquire one after reboot, or renew an existing lease. There are several possible reasons why this might happen:

- The DHCP server may have experienced a failure or may be taken down for maintenance.
- The relay agent on the client's local network may have failed.

- Another hardware malfunction or power failure may make communication impossible.
- The network may have run out of allocatable addresses.

Without a lease, the host has no IP address, and without an address, the host is effectively dead in the water. The base DHCP specification doesn't really specify any recourse for the host in the event that it cannot successfully obtain a lease. It is left up to the implementor to decide what to do, and when DHCP was first created, many host implementations would simply display an error message and leave the host unusable until an administrator or user took action.

Clearly, this is far from an ideal situation. It would be better if we could just have a DHCP client that is unable to reach a server automatically configure itself. In fact, the Internet Engineering Task Force (IETF) reserved a special IP address block for this purpose (see Chapter 17). This block, 169.254.0.1 through 169.254.255.254 (or 169.254.0.0/16 in classless notation) is reserved for autoconfiguration, as mentioned in RFC 3330, "Hosts obtain these addresses by auto-configuration, such as when a DHCP server may not be found."

Strangely, however, no TCP/IP standard was defined to specify how such autoconfiguration works. To fill the void, Microsoft created an implementation that it calls *Automatic Private IP Addressing* (APIPA). Due to Microsoft's market power, APIPA has been deployed on millions of machines, and has thus become a de facto standard in the industry. Many years later, the IETF did define a formal standard for this functionality, in RFC 3927, "Dynamic Configuration of IPv4 Link-Local Addresses."

APIPA Operation

APIPA is really so simple that it's surprising it took so long for someone to come up with the idea. It takes over at the point where any DHCP lease process fails. Instead of just halting with an error message, APIPA randomly chooses an address within the aforementioned private addressing block. It then performs a test very similar to the one in step 13 in the DHCP allocation process (see Chapter 62): It uses ARP to generate a request on the local network to see if any other client responds using the address it has chosen. If there is a reply, APIPA tries another random address and repeats the test. When the APIPA software finds an address that is not in use, it is given to the client as a default address. The client will then use default values for other configuration parameters that it would normally receive from the DHCP server. This process is illustrated in Figure 64-1.

A client using an autoconfigured address will continue to try to contact a DHCP server periodically. By default, this check is performed every five minutes. If and when it finds one, it will obtain a lease and replace the autoconfigured address with the proper leased address.

APIPA is ideally suited to small networks, where all devices are on a single physical link. Conceivably, with 20 APIPA-enabled DHCP clients on a network with a single DHCP server, you could take the server down for maintenance and still have all the clients work properly, using 169.254.x.x addresses.

Bear in mind, however, that APIPA is not a proper replacement for full DHCP.

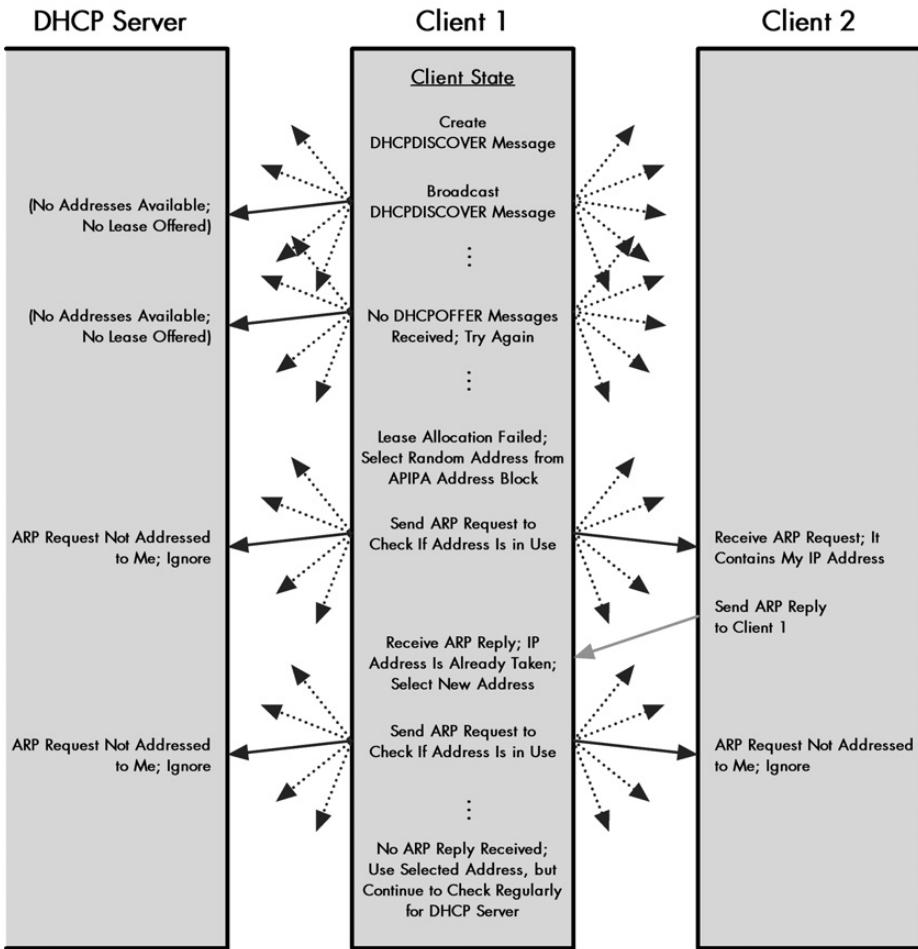


Figure 64-1: DHCP Automatic Private IP Addressing (APIPA) In this example, Client 1 is trying to get an IP address from its DHCP server, but the server is out of addresses, so it does not respond to the client's requests. The client is configured to use APIPA, so it randomly selects an address from the APIPA address block. It sends an ARP request on the local network to see if any other device is using that address. Usually, there will be no conflict, but here Client 2 is using the address, so it responds. Client 1 chooses a different address, and this time gets no reply. It begins using that address, while continuing to check regularly for a DHCP server to come online.

APIPA Limitations

The 169.254.0.0/16 block is a private IP range and comes with all the limitations of private IP addresses, including inability to use these addresses on the Internet. Also, APIPA cannot provide the other configuration parameters that a client may need to get from a DHCP server. Finally, APIPA will not work properly in conjunction with proxy ARP, because the proxy will respond for any of the private addresses, so they will all appear to be used.

Since it uses ARP to check for address conflicts, APIPA is not well suited for large internetworks. To use it on an internetwork with multiple subnets would require software that allows each subnet to use a different portion of the full 169.254.0.0/16 blocks, to avoid conflicts.

In practice, APIPA is a solution for small networks. Large internetworks deal with the problem of not being able to contact a DHCP server by taking steps to ensure that a client can always contact a DHCP server.

KEY CONCEPT An optional DHCP feature called *Automatic Private IP Addressing (APIPA)* was developed to allow clients to still be able to communicate in the event that they are unable to obtain an IP address from a DHCP server. When enabled, the client chooses a random address from a special reserved block of private IP addresses and checks to make sure the address is not already in use by another device. It continues to check for a DHCP server periodically until it is able to find one.

DHCP Server Conflict Detection

One of the primary decisions any TCP/IP administrator using DHCP must make is how many DHCP servers to deploy. A single server has the advantage of simplicity, but provides no redundancy in the event of failure. It also means that whenever the DHCP server is down, clients can't get addresses. For these reasons, most larger networks use two or more servers.

When you have two servers or more—let's say two for sake of this discussion—you then have another decision to make: How do you divide the address pool between the servers? As I explored in detail in the discussion of DHCP address pools in Chapter 61, there are two options: give the servers overlapping addresses or making them non-overlapping. Unfortunately, in classic DHCP, neither is really a great solution. Overlapping ranges mean both servers might try to assign the same address, since DHCP includes no provision for communication between servers. Non-overlapping ranges avoid this problem, but make only some of the addresses available to each server.

It's strange that the DHCP standard didn't provide better support for cross-server coordination, even though there clearly was a need for it. However, certain DHCP implementations include an optional feature to allow two servers to have overlapping scopes without address clashes occurring. This is a feature commonly found on Microsoft DHCP servers and may also be present in other implementations. It is called *DHCP server conflict detection*.

The idea behind conflict detection is very simple. Suppose a DHCP server receives a DHCPDISCOVER message from a client and decides to offer it a lease. Before sending the DHCPOFFER message, the server conducts a *probe* by sending ICMP Echo (ping) messages (see Chapter 33) out to the address it plans to offer. It then waits a short period of time to hear if it receives any ICMP Echo Reply messages back. If it does, it knows the IP address is in use and chooses a different one.

If all DHCP servers are configured to do this before offering an address, then it is possible to give all of them the same, overlapping addresses for assignment. They won't have any way of coordinating with each other, but as long as they ask first by doing an ICMP check, there won't be any problems. This provides an administrator

with the advantages of overlapping address ranges—simplicity and access to all addresses by all servers—without risk of address conflicts. The only small drawback is a little extra network traffic to perform the check, and possibly a few milliseconds of server CPU time if a new address needs to be chosen.

If you were paying attention when you read about the DHCP allocation process in Chapter 62, you may have noticed that what I am describing here sounds familiar. In fact, it's true that this feature isn't anything new. The use of ICMP to check an address before offering it is actually mentioned in RFC 2131 as part of the standard DHCP allocation process, and you can find it mentioned as step 5 in the allocation process description.

So why was conflict detection required to be an extra feature? The reason is that the use of ICMP wasn't mandatory because the standard says servers *should* do it, not that they *must* do it. This choice was made to provide flexibility in implementing DHCP, but that flexibility comes at a cost. So, if you want to use this feature, you need to look for support for it in your server software.

KEY CONCEPT Some DHCP implementations include a feature called *server conflict detection*. When this feature is activated, it causes each server to always check to make sure an address is not in use before granting it to a client. When conflict detection is used by all DHCP servers on a network, the servers can be given overlapping scopes, so each can assign any of the organization's IP addresses, while at the same time not needing to be concerned about two clients being assigned the same address by different servers.

DHCP and BOOTP Interoperability

I've talked extensively about how DHCP was designed based on BOOTP and how they use the same basic communication method and message format. This was done for several reasons, one of the most important of which was ensuring interoperability of the two protocols. Given this, you might expect that we could simply say that BOOTP and DHCP are compatible with each other, and that's that.

It is true that DHCP was intended to be compatible with BOOTP. RFC 2131 lists the following as one of DHCP's design goals: "DHCP must provide service to existing BOOTP clients." This seems pretty clear. The reuse of the BOOTP message format is one of the keys to DHCP and BOOTP compatibility. DHCP functionality is implemented not through new fields, but rather through DHCP-specific options, such as the DHCP Message Type option that specifies the all-important type of DHCP messages. DHCP devices can look for this extra information, while BOOTP devices can ignore it.

However, while DHCP and BOOTP are similar, they are not the same, and so there are some interoperability concerns that crop up when they are used together. The DHCP message format is structurally the same as the BOOTP format, but the interpretation of certain fields is slightly different. BOOTP clients don't understand DHCP, so when BOOTP and DHCP are used together, the DHCP client or server must sometimes behave slightly differently to compensate. Further complicating matters are the facts that not all implementations of DHCP and BOOTP are necessarily exactly the same and that certain specifications in the DHCP standard are not mandatory.

For these reasons, we cannot just assume that DHCP and BOOTP will work together. To address some of these issues, the IETF published RFC 1534, “Inter-operation Between DHCP and BOOTP,” at the same time that DHCP was originally created. This document looks at how the protocols work together, focusing on the two distinct client/server interoperating combinations: a BOOTP client connecting to a DHCP server, and a DHCP client connecting to a BOOTP server. Let’s consider each case.

BOOTP Clients Connecting to a DHCP Server

As indicated by the preceding quote from RFC 2131, DHCP was specifically intended to allow a DHCP server to handle requests from BOOTP clients. The protocol itself is set up to enable this, but it does require that the DHCP server be given certain intelligence to know how to deal with BOOTP clients.

One of the most important issues is that BOOTP clients will follow the BOOTP configuration process and not the DHCP leasing processes. The DHCP server must use BOOTP messages with the BOOTP meanings for fields when dealing with BOOTP clients. A server determines that a client is using BOOTP instead of DHCP by looking for the presence of the DHCP Message Type option, which must be present in all DHCP messages but is not used for BOOTP.

If a DHCP server detects that it is dealing with a BOOTP client, it can respond with configuration information for the client. The server can use either manual or automatic allocation for the client. Automatic allocation means the server chooses an address from its pool of unused addresses, but assigns it permanently. BOOTP clients are not capable of dynamic allocation, since BOOTP is static in nature.

A DHCP server may include BOOTP vendor information fields in its response to a BOOTP client, including ones defined since BOOTP was created. However, it obviously must not send any DHCP-specific options.

DHCP Clients Connecting to a BOOTP Server

A DHCP client can obtain configuration information from a BOOTP server, because the server will respond to the client’s initial DHCPDISCOVER message as if it were a BOOTP BOOTREQUEST message. The DHCP client can tell that a BOOTP reply has been received because there will be no DHCP Message Type option.

A response from a BOOTP server should be treated as an infinite lease, since again, that’s all that BOOTP supports. Note that if a DHCP client receives a response from both a BOOTP server and a DHCP server, it should use the DHCP response and not the BOOTP response (even if this means it gets a shorter lease).

DHCP Security Issues

DHCP was designed in the early 1990s, when the number of organizations on the Internet was relatively small. Furthermore, it was based on BOOTP, which was created in the 1980s, when the Internet as we know it today barely even existed. In those days, Internet security wasn’t a big issue, because it was mostly a small group

of research and educational organizations using TCP/IP on the Internet. As a result, DHCP, like many protocols of that era, doesn't do much to address security concerns.

Actually, this is a bit understated. Not only does DHCP run over the Internet Protocol (IP) and the User Datagram Protocol (UDP), which are inherently insecure, but the DHCP protocol itself has no security provisions whatsoever. This is a fairly serious issue in modern networks, because of the sheer power of DHCP, which deals with critical configuration information.

DHCP Security Concerns

There are two different classes of potential security problems related to DHCP:

Unauthorized DHCP Servers If a malicious person plants a rogue DHCP server, it is possible that this device could respond to client requests and supply them with spurious configuration information. This could be used to make clients unusable on the network, or worse, set them up for further abuse later on. For example, a hacker could exploit a bogus DHCP server to direct a DHCP client to use a router under the hacker's control, rather than the one the client is supposed to use.

Unauthorized DHCP Clients A client could be set up that masquerades as a legitimate DHCP client and thereby obtain configuration information intended for that client. This information could then be used to compromise the network later on. Alternatively, a malicious person could use software to generate a lot of bogus DHCP client requests to use up all the IP addresses in a DHCP server's pool. More simply, this could be used by a thief to steal an IP address from an organization for his own use.

These are obviously serious concerns. The normal recommended solutions to these risks generally involve providing security at lower layers. For example, one of the most important techniques for preventing unauthorized servers and clients is careful control over physical access to the network: layer 1 security. Security techniques implemented at layer 2 may also be of use—for example, in the case of wireless LANs. Since DHCP runs over UDP and IP, one could use IPSec at layer 3 to provide authentication.

DHCP Authentication

To try to address some of the more specific security concerns within DHCP itself, in June 2001, the IETF published RFC 3118, "Authentication for DHCP Messages." This standard describes an enhancement that replaces the normal DHCP messages with authenticated ones. Clients and servers check the authentication information and reject messages that come from invalid sources. The technology involves the use of a new DHCP option type, the Authentication option, and operating changes to several of the leasing processes to use this option.

Unfortunately, 2001 was pretty late in the DHCP game, and there are millions of DHCP clients and servers around that don't support this new standard. Both the client and server must be programmed to use authentication for this method to

have value. A DHCP server that supports authentication could use it for clients that support the feature and skip it for those that do not. However, the fact that this option is not universal means that it is not widely deployed, and most networks must rely on more conventional security measures.

DHCP for IP Version 6 (DHCPv6)

DHCP is currently the standard host configuration protocol for the TCP/IP protocol suite. TCP/IP is built on version 4 of IP (IPv4). However, development work has been under way since the early 1990s on a successor to IPv4: version 6 of the Internet Protocol (IPv6; see Part II-4 for more information). This new IP standard will be the future of TCP/IP.

While most of the changes that IPv6 brings impact technologies at the lower layers of the TCP/IP architectural model, the significance of the modifications means that many other TCP/IP protocols are also affected. This is particularly true of protocols that work with addresses or configuration information, including DHCP. For this reason, a new version of DHCP is required for IPv6. Development has been under way for quite some time on *DHCP for IPv6*, also sometimes called *DHCPv6*. At the time of writing, DHCPv6 has not yet been formally published—it is still an Internet draft under discussion.

NOTE In discussions purely oriented around IPv6, DHCPv6 is sometimes just called DHCP, and the original DHCP is called DHCPv4.

Two Methods for Autoconfiguration in IPv6

One of the many enhancements introduced in IPv6 is an overall strategy for easier administration of IP devices, including host configuration. There are two basic methods defined for autoconfiguration of IPv6 hosts:

Stateless Autoconfiguration A method defined to allow a host to configure itself without help from any other device.

Stateful Autoconfiguration A technique where configuration information is provided to a host by a server.

Which of these methods is used depends on the characteristics of the network. Stateless autoconfiguration is described in RFC 2462 and discussed in Chapter 24. Stateful autoconfiguration for IPv6 is provided by DHCPv6. As with regular DHCP, DHCPv6 may be used to obtain an IP address and other configuration parameters, or just to get configuration parameters when the client already has an IP address.

DHCPv6 Operation Overview

The operation of DHCPv6 is similar to that of DHCPv4, but the protocol itself has been completely rewritten. It is not based on the older DHCP or on BOOTP, except in conceptual terms. It still uses UDP, but it uses new port numbers, a new

message format, and restructured options. All of this means that the new protocol is not strictly compatible with DHCPv4 or BOOTP, though I believe work is under way on a method to allow DHCPv6 servers to work with IPv4 devices.

KEY CONCEPT Since DHCP works with IP addresses and other configuration parameters, the change from IPv4 to IPv6 requires a new version of DHCP commonly called *DHCPv6*. This new DHCP represents a significant change from the original DHCP and is still under development. DHCPv6 is used for IPv6 *stateful autoconfiguration*. The alternative is *stateless autoconfiguration*, a feature of IPv6 that allows a client to determine its IP address without need for a server.

DHCPv6 is also oriented around IPv6 methods of addressing, especially the use of link-local scoped multicast addresses (see Chapter 25). This allows efficient communication even before a client has been assigned an IP address. Once a client has an address and knows the identity of a server, it may communicate with the server directly using unicast addressing.

DHCPv6 Message Exchanges

There are two basic client/server message exchanges that are used in DHCPv6: the *four-message exchange* and the *two-message exchange*. The former is used when a client needs to obtain an IPv6 address and other parameters. This process is similar to the regular DHCP address allocation process. Highly simplified, it involves these steps:

1. The client sends a multicast Solicit message to find a DHCPv6 server and ask for a lease.
2. Any server that can fulfill the client's request responds to it with an Advertise message.
3. The client chooses one of the servers and sends a Request message to it, asking to confirm the offered address and other parameters.
4. The server responds with a Reply message to finalize the process.

There is also a shorter variation of the four-message process above, where a client sends a Solicit message and indicates that a server should respond back immediately with a Reply message.

If the client already has an IP address, either assigned manually or obtained in some other way, a simpler process can be undertaken, similar to how in regular DHCP the DHCPINFORM message is used:

1. The client multicasts an Information-Request message.
2. A server with configuration information for the client sends back a Reply message.
3. As in regular DHCP, a DHCPv6 client renews its lease after a period of time by sending a Renew message. DHCPv6 also supports relay agent functionality, as in DHCPv4.

PART III-4

TCP/IP NETWORK MANAGEMENT FRAMEWORK AND PROTOCOLS

Modern networks and internetworks are larger, faster, and more capable than their predecessors of years gone by. As we expand, speed up, and enhance our networks, they become more complex, and as a result, more difficult to manage. Years ago, an administrator could get by with very simple tools to keep a network running, but today, more sophisticated network management technologies are required to match the sophistication of our networks.

Some of the most important tools in the network manager's toolbox are now in the form of software, not hardware. To manage a sprawling, heterogeneous, and complex internetwork, we can employ software applications to gather information and control devices using the internetwork itself. TCP/IP, being the most popular internetworking suite, has such software tools. One of the most important is a pair of protocols that have been implemented as part of an overall method of network management called the *TCP/IP Internet Standard Management Framework*.

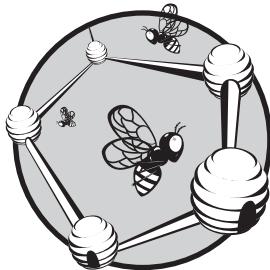
This part describes the TCP/IP Internet Standard Management Framework, looking at each of its architectural and protocol components and how they interoperate. The first chapter provides an overview of the network management framework itself and serves as an introduction to the chapters that follow. The second chapter discusses the way that network management information is structured and arranged into information stores

called *management information bases (MIBs)*. The third chapter describes the concepts behind and operation of the key protocol in TCP/IP network management: the Simple Network Management Protocol (SNMP). The fourth chapter provides details on SNMP's messaging and message formats. Finally, the fifth chapter takes a brief look at Remote Network Monitoring (RMON), an enhancement of SNMP—sometimes called a protocol, even though it really isn't—that provides administrators with greater management and monitoring abilities on a TCP/IP internetwork.

Note that while you may be tempted to jump straight to the chapter on SNMP, what is written there will make a lot more sense if you read the chapters of this part in order.

65

TCP/IP INTERNET STANDARD MANAGEMENT FRAMEWORK OVERVIEW



TCP/IP network management functions are most commonly associated with the key protocol responsible for implementing those functions: the *Simple Network Management Protocol (SNMP)*. Many people have heard of SNMP, and it is common for SNMP to be considered “the” way that network management is performed in TCP/IP. This is true to an extent, but is really an oversimplification. The actual SNMP protocol is only one part of a higher-level network management strategy called the *Internet Standard Management Framework*. In order to really understand how SNMP works, you need to first have some background on the way this network management is structured as a whole.

In this chapter, I provide an introduction to TCP/IP network management by describing the concepts and components of the TCP/IP Internet Standard Management Framework. I begin with an overview and history of the framework, and discuss how it is related to SNMP. I describe the TCP/IP network management model and the key components that compose a network management system. I provide a summary of the architecture of the

Internet Standard Management Framework. I then describe the three main versions of the Framework and SNMP and how they compare. I conclude with a discussion of the many standards used to describe this technology.

Overview and History of the TCP/IP Internet Standard Management Framework and Simple Network Management Protocol (SNMP)

An adage from the world of professional sports says that a baseball umpire is doing a good job when you forget that he is there. In many ways, the same could be said of a network administrator. The administrator is doing a good job when the network is running so smoothly and efficiently that users forget that the administrator exists. Because, as the administrator knows all too well, the second there is a problem, the users will all remember very quickly that he or she is there.

A primary job of a network administrator is to keep tabs on the network and ensure that it is operating normally. Information about the hardware and software on the network is a key to performing this task properly.

When networks were small, an administrator could stay informed about the status of hardware and software using simple means, such as physically walking over to a computer and using it, or using a low-level link layer management protocol. This is simply not possible with modern internetworks, which are large, geographically diverse, and often consist of many different lower-layer technologies. Usually, the only thing all the devices on the network have in common is an implementation of a particular internetworking protocol suite, such as TCP/IP. This makes the internetwork itself a logical way to facilitate the communication of network management information between devices and a network administrator.

Early Development of SNMP

Many people recognized during the early days of the Internet that some sort of network management technology would be needed for TCP/IP. Unfortunately, at first there was no single standard. In the 1980s, several different technologies were developed by different working groups. There were three main contestants: the *High-level Entity Management System (HEMS)/High-level Entity Management Protocol (HEMP)* as defined by RFCs 1021 through 1024; the *Simple Gateway Monitoring Protocol (SGMP)*, defined by RFC 1028; and the *Common Management Information Protocol (CMIP)*, which is actually part of the OSI protocol suite.

The Internet Engineering Task Force (IETF) recognized the importance of having a unifying management standard for TCP/IP, and in 1988, published RFC 1052, “IAB Recommendations for the Development of Internet Network Management Standards.” This memo is not a standard, but more a statement of intention and documentation of a meeting held on this subject. The conclusion of RFC 1052 was that SGMP be used as the basis of a new Internet standard to be called the *Simple Network Management Protocol (SNMP)*. This development was to be carried out by the SNMP Working Group.

The Two Meanings of SNMP

The rationale of the middle two words in the name Simple Network Management Protocol is obvious, but the other two words are slightly more problematic. The word *Protocol* implies that SNMP is just a TCP/IP communication protocol, like other protocols, such as the Dynamic Host Configuration Protocol (DHCP) and the File Transfer Protocol (FTP). Unfortunately, this is both true and untrue: the name is ambiguous.

At a lower level, SNMP does indeed refer specifically to the actual protocol that carries network management information between devices. This is what most people think of when they talk about SNMP. However, as defined by the SNMP working group, the TCP/IP network management solution as a whole consists of a number of different elements arranged in an architecture. This architecture originally had no specific name, but is now called the *Internet Standard Management Framework*. Oddly, this higher-level framework is not abbreviated ISMF, but is *also* called SNMP, which means that context is important in understanding that term.

NOTE To avoid confusion, I will often use the phrases *SNMP Framework* and *SNMP protocol* to differentiate these two uses of the term *SNMP*.

Design Goals of SNMP

The word *Simple* in the protocol's name is another problem. Even in its first iteration, it was only somewhat simple. The most current version of SNMP is fairly complicated indeed, with many different standards defining the SNMP Framework, the SNMP protocol itself, and a number of supporting elements.

So why is it called *Simple*? Well, as they say, everything is relative. SNMP is simple when compared to other protocols that are even more complex. Some of this can be seen by looking at the basic goals of the Internet Standard Management Framework and the SNMP protocol as a whole:

- SNMP defines a universal way that management information can be easily defined for any object, and then exchanged between that object and a device designed to facilitate network management.
- SNMP separates the functions of defining and communicating management information from the applications that are used for network management.
- The actual SNMP protocol is fairly simple, consisting of only a few easy-to-understand protocol operations.
- The implementation of SNMP is relatively simple for the designers and manufacturers of products.

KEY CONCEPT The *Simple Network Management Protocol (SNMP)* defines a set of technologies that allow network administrators to remotely monitor and manage TCP/IP network devices. The term SNMP refers both to a specific communication protocol (sometimes called the *SNMP protocol*) and an overall framework for Internet management (the *SNMP Framework*).

Since SNMP is a TCP/IP application layer protocol, it can theoretically run over a variety of transport mechanisms. It is most commonly implemented over the Internet Protocol (IP), but the most recent versions also define *transport mappings* that can allow SNMP information to be carried over other internetworking technologies.

Further Development of SNMP and the Problem of SNMP Variations

The first Internet Standard Management Framework developed (in 1988) is now called *SNMP version 1 (SNMPv1)*. This initial version of SNMP achieved widespread acceptance, and it is still probably the most common version of SNMP.

Much of the history of SNMP since that time has been a rather confusing standards nightmare. SNMPv1 had a number of weaknesses, particularly in the area of security. For this reason, shortly after SNMPv1 was done, work began on a new version of SNMP. Unfortunately, this effort became a quagmire, with many competing variations of SNMPv2 being created. After many years of confusion, none of the SNMPv2 variants achieved significant success.

Recently, a third version of the SNMP Framework and protocol has been published, which adds new features and reunites SNMP under a single, universal protocol again. The discussions of SNMP versions and SNMP standards later in this chapter further explore the history of SNMP since 1988. They can be considered a continuation of this historical overview, as they help clarify the very confusing story behind SNMP versions over the last decade and a half.

RELATED INFORMATION More background on the SNMP protocol proper can be found in the overview of the actual protocol itself, in Chapter 67.

TCP/IP SNMP Operational Model, Components, and Terminology

So, it seems the *Simple* Network Management Protocol isn't quite so simple after all. There are many versions and standards and uses of SNMP, and so a lot to learn. I think a good place to start in understanding what SNMP does is to look at its *model of operation*. Then we can examine the components that compose a TCP/IP network management system and define the terminology used to describe them.

SNMP Device Types

The overall idea behind SNMP is to allow the information needed for network management to be exchanged using TCP/IP. More specifically, the protocol allows a network administrator to make use of a special network device that interacts with other network devices to collect information from them and modify how they operate. In the simplest sense, two different basic types of hardware devices are defined:

Managed Nodes Regular nodes on a network that have been equipped with software to allow them to be managed using SNMP. These are, generally speaking, conventional TCP/IP devices. They are also sometimes called *managed devices*.

Network Management Station (NMS) A designated network device that runs special software to allow it to manage the regular managed nodes mentioned just above. One or more NMSs must be present on the network, as these devices are the ones that really run SNMP.

SNMP Entities

Each device that participates in network management using SNMP runs a piece of software, generically called an *SNMP entity*. The SNMP entity is responsible for implementing all of the various functions of the SNMP protocol. Each entity consists of two primary software components. Which components make up the SNMP entity on a device depends on whether the device is a managed node or an NMS.

Managed Node Entities

The SNMP entity on a managed node consists of the following software elements and constructs:

SNMP Agent A software program that implements the SNMP protocol and allows a managed node to provide information to an NMS and accept instructions from it.

SNMP Management Information Base (MIB) An MIB defines the types of information stored about the node that can be collected and used to control the managed node. Information exchanged using SNMP takes the form of objects from the MIB.

Network Management Station Entities

The SNMP entity on an NMS consists of the following:

SNMP Manager A software program that implements the SNMP protocol, allowing the NMS to collect information from managed nodes and to send instructions to them.

SNMP Applications One or more software applications that allow a human network administrator to use SNMP to manage a network.

SNMP Operational Model Summary

So, to integrate and reiterate all of this, let's summarize. SNMP consists of a small number of *network management stations (NMSs)* that interact with regular TCP/IP devices that are called *managed nodes*. The *SNMP manager* on the NMS and the *SNMP agents* on the managed nodes implement the SNMP protocol and allow network management information to be exchanged. *SNMP applications* run on the NMS and provide the interface to the human administrator, and allow information to be collected from the *management information bases (MIBs)* at each SNMP agent. Figure 65-1 illustrates the SNMP operational model.

An SNMP managed node can be pretty much any network device that can communicate using TCP/IP, as long as it is programmed with the proper SNMP entity software. SNMP is designed to allow regular hosts to be managed, as well as intelligent network interconnection devices, such as routers, bridges, hubs, and

switches. Other devices—printers, scanners, consumer electronic devices, specialized medical devices, and so on—can also be managed, as long as they connect to a TCP/IP internetwork.

On a larger network, an NMS may be a separate, high-powered TCP/IP computer dedicated to network management. However, it is really software that makes a device into an NMS, so the NMS may not be a separate hardware device. It may act as an NMS and also perform other functions on the network.

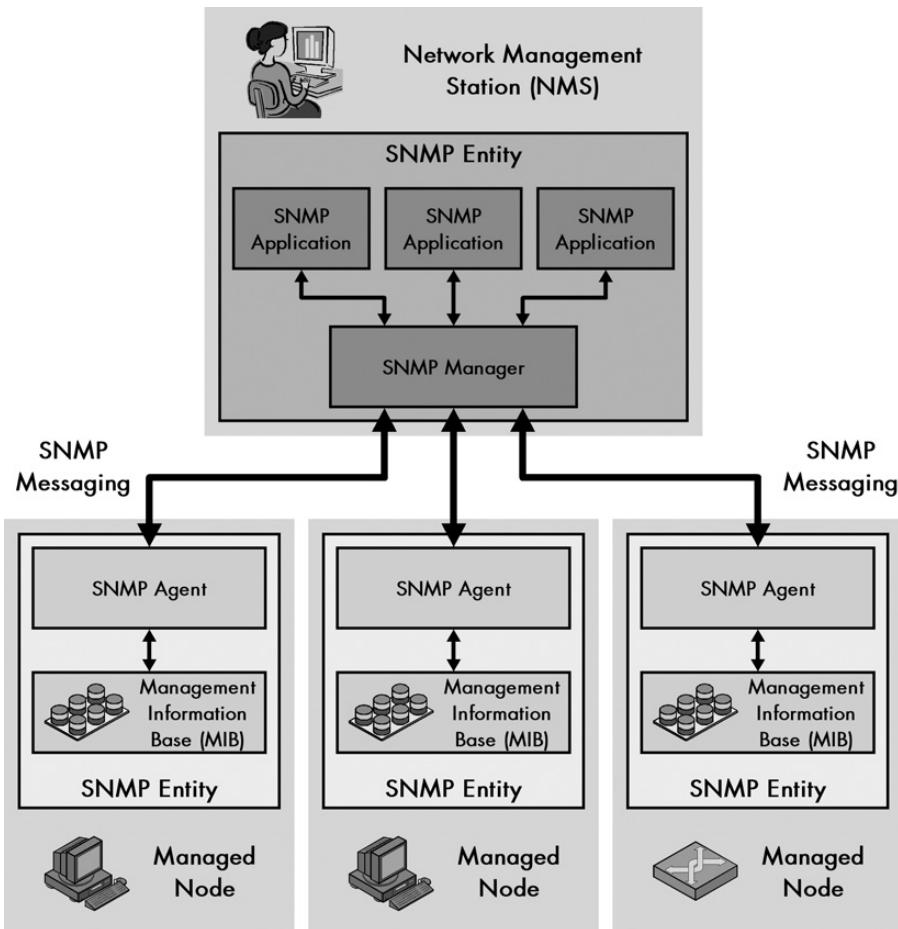


Figure 65-1: SNMP operational model This diagram shows a simplified implementation of SNMP, with one network management station (NMS) used to maintain three managed nodes. Each device has an SNMP entity, and they communicate using SNMP messages. The SNMP entity of the NMS consists of the SNMP manager and one or more SNMP applications. The managed nodes each run an SNMP agent and maintain a management information base (MIB).

KEY CONCEPT SNMP allows a network administrator using a *network management station (NMS)* to control a set of managed nodes. Each device incorporates an SNMP entity that implements the technology. In an NMS, the entity consists of an *SNMP manager* module and a set of SNMP applications. In a managed node, the entity consists of an *SNMP agent* and *management information base (MIB)*.

TCP/IP Internet Standard Management Framework

Architecture and Protocol Components

The Internet Standard Management Framework encompasses all of the technologies that compose the TCP/IP network management solution. The SNMP Framework consists of a number of architectural components that define how management information is structured, how it is stored, and how it is exchanged using the SNMP protocol. The Framework also describes how the different components fit together, how SNMP is to be implemented in network devices, and how the devices interact.

SNMP Framework Components

As we will explore in more detail in the next chapter, the Internet Standard Management Framework is entirely *information-oriented*. It includes four primary components (see Figure 65-2):

Structure of Management Information (SMI) To ensure interoperability of various devices, we want to have a consistent way of describing the characteristics of devices to be managed using SNMP. In computer science, a *data description language (DDL)* is the tool for this job. The *SMI* is a standard that defines the structure, syntax, and characteristics of management information in SNMP.

Management Information Bases (MIBs) Each managed device contains a set of variables that is used to manage it. These variables represent information about the operation of the device that is sent to an NMS, and/or parameters sent to the managed device to control it. The *MIB* is the full set of these variables that describe the management characteristics of a particular type of device. Each variable in a MIB is called a *MIB object*, and it is defined using the SMI data description language. A device may have many objects, corresponding to the different hardware and software elements it contains.

NOTE Initially, a single document defined the MIB for SNMP, but this model was inflexible. To allow new MIB objects to be more easily defined, groups of related MIB objects are now defined in separate RFC standards called MIB modules. More than 100 such MIB modules have been defined so far.

Simple Network Management Protocol (SNMP) This is the actual SNMP protocol itself. It defines how information is exchanged between SNMP agents and NMSs. The SNMP *protocol operations* define the various SNMP messages and how they are created and used. SNMP *transport mappings* describe how SNMP can be used over various underlying internetworks, such as TCP/IP, IPX, and others.

Security and Administration To the previous three main architectural components, the SNMP Framework adds a number of supporting elements. These provide enhancements to the operation of the SNMP protocol for security and address issues related to SNMP implementation, version transition, and other administrative issues.

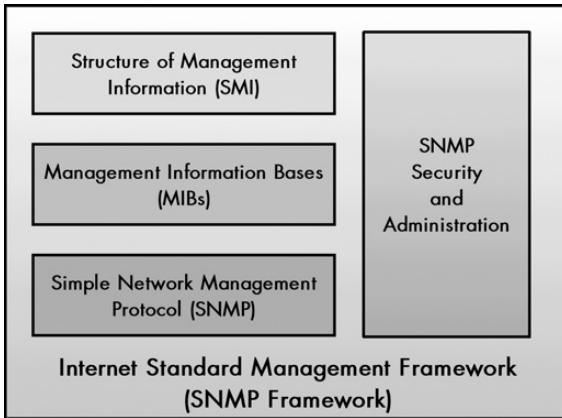


Figure 65-2: Components of the TCP/IP Internet Standard Management Framework

KEY CONCEPT The three main components of the Internet Standard Management Framework (SNMP Framework) are the Structure of Management Information (SMI), management information bases (MIBs), and the SNMP protocol itself. These are supported by SNMP security and administration elements.

SNMP Framework Architecture

The creators of SNMP specifically designed the Framework to be modular, because when SNMP was originally created, it was seen as only a temporary solution until a transition could be made to another network management protocol from the OSI protocol suite. The modular architecture separated definitional, data, and functional (protocol) elements, to allow the SNMP protocol itself to be replaced without changing how network management information was defined and described.

This transition to the OSI protocol never occurred, but the architecture has still proven valuable in defining the entire scope of SNMP and in making its implementation much simpler. Each of the major components discussed in the previous section—the SMI, MIBs, and SNMP itself—are described in different standards. The modularity of the SNMP Framework has also allowed changes to be made to these components relatively independently of each other, making the transition between SNMP versions easier than it would have been if one huge document defined everything.

TCP/IP Internet Standard Management Framework and SNMP Versions (SNMPv1, SNMPv2 Variants, and SNMPv3)

In Chapter 3, I explained the differences between proprietary, de facto, and open standards, and described the many benefits of open standards. History is replete with examples of technologies that have succeeded because they used an open standard when a competing standard was proprietary.

TCP/IP and the Internet are often held up as a model for proper open-standards development. Thousands of TCP/IP standards have been developed and published using the well-known Request for Comments (RFC) standardization process. The result has been the most successful set of internetworking protocols in computing history, accepted and used worldwide.

Nobody is perfect, however, and no process is perfect either. Some problems occurred in the introduction of SNMP version 2, leading to a virtual breakdown in the normally smooth protocol standardization method, and a proliferation of incompatible variants that we aren't used to seeing in TCP/IP. The story behind this is a continuation of the general SNMP overview and history from earlier in this chapter, and it explains the many SNMP standard names and numbers, so you can make sense of them. At the same time, the discussion serves as a vivid reminder of how important proper standard development is, and what the consequences are when there isn't universal agreement on how a standard should evolve.

SNMPv1

The first version of SNMP was developed in early 1988 and published in the form of three RFC standards in August 1988. This first version is now known as *SNMP version 1* or *SNMPv1*. The three SNMPv1 standards provided the initial description of the three main Internet Standard Management Framework components: the SMI, MIB, and SNMP protocol itself. However, the term *Internet Standard Management Framework* was not actually used at that time.

SNMPv1 was generally accepted and widely deployed in many networks. SNMPv1 got the job done and became the standard for TCP/IP network management. It is still widely used today. It is the Old Faithful of SNMP versions. Slight revisions were made to the initial standards, and more and more MIB modules were defined over time, but the technology remained the same for a number of years.

As with any technology, users of SNMPv1 identified weaknesses in it and opportunities for improvement. One of the areas in which SNMPv1 was most criticized was the area of security. SNMPv1 used only a "trivial" (as RFC 3410 puts it) authentication scheme, employing a password-like construct called a *community string*.

The issue of security turned out to be the bone of contention that eventually led to serious problems in the development of SNMP. Some people felt that community strings were sufficient security, but many others felt it was important that better security be put into SNMP. There were many different ways proposed to add security to SNMP, but no universal agreement on how to do it. The points raised about the security weaknesses in the original SNMPv1 had some validity, as I explore in the discussion of SNMP protocol operations in Chapter 67.

SNMPsec

The first attempt to add security came in the form of three standards published in July 1992 that defined a new security mechanism using logical identifiers called *parties*. This is sometimes called *SNMP Security* or *SNMPsec*. This method was more secure than the original SNMPv1, but SNMPsec was never widely accepted, and it is now considered historical.

SNMPv2

The idea of party-based security never went away, however. It was used as the basis of the definition of the first full revision of SNMP, when *SNMP version 2 (SNMPv2)* was published in RFCs 1441 through 1452 in April 1993. This new version incorporated the new security model, as well as making changes to the actual SNMP protocol operations, changes to the SMI standard (defining version 2 of SMI, SMIv2), and formalizing the concept of the Internet Standard Management Framework.

Unfortunately, this new standard also was never universally accepted. Some people thought the whole new version was a great advance, but others took issue with the party-based security, claiming it was too complex. A great deal of debate and discussion took place over the next couple of years, as an attempt was made to get everyone on board with the new version.

SNMPv2 Variants

Acceptance of SNMPv2 never happened. Instead, different splinter groups broke off and began work on *variants* of SNMPv2. To prevent confusion, the original SNMPv2 became known as either *SNMPv2 classic* (reminiscent of the name a particular soft drink) or *SNMPv2p*, with the *p* referring to party-based security. Things got very interesting (and confusing) when the following were proposed and/or developed:

SNMPv1.5 You can tell immediately that an idea is probably going to be a problem when it proposes a version number lower than a number already standardized. SNMPv1.5 was an attempt to retain the uncontroversial elements in SNMPv2p—the enhancements to the SNMP protocol and SMI—while going back to community-based security as in SNMPv1. It never became a standard itself, but became the basis of the next variant.

Community-Based SNMPv2 (SNMPv2c) This is SNMPv2p modified to use community strings instead of party-based security; in essence, the same idea as SNMPv1.5, but with a more official-sounding name and a few changes. Interestingly, the standard that defines this, RFC 1901, still has an experimental status, despite the fact that SNMPv2c actually achieved some degree of commercial success, where the standard SNMPv2p did not. SNMPv2c was defined by standards RFC 1902 through 1908, which incorporate other changes, including a new version of SMI (SMIv2).

User-Based SNMPv2 (SNMPv2u) This is an alternative security method for SNMPv2c, which is based on users rather than community strings. It is considered simpler than party-based but more secure than community-string security. It is defined by RFC 1909 and RFC 1910. It, too, is formally considered experimental.

SNMPv2* As if all of the other variants were not enough, a well-known vendor decided to define another variant called *SNMPv2** that combined elements of SNMPv2p and SNMPv2u. This was never formally standardized. (Yes, that's an asterisk in the name. No, there's no footnote at the bottom of the page, so don't bother looking for one. Yes, putting an asterisk in a name is extremely confusing. No, I don't know how it is that marketing people get paid good money to come up with names like this.)

Now, imagine that you were a network administrator in the mid-1990s and were faced with SNMPv2p, SNMPv2c, SNMPv2u, and SNMPv2*. Which one would you choose? Well, if you are like most people, you would choose none of the above, saying, “I think I’ll stick with SNMPv1 until these version 2 folks get their act together!” And that’s basically what happened. Some proponents of these variations promoted them, but there was never any agreement, and the result was that the success of all of the various and sundry SNMPv2s was limited. As I said, this is a classic illustration of how important universal standardization is.

SNMPv3

I would imagine that, at some point, everyone realized that the situation was a mess and decided enough is enough. In 1996, work began on a new approach to resolve the outstanding issues and return universality to SNMP. In 1998, *SNMP version 3 (SNMPv3)* was developed, which includes additional enhancements to SNMP and finally gets all the players back on the same team.

SNMPv3 is the most current version of SNMP and is still being actively revised. One of the important changes in SNMPv3 is a more formalized way of handing different security approaches to SNMP—obviously, a lesson learned from the SNMPv2 experience.

SNMPv3 uses SNMPv2 protocol operations and its protocol data unit (PDU) message format, and the SMIv2 standard from SNMPv2 as well. SNMPv3 allows a number of different security methods to be incorporated into its architecture, and includes standards describing user-based security as defined in SNMPv2u and SNMPv2*, as well as a new view-based access control model. It also includes additional tools to aid in the administration of SNMP.

TCP/IP Internet Standard Management Framework and SNMP Standards

You’ve now seen that there are three different versions of the Internet Standard Management Framework. Some of these versions have different variants. Each version or variant of the Framework includes multiple modular components. Each component has one or more documents that define it. Some of these have multiple revisions. Add to that dozens of individual MIBs defined for SNMP and other support documents, and what do you have? A boatload of TCP/IP standards, that’s what. There are probably more RFCs defining parts of SNMP than any other single TCP/IP protocol or technology.

It is specifically because there are so many versions and components and documents associated with SNMP that I feel it is important to keep all the standards straight. To that end, Tables 65-1 through 65-6 show the major SNMP standards for each of the versions and variants of the SNMP Framework: SNMPv1, SNMPsec, SNMPv2p, SNMPv2c, SNMPv2u, and SNMPv3. (SNMPv2* was not standardized using the regular RFC process.) Each individual RFC defines one component of one version of the Framework.

The usual way that RFCs work is that when new versions of a standard are released that are direct replacements for older ones, the older ones are obsoleted by the new ones. With SNMP, due to the many versions and the controversy over the variants, this is a bit unclear. For example, the standards defining SNMPv2p are not considered by the IETF to obsolete the standards for SNMPv1, but the IETF says the standards for SNMPv2c and SNMPv2u do obsolete those of SNMPv2p.

To keep all of this distinct, I decided to show the standards for each version or variant separately. I put the RFC numbers for obsolete RFCs only where those RFCs are for the same SNMP version or variant. For example, RFC 3410 obsoletes 2570 because they both deal with SNMPv3 and 3410 is a direct replacement for 2570.

Also, there are a few cases where the name of a standard changed slightly between RFC numbers; I have shown the current name. A full, hyperlinked list of RFCs can be found at <http://www.rfc-editor.org/rfc-index.html>.

Table 65-1: SNMP Version 1 (SNMPv1) Standards

Obsolete RFCs	Most Recent RFC	Date of Most Recent RFC	Standard Name
1065	1155	May 1990	Structure and Identification of Management Information for TCP/IP-Based Internets
1066	1156	May 1990	Management Information Base for Network Management of TCP/IP-Based Internets
1067, 1098	1157	May 1990	Simple Network Management Protocol (SNMP)
1158	1213	March 1991	Management Information Base for Network Management of TCP/IP-Based Internets: MIB-II

Table 65-2: SNMP Security (SNMPsec) Standards

Obsolete RFCs	Most Recent RFC	Date of Most Recent RFC	Standard Name
—	1351	July 1992	SNMP Administrative Model
—	1352	July 1992	SNMP Security Protocols
—	1353	July 1992	Definitions of Managed Objects for Administration of SNMP Parties

Table 65-3: Party-Based SNMP Version 2 (SNMPv2p) Standards

Obsolete RFCs	Most Recent RFC	Date of Most Recent RFC	Standard Name
—	1441	April 1993	Introduction to Version 2 of the Internet-Standard Network Management Framework
—	1442	April 1993	Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)
—	1443	April 1993	Textual Conventions for Version 2 of the Simple Network Management Protocol (SNMPv2)
—	1444	April 1993	Conformance Statements for Version 2 of the Simple Network Management Protocol (SNMPv2)

(continued)

Table 65-3: Party-Based SNMP Version 2 (SNMPv2p) Standards (continued)

Obsolete RFCs	Most Recent RFC	Date of Most Recent RFC	Standard Name
—	1445	April 1993	Administrative Model for Version 2 of the Simple Network Management Protocol (SNMPv2)
—	1446	April 1993	Security Protocols for Version 2 of the Simple Network Management Protocol (SNMPv2)
—	1447	April 1993	Party MIB for Version 2 of the Simple Network Management Protocol (SNMPv2)
—	1448	April 1993	Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)
—	1449	April 1993	Transport Mappings for Version 2 of the Simple Network Management Protocol (SNMPv2)
—	1450	April 1993	Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)
—	1451	April 1993	Manager-to-Manager Management Information Base
—	1452	April 1993	Coexistence Between Version 1 and Version 2 of the Internet-Standard Network Management Framework

Table 65-4: Community-Based SNMP Version 2 (SNMPv2c) Standards

Obsolete RFCs	Most Recent RFC	Date of Most Recent RFC	Standard Name
—	1901	January 1996	Introduction to Community-Based SNMPv2
—	1902	January 1996	Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)
—	1903	January 1996	Textual Conventions for Version 2 of the Simple Network Management Protocol (SNMPv2)
—	1904	January 1996	Conformance Statements for Version 2 of the Simple Network Management Protocol (SNMPv2)
—	1905	January 1996	Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)
—	1906	January 1996	Transport Mappings for Version 2 of the Simple Network Management Protocol (SNMPv2)
—	1907	January 1996	Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)
—	1908	January 1996	Coexistence between Version 1 and Version 2 of the Internet-Standard Network Management Framework

Table 65-5: User-Based SNMP Version 2 (SNMPv2u) Standards

Obsolete RFCs	Most Recent RFC	Date of Most Recent RFC	Standard Name
—	1909	February 1996	An Administrative Infrastructure for SNMPv2
—	1910	February 1996	User-Based Security Model for SNMPv2

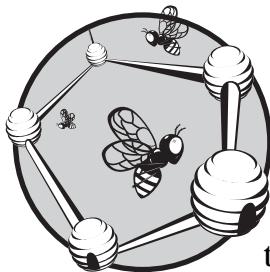
Table 65-6: SNMP Version 3 (SNMPv3) Standards

Obsolete RFCs	Most Recent RFC	Date of Most Recent RFC	Standard Name
—	2576	March 2000	Coexistence between Version 1, Version 2, and Version 3 of the Internet-Standard Network Management Framework
—	2578	April 1999	Structure of Management Information Version 2 (SMIV2)
—	2579	April 1999	Textual Conventions for SMIV2
—	2580	April 1999	Conformance Statements for SMIV2
2570	3410	December 2002	Introduction and Applicability Statements for Internet-Standard Management Framework
2261, 2271, 2571	3411	December 2002	An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks
2262, 2272, 2572	3412	December 2002	Message Processing and Dispatching for the Simple Network Management Protocol (SNMP)
2263, 2273, 2573	3413	December 2002	Simple Network Management Protocol (SNMP) Applications
2264, 2274, 2574	3414	December 2002	User-Based Security Model (USM) for Version 3 of the Simple Network Management Protocol (SNMPv3)
2265, 2275, 2575	3415	December 2002	View-Based Access Control Model (VACM) for the Simple Network Management Protocol (SNMP)
—	3416	December 2002	Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP)
—	3417	December 2002	Transport Mappings for the Simple Network Management Protocol (SNMP)
—	3418	December 2002	Management Information Base (MIB) for the Simple Network Management Protocol (SNMP)

In addition to all of the standards listed in these tables, there are dozens of supplemental RFCs that describe MIB modules and also clarify various fine points of operation related to SNMP. You can find all the MIBs in an online list of RFCs by searching for “MIB” or “SNMP.”

66

TCP/IP STRUCTURE OF MANAGEMENT INFORMATION (SMI) AND MANAGEMENT INFORMATION BASES (MIBS)



The Internet Standard Management Framework defines three major components that describe how devices can be managed on a TCP/IP internetwork. One of these, the actual Simple Network Management Protocol (SNMP) is relatively well known, but is only part of the overall picture. SNMP describes how information is exchanged between SNMP entities, but two other components are equally important, because they describe the information itself.

In this chapter, I describe these two important supporting elements of the TCP/IP Internet Standard Management Framework: the Management Information Base (MIB) standard that describes types of information that SNMP works with, and the Structure of Management Information (SMI) standard that specifies how MIB information is defined. Understanding these two parts of the SNMP Framework is an important initial step before we examine the actual SNMP protocol itself.

I begin with an overview description of the SMI data description language and how MIBs work. I discuss the MIB object name hierarchy

and the notation used to refer to names. I also describe how MIB objects work, discussing the different object types and MIB object groups. I describe MIB concepts common to all of the versions of SNMP, and discuss both of the specific versions of SMI (SMIv1 and SMIv2) used in those SNMP versions.

BACKGROUND INFORMATION *If you have not yet already read the preceding chapter describing the SNMP Internet Standard Management Framework, you should do so before proceeding here.*

TCP/IP SMI and MIBs Overview

The key to really understanding TCP/IP network management is to comprehend the *information-oriented* nature of the entire Internet Standard Management Framework (SNMP Framework). To see what I mean by this, let's step back for a moment and consider in general terms the problem of network management, and more specifically, the problem of managing devices on a network.

SNMP's Information-Oriented Design

A network administrator needs to perform two basic types of actions: gather data about devices to learn how they are functioning and give commands to devices to change how they are functioning. In the simplest terms, the first category can be considered as a read operation, and the second is comparable to a write operation.

A classic way of implementing this functionality is to define a communication protocol. Most such protocols are *command-oriented*—they consist of a specific set of commands to perform the read and write operations. For example, a network management protocol might have a read command such as “report on number of hours device has been in use,” and a write command might be something like “put this device into test mode.” The network manager would control the device by giving the appropriate commands.

A command-oriented management protocol has the advantage of simplicity, since it’s clear what the commands are for and how they are to be used. It can be reasonably well suited for use in certain environments, but it doesn’t work well on a large, heterogeneous TCP/IP internetwork. The main reason for this is that command-orientation inextricably ties the protocol to the devices being managed. Consider the following problems:

- Every type of device might require a distinct set of commands. For example, the commands given to a router might need to be different than those given to a host. This would lead either to a proliferation of commands in the protocol or to inflexibility in allowing proper management of different device types.
- Every time a company created a new type of device, or made a unique version of a type of device, the network management protocol would need to be changed.
- Whenever the operation of a kind of device changed, due perhaps to a change in another protocol, the management protocol would need to be updated.
- The protocol itself could not be easily changed without affecting a lot of hardware.

The solution to the problems of command-oriented management protocols is to use an *information-oriented* model. Instead of defining specific commands that interrogate or control devices, the devices are defined in terms of units of information that are to be exchanged between the devices and a management station.

Instead of read commands and write commands, we have *variables* that can be read or written. Take the two examples mentioned earlier. Instead of a command like “report on a number of hours device has been in use,” the device keeps a variable called “number of hours in use,” and the network management station can read this as one of many variables, with no need for a specific protocol command. Instead of a write command called “put this device into test mode,” the device has a variable called “current mode.” The network manager can change the mode of the device to test mode by changing the value of the variable.

This difference may seem subtle, but it underlies every aspect of how SNMP works. I believe part of why the SNMP Framework is hard to understand is because insufficient emphasis is placed on looking at things in the “SNMP way,” which means thinking about information objects and not commands.

KEY CONCEPT Unlike most protocols, which are *command-oriented*, SNMP is *information-oriented*. SNMP operations are implemented using objects called *variables* that are maintained in managed devices. Rather than issuing commands, a network management station checks the status of a device by reading variables, and controls the operation of the device by changing (writing) variables.

MIB and MIB Objects

Given this backdrop, we can look at the SNMP Framework in a new light. The actual SNMP protocol itself, which we’ll examine in the next couple of chapters, has only a few, generic commands to accomplish read and write tasks. It deals with only the methods by which network management information is exchanged between SNMP agents and SNMP network management stations (NMSs), which were described in the previous chapter. The network management information is really the heart of TCP/IP network management.

So, instead of SNMP being defined in terms of commands used to control particular devices, it is defined in terms of management information variables, generally called *objects*. Each object describes a particular characteristic of a device. Some objects are fairly generic and are meaningful for any device on a TCP/IP network; for example, an object describing something related to the Internet Protocol (IP) itself, such as the device’s IP address. Other objects might be particular to a specific type of device; for example, a router will have objects that a regular host’s Ethernet network interface card would not.

A collection of objects used in SNMP is called a *management information base*, or *MIB*. (In fact, SNMP objects are often called *MIB objects*.) The first version of SNMP, SNMPv1, had a single standard that defined the entire MIB for SNMP. Newer versions provide more flexibility by using different *MIB modules* that define sets of variables particular to the hardware or software used by a device.

KEY CONCEPT The management data variables in a managed device are maintained in a logical collection called a *management information base (MIB)*. The objects in the MIB are often called *MIB objects*, and they are typically collected into sets called *MIB modules*.

Defining objects using modules allows for significant flexibility in defining the variables that allow management of different types of devices. A device can incorporate all the MIB modules appropriate to the hardware and software it uses. For example, if you had a device using Ethernet, it would incorporate variables from the Ethernet MIB. A device using Token Ring would use the Token Ring MIB. Both devices would also use the common SNMP MIB that is used by all TCP/IP devices. Other modules might also be included as needed.

NOTE Due to its name, the MIB is often called a database. This is, strictly speaking, inaccurate. The MIB is a description of objects. The actual MIB in a device may be implemented as a software database, but that is not required.

Defining MIB Objects: SMI

The use of MIB objects solves the problem of the network management protocol being tied to the network management information. However, we must be very particular about how we define these objects. Again, the reason is the wide variety of devices that TCP/IP allows to be connected together. Each device may represent information in a different way. For all of them to communicate with each other, we need to ensure that management information is represented in a consistent manner.

The part of the SNMP Framework that ensures the universality of MIB objects is the *Structure of Management Information (SMI)* standard. SMI defines the rules for how MIB objects and MIB modules are constructed. In SMI, MIB objects are described using a precise set of definitions based on a data description language called the ISO *Abstract Syntax Notation 1 (ASN.1)* standard.

In essence, we really have three levels of abstraction in SNMP. The actual SNMP protocol moves values that represent the state of management devices. The MIB defines what these variables are. And the SMI defines how the variables in the MIB are themselves defined.

There are two main SMI standards. The original, *SMIv1*, was part of the first SNMP Framework, SNMPv1, defined in RFC 1155. It sets out the basic rules for MIBs and MIB variables. The second, *SMIv2*, was defined as part of SNMPv2p in RFC 1442 and further updated in RFC 2578, part of SNMPv3. It is similar to the earlier version, but defines more object types, as well as the structure of MIB modules.

These SMI standards are responsible for defining the following important information elements in SNMP:

- The general characteristics associated with all MIB objects—the standard way by which all MIB objects are described
- The different types of MIB objects that can be created, such as integers, strings, and more complex data types

- A hierarchical structure for naming MIB objects, so they can be addressed in a consistent manner without names overlapping
- The information associated with each MIB module

KEY CONCEPT The *Structure of Management Information (SMI)* standard is responsible for defining the rules for how MIB objects are structured, described, and organized. SMI allows dissimilar devices to communicate by ensuring that they use a universal data representation for all management information.

TCP/IP MIB Objects, Object Characteristics, and Object Types

As explained in the previous sections, the SNMP Framework is designed to facilitate the exchange of management information. The MIB defines a device's management information and contains a number of variables called *MIB objects*, also called *managed objects*. These objects are defined according to the rules set out in the SMI standard.

The best place to begin looking at MIB objects is by examining the SMI rules that define them. As I mentioned earlier in this chapter, two different versions of SMI have been created: SMIv1 as part of the original SNMP, and SMIv2 as part of SNMPv2 and SNMPv3. The two are similar in terms of how MIB objects are described, but SMIv2 allows more information to be associated with each object.

MIB Object Characteristics

Just as a typical protocol uses a field format for specifying the content of messages sent between devices using the protocol, SMI uses a format that specifies the fundamental characteristics of each MIB object. The most basic of these are five mandatory characteristics defined in SMIv1. These are also used in SMIv2, but a couple of names were changed, and the possible values for some of the fields were modified as well. An MIB object may have the following characteristics (see Figure 66-1):

Object Name Each object has a name that serves to uniquely identify it. Actually, that's not entirely true. Each object has *two* names: a textual name called an *object descriptor* and a numeric *object identifier*, which indicates the object's place in the MIB object name hierarchy. We'll explore these names and how they are used shortly.

Syntax Defines the object's data type and the structure that describes it. This attribute is very important because it defines the data type of information that the object contains. There are two basic categories of data types allowed:

- Regular data types are single pieces of information, of the type we are used to dealing with on a regular basis, such as integers and strings. These are called *base types* in SMIv2. SMIv1 differentiates between *primitive types* like integers defined in ASN.1, and *defined types* that are special forms of primitive types that are still single pieces of information but with certain special meaning attached to how they are used. SMIv2 doesn't use those two terms.

- Tabular data types are collections of multiple data elements. They may take the form of a list of base types or a table of base types. For example, a table of integers could be constructed to represent a set of values. In SMIv1, these are called *constructor types*, in SMIv2 they are *conceptual tables*. They can be accessed using special SNMP mechanisms designed for reading tables. See the topic on SNMP table traversal for more on tables.

Access (Max-Access in SMIv2) This field defines the ways that an SNMP application will normally use the object. In SMIv1, there are four different possible values: *read-only*, *read-write*, *write-only*, and *not-accessible*. In SMIv2 there are five values, which are described as a hierarchy of sorts. SMIv2 calls this characteristic *Max-Access (maximum access)* to make it explicit that higher access levels include the lower levels as well. For example, an object with *read-create* access can also be used in any of the modes below it, such as *read-write*, but not vice versa. The following are the five SMIv2 access values, in decreasing order of access (note that *write-only* has been removed in SMIv2):

- *read-create* (object can be read, written, or created)
- *read-write* (object can be read or written)
- *read-only* (object can only be read)
- *accessible-for-notify* (object can be used only using SNMP notification or SNMP traps)
- *not-accessible* (used for special purposes)

Status Indicates the currency of the object definition. In SMIv1 there are three values: *mandatory*, *optional*, and *obsolete*. In SMIv2, the first two are combined into simply *current*, meaning a current definition. The value *obsolete* is retained, and *deprecated* is added, meaning the definition is obsolete but maintained for compatibility.

Definition (Description in SMIv2) A textual description of the object.

Optional Characteristics SMIv2 adds the following optional characteristics that may appear in the definition of an object:

- *Units* is a textual description of the units associated with the object.
- *Reference* is a text cross-reference to a related document or other information relevant to the object.
- *Index* is a value used to define objects that are actually more complex rows of other objects.
- *Augments* is an alternative to the *Index* field.
- *DefVal* defines an acceptable default value for the object.

KEY CONCEPT Each management information variable, called an *MIB object*, has associated with it five key attributes: its name, syntax, maximum access, status, and definition. It may also have a number of optional characteristics.

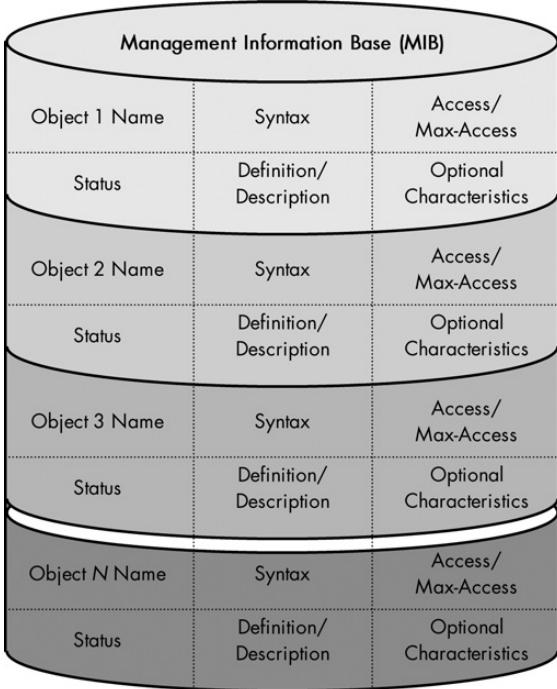


Figure 66-1: SNMP management information base (MIB) This diagram shows an SNMP MIB containing N MIB objects. Each object has five mandatory characteristics and a variable number of optional characteristics.

SMI Data Types

Table 66-1 shows the regular data types supported for objects in both SMIv1 and SMIv2. (The names with 32 in them are the ones used in SMIv2; they were changed to make the type's bit size explicit.) The first five entries in the table are primitive types; the rest are defined types, using the SMIv1 terminology.

Table 66-1: SNMP SMI Regular Data Types

Data Type Code	Description	In SMIv1?	In SMIv2?
Integer/Integer32	A 32-bit signed integer in two's complement notation, capable of holding a value from -2,147,483,648 to +2,147,483,647. Can also be used to represent an enumerated type; for example, where 1 represents a particular constant, 2 represents a different one, and so on.	Yes	Yes
Octet String	A variable-length string of binary or text data.	Yes	Yes
Null	Nothing.	Yes	No
Bits	An enumeration of named bits. Used to allow a set of bit flags to be treated as a single data type.	No	Yes
Unsigned	A 32-bit unsigned integer, from 0 to 4,294,967,295.	No	Yes
Network Address/ IpAddress	An IP address, encoded as a 4-byte octet string.	Yes	Yes <i>(continued)</i>

Table 66-1: SNMP SMI Regular Data Types (continued)

Data Type Code	Description	In SMIv1?	In SMIv2?
Counter/Counter32	A 32-bit unsigned integer that begins at 0 and increases up to 4,294,967,295, then wraps back to 0.	Yes	Yes
Gauge/Gauge32	A 32-bit unsigned integer that may have a value from 0 to 4,294,967,295 and may increase or decrease, like a gauge. A minimum and maximum value are associated with the gauge, indicating its normal range.	Yes	Yes
TimeTicks	A 32-bit unsigned integer that indicates the number of hundredths of seconds since some arbitrary start date. Used for timestamping and to compute elapsed time.	Yes	Yes
Opaque	Data using arbitrary ASN.1 syntax that is to be passed between devices without being interpreted. As in the Network File System's (NFS) XDR (see Chapter 58), the term <i>opaque</i> means that the data is treated like a black box, whose internal details cannot be seen.	Yes	Yes
Counter64	A counter like Counter32 but 64 bits wide, allowing a value from 0 to 18,446,744,073,709,551,615.	No	Yes

In addition to the types shown in Table 66-1, other defined types are also created to indicate more specific semantics for a particular data type. These are called *textual conventions* and are described in RFC 2579 for SMIv2. For example, a type called *TimeStamp* is the same as *TimeTicks*. However, seeing an object using the former rather than the latter makes it more clear that the variable is representing a particular timestamp value. Another is called *TimeInterval*, which is also just an integer underneath its name, but conveys a different interpreted meaning.

If all of this seems very confusing to you, note that this description is actually a significant simplification of SMI's object definitions. Check out Listing 66-1, which shows an object definition from RFC 3418, using SMIv2.

```
sysLocation OBJECT-TYPE
    SYNTAX DisplayString (SIZE (0..255))
    MAX-ACCESS read-write
    STATUS current
    DESCRIPTION "The physical location of this node
        (e.g., 'telephone closet, 3rd floor'). If the location
        is unknown, the value is the zero-length string."
        ::= { system 6 }
```

Listing 66-1: Example SNMP SMIv2 object definition

Note that *DisplayString* is a textual convention for a displayed text string. The last part, { system 6 }, will be explained in the next section.

TCP/IP MIB Object Descriptors and Identifiers and the Object Name Hierarchy

Of the many MIB object characteristics, only one is sufficiently interesting that it really deserves its own exposition. Or perhaps I should say that only one is

sufficiently complicated to require further explanation. This is the object name, part of the larger naming system used for MIB objects.

Each MIB object actually has two names: an *object descriptor* and an *object identifier*.

Object Descriptors

The object descriptor is a conventional text name that provides a user-friendly handle to refer to the object. The name is assigned based on the particular MIB object group in which the object is located. In the previous example, `sysLocation` is the object descriptor for that MIB object. I describe these names in greater detail later in this chapter, when I discuss MIB modules and object groups.

Object Identifiers

Text names are convenient, but they are generally unstructured. There are at present more than 10,000 different MIB objects, and even if each has a distinct text name, a huge collection of such names doesn't help us to manage these objects and see how they are related. For this, we need a more structured approach to categorizing and naming objects.

This problem is similar to another problem that you may recall reading about: the problem of how to assign names on the Internet. Originally, names for hosts were simple, flat names, but this quickly grew unwieldy. The DNS hierarchical name space (see Chapter 53) allows every device to be arranged into a single hierarchical tree structure. The name of the device can be formed by traversing the tree from the top down to the location of the device, listing the labels traversed separated by dots. For example, the web server of The PC Guide is at <http://www.pcguide.com>.

This same concept is used to organize MIB objects in SNMP. A single, universal hierarchy that contains all MIB objects is used. It is hierarchical in nature, and it is split into levels from the most general to the most specific. Each object has a particular place in the hierarchy.

There is an important difference between the MIB name hierarchy and the DNS one: the MIB name hierarchy is even more universal than the one for DNS. The entire subtree of all MIB objects is just one branch of the full, international object hierarchy maintained by the International Organization for Standardization (ISO) and the International Telecommunication Union (ITU). This object identification hierarchy is so general that it can contain a name for every object or variable in use by any technology in the entire world (and possibly other planets or solar systems).

The reason for my jocularity will become apparent in a moment. Suffice it to say that this object tree is enormous. Each node in this tree is identified with both a label and an integer. The labels are for descriptive purposes. Object (or subtree) identifiers are formed by listing the numbers in sequence from the top of the tree down to the node, separated by dots. SNMP doesn't reverse the order of the labels the way DNS does, however. They are listed top-down from left to right. (The text labels can be used for names, too, but they are not because they would get very long due to how deep the tree structure is.)

KEY CONCEPT SNMP MIB objects have two names. The first is a *text object descriptor*, which provides a means of addressing the object in a way that is familiar and easy for humans. The second is the *object identifier*, which consists of a sequence of integers that specifies the location of the object in the global object hierarchy maintained by the international standards bodies ISO and ITU.

Structure of the MIB Object Name Hierarchy

Let's explore how the MIB object tree is structured, and more important, how SNMP MIB objects fit into it. Figure 66-2 illustrates the global object name hierarchy and SNMP MIB hierarchies.

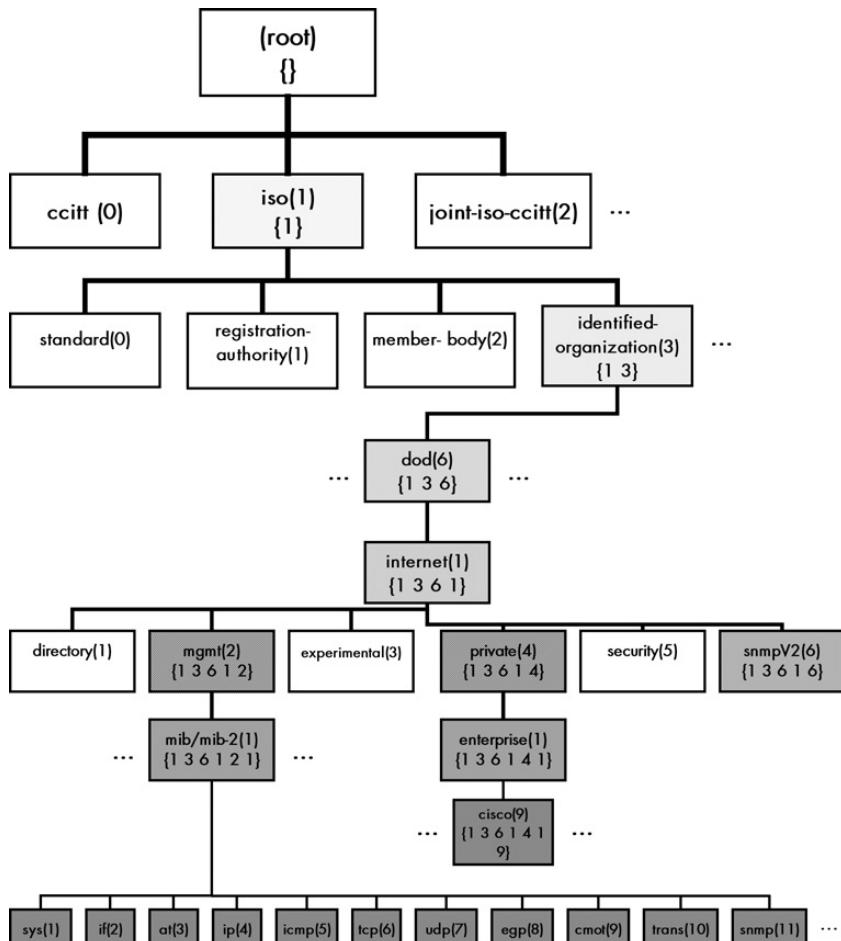


Figure 66-2: Global object name hierarchy and SNMP MIB hierarchies This diagram shows the object name hierarchy defined by ISO and CCITT (ITU) to allow all types of objects to be universally represented. The path within this larger tree to the tree branches relevant to SNMP can be found by following the shaded boxes. The two subtrees used for SNMP are shown as the hatched boxes under *internet(1)*. Each contains its own substructure (some of which is illustrated here) defining thousands of different MIB objects. The branch on the left side is used for generic MIB objects and the one on the right for private ones. A separate hierarchy is also defined for SNMPv2.

The tree's root has no label, and has three children:

- ccitt(0) for ITU (formerly the CCITT) standards (also seen as itu(0)).
- iso(1) for ISO standards.
- joint-iso-ccitt(2) for joint standards (also seen as joint-iso-itu(2)).

Following the iso(1) node, we see the following at the next several levels:

- Within iso(1), the ISO has created a subtree for use by other organizations, called org(3).
- Within org(3), there is a subtree for the United States Department of Defense (which, as you may recall, was the originator of the Internet): dod(6).
- Within dod(6), there is a subtree called internet(1).

Everything we work with in SNMP is under this one very specific subtree:

1.3.6.1, which if we used the text labels would be iso.org.dod.internet. Within this part of the name space, there are six subtrees below:

- directory(1) is reserved for future use by ISO.
- mgmt(2) is the primary subtree where MIB objects are located. This is 1.3.6.1.2. It contains a subtree called mib(1), which is 1.3.6.1.2.1. When MIB-II was created, a subtree called mib-2(1) was created using the same number, 1.3.6.1.2.1.
- experimental(3) contains objects used for standards under development. This is 1.3.6.1.3.
- private(4) is used for objects defined by private companies. This node, 1.3.6.1.4, has a subtree called enterprise(1), which is 1.3.6.1.4.1.
- security(5) is reserved for security use.
- snmpV2(6) defines objects used specifically for SNMP version 2.

So, what's the bottom line of all this? Well, basically all MIB module objects are named within one of these two branches of the overall object tree:

Regular MIB Objects These are in the mib(1) subtree under mgmt(2):
1.3.6.1.2.1.

Private MIB Objects These are in the enterprise(1) subtree under private(4), which is 1.3.6.1.4.1. For example, within enterprise(1), there is an entry cisco(9) for Cisco Systems. So all Cisco-specific MIB objects start with 1.3.6.1.4.1.9.

Clear as mud, right? Why didn't they just make a separate hierarchy where "mib" was at the top instead of six levels deep? How dare you even suggest such a thing? Don't you understand the importance of global standards?

All facetiousness aside, this name hierarchy is a bit cumbersome to deal with (okay, more than a bit), but it does allow us to keep MIB objects organized in a sensible way. Within the 1.3.6.1.2.1 subtree, we find most of the regular MIB objects used in SNMP. Each subtree within 1.3.6.1.2.1 corresponds to one of the regular SNMP object groups or a particular MIB module.

KEY CONCEPT All MIB objects have object identifiers that fit within two branches of the global object hierarchy. Regular MIB objects (which are not vendor-specific) fit in the mib(1) subtree under mgmt[2]: 1.3.6.1.2.1. Private objects, which can be created by a hardware vendor to assist in managing that vendor's products, are in the enterprise(1) subtree under private[4], which is 1.3.6.1.4.1.

Recursive Definition of MIB Object Identifiers

An object is given a text object descriptor by putting its name at the start of the object, as shown in Listing 66-1, but the definition of numeric object identifiers is, again, more complex. It is done by defining only the number of the object within its particular subtree. This means the object identifiers are defined recursively (one based on another) and are not explicitly stated for each object. This is syntactically precise, but makes it hard to see at a glance what the number is for any particular object.

Consider again the example in Listing 66-1. For this object, sysLocation is the object descriptor and { system 6 } is the object identifier. This means it is object number 6 within the node system, which is in turn defined as { mib-2 1 }—it is the first node within the mib-2 subtree. Since mib-2 is 1.3.6.1.2.1, as noted in the previous section, this means system is 1.3.6.1.2.1.1 and sysLocation is 1.3.6.1.2.1.1.6.

TCP/IP MIB Modules and Object Groups

The MIB contains the collection of MIB objects that describe the characteristics of a device using the SNMP Framework. When SNMP was first created, there were not that many objects in the MIB. Furthermore, they were mostly generic objects that applied fairly universally to TCP/IP devices as a whole. In fact, most of the MIB objects were variables related to the operation of TCP/IP protocols such as IP, the Transmission Control Protocol (TCP), and the Internet Control Message Protocol (ICMP).

For this reason, at first, a single document defined “the” MIB for SNMP. The first of these documents was RFC 1066, part of the initial SNMPv1 specification. It was then revised in RFC 1156. In RFC 1158, a second version of the MIB, *MIB II*, was defined, which was essentially the same but made a few changes.

The Organization of MIB Objects into Object Groups

The number of MIB objects defined in the early MIB standards was relatively small. However, there were still several dozen of them, and it was recognized from the start that more would be created in time. To help organize the objects in a logical way, they were arranged into *object groups*. These groups serve the purpose of separating the objects and defining how they should be given object identifiers in the overall object name hierarchy.

Each group has associated with it three important pieces of information:

Group Name This is a name that is used as a text label in the object identification tree described earlier in this chapter (see Figure 66-2). These objects are all located within the iso.org.dod.internet.mgmt.mib subtree. So, for example, the group system would be iso.org.dod.internet.mgmt.mib.system.

Group Number This number corresponds to the group name used for making numeric identifiers from the object name tree. For example, the group system has the number 1, and so the group's object identifier is 1.3.6.1.2.1.1. All objects in that group will be under that tree; for example, sysUpTime is 1.3.6.1.2.1.1.3.

Group Code This is a text label that may be the same as the group name or may be an abbreviation. It is used as a prefix in making object descriptors (the text names of objects). For example, for the group system, the code is sys, and so an object in this group is sysUpTime.

Table 66-2 shows the eight generic SNMP groups defined in RFC 1158, along with their codes, names, and numbers.

Table 66-2: SNMP Generic MIB Object Groups

Group Name	Group Code	Group Number	Full Group Identifier	Description
system	sys	1	1.3.6.1.2.1.1	General objects of relevance to all or most devices. For example, a general description of the device is an object in this group, as is the identifier of the object. Later MIB versions greatly expanded the number of variables in this group.
interfaces	if	2	1.3.6.1.2.1.2	Objects related to the IP interfaces between this device and the internetwork. (Recall that a regular host normally has one interface, while a router has two or more.)
at (address translation)	at	3	1.3.6.1.2.1.3	Objects used for IP address translation. (No longer used.)
ip	ip	4	1.3.6.1.2.1.4	Objects related to the IP layer of the device as a whole (as opposed to interface-specific information in the if group).
icmp	icmp	5	1.3.6.1.2.1.5	Objects related to the operation of ICMP.
tcp	tcp	6	1.3.6.1.2.1.6	Objects related to the operation of the TCP.
udp	udp	7	1.3.6.1.2.1.7	Objects related to the operation of the User Datagram Protocol (UDP).
egp	egp	8	1.3.6.1.2.1.8	Objects related to the operation of the Exterior Gateway Protocol (EGP).
cmot	cmot	9	1.3.6.1.2.1.9	Objects related to running the CMIP protocol over TCP (historical, not used).
transmission	trans	10	1.3.6.1.2.1.10	Objects related to the specific method of information transmission used by each interface on the system.
snmp	snmp	11	1.3.6.1.2.1.11	Objects used to manage SNMP itself.

All of the groups in this table are fairly generic, and with the exception of the one about EGP, apply to pretty much every TCP/IP system using SNMP. (The mention of EGP, a routing protocol now considered obsolete, shows the age of this list.) The first five groups and the last one are mandatory for all systems. The others are used only by devices that use the indicated protocols or functions.

MIB Modules

What's most conspicuous about the object groups listed in Table 66-2 is the groups that are not included. There are no groups for most of the other TCP/IP protocols, nor any for variables that might be needed for specific hardware types. For example, most hosts will have a network card in them using a layer 2 protocol like Ethernet or Token Ring. How does a manager check or control the operation of this hardware? What about newer routing protocols like Open Shortest Path First (OSPF) or Border Gateway Protocol (BGP)? How about objects related to running the Domain Name System (DNS)?

Updating the MIB document constantly would have been impractical. Instead, in SNMPv2, the MIB was changed from a single document to a group of documents. The basic organization into groups of objects was retained, but instead of all groups being in the same standard, they are divided into multiple standards. A method was also defined for how to create *MIB modules* that describe new groups of objects specific to a particular technology. A list of these modules is maintained by the *Internet Assigned Numbers Authority* (IANA), the organization that maintains all of these sorts of numbers. The current list of SNMP MIB modules can be found at <http://www.iana.org/assignments/smi-numbers>.

The use of MIB modules makes putting SNMP support into a device somewhat like going shopping. The basic groups common to all devices are incorporated into each device, and then other modules/groups are used as needed. Table 66-3 provides a brief selection of MIB modules to give you an idea of what is out there, also showing the module's group number (within the 1.3.6.1.2.1 name subtree). There are many, many more modules than listed in this table.

Table 66-3: Some Common SNMP MIB Modules

MIB Module Name	Group Number	Description
ospf	14	Objects related to OSPF
bgp	15	Objects related to BGP
rmon	16	Objects used as part of Remote Network Monitoring (RMON)
snmpDot3 RptrMgt	22	Objects related to IEEE 802.3 (Ethernet) repeaters
rip-2	23	Objects used as part of version 2 of the Routing Information Protocol (RIP)
snmpDot3 MauMgt	26	Objects related to IEEE 802.3 (Ethernet) medium attachment units
etherMIB	35	Ethernet-like generic objects

(continued)

Table 66-3: Some Common SNMP MIB Modules (continued)

MIB Module Name	Group Number	Description
mipMIB	44	Mobile IP objects
ipMIB	48	IP objects for SNMPv2
tcpMIB	49	TCP objects for SNMPv2
udpMIB	50	UDP objects for SNMPv2

The last three entries in Table 66-3 might seem a bit confusing, since there were already groups for IP, TCP, and UDP, as shown in Table 66-2. The reason for these is that when the new modular architecture for MIB objects was created in SNMPv2, the definition of objects for the individual protocols that was part of the one document in SNMPv1 was separated out into individual MIB documents for consistency and to allow them to be updated independently. In fact, the base SNMPv2 and SNMPv3 MIB documents now define only objects in the `system` and `snmp` groups.

KEY CONCEPT MIB objects created early in SNMP's history were organized into *MIB object groups* that reside within the `mib(1)` subtree, starting with identifier code `1.3.6.1.2.1`. As the popularity of TCP/IP grew, it became impractical to centrally define all MIB objects, so sets of objects particular to different hardware devices are now specified in *MIB modules*.

MIB Module Format

The format for MIB modules is described in the SMI standard, version 2 (SMIv2). This document specifies how modules are to be defined in a way similar to how objects themselves are defined: by listing a set of characteristics that must be included in each module description. The module fields are as follows:

Module Name The name of the module. Remember that modules are really objects, syntactically, so like regular objects, they have a textual object descriptor (like `tcpMIB`) and an object identifier (in the case of `tcpMIB`, the number 50).

Last Updated The date and time that the module was last revised.

Organization The name of the organization that is managing the development of the module.

Contact Information The name, address, telephone number, and email address of the point person for this module.

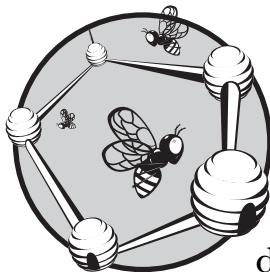
Description A description of the module.

Revision and Revision Description One Revision entry is placed for each revision of the module to show its history. Each entry has a description associated with it.

After the definition of the module itself, the objects in the module are described. For an example, see RFC 2012, which defines the SNMPv2 TCP MIB.

67

TCP/IP SIMPLE NETWORK MANAGEMENT PROTOCOL (SNMP) CONCEPTS AND OPERATION



The overall network management solution for TCP/IP networks is the Internet Standard Management Framework. In the previous two chapters, we have taken a look at the Framework as a whole, and also discussed the two components that define the management information transmitted between TCP/IP devices to accomplish network management. The third major part of the SNMP Framework is the actual *Simple Network Management Protocol (SNMP)*, which is responsible for moving management information between devices.

The core of the protocol consists of a set of *protocol operations* that allow management information to be exchanged between SNMP agents and managers. Having previously examined the generalities of SNMP and what management information base (MIB) objects are, we can now get down to the nitty gritty of how management information is actually communicated using SNMP.

In this chapter, I provide a detailed description of the operations performed by the SNMP protocol. I begin with a brief overview and history of the protocol. I then provide a general description of how SNMP operates and the two basic methods that devices use to communicate. I also describe SNMP's message classes and the basic operations performed in SNMP: basic request/response, table traversal, object modification, and notification. I conclude with a discussion of SNMP security issues and a summary of the security methods in each of the SNMP versions.

NOTE *The number and types of protocol operations in SNMP changed between SNMPv1 and SNMPv2. The operations defined in SNMPv2 have been carried forward into the newest version, SNMPv3. Most of the discussion focuses on SNMPv3 as the newest implementation, noting the differences between it and the original and still widely used SNMPv1.*

SNMP Protocol Overview

As explained in the previous chapters, the SNMP Framework is often described as being *information-oriented*. A specific decision was made in the design of the SNMP Framework to decouple the management information conveyed between SNMP agents and SNMP managers from the protocol used to carry that information. This provides numerous benefits to the technology as a whole, chief among them flexibility and modularity.

In this model, the operation of the management protocol is *not* defined in terms of specific commands made to check the status of a device or change how it operates. Instead, the protocol is defined in terms of management information variables called *objects*, and a communication protocol that allows these objects to be either examined or changed by a network administrator. I describe this concept thoroughly in the previous chapter.

The MIB and Structure of Management Information (SMI) spell out the rules for how MIB objects are created and described. These MIB objects describe the types of information that can be read from the device or written to the device. The last piece of the puzzle is the actual protocol that is responsible for these read- and write-type operations. This is SNMP itself, which I give the somewhat redundant name *SNMP protocol* to differentiate it from the SNMP Framework.

The result of the separation of the protocol from the management information it carries is that the protocol itself becomes significantly reduced in complexity. Instead of the SNMP protocol needing to define dozens or even hundreds of operations that specify particular network management functions, it needs to deal with only the transmission of MIB object information between SNMP agents and managers. The SNMP protocol itself does not pay attention to what is in these objects; it is merely concerned with moving them around. In some ways, the SNMP protocol is the only really simple part of SNMP!

Early Development of SNMPv1

The history of the SNMP protocol goes back to the predecessor of the SNMP Framework, the *Simple Gateway Monitoring Protocol (SGMP)*, which was defined in RFC 1028 in 1987. SGMP was designed as an interim solution for network management while

larger issues were being explored, as I explained in Chapter 65. However, this standard is where many of the basic design concepts underlying the modern SNMP protocol can be found.

The SGMP standard specified the basic design model used in SNMP by describing SGMP in terms of only retrievals of, or alterations to, variables stored on an Internet gateway (router). The standard also outlines the small number of protocol operations that are still the basis for SNMP's operation today.

The first version of the SNMP Framework, SNMPv1, included the first formal definition of the SNMP protocol in RFC 1067 (later revised by RFCs 1098 and 1157). This standard refines the protocol operations given in the SGMP document. It makes the operation of the SNMP protocol fit into the overall SNMP Framework, working with formally defined MIB objects.

SNMPv2 and the Division of SNMP into Protocol Operations and Transport Mappings

When SNMPv2 was created, the single document describing the SNMP protocol was split into two standards, to make the protocol more modular and better reflective of the layers used in internetworks:

Protocol Operations The first document of the pair describes the actual mechanics by which MIB objects are moved between SNMP devices using particular SNMP message types. In SNMPv3, it is RFC 3416, “Version 2 of the Protocol Operations for the Simple Network Management Protocol (SNMP).” When people talk about just “the SNMP standard,” this is the document they usually mean.

Transport Mappings The second document details how the SNMP protocol operations described in the first standard can be transported over a variety of different protocol suites. By using the correct mapping, SNMP operations can be carried out using lower-layer technologies other than the Internet Protocol (IP). This standard is represented in SNMPv3 by RFC 3417, “Transport Mappings for the Simple Network Management Protocol (SNMP).”

KEY CONCEPT The actual mechanism used to communicate management information between network management stations (NMSs) and managed devices is called the *Simple Network Management Protocol*, which may be called the *SNMP protocol* to differentiate it from the SNMP Framework. It consists of a number of *protocol operations* that describe the actual message exchanges that take place between devices, and a set of *transport mappings* that define how these messages are carried over various types of internetworks. The Internet Protocol (IP) is the most common transport mapping used for SNMP.

I discuss transport mappings in a little more detail in the description of SNMP messaging later in this chapter, but since the IP/User Datagram Protocol (UDP) method is by far the most common transport mechanism, there isn't a great deal to say about that aspect of the SNMP protocol.

SNMP Communication Methods

For SNMP to be useful in enabling the management of a network, it must allow a network administrator using a network management station (NMS) to easily check the status of SNMP agents in managed devices. In data communications, there are two general techniques that are used in a situation where one entity needs to be kept informed about activity or occurrences on another:

Poll-Driven Communication This term refers to the general technique of having the one who wants the information ask for it—just like someone might conduct a political poll. In SNMP, the NMS would poll SNMP agents for information. A common real-life example of polling is the model used by the regular mail service; every day you go to check your mailbox to see if you have any mail.

Interrupt-Driven Communication This term refers to having a device with information that another needs to know decide to send the information of its own volition. In SNMP, this would refer to an SNMP agent sending information to an NMS without being asked. This is the model used by that most famous of interrupters—the telephone.

Which communication method is better? The usual answer applies here: Neither is better or worse universally, which is why both options exist. Due to the obvious strengths and weaknesses of these models, the SNMP protocol is designed to use both. Polling is used for the periodic gathering of routine information, such as checking the usage statistics and general status of a device. Interrupts are used in the form of *traps* that a network administrator can set on a managed device. These traps cause an SNMP agent to interrupt an NMS when an event of importance occurs.

KEY CONCEPT SNMP uses two basic methods for exchanging management information. Routine communication uses a *poll-driven* technique, where the network management station (NMS) requests information from managed nodes. An *interrupt-driven* model is also supported. In situations where a managed device needs to tell an NMS about an occurrence immediately, it can send a *trap* message without waiting for a request from the NMS.

The focus of most of our look at SNMP in this chapter will concentrate on SNMP protocol operations: what messages are used, how they are structured, and how they are exchanged. In examining these messages, we will see the two main ways that information exchanges occur in SNMP—by polling and by interrupt—and also discover how the SNMP protocol works with MIB objects.

SNMP Protocol Operations

The actual communication of information in the SNMP protocol is performed in a manner similar to most other protocols, through the exchange of SNMP messages. These messages are sometimes called *protocol data units* or *PDUs*. This is a term you may have heard used in other protocols, and it is part of the formal definition of

data encapsulation in the OSI Reference Model, as explained in Chapter 5. A message is, of course, a data unit used by the protocol. SNMP messages all have *-PDU* at the ends of their names to identify them.

Some consider *protocol data unit* to be analogous to the military using oblong, metallic-headed, manually operated, fastener-acceleration device to refer to a hammer. To be fair though, strictly speaking, in SNMP, a PDU and a message are not exactly the same. The PDU is the higher-layer data that SNMP encapsulates, as described by the OSI model. The SNMP message format is a *wrapper* that encapsulates a PDU along with header fields, as I describe in the next chapter on SNMP messaging. However, the point of a message is to send a PDU, so the two are close enough, and the terms are sometimes used interchangeably.

SNMP PDU Classes

SNMPv1 originally defined six PDUs. The number of PDUs was expanded, and some changes were made to their names and uses in SNMPv2 and SNMPv3. The current SNMP Framework categorizes the PDUs into different *classes*. These classes describe both the function of each message type and the kind of communication they use to perform their task (polling versus interrupting).

KEY CONCEPT SNMP messages consist of a set of fields wrapped around a data element called a *protocol data unit* or *PDU*. In some cases, the terms message and PDU are used interchangeably, although they are technically not the same. SNMP PDUs are arranged into *classes* based on their function.

Table 67-1 lists the main SNMPv2/SNMPv3 PDU classes, describes them, and shows which PDUs are in each class in SNMPv2/SNMPv3. These classes were not used in SNMPv1, but for clarity, I also show which messages from SNMPv1 fall into the classes conceptually.

Table 67-1: SNMP PDU (Message) Classes

SNMPv3 PDU Class	Description	SNMPv1 PDUs	SNMPv2/SNMPv3 PDUs
Read	Messages that read management information from a managed device using a polling mechanism.	GetRequest-PDU, GetNextRequest-PDU GetBulkRequest-PDU	GetRequest-PDU, GetNextRequest-PDU, GetBulkRequest-PDU
Write	Messages that change management information on a managed device to affect the device's operation.	SetRequest-PDU	SetRequest-PDU
Response	Messages sent in response to a previous request.	GetResponse-PDU	Response-PDU
Notification	Messages used by a device to send an interrupt-like notification to an SNMP manager.	Trap-PDU	Trapv2-PDU, InformRequest-PDU

The GetBulkRequest-PDU and InformRequest-PDU messages are new in SNMPv2/v3. The GetResponse-PDU message was renamed Response-PDU (since it is a response and not a message that gets anything), and the new Trapv2-PDU replaces Trap-PDU.

There are three other special classes defined by the current SNMP Framework that are of less interest to us because they don't define actively used messages, but which I should mention for completeness. The Internal class contains a special message called Report-PDU defined for internal SNMP communication. The SNMP standards also provide two classes called Confirmed and Unconfirmed, which are used to categorize the messages listed in Table 67-1 based on whether or not they are acknowledged. The Report-PDU, Trapv2-PDU, and Response-PDU messages are considered Unconfirmed, and the rest are Confirmed.

Now we will look at how the major message types in the four main classes are used. Note that in general terms, all protocol exchanges in SNMP are described in terms of one SNMP entity sending messages to another. Most commonly, the entity sending requests is an SNMP manager, and the one responding is an SNMP agent, except for traps, which are sent by agents. For greater clarity, I try to use these more specific terms (*manager* or *agent*) when possible, rather than just *entity*.

Basic Request/Response Information Poll Using GetRequest and (Get)Response Messages

The obvious place to begin our detailed look at SNMP protocol operations is with the simplest type of information exchange. This would be a simple *poll* operation to read one or more management information variables, used by one SNMP entity (typically an SNMP manager) to request or read information from another entity (normally an SNMP agent on a managed device). SNMP implements this as a simple, two-message request/response protocol exchange, similar to the request/reply processes found in so many TCP/IP protocols.

This information request process typically begins with the user of an application wanting to check the status of a device or look at information about it. As we've seen, all this information is stored on the device in the form of MIB objects. The communication, therefore, takes the form of a request for particular MIB objects and a reply from the device containing those objects' values. In simplified form, the steps in the process are as follows (see Figure 67-1):

1. **SNMP Manager Creates GetRequest-PDU** Based on the information required by the application and user, the SNMP software on the NMS creates a GetRequest-PDU message. It contains the names of the MIB objects whose values the application wants to retrieve.
2. **SNMP Manager Sends GetRequest-PDU** The SNMP manager sends the PDU to the device that is being polled.
3. **SNMP Agent Receives and Processes GetRequest-PDU** The SNMP agent receives and processes the request. It looks at the list of MIB object names contained in the message and checks to see if they are valid (ones the agent actually implements). It looks up the value of each variable that was correctly specified.
4. **SNMP Agent Creates Response-PDU** The agent creates a Response-PDU to send back to the SNMP manager. This message contains the values of the MIB objects requested and/or error codes to indicate any problems with the request, such as an invalid object name.

5. **SNMP Agent Sends Response-PDU** The agent sends the response back to the SNMP manager.
6. **SNMP Manager Processes Response-PDU** The manager processes the information in the Response-PDU received from the agent.

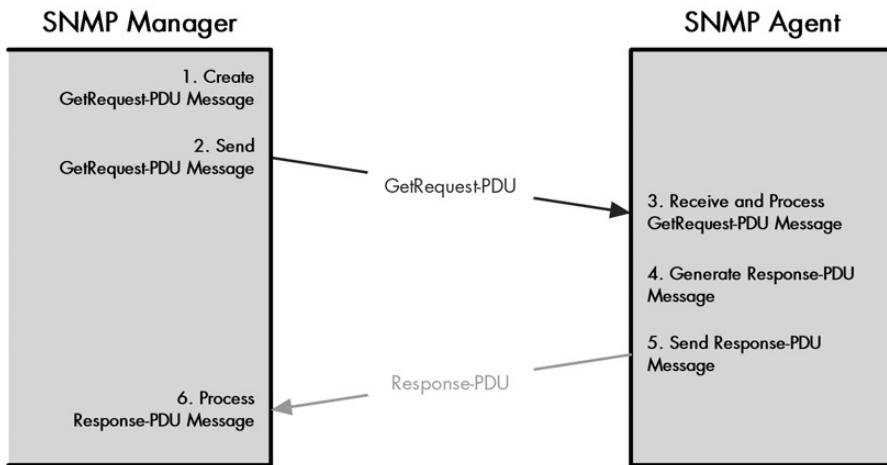


Figure 67-1: SNMP information poll process The basic SNMP information polling process involves a simple exchange of a GetRequest-PDU sent by an SNMP manager and a Response-PDU returned by an SNMP agent.

KEY CONCEPT The most basic type of communication in SNMP is an *information poll*, which allows an NMS to read one or more MIB objects from a managed node using a simple request/reply message exchange.

The Response-PDU message is called GetResponse-PDU in SNMPv1. Presumably, this name was chosen based on the fact that it was a response to a get operation, to make the names GetRequest-PDU and GetResponse-PDU somewhat symmetric. The problem is that this name is confusing, for two reasons. First, it sounds to some people like the purpose of the PDU is to “get a response.” Second, the GetResponse-PDU was also defined as the response message for operations other than get operations, including the reply message for SetRequest-PDU. Having a GetResponse message be sent in reply to a SetRequest message is disconcerting. The new name is more generic and avoids these problems.

Table Traversal Using GetNextRequest and GetBulkRequest Messages

The GetRequest-PDU message is used by applications to request values for regular, single variables in an SNMP managed object’s MIB. As I mentioned in Chapter 66, however, the SMI also allows an MIB to contain *tabular data*.

MIB tables are a useful way for a device to store and organize a set of related data items. It would be far from ideal to try to structure these items just as collections of regular objects. For example, a device may have multiple IP addresses. It would

be inefficient to define one MIB object called ipAddr1, another called ipAddr2, and so on to store IP address information. Instead, an object called ipAddrTable is defined in the original SNMPv1 MIB, which specifies a table containing one or more entries called ipAddrEntry. Each entry contains the IP address and subnet mask for one of the interfaces of the device.

SNMPv1 Table Traversal Using GetNextRequest

There needs to be a way to let an SNMP manager read the contents of these tables from a device. This can be done using the regular GetRequest-PDU message, by specifying each entry in the table, one after the other. However, this is somewhat crude, and it leaves a problem: the SNMP manager may not know how many entries are in the table, and therefore, how many entries it should request.

The problem of *table traversal* was addressed in SNMPv1 through the creation of a new message type called GetNextRequest-PDU. You can think of this as a relative of the regular GetRequest-PDU. The GetNextRequest-PDU contains the name of a tabular variable, as well as a particular entry in the table. The device receiving the GetNextRequest-PDU uses this to look up the next value in the table and return it in a GetResponse-PDU message.

The actual protocol exchange is about the same as that described in the previous section: a request is sent by the SNMP manager, and a reply is returned by the SNMP agent. The difference is that instead of the SNMP agent returning the value for the variable specified, it returns the value of the *next* variable in the table. This is then used as the value for the next request, and so on, until the last entry in the table is reached. Once this happens and a GetNextRequest-PDU is sent that contains this last entry, the responding device indicates this by returning the MIB object that conceptually follows the table in the implementation of the MIB. This signals to the SNMP manager that the table has been fully traversed.

KEY CONCEPT The SNMP GetNextRequest-PDU message allows an NMS to request a series of consecutive variables in an MIB. This is most commonly used to allow tabular data to be more easily retrieved, without requiring that each variable in the table be individually specified.

SNMPv2/v3 Table Traversal Using GetBulkRequest

The GetNextRequest-PDU message is functional, but while it is more elegant than using regular GetRequest-PDU messages, it is not any more efficient—each entry in the table must still be requested one at a time. This means that retrieving the information in a table takes a long time and also results in a great deal of traffic being generated, due to the number of requests and replies that must be sent.

To make table traversal easier and more conservative in its use of network resources, SNMPv2 introduced a new message type called GetBulkRequest-PDU. You can probably surmise the idea here from the name. Instead of specifying a particular MIB object to get or to get next, a GetBulkRequest-PDU allows an SNMP manager to send a single request that results in a number of entries in a table being returned in a Response-PDU message.

The GetBulkRequest-PDU is designed to allow both regular variables and tables to be retrieved in a single request. The PDU includes a list of objects, just as

in a GetRequest-PDU or GetNextRequest-PDU. The list is organized so that regular objects appear first and table objects come afterwards. Two special parameters are included in the request:

Non Repeaters Specifies the number of nonrepeating, regular objects to be retrieved. This is the number of regular objects at the start of the object list.

Max Repetitions Specifies the number of iterations, or entries, to read for the remaining tabular objects.

For example, suppose an SNMP manager wanted to request four regular variables and three entries from a table. The GetNextRequest-PDU would contain five MIB object specifications, with the table last. The Non Repeaters field would be set to 4, and the Max Repetitions field set to 3.

KEY CONCEPT To improve the efficiency of table traversal, SNMPv2 introduced the GetBulkRequest-PDU message, which allows an NMS to request a sequence of MIB objects from a table using a single request to a managed node.

The original method of traversing tables using GetRequest-PDU and GetNextRequest-PDU from SNMPv1 was retained in SNMPv2 and SNMPv3 when they were developed. However, the introduction of the more efficient GetBulkRequest-PDU means that GetNextRequest-PDU is not as important as it was in SNMPv1. Bear in mind, however, that using GetBulkRequest-PDU does require that the requesting entity know how many entries to ask for. So, some trial and error, or multiple requests, may be required to get a whole table if the number of entries is not known.

Object Modification Using SetRequest Messages

The GetRequest-PDU, GetNextRequest-PDU, and GetBulkRequest-PDU messages are the three members of the SNMP Read class of PDUs—they are used to let an SNMP manager read MIB objects from an SNMP agent. The opposite function is represented by the SNMP Write class, which contains a single member: the SNMP SetRequest-PDU message.

The use of this PDU is fairly obvious; where one of the three Get PDUs specifies a variable whose value is to be retrieved, the SetRequest-PDU message contains a specification for variables whose values are to be modified by the network administrator. Remember that SNMP does not include specific commands to let a network administrator control a managed device. This is the *control* method, which works by setting variables that affect the operation of the managed device.

The set process is the complement of the get process, using the same basic idea, but a reversal in how the object values travel and what is done with them. The process follows these steps (see Figure 67-2):

1. **SNMP Manager Creates SetRequest-PDU** Based on the information changes specified by the user through the SNMP application, the SNMP software on the

NMS creates a SetRequest-PDU message. It contains a set of MIB object names and the values to which they are to be set.

2. **SNMP Manager Sends SetRequest-PDU** The SNMP manager sends the PDU to the device being controlled.
3. **SNMP Agent Receives and Processes SetRequest-PDU** The SNMP agent receives and processes the set request. It examines each object in the request, along with the value to which the object is to be set, and determines if the request should or should not be honored.
4. **SNMP Agent Makes Changes and Creates Response-PDU** Assuming that the information in the request was correct (and any security provisions have been satisfied), the SNMP agent makes changes to its internal variables. The agent creates a Response-PDU to send back to the SNMP manager, which either indicates that the request succeeded or contains error codes to indicate any problems with the request found during processing.
5. **SNMP Agent Sends Response-PDU** The agent sends the response back to the SNMP manager.
6. **SNMP Manager Processes Response-PDU** The manager processes the information in the Response-PDU to see the results of the set.

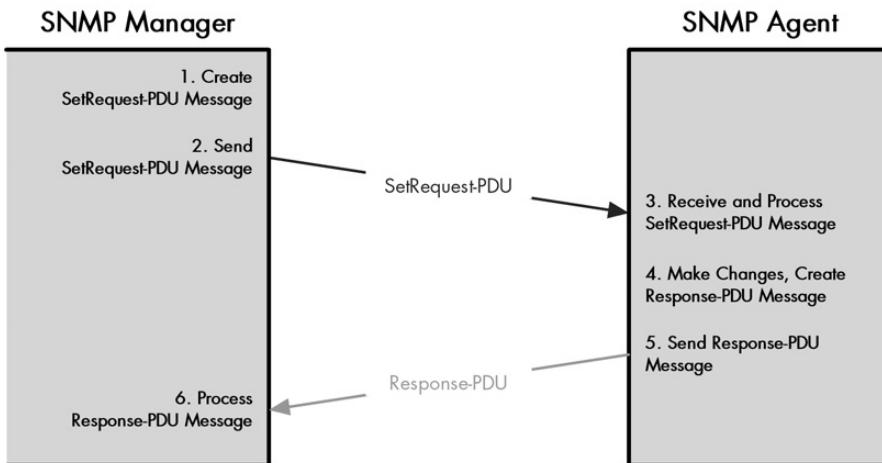


Figure 67-2: SNMP object modification process The communication process for setting a MIB object value is very similar to that used for reading one. The main difference is that the object values are sent from the SNMP manager to the SNMP agent, carried in the SetRequest-PDU message.

Obviously, telling a device to change a variable's value is a more significant request than just asking the device to read the value. For this reason, the managed device must very carefully analyze and verify the information in the request to ensure that the request is valid. The checks performed include the following:

- Verifying the names of the objects to be changed
- Verifying that the objects are allowed to be modified (based on their Access or Max-Access object characteristic, as described in Chapter 66)

- Checking the value included in the request to ensure that its type and size are valid for the object to be changed

This is also a place where general protocol security issues become more important, as I'll discuss near the end of this chapter.

KEY CONCEPT SNMP NMSs control the operation of managed devices by changing MIB objects on those devices. This is done using the SetRequest-PDU message, which specifies the objects to be modified and their values.

Information Notification Using Trap and InformRequest Messages

Earlier in this chapter, I introduced the two basic methods of communicating information between SNMP devices: using polls or interrupts. All of the message types and exchanges we have examined thus far in this section have been poll-driven. They consist of an SNMP manager making a specific request that results in action being taken, and a response being generated by an SNMP agent.

Polling is ideal for the exchange of routine information that needs to be gathered on a regular basis. For example, the regular get requests could be used to verify the settings on a device, examine error counts over a period of time, or check its uptime or use statistics. And, obviously, polling is the only real method for performing a set operation, where data is changed.

But polling is not well suited for important information that needs to be communicated quickly. The reason is that poll-driven communication is always initiated by the recipient of the information: the SNMP manager. If something significant occurs on a managed device that the manager wasn't expecting, the manager won't find out about it unless it specifically asks to see the variable that has changed. This means that important variables would need to be checked all the time by the SNMP manager, which is highly inefficient.

In the real world, using polling to implement situations where critical information needs to be sent would be like having the emergency response service in your town call everyone every hour to find out if they needed an ambulance or fire truck. Similarly, in SNMP, a mechanism was needed to let an SNMP agent initiate the communication of information. This capability was originally made part of the SNMPv1 protocol through the inclusion of the Trap-PDU message type.

In computer science, a *trap* is simply a set of conditions that a device monitors continuously. If the appropriate conditions occur, the trap is *triggered* and causes some sort of action to be taken. In SNMP, traps are programmed into SNMP agents, and when they are triggered, an SNMP Trap-PDU message is sent to an SNMP manager to inform it of the occurrence. Examples of traps in the SNMPv1 specification include ones that trigger in the event of a communication link failure, restart of the device, or an authentication problem.

Use of SNMP Trap and Trapv2 Messages

The communication in the case of a trap is trivial. The SNMP agent sends the trap, and the SNMP manager is thereby considered informed of what happened. That's

pretty much it. These are Unconfirmed messages, and no reply is made back to the SNMP agent. The triggering of the trap may lead the network administrator to take follow-up action at the device that sent the trap.

The designer of a particular MIB must determine which traps to create for a particular group of objects. The implementation must specify the conditions under which the traps will trigger and also the destination to which the Trap-PDU message will be sent when this occurs. In SNMPv2, the trap notification message was retained in the form of the Trapv2-PDU message.

Use of the SNMPv2 InformRequest Message

SNMPv2 also incorporates a second notification message type: the InformRequest-PDU message. This type of message is not the same as a trap, but it is related to traps for two reasons: Both message types are used to communicate information without the recipient initiating the process, and the two messages are sometimes used in conjunction.

The purpose of the InformRequest-PDU is actually to facilitate the communication of information between NMSs. The SNMP manager on one NMS can choose to inform another of some piece of information by sending an InformRequest-PDU to that other SNMP manager. The receiving manager then replies back with a Response-PDU to the one that sent the InformRequest-PDU, confirming receipt of the inform message.

A common way that this message is used is to spread the news when a trap occurs. Suppose a device experiences a power failure, which results in a Trapv2-PDU being sent to NMS 1. The network administrator may want to set up NMS 1 so that receipt of particular traps causes the information in the trap to be forwarded to another NMS. The InformRequest-PDU would be used to carry that information from NMS 1 to, say, NMS 2.

KEY CONCEPT SNMP managed devices can inform an NMS of an important occurrence by sending it a Trap-PDU or Trapv2-PDU message. Network administrators determine the circumstances under which one of these messages should be transmitted. SNMPv2 adds to this capability the InformRequest-PDU message, which can be used to propagate information about an event between management stations.

SNMP Protocol Security Issues and Methods

In my description of the various SNMP versions in Chapter 65, it's possible that I may have been a bit harsh on those who worked on SNMP during the 1990s. The proliferation of many SNMP version 2 variants really was unfortunate, and not something we often see in the world of TCP/IP. However, now that we've seen the sort of work that SNMP does, the desire for security in the protocol would seem to be clear. Given that, and given the very low level of security in the initial SNMPv1 protocol, it's understandable to some extent why a conflict over security issues arose.

The need for security in SNMP is obvious because the MIB objects being communicated contain critical information about network devices. We don't want just anyone snooping into our network to find out our IP addresses, how long our machines have been running, whether our links are down, or pretty much anything

else. When it comes to object write operations using a SetRequest-PDU, the concerns are magnified even more, because we definitely don't want strangers being able to control or interfere with our managed devices by issuing bogus commands to change MIB objects that control device operation!

Problems with SNMPv1 Security

Unfortunately, the security incorporated into SNMPv1 was extremely limited. It really took the form of only one policy and one simple technology.

SNMP was created with the mindset that the MIB objects used in the protocol would be relatively *weak*. This means that the objects are designed so that any problems in working with them result in minimal damage. The policy of the designers of SNMP was that MIB objects that are normally read should not contain critical information, and objects that are written should not control critical functions.

So, a read-only MIB object containing a description of a machine is fine, but one containing the administrative password is not. Similarly, a read-write MIB object that controls when the computer next reboots is acceptable, but one that tells the object to reformat its hard disk is definitely not!

All the devices in an SNMP network managed by a particular set of NMSs are considered to be in a *community*. Each SNMPv1 message sent between members of the community is identified by a *community string* that appears in a field in the message header. This string is like a simple password. Any messages received with the wrong string will be rejected by the recipient.

These security features are better than nothing, but not much. The use of weak objects is comparable to a policy that says not to leave your car in front of the convenience store with the doors unlocked and the key in the ignition—it is basically saying, “Don’t ask for trouble.” This is wise, but it’s not a complete security solution.

The community strings protect against obvious tampering in the form of unauthorized messages. However, the strings are sent in plain text, and they can easily be discovered and then used to compromise the community. So, this is like locking your doors when parking your car—it protects against the casual thief but not a pro.

Of course, for some people, not leaving their car running and locking the doors when they park provide enough security, and SNMPv1’s security was also sufficient for some users of SNMP. But in newer, larger internetworks, especially ones spanning large distances or using public carriers, SNMPv1 wasn’t up to the task. This is why all that fun stuff occurred with SNMPv2.

SNMPv2/v3 Security Methods

During the evolution of SNMPv2 variants, and eventually the creation of SNMPv3, several new security models were created to improve SNMPv1’s security:

Party-Based Security Model Party-based security was the model for the original SNMPv2 standard, now called *SNMPv2p*. A logical entity called a *party* is defined for communication that specifies a particular authentication protocol and a privacy (encryption) protocol. The information is used to verify that a particular request is authentic, and to ensure that the sender and receiver agree on how to encrypt and decrypt data.

User-Based Security Model (USM) USM was developed in the SNMPv2u variant and used in SNMPv2* (SNMPv2 asterisk). It eventually was adopted in SNMPv3. The idea here is to move away from tying security to the machines and instead use more traditional security based on access rights of a user of a machine. A variety of authentication and encryption protocols can be used to ensure access rights are respected and to protect message privacy. The method relies on timestamps, clock synchronization, and other techniques to protect against certain types of attacks.

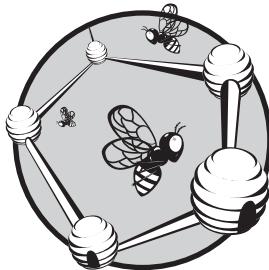
View-Based Access Control Model (VACM) VACM is part of SNMPv3, and it defines a method where more fine control can be placed on access to objects on a device. A *view* specifies a particular set of MIB objects that can be accessed by a particular group in a particular context. By controlling these views, an administrator can manage what information is accessed by whom.

Party-based security pretty much died with SNMPv2p. USM and VACM are part of SNMPv3 and provide enhanced security for those who need it. Again, it's interesting to note how many networks continue to use SNMPv1, security warts and all.

SNMPv3 took another important security-related step in redefining the SNMP architecture to seamlessly support multiple security models. This enables different implementations to choose the security model that is best for them. USM is the default model in SNMPv3.

68

SNMP PROTOCOL MESSAGING AND MESSAGE FORMATS



As we saw extensively in the previous chapter, the communication of management information is accomplished through the exchange of Simple Network Management Protocol (SNMP) messages that contain *protocol data units (PDUs)*. Like the messages of most TCP/IP protocols, these PDUs are designed to use a particular field format, and are created, addressed, and transported according to specific protocol rules. SNMP messages include fields that control the operation of the protocol, and they carry a payload of management information in the form of management information base (MIB) objects.

In this chapter, I describe the details of how messaging is accomplished in the SNMP protocol. I begin with a general discussion of issues related to message generation, addressing, and transport, and a description of how retransmission of messages is handled when necessary. I discuss the way fields are defined in SNMP messages and describe their general format, explaining the difference between the overall message and the PDU it contains. I then examine the message format used in all of the important SNMP versions, showing the structure of each message type and the fields used.

SNMP Protocol Message Generation

Message generation in SNMP is a bit different than the typical TCP/IP client/server model used for most other protocols. There aren't really any formal clients and servers in SNMP, since management information can be obtained from any device; it is distributed. Most of the message exchanges use a matched pair of request and reply messages. The network management station (NMS) usually *acts* as the client in these exchanges, sending a particular get or set request to an SNMP agent, which plays the role of server for the information it contains. However, SNMP agents aren't usually considered servers in the conventional sense.

SNMP traps deviate from the normal request/reply model of message generation entirely. When a trap is triggered, an SNMP agent sends a trap message to an NMS on its own, not in reaction to receiving a request. Since trap messages are unconfirmed, there is no reply. Note, however, that the SNMP versions 2 and 3 (SNMPv2 and SNMPv3) InformRequest-PDU message (discussed later in this chapter) is confirmed, and a response message is thus sent back to the NMS that generates it.

SNMP Transport Mappings

Once a message has been generated, it is sent using the protocols at the levels below the application layer where SNMP resides. As you saw in the overview of the SNMP protocol in the previous chapter, the current SNMP standard set separates the description of protocol operations and PDUs from the methods used to actually send them.

Starting with version 2, SNMP has defined several *transport mappings* that describe how SNMP PDUs can be sent over a variety of internetworking protocol suites, including TCP/IP, OSI, IPX/SPX (Novell), and AppleTalk. Many of the specific details of SNMP messaging depend on the transport mapping that is used in a particular implementation. SNMP is primarily used on TCP/IP internetworks, and TCP/IP is where our interest lies here, so the rest of this discussion will deal with transport issues when SNMP is used over the Internet Protocol (IP).

The standard IP transport mapping for SNMP calls for it to be carried using the User Datagram Protocol (UDP). This decision goes back to the initial implementation of SNMPv1 (before there were distinct transport mappings). UDP was likely chosen because it is more efficient for the simple request/reply messaging scheme SNMP uses. The many Transmission Control Protocol (TCP) features were not considered necessary and add overhead that SNMP's designers wanted to avoid. It is possible that TCP could be used to carry SNMP, defined as a different transport mapping, but I don't believe this is actually done.

Two well-known UDP port numbers are reserved for SNMP. The first is port 161, which is the general-purpose SNMP number. All devices that are set up to listen for SNMP requests—both agents and managers—listen on port 161. Each device receives any messages sent and replies back to the client, the SNMP entity that issued the request, which uses an ephemeral port number to identify the requesting process. The second UDP port number is 162, which is reserved for

SNMP traps. Having two numbers allows regular messages and traps to be kept separate. Normally, only NMSs would listen on port 162, since agents are not recipients of traps.

The use of UDP allows SNMP information communication to be streamlined, since there is no need to establish a TCP connection, and since message headers are shorter and processing time slightly reduced. But the use of UDP introduces a couple of issues that SNMP implementations must be concerned with, including message size and lost messages.

UDP Message Size Issues

The first issue is that of message length. SNMP PDUs can carry many MIB objects, which means they could potentially be rather large. However, UDP is limited in the size of message it can carry (where TCP is not). The standards specify that SNMP entities must accept messages up to at least 484 bytes in size. They also recommend that SNMP implementations be able to accept even larger messages, up to 1,472 bytes, which would correspond to the largest size message that can be encapsulated in an Ethernet frame (1,500 bytes, allowing 20 bytes for the IP header and 8 for the UDP header).

The use of the GetBulkRequest-PDU message type in SNMPv2 and SNMPv3 requires particular care, since it allows a single request to result in many MIB objects being sent back in a response. The Max Repetitions parameter must be chosen conservatively so the SNMP agent doesn't try to send an enormous message that won't fit.

Lost Transmission Issues

The second issue with UDP is the price we pay for its efficiency and simplicity: a lack of transport features. UDP doesn't guarantee data delivery or handle retransmissions, which means a request or reply could, in theory, be lost in transit. Only the device that initially sends a request can know if there was a problem with transport. It sends the request, and if it receives no reply, it knows either the request or response got lost. This puts the responsibility for retransmission on the device that sends the request message.

NMSs sending requests to SNMP agents generally use a timer to keep track of how much time has elapsed since a request was sent. If the response doesn't arrive within a certain time interval, the request is sent again. Because of how SNMP works, having a request be received more than once accidentally will normally not cause any problems (a property known as *idempotence*). The NMS does need to employ an algorithm to ensure that it does not generate too many retransmissions and clog the network (especially since congestion might be causing the loss of its messages in the first place).

Since traps are unconfirmed, there is no way for the intended recipient of a trap PDU to know if did not arrive, nor is there any way for the sender of the trap PDU to know. This is just a weakness in the protocol; the overall reliability of TCP/IP (and the underlying networks) ensures that these messages are not lost very often.

KEY CONCEPT SNMP is designed with a separately defined set of *protocol operations* and *transport mappings*, so it can be carried over many different internetworking technologies. The most common of these transport mechanisms is TCP/IP, where SNMP makes use of UDP running over IP, for its efficient and simple communication. The lack of reliability features in UDP means that requests must be tracked by the device sending them and retransmitted if no reply is received. The limited size of UDP messages restricts the amount of information that can be sent in any SNMP PDU.

SNMP General Message Format

To structure its messages for transport, SNMP uses a special field format, like most protocols. What's interesting about SNMP, however, is that its standards do not describe the SNMP message format using a simple list of fields the way most TCP/IP standards do. Instead, SNMP messages are defined using the same data description language (*Abstract Syntax Notation 1* or ASN.1) that is used to describe MIB objects.

The reason for this is that SNMP messages implement the various SNMP protocol operations with the ultimate goal of allowing MIB objects to be conveyed between SNMP entities. These MIB objects become fields within the messages to be sent. The MIB objects carried in SNMP messages are defined using ASN.1 as described in the Structure of Management Information (SMI) standard. So, it makes sense to define SNMP messages and all their fields using the same syntax.

Since all SNMP fields are defined like MIB objects, they are like objects in that they have certain characteristics. Specifically, each field has a name, and its contents are described using one of the standard SMI data types. So, unlike normal message formats where each field has just a name and a length, an SNMP message format field has a name and a *syntax*, such as Integer, Octet String, or IpAddress. The syntax of the field defines its length and how it is formatted and used.

Just as regular message formats use integers to represent specific values (for example, the numeric Opcode field in the DNS message header, which indicates the DNS message type), this can be done in SNMP using an enumerated integer type. An example would be the Error Status field, where a range of integer values represents different error conditions.

The decision to define SNMP messages using ASN.1 allows the message format description to be consistent with how the objects in the format are described, which is nice. Unfortunately, it means that the field formats are very hard to determine from the standards, because they are not described in one place. Instead, the overall message format is defined as a set of components, and those components contain subcomponents that may be defined elsewhere, and so on. In fact, the full message format isn't even defined in one standard; parts are spread across several standards. So, you can't look in one place and see the whole message format. Well, I should say that you can't if you use the standards, but you can if you look here.

To make things easier for you, I have converted these distributed syntax descriptions into the same tabular field formats I use throughout the rest of this book. I will begin here by describing the general format used for SNMP messages, and in the remainder of the chapter, explore the specific formats used in each version of SNMP.

The Difference Between SNMP Messages and PDUs

To understand SNMP messages, it is important that you first grasp the difference between SNMP messages and SNMP PDUs. We've seen in looking at SNMP protocol operations that the two terms are often used interchangeably. This is because each message carries one PDU, and the PDU is the most important part of the message.

However, strictly speaking, an SNMP PDU and an SNMP message are not exactly the same. The PDU is the actual piece of information that is being communicated between SNMP entities. It is carried within the SNMP message along with a number of header fields, which are used to carry identification and security information. Thus, conceptually, the SNMP message format can be considered to have two overall sections:

Message Header Contains fields used to control how the message is processed, including fields for implementing SNMP security.

Message Body (PDU) Contains the main portion of the message. In this case, the message body is the PDU being transmitted.

The overall SNMP message is sometimes called a *wrapper* for the PDU, since it encapsulates the PDU and precedes it with additional fields. The distinction between the PDU and the message format as a whole began as a formality in SNMPv1, but it became quite important in later versions. The reason is that it allows the fields used for basic protocol operations (which are in the PDU) to be kept separate from fields used to implement security features. In SNMPv2, the implementation of security became a very big deal indeed, so this flexibility was quite important.

General PDU Format

The fields in each PDU depend on the PDU type, but can be divided into the following general substructure:

PDU Control Fields A set of fields that describe the PDU and communicate information from one SNMP entity to another.

PDU Variable Bindings A set of descriptions of the MIB objects in the PDU. Each object is described as a *binding* of a name to a value.

Each PDU will follow this general structure, which is shown in Figure 68-1, differing only in the number of control fields and variable bindings and how they are used. In theory, each PDU could have a different message format using a distinct set of control fields, but in practice, most PDUs for a given SNMP version use the same control fields (with some exceptions).

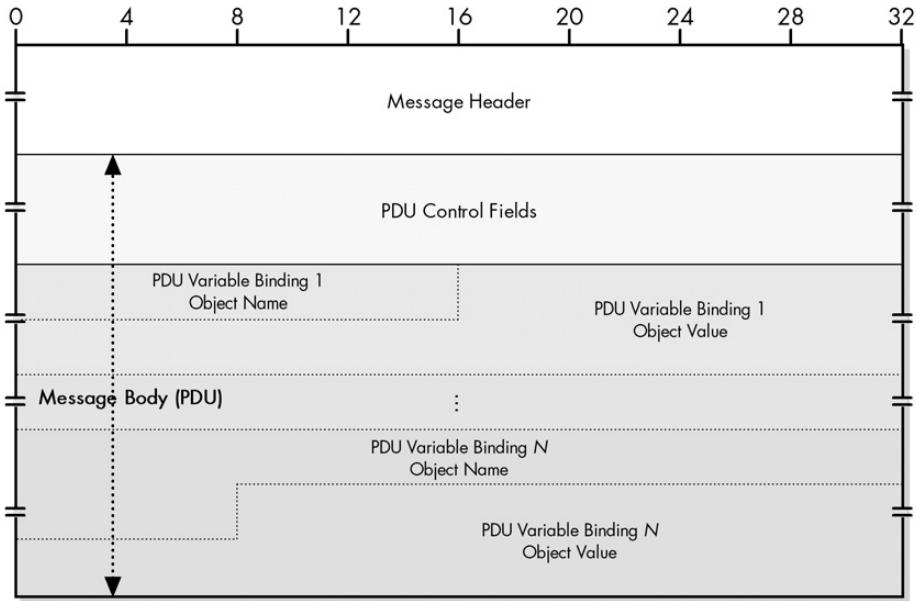


Figure 68-1: SNMP general message format

Each variable binding describes one MIB object. The binding consists of a pair of subfields, one specifying the name of the object in standard SNMP object identifier notation and one its value, formatted to match the object's SMI syntax. For example, if the object were of type Integer, the value field would be four bytes wide and contain a numeric integer value. Table 68-1 describes the subfield format for each PDU variable binding.

Table 68-1: SNMP Variable Binding Format

Subfield Name	Syntax	Size (Bytes)	Description
Object Name	Sequence of Integer	Variable	The numeric object identifier of the MIB object, specified as a sequence of integers. For example, the object sysLocation has the object identifier 1.3.6.1.2.1.1.6, so it would be specified as 1 3 6 1 2 1 1 6 using ASN.1.
Object Value	Variable	Variable	In any type of get request, this subfield is a placeholder; it is structured using the appropriate syntax for the object but has no value (since the get request is asking for that value!). In a set request (SetRequest-PDU) or in a reply message carrying requested data (GetResponse-PDU or Response-PDU), the value of the object is placed here.

KEY CONCEPT The general format of SNMP messages consists of a *message header* and a *message body*. The body of the message is also called the *protocol data unit*, or *PDU*, and contains a set of *PDU control fields* and a number of *variable bindings*. Each variable binding describes one MIB object and consists of the object's name and value.

SNMP Version 1 (SNMPv1) Message Format

The SNMP general message format was first used to define the format of messages in the original SNMP protocol, SNMPv1. This first version of SNMP is probably best known for its relative simplicity compared to the versions that followed it. This is reflected in its message format, which is quite straightforward.

SNMPv1 General Message Format

The general message format in SNMPv1 is a wrapper consisting of a small header and an encapsulated PDU. Not very many header fields were needed in SNMPv1 because the community-based security method in SNMPv1 is very rudimentary. The overall format for SNMPv1 messages is described in Table 68-2 and illustrated in Figure 68-2.

Table 68-2: SNMP Version 1 (SNMPv1) General Message Format

Field Name	Syntax	Size (Bytes)	Description
Version	Integer	4	Version Number: Describes the SNMP version number of this message; used for ensuring compatibility between versions. For SNMPv1, this value is actually 0, not 1.
Community	Octet String	Variable	Community String: Identifies the SNMP community in which the sender and recipient of this message are located. This is used to implement the simple SNMP community-based security mechanism, described in the previous chapter.
PDU	—	Variable	Protocol Data Unit: The PDU being communicated as the body of the message.

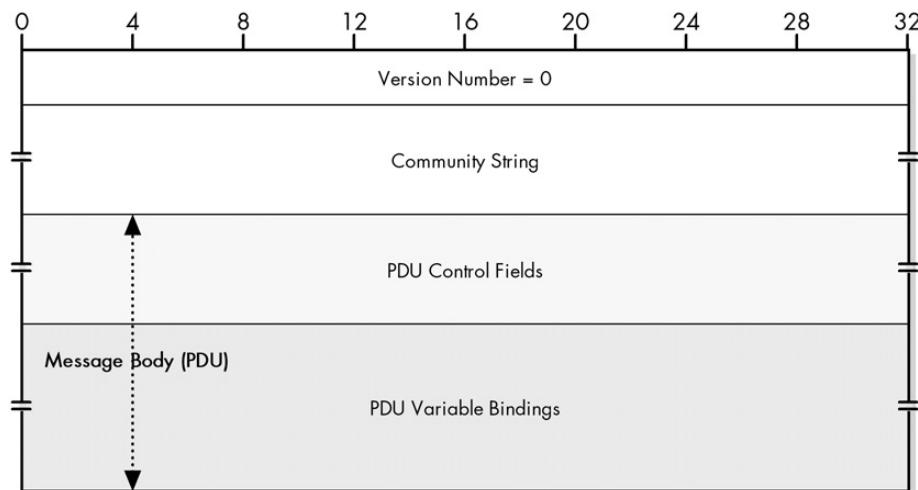


Figure 68-2: SNMPv1 general message format

SNMPv1 PDU Formats

All of the PDUs in SNMPv1 have the same format, with one exception: Trap-PDU. The exact semantics of each field in the PDU depend on the particular message. For example, the ErrorStatus field only has meaning in a reply and not a request, and object values are used differently in requests and replies as well.

Table 68-3 shows the common format for most of the SNMPv1 PDUs: GetRequest-PDU, GetNextRequest-PDU, SetRequest-PDU, and GetResponse-PDU.

Table 68-3: SNMPv1 Common PDU Format

Field Name	Syntax	Size (Bytes)	Description
PDU Type	Integer (Enumerated)	4	PDU Type: An integer value that indicates the PDU type: 0 = GetRequest-PDU 1 = GetNextRequest-PDU 2 = GetNextRequest-PDU 3 = SetRequest-PDU
Request ID	Integer	4	Request Identifier: A number used to match requests with replies. It is generated by the device that sends a request and copied into this field in a GetResponse-PDU by the responding SNMP entity.
Error Status	Integer (Enumerated)	4	Error Status: An integer value that is used in a GetResponse-PDU to tell the requesting SNMP entity the result of its request. A value of zero indicates that no error occurred; the other values indicate what sort of error happened, as listed in Table 68-4.
Error Index	Integer	4	Error Index: When Error Status is nonzero, this field contains a pointer that specifies which object generated the error. Always zero in a request.
Variable Bindings	Variable	Variable	Variable Bindings: A set of name/value pairs identifying the MIB objects in the PDU, and in the case of a SetRequest-PDU or GetResponse-PDU, containing their values. See the discussion of the general SNMP general PDU format earlier in this chapter for more on these bindings.

Table 68-4: SNMPv1 Error Status Field Values

Error Status Value	Error Code	Description
0	noError	No error occurred. This code is also used in all request PDUs, since they have no error status to report.
1	tooBig	The size of the GetResponse-PDU would be too large to transport.
2	noSuchName	The name of a requested object was not found.
3	badValue	A value in the request didn't match the structure that the recipient of the request had for the object. For example, an object in the request was specified with an incorrect length or type.
4	readOnly	An attempt was made to set a variable that has an Access value indicating that it is read-only.
5	genErr	An error other than one of the preceding four specific types occurred.

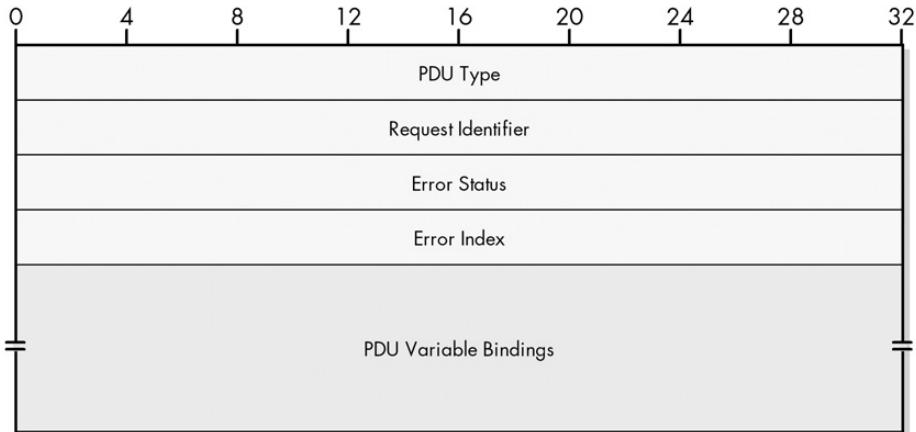


Figure 68-3: SNMPv1 common PDU format

Table 68-5 describes the special format for the SNMPv1 Trap-PDU, and it is illustrated in Figure 68-4.

Table 68-5: SNMPv1 Trap-PDU Format

Field Name	Syntax	Size (Bytes)	Description
PDU Type	Integer (Enumerated)	4	PDU Type: An integer value that indicates the PDU type, which is 4 for a Trap-PDU message.
Enterprise	Sequence of Integer	Variable	Enterprise: An object identifier for a group, which indicates the type of object that generated the trap.
Agent Addr	NetworkAddress	4	Agent Address: The IP address of the SNMP agent that generated the trap. This is also in the IP header at lower levels but inclusion in the SNMP message format allows for easier trap logging within SNMP. Also, in the case of a multihomed host, this specifies the preferred address.
Generic Trap	Integer (Enumerated)	4	Generic Trap Code: A code value specifying one of a number of predefined generic trap types.
Specific Trap	Integer	4	Specific Trap Code: A code value indicating an implementation-specific trap type.
Time Stamp	TimeTicks	4	Time Stamp: The amount of time since the SNMP entity sending this message last initialized or reinitialized. Used to time stamp traps for logging purposes.
Variable Bindings	Variable	Variable	Variable Bindings: A set of name/value pairs identifying the MIB objects in the PDU. See the discussion of the general SNMP general PDU format earlier in this chapter for more on these bindings.

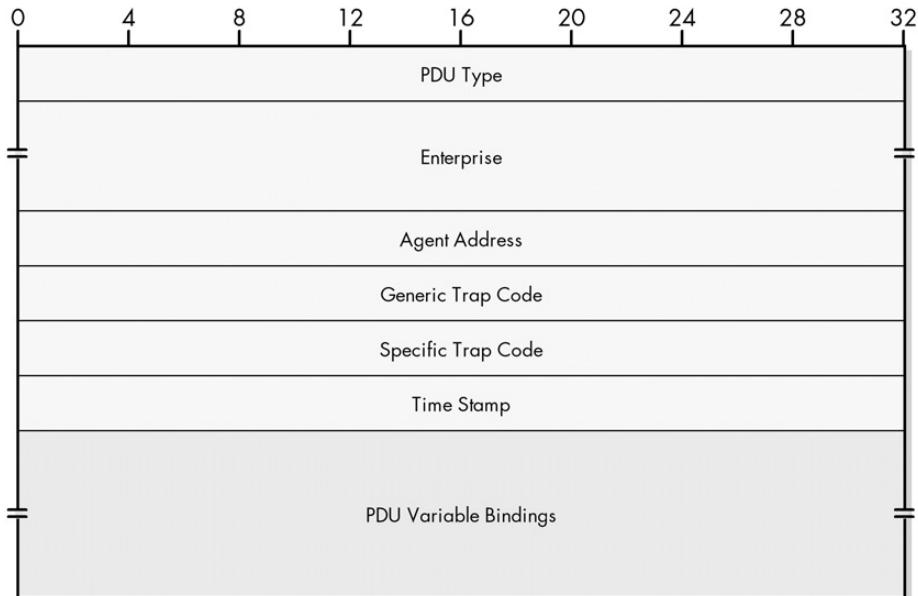


Figure 68-4: SNMPv1 Trap-PDU format

SNMP Version 2 (SNMPv2) Message Formats

After SNMPv1 had been in use for several years, certain issues with it were noticed and areas for improvement identified. This led to the development of the original SNMPv2, which was intended to enhance SNMPv1 in many areas, including MIB object definitions, protocol operations, and security. This last area, security, led to the proliferation of SNMPv2 version variants that I described in Chapter 65.

Since there are several different SNMPv2s, there are also several message formats for SNMPv2. This is confusing, but it would be even worse without the modular nature of SNMP messages coming to the rescue. The protocol operations in SNMPv2 were changed from SNMPv1, which necessitated some modifications to the format of SNMPv2 PDUs. However, the protocol operations are the same for all the SNMPv2 variations. The differences between SNMPv2 variants are in the areas of security implementation. Thus, the result of this is that the PDU format is the same for all the SNMPv2 types, while the overall message format differs for each variant. (This is why the distinction between a PDU and a message is not just an academic one!)

During the SNMPv2 divergence, four variations were defined: the original SNMPv2 (SNMPv2p), community-based SNMPv2 (SNMPv2c), user-based SNMPv2 (SNMPv2u), and SNMPv2 asterisk (SNMPv2*). Of these, the first three were documented in sets of SNMP RFC standards, as discussed in Chapter 65; the fourth was not. The structure of the overall message format for each variant is discussed in an administrative or security standard for the variation in question, which makes reference to the shared SNMPv2 standard for the PDU format (RFC 1905).

SNMP Version 2 (SNMPv2p) Message Format

The party-based security model is quite complex, but the basic messaging in this version is described through the definition of a *management communication*, which describes the source and destination party and makes reference to a *context* for the communication. The overall message format is described in detail in RFC 1445.

This information is summarized in Table 68-6 and shown graphically in Figure 68-5.

Table 68-6: SNMP Version 2 (SNMPv2p) General Message Format

Field Name	Syntax	Size (Bytes)	Description
Version	Integer	4	Version Number: Describes the SNMP version number of this message; used for ensuring compatibility between versions. For SNMPv2p, this value is 2.
Dst Party	Sequence of Integer	Variable	Destination Party: An object identifier that specifies the party that is the intended recipient of the message.
Src Party	Sequence of Integer	Variable	Source Party: An object identifier that specifies the party that is the sender of the message.
Context	Sequence of Integer	Variable	Context: Defines a set of MIB object resources that is accessible by a particular entity.
PDU	—	Variable	PDU: The protocol data unit of the message.

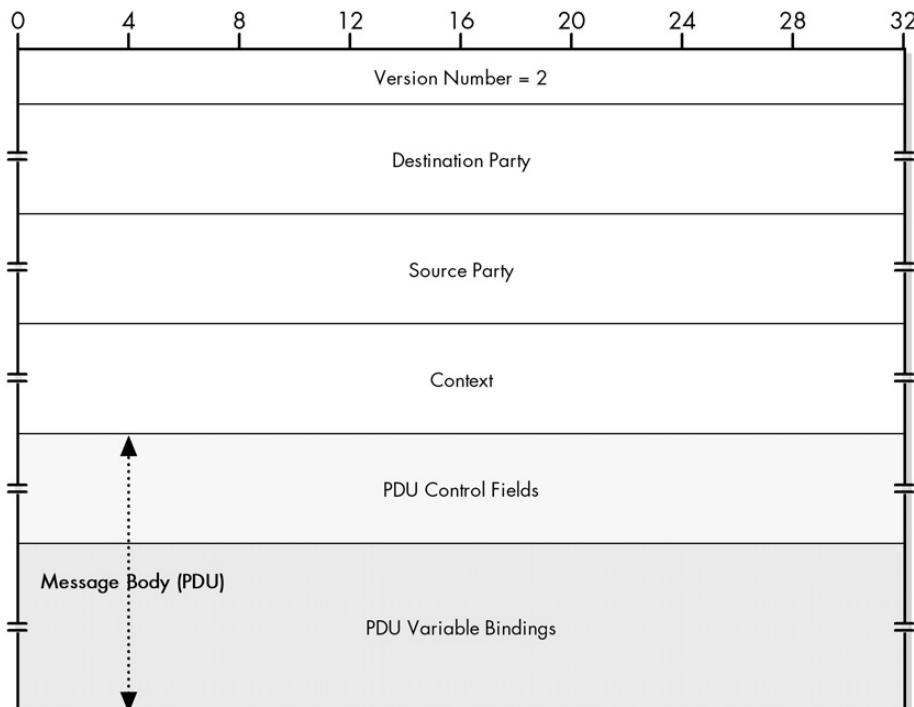


Figure 68-5: SNMPv2p general message format

Community-Based SNMP Version 2 (SNMPv2c) Message Format

The community-based version of SNMPv2 was intended to keep the new protocol enhancements introduced by SNMPv2p but go back to the simple SNMPv1 security model. As such, the defining document for SNMPv2c, RFC 1901, specifies that its overall message format is the same as that of SNMPv1, except that the version number is changed. This is shown in Table 68-7 and illustrated in Figure 68-6.

Table 68-7: Community-Based SNMP Version 2 (SNMPv2c) General Message Format

Field Name	Syntax	Size (Bytes)	Description
Version	Integer	4	Version Number: Describes the SNMP version number of this message; used for ensuring compatibility between versions. For SNMPv2c, this value is 1.
Community	Octet String	Variable	Community String: Identifies the SNMP community in which the sender and recipient of this message are located.
PDU	—	Variable	Protocol Data Unit: The PDU being communicated as the body of the message.

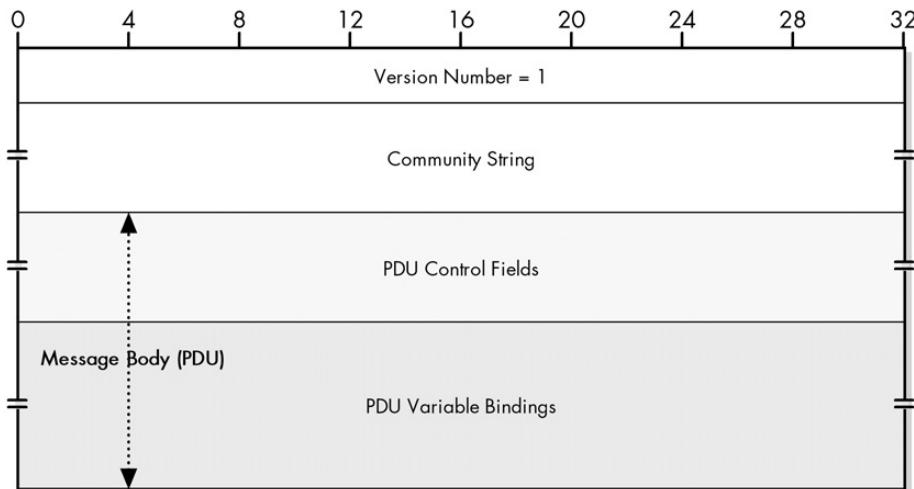


Figure 68-6: SNMPv2c general message format

User-Based SNMP Version 2 (SNMPv2u) Message Format

The user-based version of SNMPv2 was defined as an optional security model at the time that SNMPv2c was standardized. RFC 1910 defines the user-based security model and the message format described in Table 68-8 and illustrated in Figure 68-7.

Table 68-8: User-Based SNMP Version 2 (SNMPv2u) General Message Format

Field Name	Syntax	Size (Bytes)	Description
Version	Integer	4	Version Number: Describes the SNMP version number of this message; used for ensuring compatibility between versions. For SNMPv2u, this value is 2. Note that this is the same value as used for SNMPv2p.
Parameters	Octet String	Variable	Parameters: A string of parameters used to implement the user-based security model, which are briefly described in Table 68-9.
PDU	—	Variable	Protocol Data Unit: The PDU being communicated as the body of the message. This may be in either encrypted or unencrypted form.

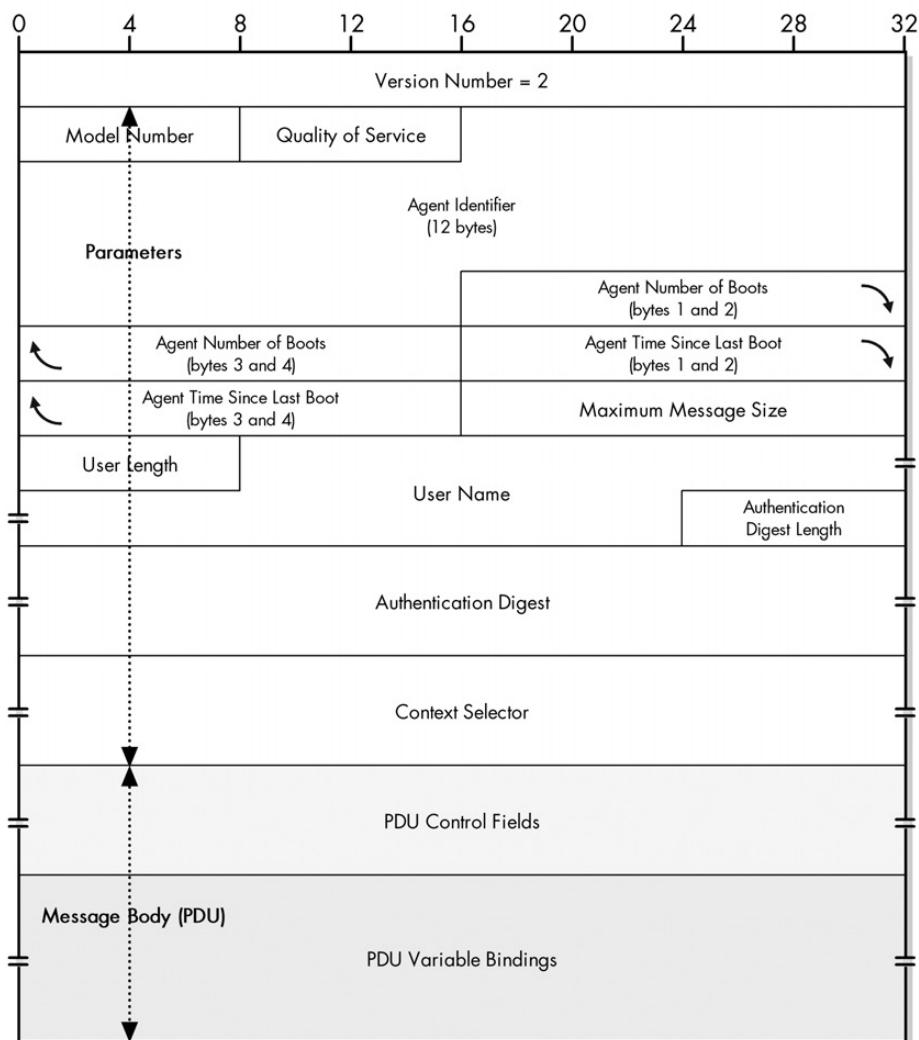


Figure 68-7: SNMPv2u general message format

Table 68-9: SNMPv2u Parameter Field Subfields

Subfield Name	Size (Bytes)	Description
Model	1	Model Number: Set to 1 to identify the user-based model.
QoS	1	Quality of Service: Indicates whether authentication and/or privacy (encryption) have been used and whether generation of a Report-PDU is allowed.
Agent ID	12	Agent Identifier: The identifier of the agent sending the message. Used to defeat replay attacks and certain other types of security attacks.
Agent Boots	4	Agent Number of Boots: The number of times the agent has been booted or rebooted since its Agent ID was set; used to defeat certain security attacks.
Agent Time	4	Agent Time Since Last Boot: The number of seconds since the last boot of this agent. Again, used to defeat replay and other security attacks.
Max Size	2	Maximum Message Size: The maximum size of message that the sender of this message can receive.
User Len	1	User Length: The length of the User Name field below.
User Name	Variable (1 to 16)	User Name: The name of the user on whose behalf the message is being sent.
Auth Len	1	Authentication Digest Length: The length of the Auth Digest field.
Auth Digest	Variable (0 to 255)	Authentication Digest: An authentication value used to verify the identity and genuineness of this message, when authentication is used.
Context Selector	Variable (0 to 40)	Context Selector: A string that is combined with the Agent ID to specify a particular context that contains the management information referenced by this message.

SNMPv2 PDU Formats

The format of protocol data units in SNMPv2 is described in RFC 1905, and it is similar to that of SNMPv1. The format for all PDUs in SNMPv2 is the same, except for the GetBulkRequest-PDU message. (Oddly, this includes the Trapv2-PDU message, even though the Trap-PDU message in SNMPv1 used a distinct format.)

Table 68-10 shows the common SNMPv2 PDU format. Table 68-11 contains a listing of the different values for the Error Status field and how they are interpreted. Figure 68-8 illustrates the SNMPv2 common PDU format.

Table 68-10: SNMPv2 Common PDU Format

Field Name	Syntax	Size (Bytes)	Description
PDU Type	Integer (Enumerated)	4	PDU Type: An integer value that indicates the PDU type: 0 = GetRequest-PDU 1 = GetNextRequest-PDU 2 = Response-PDU 3 = SetRequest-PDU 4 = Obsolete, not used (this was the old Trap-PDU in SNMPv1) 5 = GetBulkRequest-PDU (has its own format; see Table 68-12) 6 = InformRequest-PDU 7 = Trapv2-PDU 8 = Report-PDU

(continued)

Table 68-10: SNMPv2 Common PDU Format (continued)

Field Name	Syntax	Size (Bytes)	Description
Request ID	Integer	4	Request Identifier: A number used to match requests with replies. It is generated by the device that sends a request and copied into this field in a Response-PDU by the responding SNMP entity.
Error Status	Integer (Enumerated)	4	Error Status: An integer value that is used in a Response-PDU to tell the requesting SNMP entity the result of its request. A value of zero indicates that no error occurred; the other values indicate what sort of error happened (see Table 68-11).
Error Index	Integer	4	Error Index: When Error Status is nonzero, this field contains a pointer that specifies which object generated the error. Always zero in a request.
Variable Bindings	Variable	Variable	Variable Bindings: A set of name/value pairs identifying the MIB objects in the PDU, and in the case of messages other than requests, containing their values. See the discussion of the general SNMP general PDU format earlier in this chapter for more on these bindings.

NOTE The first six Error Status field values (0 to 5) are maintained as used in SNMPv1 for compatibility, but SNMPv2 adds many new error codes that provide more specific indication of the exact nature of an error in a request. The genErr code is still used only when none of the specific error types (either the old codes or the new ones) apply.

Table 68-11: SNMPv2 PDU Error Status Field Values

Error Status Value	Error Code	Description
0	noError	No error occurred. This code is also used in all request PDUs, since they have no error status to report.
1	tooBig	The size of the Response-PDU would be too large to transport.
2	noSuchName	The name of a requested object was not found.
3	badValue	A value in the request didn't match the structure that the recipient of the request had for the object. For example, an object in the request was specified with an incorrect length or type.
4	readOnly	An attempt was made to set a variable that has an Access value indicating that it is read-only.
5	genErr	An error occurred other than one indicated by a more specific error code in this table.
6	noAccess	Access was denied to the object for security reasons.
7	wrongType	The object type in a variable binding is incorrect for the object.
8	wrongLength	A variable binding specifies a length incorrect for the object.
9	wrongEncoding	A variable binding specifies an encoding incorrect for the object.
10	wrongValue	The value given in a variable binding is not possible for the object.
11	noCreation	A specified variable does not exist and cannot be created.
12	inconsistentValue	A variable binding specifies a value that could be held by the variable but cannot be assigned to it at this time.

(continued)

Table 68-11: SNMPv2 PDU Error Status Field Values (continued)

Error Status Value	Error Code	Description
13	resourceUnavailable	An attempt to set a variable required a resource that is not available.
14	commitFailed	An attempt to set a particular variable failed.
15	undoFailed	An attempt to set a particular variable as part of a group of variables failed, and the attempt to then undo the setting of other variables was not successful.
16	authorizationError	A problem occurred in authorization.
17	notWritable	The variable cannot be written or created.
18	inconsistentName	The name in a variable binding specifies a variable that does not exist.

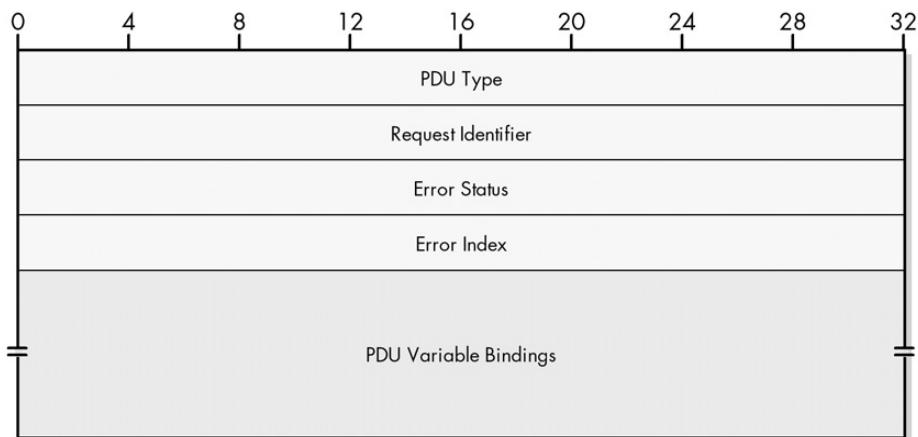


Figure 68-8: SNMPv2 common PDU format

The special format of the SNMPv2 GetBulkRequest-PDU message is shown in Table 68-12 and illustrated in Figure 68-9.

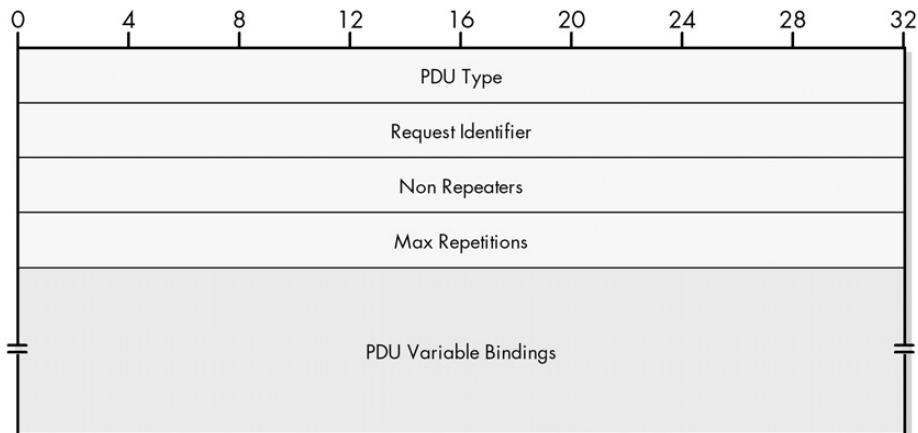


Figure 68-9: SNMPv2 GetBulkRequest-PDU format

Table 68-12: SNMPv2 GetBulkRequest-PDU Format

Field Name	Syntax	Size (Bytes)	Description
PDU Type	Integer (Enumerated)	4	PDU Type: An integer value that indicates the PDU type, which is 5 for a Get-BulkRequest-PDU message.
Request ID	Integer	4	Request Identifier: A number used to match requests with replies. It is generated by the device that sends a request and copied into this field in a Response-PDU by the responding SNMP entity.
Non Repeaters	Integer	4	Non Repeaters: Specifies the number of nonrepeating, regular objects at the start of the variable list in the request.
Max Repetitions	Integer	4	Maximum Repetitions: The number of iterations in the table to be read for the repeating objects that follow the nonrepeating objects.
Variable Bindings	Variable	Variable	Variable Bindings: A set of name/value pairs identifying the MIB objects in the PDU. See the discussion of the general SNMP general PDU format earlier in this chapter for more on these bindings.

Chapter 67 contains full details on how the Non Repeaters and Max Repetitions fields are used.

SNMP Version 3 (SNMPv3) Message Format

In the late 1990s, SNMPv3 was created to resolve the problems that occurred with the many different variations of SNMPv2. The SNMPv3 Framework adopts many components that were created in SNMPv2, including the SNMPv2 protocol operations, PDU types, and PDU format. The significant changes made in SNMPv3 include a more flexible way of defining security methods and parameters to allow the coexistence of multiple security techniques.

The general message format for SNMPv3 still follows the same idea of an overall message wrapper that contains a header and an encapsulated PDU, but it is further refined. The fields in the header have themselves been divided into those dealing with security and those that do not deal with security matters. The fields not related to security are common to all SNMPv3 implementations. The use of the security fields can be tailored by each SNMPv3 security model, and processed by the module in an SNMP entity that deals with security. This solution provides considerable flexibility while avoiding the problems that plagued SNMPv2.

The overall SNMPv3 message format is described in RFC 3412, which specifies its message processing and dispatching. Table 68-13 describes the SNMPv3 message format, and it is illustrated in Figure 68-10.

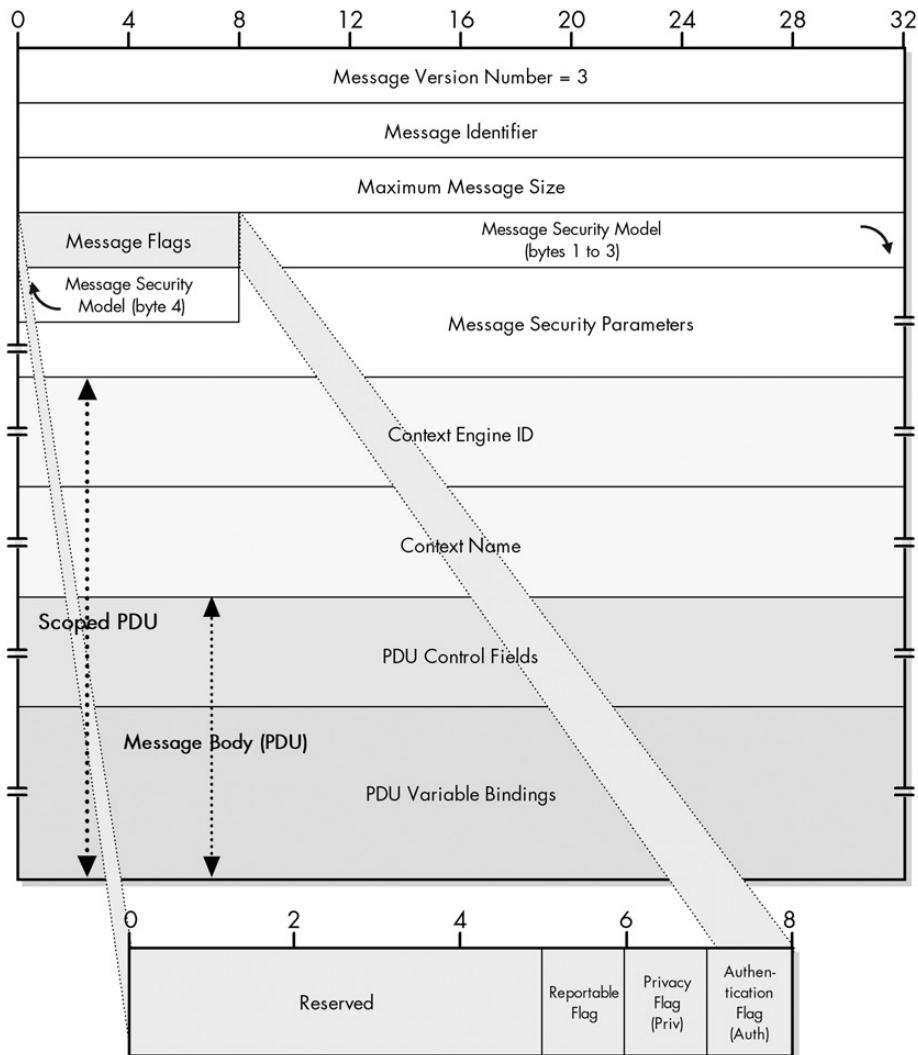


Figure 68-10: SNMPv3 general message format

Table 68-13: SNMP Version 3 (SNMPv3) General Message Format

Field Name	Syntax	Size (Bytes)	Description
Msg Version	Integer	4	Message Version Number: Describes the SNMP version number of this message; used for ensuring compatibility between versions. For SNMPv3, this value is 3.
Msg ID	Integer	4	Message Identifier: A number used to identify an SNMPv3 message and to match response messages to request messages. The use of this field is similar to that of the Request ID field in the SNMPv2 PDU format (see Table 68-10), but they are not identical. This field was created to allow matching at the message-processing level, regardless of the contents of the PDU, to protect against certain security attacks. Thus, Msg ID and Request ID are used independently.

(continued)

Table 68-13: SNMP Version 3 (SNMPv3) General Message Format (continued)

Field Name	Syntax	Size (Bytes)	Description
Msg Max Size	Integer	4	Maximum Message Size: The maximum size of message that the sender of this message can receive. Minimum value of this field is 484.
Msg Flags	Octet String	1	Message Flags: A set of flags that controls processing of the message. The current substructure of this field is shown in Table 68-14.
Msg Security Model	Integer	4	Message Security Model: An integer value indicating which security model was used for this message. For the user-based security model (the default in SNMPv3), this value is 3.
Msg Security Parameters	—	Variable	Message Security Parameters: A set of fields that contain parameters required to implement the particular security model used for this message. The contents of this field are specified in each document describing an SNMPv3 security model. For example, the parameters for the user-based model are in RFC 3414.
Scoped PDU	—	Variable	Scoped PDU: Contains the PDU to be transmitted, along with parameters that identify an SNMP context, which describes a set of management information accessible by a particular entity. The PDU is said to be <i>scoped</i> because it is applied within the scope of this context. (Yes, security stuff is confusing, sorry; it would take pages and pages to properly explain contexts; see RFC 3411.) The field may be encrypted or unencrypted depending on the value of <i>Priv Flag</i> . Its structure is shown in Table 68-15.

Table 68-14: SNMPv3 Msg Flags Subfields

Subfield Name	Size (Bits)	Description
Reserved	5	Reserved: Reserved for future use.
Reportable Flag	1	Reportable Flag: When set to 1, a device receiving this message must send back a Report-PDU whenever conditions arise where such a PDU should be generated.
Priv Flag	1	Privacy Flag: When set to 1, indicates that encryption was used to protect the privacy of the message. May not be set to 1 unless Auth Flag is also set to 1.
Auth Flag	1	Authentication Flag: When set to 1, indicates that authentication was used to protect the authenticity of this message.

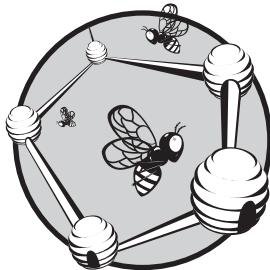
Table 68-15: SNMPv3 Scoped PDU Subfields

Subfield Name	Syntax	Size	Description
Context Engine ID	Octet String	Variable	Used to identify to which application the PDU will be sent for processing.
Context Name	Octet String	Variable	An object identifier specifying the particular context associated with this PDU.
PDU	—	Variable	The protocol data unit being transmitted.

Fortunately, SNMPv3 uses the protocol operations from SNMPv2, as described in RFC 3416, which is just an update of RFC 1904. Thus, the PDU formats for SNMPv3 are the same as those of SNMPv2 (see Tables 68-10 through 68-12 and Figures 68-8 and 68-9).

69

TCP/IP REMOTE NETWORK MONITORING (RMON)



We've seen in the preceding chapters of this part that the Simple Network Management Protocol (SNMP) defines both a framework and a specific protocol for exchanging network information on a TCP/IP internetwork. The general model used by SNMP is that of a network management station (NMS) that sends requests to SNMP agents running on managed devices. The SNMP agents may also initiate certain types of communication by sending *trap* messages to tell the NMS when particular events occur.

This model works well, which is why SNMP has become so popular. However, one fundamental limitation of the protocol and the model it uses is that it is oriented around the communication of network information from SNMP agents that are normally part of regular TCP/IP devices, such as hosts and routers. The amount of information gathered by these devices is usually somewhat limited, because obviously hosts and routers have real work to do—that is, doing the jobs of being hosts and routers. They can't devote themselves to network management tasks.

Thus, in situations where more information is needed about a network than is gathered by traditional devices, administrators often use special hardware units called *network analyzers*, *monitors*, or *probes*. These are dedicated pieces of equipment that are connected to a network and used strictly for the purpose of gathering statistics and watching for events of interest or concern to the administrator. It would obviously be very useful if these devices could use SNMP to allow the information they gather to be retrieved, and to let them generate traps when they notice something important. To enable this, the *Remote Network Monitoring (RMON)* specification was created.

RMON Standards

RMON is often called a protocol, and you will sometimes see SNMP and RMON referred to as the TCP/IP network management protocols. However, RMON really isn't a separate protocol at all—it defines no protocol operations. RMON is actually part of SNMP, and the RMON specification is simply a management information base (MIB) module that defines a particular set of MIB objects for use by network monitoring probes. Architecturally, it is just one of the many MIB modules that compose the SNMP Framework.

KEY CONCEPT SNMP *Remote Network Monitoring (RMON)* was created to enable the efficient management of networks using dedicated management devices such as network analyzers, monitors, or probes. RMON is often called a protocol, but it does not define any new protocol operations. It is actually an MIB module for SNMP that describes objects that permit advanced network management capabilities.

The first standard documenting RMON was RFC 1271, “Remote Network Monitoring Management Information Base,” published in 1991. RFC 1271 was replaced by RFC 1757 in 1995, which made a couple of changes to the specification. RFC 2819, published in May 2000, updates RMON to use the new Structure of Management Information version 2 (SMIV2) specification that is part of SNMPv2 but is functionally the same as RFC 1757.

RMON MIB Hierarchy and Object Groups

Since RMON is a MIB module, it consists almost entirely of descriptions for MIB objects, each with the standard characteristics belonging to all such objects. All the objects within RMON are arranged into the SNMP object name hierarchy within the *rmon* group, which is group number 16 within the SNMP mib (mib-2) object tree, 1.3.6.1.2.1. So, all RMON objects have identifiers starting with 1.3.6.1.2.1.16. This single RMON group is broken down into several lower-level groups that provide more structure for the RMON objects defined by the specification. Figure 69-1 shows this structure.

Table 69-1 describes each of the RMON groups, showing its name, group code (which is used as the prefix for object descriptors in the group), and RMON group number and SNMP object hierarchy identifier.

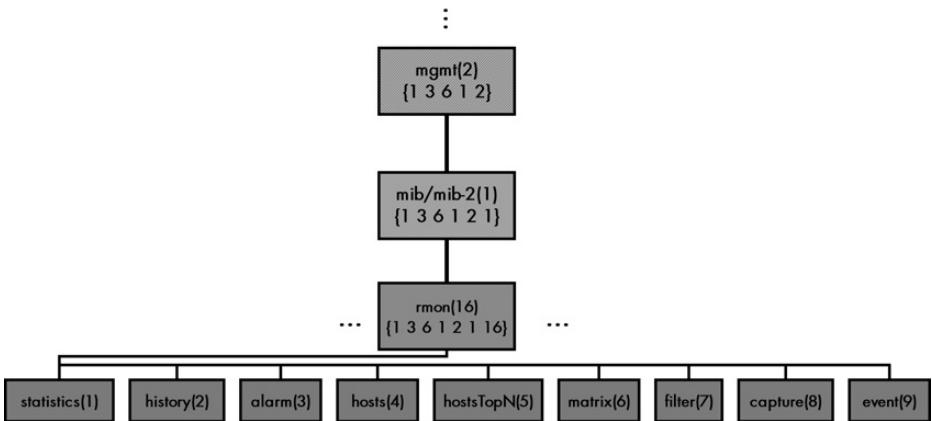


Figure 69-1: SNMP Remote Network Monitoring (RMON) MIB hierarchy RMON uses a special MIB module, *rmon(16)*, which fits into the overall SNMP object hierarchy tree under *mib/mib-2(1)* within *mgmt(2)*—just like other MIB object groups such as *sys(1)* and *if(2)*; see Figure 66-2 in Chapter 66. Within this group, which has the group identifier *1.3.6.1.2.1.16*, are nine subgroups of RMON objects.

Table 69-1: SNMP RMON MIB Object Groups

RMON Group Name	RMON Group Code	RMON Group Number	Full Group Identifier	Description
statistics	etherStats	1	1.3.6.1.2.1.16.1	This group contains objects that keep track of network statistics measured by the device. Statistics may include network traffic load, average packet size, number of broadcasts observed, counts of errors that have occurred, the number of packets in various size ranges, and so forth.
history	history, etherHistory	2	1.3.6.1.2.1.16.2	The history group contains a single table object that controls how often statistical data is sampled by the probe. The additional etherHistory group is optional and contains extra Ethernet-specific information; it is contained logically within the history group.
alarm	alarm	3	1.3.6.1.2.1.16.3	This group defines the parameters under which an alarm may be generated to inform an administrator of an occurrence of import. The alarm group contains a table that describes the thresholds that will cause an event to be triggered (see the event group description in this table).
hosts	host	4	1.3.6.1.2.1.16.4	This group contains objects that keep track of information for each host on a network.
hostsTopN	hostTopN	5	1.3.6.1.2.1.16.5	This group contains objects that facilitate reporting of hosts sorted in a particular way. The administrator determines how these ordered statistics are tracked. For example, an administrator could generate a report listing hosts sorted by the number of packets transmitted, showing the most active devices.

(continued)

Table 69-1: SNMP RMON MIB Object Groups (continued)

RMON Group Name	RMON Group Code	RMON Group Number	Full Group Identifier	Description
matrix	matrix	6	1.3.6.1.2.1.16.6	This group keeps track of statistics for data exchanges between particular pairs of hosts. The amount of data sent between any two devices on the network could be tracked here. Since a large network could have thousands of such device pairs, to conserve resources on the probe, often only the most recent conversations between device pairs are kept in the MIB.
filter	filter	7	1.3.6.1.2.1.16.7	This group allows an administrator to set up filters that control what sorts of network packets the probe will capture.
capture	buffer, capture	8	1.3.6.1.2.1.16.8	This group is used to allow a probe to capture packets based on particular parameters set up in the filter group.
event	event	9	1.3.6.1.2.1.16.9	When a particular alarm is triggered based on the parameters in the objects in the alarm group, an event is generated. This group controls how these events are processed, including creating and sending an SNMP trap message to an NMS.

The original RMON standard was heavily oriented around Ethernet local area networks (LANs), and you can see some of that in Table 69-1. Probes can also gather and report information related to other networking technologies by using other RMON groups created for that purpose. The best example of this was the definition of a set of groups specifically for Token Ring, which was defined in RFC 1513 in 1993.

RMON Alarms, Events, and Statistics

Alarms and events are particularly useful constructs in RMON, as they allow the immediate communication of important information to an NMS. The administrator has full control over what conditions will cause an alarm to be sounded and how an event is generated. This includes specifying which variables or statistics to monitor, how often to check them, and what values will trigger an alarm. A log entry may also be recorded when an event occurs. If an event results in transmission of a trap message, the administrator will thus be notified and can decide how to respond, depending on the severity of the event.

Like all MIB modules and groups, a particular manufacturer may decide which RMON groups to implement. However, certain groups—such as alarm and event—are related, and some groups—such as statistics—are usually implemented in all RMON probes. Obviously, when RMON is used, the NMS must be aware of RMON groups and must allow a network management application to be run that will exploit the capabilities of the RMON MIB objects.

PART III-5

TCP/IP APPLICATION LAYER ADDRESSING AND APPLICATION CATEGORIES

The TCP/IP protocol suite is the foundation of modern internetworking, and for this reason, has been used as the primary platform for the development and implementation of networking applications. Over the past few decades, as the global TCP/IP Internet has grown, hundreds of new applications have been created. These programs support a myriad of different tasks and functions, ranging from implementing essential business tasks to providing pure entertainment. Users may be in the same room or on different continents.

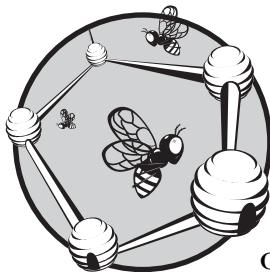
Of the many TCP/IP applications, a small number are widely considered to be key applications of TCP/IP. Most have been around for a very long time—in some cases, longer than even the modern Internet Protocol (IP) itself. Many of these protocols deal specifically with the sending of information that has been arranged into discrete units called *files* or *messages*. For this reason, one of the most important groups of TCP/IP applications is the one that describes the basic mechanisms for moving these files between internetworked devices: *file and message transfer applications*.

The rest of this book deals with the most common definitive TCP/IP applications and application layer protocols. Before describing the applications themselves, however, I need to lay some groundwork related to application protocols as a whole. To that end, this part contains two chapters. In the

first, I explain the universal system set up for TCP/IP applications to use for addressing Internet resources: Uniform Resource Identifiers (URIs), which include Uniform Resource Locators (URLs) and Uniform Resource Names (URNs). In the second chapter, I provide an overview of file and message transfer applications, including a description of the differences between them.

70

TCP/IP APPLICATION LAYER ADDRESSING: UNIFORM RESOURCE IDENTIFIERS, LOCATORS, AND NAMES (URIs, URLs, AND URNs)



The Internet consists of millions of interconnected servers, each of which is capable of providing useful information to Internet users who request it. The more information a network has, the richer it is, but the more difficult it becomes to locate. In order to use information, we need to be able to find it, and that requires, at a minimum, that we employ some means for labeling each file or object.

For this purpose, TCP/IP has defined a system of Uniform Resource Identifiers (URIs) that can be used both on the Internet and on private TCP/IP networks. Each URI uniquely specifies how a client can locate a particular resource and access it so it can be used. URIs are subdivided into Uniform Resource Locators (URLs) and Uniform Resource Names (URNs), which serve a similar purpose but work in different ways.

In this chapter, I describe the system of addressing used on the Internet to identify files, objects, and resources. I begin with an overview, which introduces the concept of URIs and explains the differences between URIs, URLs, and URNs. I then provide a detailed explanation of URLs and how

they are used. This includes an overview of the general syntax used for URLs, a description of the URL schemes used for the most common applications, a discussion of relative URLs and how they work, and a comprehensive look at real-world issues associated with URLs, including the intentional obfuscation games being played by some unscrupulous people. Finally, I discuss URNs, including how they solve a major problem with URLs and the impediments to their use.

URI Overview and Standards

If you've been working your way up the OSI Reference Model layers in reading this book, you might have expected that you would be done with addressing by this point. After all, we have already discussed MAC addresses at layer 2, IP addresses at layer 3, and mechanisms for converting between them (see Chapters 13 and 14). We even have ports and sockets that provide transport layer addressing capabilities to let each device run multiple software applications (see Chapter 43). Given all this, the idea of application layer addressing may seem a bit strange, and I am aware that using the term to refer to the subject of this chapter may be a bit unorthodox.

The concept isn't really as odd as it might seem at first, however. It's true that with an IP address and a port number, we can theoretically access any resource on a TCP/IP internetwork; the problem is finding it.

Application layer addressing is not something that is required by the computer software. It is something that makes it easier for humans to identify and locate resources. This is very much the same rationale that is used to justify the creation of name systems, such as the Domain Name System (DNS; see Part III-1). DNS is a form of high-level addressing that allows names to be used instead of IP addresses. It is helpful to people, who find it easier to understand www.intel.com than 198.175.96.33.

The idea behind a comprehensive application layer addressing scheme is to extend to the next level what DNS has already accomplished. DNS names provide essential high-level abstract addressing, but only of whole devices (whether real or virtual). These names can be used as the basis for a more complete labeling scheme that points not just to a site or device, but to a specific file, object, or other resource. In TCP/IP, these labels are called Uniform Resource Identifiers (URIs).

URIs were one of the key technologies developed as part of the World Wide Web (WWW), and they are still most often associated with the Web and the protocol that implements it, the Hypertext Transfer Protocol (HTTP; see Part III-8). You have likely used URIs thousands of times in the past; whenever you enter something like <http://www.myfavoritewebsite.com> into a web browser, you are using a URI.

The reason why URIs are so important to the Web is that they combine into one string all of the information necessary to refer to a resource. This compactness of expression is essential to the entire concept of hypertext resource linking. If we want to be able to have an object in one document point to another, we need to have a simple way of describing that object without requiring a whole set of instructions. URIs allow us to do exactly that.

In fact, URIs are so associated with the Web that they are usually described as being part of Web technology specifically. They are not, however, unique to the Web, which is why this chapter is separate from the discussion of WWW and HTTP.

URI Categories: URLs and URNs

URIs are a general-purpose method for referring to many kinds of TCP/IP resources. They are currently divided into two primary categories based on how they describe a resource:

Uniform Resource Locators (URLs) A URL is a URI that refers to a resource through the combination of a protocol or access mechanism and a specific resource location. A URL begins with the name of the protocol to be used for accessing the resource, and then contains sufficient information to point to how it can be obtained.

Uniform Resource Names (URNs) A URN is a URI that provides a way of uniquely naming a resource without specifying an access protocol or mechanism, and without specifying a particular location.

The difference between a URL and a URN is that the former is much more specific and oriented around how to access a resource, while the latter is more abstract and designed more to identify what the resource is than describe how to get it.

Giving someone a URL is like giving them directions to find a book, as follows: “Take the train to Albuquerque, then Bus #11 to 41 Albert Street, a red brick house owned by Joanne Johnson. The book you want is the third from the right on the bottom of the bookshelf on the second floor.”

A URN is more like referring to a book using its International Standard Book Number (ISBN); it uniquely identifies the book, regardless of where the book may be located, and doesn't tell you how to access it. (In fact, ISBNs are one of the identification systems used with URNs, as you will see in the section about URNs at the end of this chapter.)

While URLs and URNs are theoretical peers, in practice, URLs are used far more often than URNs. In fact, URLs are so dominant that most people have never even heard of URIs or URNs. The reason is that even though the example of how to find a book suggests that URNs are more natural than URLs, URLs are easier to use in practice. URLs provide the information needed to access a resource, and without being able to access a resource, simply knowing how to identify it is of limited value.

URNs are an attractive concept because they identify a resource without tying it to a specific access mechanism or location. However, the implementation of URNs requires some means of tying the permanent identifier of a resource to where it is at any given moment, which is not a simple task. For this reason, URNs and the methods for using them have been in development for a number of years, while URLs have been in active use all that time.

KEY CONCEPT Some sort of mechanism is needed on any internetwork to allow resources such as files, directories, and programs to be identified and accessed. In TCP/IP, Uniform Resource Identifiers (URIs) are used for this sort of “application layer addressing.” The two types of URIs are Uniform Resource Locators (URLs), which specify how to access an object using a combination of an access method and location, and Uniform Resource Names (URNs), which identify an object by name but do not indicate how to access it.

While URLs began with the Web and most URLs are still used with HTTP, they can and do refer to resources that are accessed using many other protocols, such as the File Transfer Protocol (FTP) and Telnet. The compactness of URIs makes them very powerful for such uses. With a URL, we can use one string to tell a program to retrieve a file using FTP. This replaces the complete FTP process of starting an FTP client, establishing a session, logging in, and issuing commands.

URI Standards

A number of Internet standards published in the 1990s describe the syntax and basic use of URIs, URLs, and URNs. The first was RFC 1630, “Universal Resource Identifiers in WWW,” which was published in 1994 and is still a good overview of the topic. In December 1994, a pair of documents, RFCs 1737 and 1738, provided more specific information about URNs and URLs, respectively. RFC 1808 describes how to define and use relative URLs. RFC 2141 provides more information about the URN syntax.

RFC 2396, “Uniform Resource Identifiers (URI): Generic Syntax,” was published in August 1998 to revise and replace some of the information in many of the previous RFCs just mentioned. It is probably the definitive standard on URIs at the present time, although RFCs continue to be published discussing issues related to URIs. This is especially true of URNs, which as I noted earlier, are still in active development.

The base documents such as RFC 2396 describe how URLs can be specified for a number of common protocols (called *schemes* in URL-speak, as we will see when we look at URLs more closely). To provide flexibility, a mechanism was also defined to allow new URL schemes to be registered. This is described in RFC 2717, “Registration Procedures for URL Scheme Names,” and RFC 2718, “Guidelines for new URL Schemes.” There are also a few RFCs that describe specific URL schemes for different protocols, including RFCs 2192 (IMAP), 2224 (NFS), 2368 (email), and 2384 (POP).

URL General Syntax

URLs are text strings that allow a resource such as a file or other object to be labeled based on its location on an internetwork and the primary method or protocol by which it may be accessed. URLs have become the most common type of URI used for application layer addressing in TCP/IP because of their simplicity.

URLs consist of two components that identify how to access a resource on a TCP/IP internetwork: the location of the resource and the method to be used to access it. These two pieces of information, taken together, allow a user with the appropriate software to obtain, read, or otherwise work with many different kinds of resources, such as files, objects, programs, and much more.

The most general form of syntax for a URL contains only two elements, which correspond to the two pieces of information just described: <scheme>:<scheme-specific-part>. The term *scheme* refers to a type of access method, which describes the way that the resource is to be used. It usually refers to either an application protocol, such as `http` or `ftp`, or a resource type, such as `file`. A scheme name

must contain only letters, plus signs (+), periods (.), and hyphens (-). In practice, scheme names usually contain only letters. Schemes are case-insensitive but usually expressed in lowercase.

The rest of the URL after the scheme (and the required colon separator) is scheme-specific. This is necessary because various protocols and access methods require different types and quantities of information to identify a particular resource. When a URL is read, the scheme name tells the program parsing it how to interpret the syntax of the rest of the URL.

Common Internet Scheme Syntax

In theory, each scheme may use a completely different syntax for the <scheme-specific-part> of a URL. However, many of these schemes share a common syntax for this part, by virtue of the similarities in how they refer to internetwork devices and resources on those devices. For example, both HTTP and FTP are used to point to specific TCP/IP devices using a DNS name or IP address, and then access resources stored in a hierarchical directory structure. It makes sense that their URLs would be at least somewhat similar.

KEY CONCEPT URLs are the most widely used type of URI. In its most basic form, a URL consists of two elements: a scheme that defines the protocol or other mechanism for accessing the resource, and a scheme-specific part that contains information that identifies the specific resource and indicates how it should be used. Some schemes use a common syntax for their scheme-specific parts; others use a syntax unique to the scheme.

The most general form of this common Internet scheme syntax is as follows:

```
<scheme>://<user>:<password>@<host>:<port>/<url-path>;<params>?<query>#<fragment>
```

The syntax elements are as follows:

<scheme> The URL scheme, which refers to a type of access method.

<user> and <password> Authentication information for schemes requiring a login, in the form of a user name and password.

<host> An Internet host, usually specified either as a fully qualified DNS domain name or an IP address in dotted decimal notation.

<port> A Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) port number to use when invoking the protocol appropriate to the scheme.

<url-path> A resource location path. This is usually a full directory path expressing the sequence of directories to be traversed from the root directory to the place where the resource is located, and then the resource's name. For example, if on a device there is a directory called project1 and within it a subdirectory called memos containing a text file called June11th-minutes.txt, the URL path project1/memos/June11th-minutes.txt would refer to that resource. Note that the slash before the <url-path> is required, and while it is technically not considered part of the path,

it serves the purpose of acting like the slash denoting the root directory in many file systems. Also, the <url-path> may end in a slash, which means that the path refers specifically to a directory. However, this is often not required, as the server will treat the URL as a directory reference by context when needed. A path may also refer to a virtual file, program, or another type of resource.

<params> Scheme-specific parameters included to control how the scheme is used to access the resource. Each parameter is generally of the form <parameter>=<value>, with each parameter specification separated from the next using a semicolon.

<query> An optional query or other information to be passed to the server when the resource is accessed.

<fragment> Identifies a particular place within a resource that the user of the URL is interested in.

Figure 70-1 illustrates this common syntax and its elements using an example of an HTTP URL.

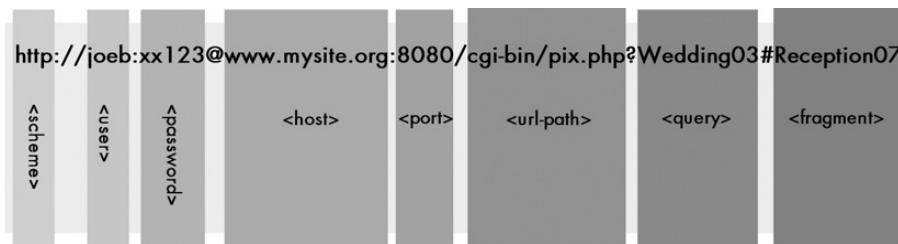


Figure 70-1: Example of a Uniform Resource Locator (URL) This diagram shows a sample URL that includes almost all of the possible elements in the general scheme syntax, each of them highlighted using shaded boxes. This URL identifies a Web (HTTP) resource that must be accessed using a particular password at the site `www.mysite.org` using port 8080. The resource in this case is a PHP program in the site's `cgi-bin` directory that causes a particular page of photographs to be displayed. The <fragment> specifier will cause the picture `Reception07` on the retrieved page of wedding photos to be displayed to the user.

Omission of URL Syntax Elements

The full URL syntax may seem very complicated, but bear in mind that this is a formal definition and shows all of the possible elements in a URL at once. Most schemes do not use every one of these elements, and furthermore, many of them are optional, even when they are valid in a particular scheme. For example, the <login> and <password> elements are officially supported for HTTP URLs, but they are very rarely used. Similarly, port numbers are often omitted, telling the client software to just use the default port number for the scheme. The “URL Schemes and Scheme-Specific Syntaxes” section of this chapter describes some of the most common URL schemes and the specific syntaxes used for them, including how and when these elements are employed.

Even though the richness of the URL syntax isn't often needed, it can be useful for supplying a wide variety of information in special cases. URLs are also very flexible in terms of how they may be expressed. For example, while a <host> element is usually a DNS name, it can also be an IP address expressed in many forms, including dotted decimal, regular decimal, hexadecimal, octal, and even a

combination of these. Unfortunately, the lack of familiarity that most people have with some of these refinements has led to URLs being abused through deliberate obscuration, to get people to visit “resources” they would normally want to avoid. We’ll explore this later in this chapter, in the “URL Obscuration, Obfuscation, and General Trickery” section.

URL Fragments

It’s worth noting that, technically, a `<fragment>` element is not considered a formal part of the URL by the standards that describe resource naming. The reason is that it identifies only a portion of a resource, and it is not part of the information required to identify the resource itself. It is not sent to the server but retained by the client software, to guide it in how to display or use the resource. Some would make a valid argument, however, that this distinction is somewhat arbitrary. Consider, for example, that the scheme itself is also used only by the client, as is the host itself.

The most common example of a URL fragment is specifying a particular bookmark to scroll to in displaying a web page. In practice, a fragment identifier is often treated as if it were part of a URL, since it is part of the string that specifies a URL.

Unsafe Characters and Special Encodings

URLs are normally expressed in the standard US ASCII character set, which is the default used by most TCP/IP application protocols. Certain characters in the set are called unsafe, because they have special meaning in different contexts, and including them in a URL would lead to ambiguity or problems in of how they should be interpreted. The space character is the classic unsafe character, because spaces are normally used to separate URLs, so including one in a URL would break the URL into pieces. Other characters are unsafe because they have special significance in a URL, such as the colon (:).

The safe characters in a URL are alphanumerics (A to Z, a to z, and 0 to 9) and the following special characters: the dollar sign (\$), hyphen (-), underscore (_), period (.), plus sign (+), exclamation point (!), asterisk (*), apostrophe ('), left parenthesis ((,), and right parenthesis ()). All other unsafe characters can be represented in a URL using an encoding scheme consisting of a percent sign (%) followed by the hexadecimal ASCII value of the character. The most common examples are given in Table 70-1.

Table 70-1: URL Special Character Encodings

Character	Encoding	Character	Encoding	Character	Encoding
<code><space></code>	%20	<	%3C	>	%3E
#	%23	%	%25	{	%7B
}	%7D		%7C	\	%5C
^	%5E	~	%7E	[%5B
]	%5D		`%60	;	%3B

(continued)

Table 70-1: URL Special Character Encodings (continued)

Character	Encoding	Character	Encoding	Character	Encoding
/	%2F	?	%3F	:	%3A
@	%40	=	%3D	&	%26

When these sequences are encountered, they are interpreted as the literal character they represent, without any significance. So, the URL `http://www.myfavesite.com/are%20you%20there%3F` points to a file called “are you there?” on www.myfavesite.com. The %20 codes prevent the spaces from breaking up the URL, and the 3F prevents the question mark in the filename from being interpreted as a special URL character.

NOTE Since the percent sign is used for this encoding mechanism, it itself is special. When it is encountered, the next values are interpreted as character encodings. So, to embed a literal percent sign, it must be encoded as %25.

Again, these encodings are sometimes abused for nefarious purposes, unfortunately, such as using them for regular ASCII characters to obscure URLs.

URL Schemes and Scheme-Specific Syntaxes

As explained in the previous sections, URLs use a general syntax that describes the location and method for accessing a TCP/IP resource:

```
<scheme>://<user>:<password>@<host>:<port>/<urlpath>;<params>?<query>#<fragment>
```

Each access method, called a *scheme*, has its own specific URL syntax, including the various pieces of information required by the method to identify a resource. RFC 1738 includes a description of the specific syntaxes used by several popular URL schemes. Others have been defined in subsequent RFCs using the procedure established for URL scheme registration.

Several of the URL schemes use the common Internet pattern shown in Figure 70-1 earlier in the chapter. Other schemes use entirely different (usually simpler) structures based on their needs.

The following sections describe the most common URL schemes and the scheme-specific syntaxes they use.

World Wide Web/Hypertext Transfer Protocol Syntax (`http`)

The Web potentially uses most of the elements of the common Internet scheme syntax, as follows:

```
http://<user>:<password>@<host>:<port>/<url-path>?<query>#<bookmark>
```

As discussed in the overview of resource identifiers, the Web is the primary application using URLs today. A URL can theoretically contain most of the common URL syntax elements, but in practice, most are omitted. Most URLs

contain only a host and a path to a resource. The port number is usually omitted, implying that the default value of 80 should be used. The `<query>` construct is often used to pass arguments or information from the client to the web server.

I have provided full details on how Web URLs are used in Chapter 79.

File Transfer Protocol Syntax (ftp)

The syntax for FTP URLs is:

```
ftp://<user>:<password>@<host>:<port>/<url-path>;type=<typecode>
```

FTP (see Chapter 72) is an interactive command-based protocol, so it may seem odd to use a URL for FTP. However, one of the most common uses of FTP is to access and read a single file, and this is what an FTP URL allows a client to do, quickly and easily. The `<user>` and `<password>` elements are used for login and may be omitted for anonymous FTP access. The port number is usually omitted and defaults to the standard FTP control channel port, 21.

The `<url-path>` is interpreted as a directory structure and filename. The appropriate CWD (change working directory) commands are issued to go to the specified directory, and then a RETR (retrieve) command is issued for the named file. The optional type parameter can be used to indicate the file type: `a` to specify an ASCII file retrieval or `i` for an image (binary) file. The type parameter is often omitted from the URL, with the correct mode being set automatically by the client based on the name of the file.

For example, consider this URL:

```
ftp://ftp.hardwarecompanyx.com/drivers/widgetdriver.zip
```

This is equivalent to starting an FTP client, making an anonymous FTP connection to `ftp.hardwarecompanyx.com`, then changing to the `drivers` directory and retrieving the file `widgetdriver.zip`. The client will retrieve the file in binary mode because it is a compressed ZIP file.

It is also possible to use an FTP URL to get a listing of the files within a particular directory. This allows users to navigate an FTP server's directory structure using URL links to find the file they want, and then retrieve it. This is done by specifying a directory name for the `<url-path>` and using the type parameter with a `<typecode>` of `d` to request a directory listing. Again, the type parameter is usually omitted, and the software figures out to send a LIST command to the server when a directory name is given in a URL.

Electronic Mail Syntax (mailto)

A special syntax is defined to allow a URL to represent the command to send mail to a user:

```
mailto:<email-address>
```

The email address (see Chapter 75) is in standard Internet form: <username>@<domainname>. This is really an unusual type of URL because it does not really represent an object at all, though a person can be considered a type of resource. Note that optional parameters, such as the subject of the email, can also be included in a mailto URL. This facility is not often used, however.

Gopher Protocol Syntax (gopher)

The syntax for the Gopher protocol is similar to that of HTTP and FTP:

gopher://<host>:<port>/<gopher-path>

See Chapter 86 for more information about the Gopher protocol.

Network News/Usenet Syntax (news)

Two syntaxes are defined for Usenet newsgroup access:

news://<newsgroup-name>
news://<message-id>

Both of these URLs are used to access a Usenet newsgroup (see Chapter 85) or a specific message, referenced by message ID. Like the mailto scheme, this is a special type of URL because it defines an access method but does not provide the detailed information to describe how to locate a newsgroup or message.

By definition, the first form of this URL is interpreted as being local. So, for example, news://alt.food.sushi means, “Access the newsgroup alt.food.sushi on the local news server, using the default news protocol.” The default news protocol is normally the Network News Transfer Protocol (NNTP). The second URL form is global, because message IDs are unique on Usenet (or at least, they are supposed to be!).

Network News Transfer Protocol Syntax (nntp)

The nntp form is a different URL type for news access:

nntp://<host>:<port>/<newsgroup-name>/<article-number>

Unlike news, this URL form specifically requests the use of NNTP (see Chapter 85) and identifies a particular NNTP server. Then it tells the server which newsgroup to access and which article number within that newsgroup. Note that articles are numbered using a different sequence by each server, so this is still a local form of news addressing. The port number defaults to 119.

Even though the `nntp` form seems to provide a more complete resource specification, the `news` URL is more often used, because it is simpler. It's easier just to set up the appropriate NNTP server in the client software once than to specify it each time, since clients usually use only one NNTP server.

Telnet Syntax (`telnet`)

This syntax is used to open a Telnet connection to a server (see Chapter 87):

```
telnet://<user>:<password>@<host>:<port>
```

In practice, the user name and password are often omitted, which causes the Telnet server to prompt for this information. Alternatively, the `<user>` can be supplied and the password left out (to prevent it being seen), and the server will prompt for just the password. The port number defaults to the standard port for Telnet, 23, and is also often omitted.

This type of URL is interesting in that it identifies a resource that is not an object but rather a service.

Local File Syntax (`file`)

A special URL type is used for referring to files on a particular host computer. The standard syntax is:

```
file://<host>:<url-path>
```

This type of URL is also somewhat interesting, in that it describes the location of an object but not an access method. It is not sufficiently general to allow access to a file anywhere on an internetwork, but is often used for referencing files on computers on a local area network (LAN) where names have been assigned to different devices.

A special syntax is also defined to refer specifically to files on the local computer:

```
file:///<url-path>
```

Here, the entire `//<host>:` element has been replaced by a set of three slashes, meaning to look on the local host.

Special Syntax Rules

Additional syntax rules are often used by browsers to support the quirks of Microsoft operating systems, especially for the file scheme. First, the backslashes used by Microsoft Windows are expressed as forward slashes as required by TCP/IP. Second, since colons are used in drive letters specifications in Microsoft operating systems, these are replaced by the vertical pipe character (!).

So, to refer to the file C:\WINDOWS\SYSTEM32\DRIVERS\ETC\HOSTS, the following URL could be used:

file:///C:/WINDOWS/SYSTEM32/DRIVERS/ETC/HOSTS

Note, however, that some browsers actually do allow the colon in the drive specification.

URL Relative Syntax and Base URLs

The URL syntax described so far is sometimes said to specify an *absolute URL*. This is because the information in the URL is sufficient to completely identify the resource. Absolute URLs thus have the property of being context-independent, meaning that users can access and retrieve the resource using the URL without any additional information required.

Since the entire point of a URL is to provide the information needed to locate and access a resource, it makes sense that we would want them to be absolute in definition most of the time. The problem with absolute URLs is that they can be long and cumbersome. There are cases where many different resources need to be identified that have a relationship to each other; the URLs for these resources often have many common elements. Using absolute URLs in such situations leads to a lot of excess and redundant verbiage.

In the overview of URIs at the beginning of this chapter, I gave a real-world analogy to a URL in the form of a description of an access method and location for a person retrieving a book: “Take the train to Albuquerque, then Bus #11 to 41 Albert Street, a red brick house owned by Joanne Johnson. The book you want is the third from the right on the bottom of the bookshelf on the second floor.”

What if I also wanted the same person to get a second book located in the same house on the ground floor after getting the first one? Should I start by saying again, “Take the train to Albuquerque, then Bus #11 to 41 Albert Street, a red brick house owned by Joanne Johnson?” Why bother, when they are already there at that house? No, I would give a second instruction in relative terms: “Go downstairs, and also get the blue book on the wood table.” This instruction only makes sense in the context of the original one.

The same need arises in URLs. Consider a web page located at <http://www.longdomainnamesareirritating.com/index.htm> that has 37 embedded graphic images in it. The poor guy stuck with maintaining this site doesn’t want to have to put <http://www.longdomainnamesareirritating.com/> in front of the URL of every image.

Similarly, if we have just taken a directory listing at <ftp://ftp.somesitesomewhere.org/very/deep/directory/structures/also/stink/>, and we want to explore the parent directory, we would like to just say “go up one level,” without having to say <ftp://ftp.somesitesomewhere.org/very/deep/directory/structures/also/>.

It is for these reasons that URL syntax was extended to include a relative form. In simplest terms, a relative URL is the same as an absolute URL, but with pieces of information omitted that are implied by context. Like our “Go downstairs”

instruction, a relative URL does not by itself contain enough information to specify a resource. A relative URL must be interpreted within a context that provides the missing information.

Interpretation Rules for Relative URLs

The context needed to find a resource from a relative URL is provided in the form of a base URL that provides the missing information. A base URL must be either a specific absolute URL or itself a relative URL that refers to some other absolute base. The base URL may be either explicitly stated or may be inferred from use. The RFCs dealing with URLs define the following three methods for determining the base URL, in the precedence in which they are listed here:

Base URL Within Document Some documents allow the base URL to be explicitly stated. If present, this specification is used for any relative URLs in the document.

Base URL from Encapsulating Entity In cases where no explicit base URL is specified in a document, but the document is part of a higher-level entity enclosing it, the base URL is the URL of the parent document. For example, a document within a body part of a MIME multipart message (see Chapter 76) can use the URL of the message as a whole as the base URL for relative references.

Base URL from Retrieval URL If neither of those two methods are feasible, the base URL is inferred from the URL used to retrieve the document containing the relative URL.

Of these three methods, the first and third are the most common. HTML, the language used for the Web, allows a base URL to be explicitly stated, which removes any doubt about how relative URLs are to be interpreted. Failing this, the third method is commonly used for images and other links in HTML documents that are specified in relative terms.

KEY CONCEPT Regular URLs are absolute, meaning that they include all of the information needed to fully specify how to access a resource. In situations where many resources need to be accessed that are approximately in the same place or are related in some way, completely specifying a URL can be inefficient. Instead, relative URLs can be used, which specify how to access a resource relative to the location of another one. A relative URL can be interpreted only within the context of a base URL that provides any information missing from the relative reference.

For example, let's go back to the poor slob maintaining <http://www.longdomainnamesareirritating.com/index.htm>. By default, any images referenced from that index.htm HTML document can use relative URLs—the base URL will be assumed from the name of the document itself. So he can just say `companylogo.gif` instead of <http://www.longdomainnamesareirritating.com/companylogo.gif>, as long as that file is in the same directory on the same server as index.htm.

If all three of these methods fail for whatever reason, then no base URL can be determined. Relative URLs in such a document will be interpreted as absolute URLs, and since they do not contain complete information, they will not work properly.

Practical Interpretation of Relative URLs

This probably seems confusing, but relative URLs are actually fairly easy to understand, because they are interpreted in a rather common-sense way. You simply take the base URL and the relative URL, and you substitute whatever information is in the relative URL for the appropriate information in the base URL to get the resulting equivalent absolute reference. In so doing, you must drop any elements that are more specific than the ones being replaced.

What do I mean by “more specific?” Well, most URLs can be considered to move from most general to most specific in terms of the location they specify. As you go from left to right, you go through the host name, then high-level directories, subdirectories, the filename, and optionally, the parameters, query, and fragment applied to the filename. If a relative URL specifies a new file name, it replaces the file name in the base URL, and any parameters, query, and fragment elements are dropped, as they no longer have meaning given that the file name has changed. If the relative URL changes the host name, the entire directory structure, filename, and everything else to the right of the host name goes away, replaced with any that might have been included in the new host name specification.

This is hard to explain in words but easy to understand with a few examples. Let’s assume we start with the following explicit base URL:

`http://site.net/dir1/subdir1/file1?query1#bookmark1`

Table 70-2 shows some examples of relative URLs and how they would be interpreted.

Table 70-2: Relative URL Specifications and Absolute Equivalents

Relative URL	Equivalent Absolute URL	Explanation
#bookmark2	<code>http://site.net/dir1/subdir1/file1#bookmark2</code>	The URL is the same as the base URL, except that the bookmark is different. This can be used to reference different places in the same document in HTML. Technically, the URL has not changed here, since the fragment (bookmark) is not part of the actual URL. A web browser given a new bookmark name will usually not try to reaccess the resource.
?query2	<code>http://site.net/dir1/subdir1/file1?query2</code>	The same file as given by the base URL, but with a different query string. Note that the bookmark reference from the base URL is stripped off.
file2	<code>http://site.net/dir1/subdir1/file2</code>	This refers to a file using the name file2, which replaces file1 in the base URL. Here, both the query and bookmark are removed.

(continued)

Table 70-2: Relative URL Specifications and Absolute Equivalents (continued)

Relative URL	Equivalent Absolute URL	Explanation
/file2	http://site.net/file2	Since a single slash was included, this means file2 is in the root directory. This relative URL replaces the entire <url-path> of the base URL.
..	http://site.net/dir1/	The pair of dots refers to the parent directory of the one in the base URL. Since the directory in the base URL is dir1/subdir1. This refers to dir1/.
../file2	http://site.net/dir1/file2	This specifies that we should go up to the parent directory to find the file file2 in dir1.
../subdir2/file2	http://site.net/dir1/subdir2/file2	This says go up one directory with .., then enter the subdirectory subdir2 to find file2.
../../dir2/subdir2/file2	http://site.net/dir2/subdir2/file2	This is the same as the previous example, but going up two directory levels, then down through dir2 and subdir2 to find file2.
//file2	http://file2	Two slashes means that file2 replaces the host name, causing everything to the right of the host name to be stripped. This is probably not what was intended, and it shows how important it is to watch those slashes.
//www.newsite.net/otherfile.htm	http://www.newsite.net/otherfile.htm	In this example, everything but the scheme has been replaced. In practice, this form of relative URL is not that common—the scheme is usually included if the site name is specified, for completeness.
file2?query2#bookmark2	http://site.net/dir1/subdir1/file2?query2#bookmark2	This replaces the filename, query name, and bookmark name.
ftp://differentsite.net/whatever	ftp://differentsite.net/whatever	Using a new scheme forces the URL to be interpreted as absolute.

Relative URLs have meaning only for certain URL schemes. For others, they make no sense and cannot be used. In particular, relative URLs are never used for the telnet, mailto, and news schemes. They are very commonly used for HTTP documents, and may also be used for FTP and file URLs.

Incidentally, there is one other very important benefit of using relative URLs: Avoiding absolute URLs in a document allows it to be more portable by eliminating hard-coded references to names that might change. Going back to our previous example, if the guy maintaining the site <http://www.longdomainnamesareirritating.com> uses only relative links to refer to graphics and other embedded objects, then if the site is migrated to www.muchshortername.com, he will not need to edit all of his links to the new name. The significance of this in Web URLs is explored further in the detailed discussion of HTTP URLs in Chapter 79.

KEY CONCEPT In addition to being more efficient than absolute URLs, relative URLs have the advantage that they allow a resource designer to avoid the specific mention of names. This increases the portability of documents between locations within a site or between sites.

URL Length and Complexity Issues

URLs are the most ubiquitous form of resource addressing for some very good reasons: They represent a simple, convenient, and easy-to-understand way of finding documents. Popularized by their use on the Web, URLs can now be seen in everything from electronic document lists to television commercials—a testament to their universality and ease of use.

At least, this is true most of the time.

When URLs work, they work very well. Unfortunately, there are also some concerns that arise with respect to how URLs are used. Both accidental and intentional misuse of URLs occurs on a regular basis. Part of why I have devoted so much effort to describing URLs is that most people don't really understand how they work, and this is part of why problems occur.

Many of the issues with URLs are directly due to the related matters of length and complexity. URLs work best when they are short and simple, so it is clear what they are about and so they are easy to manipulate. For example, <http://www.ibm.com> is recognizable to almost everyone as the website of the International Business Machines Corporation (IBM). Similarly, you can probably figure out what this URL does without any explanation: <ftp://www.somecomputercompany.com/drivers/video-drivers.zip>.

However, as you have seen earlier in this chapter, URLs can be much more complex. In particular, the common Internet syntax used by protocols such as HTTP and FTP is extremely flexible, containing a large number of optional elements that can be used when required to provide the information necessary for a particular resource access.

The point that many elements in URL syntax are optional is important. The majority of the time, most of these optional parts are omitted, which makes URLs much simpler in practical use than they are in their descriptions. For example, even though an HTTP URL theoretically contains a user name, password, host, port, path, query, and bookmark, most URLs use only a host name and a path. This is what helps keep URLs short and easy to use.

Despite this, you will still find some rather long URLs used on the Internet, for a variety of reasons:

Long DNS Domain and Host Names Some people don't realize that long host names are hard to remember. If you run the Super Auto Body Shop & Pizza Parlor, having a website called www.superauto.com will make it easier for your customers to find you than trying to register www.superautobodyshopandpizza.com. Yet DNS names of 15, 20, or even more characters are surprisingly common.

Long Document or Directory Names Similarly, short filenames are better than long ones, and again, many people don't think about this before putting files on the Internet, which makes things more difficult for those who must access them.

Use of Unsafe Characters As discussed saw earlier in this chapter, URLs have a mechanism for dealing with unsafe characters, but it makes them longer and harder to decipher. If you have a file called “{ABC Corp} budget; draft #3; third quarter 2004.htm,” the URL for it will have to be [{ABC Corp}%20budget%20draft%20third%20quarter%202004.htm](http://www.superauto.com/budget%20draft%20third%20quarter%202004.htm).

The original long filename was readable, but the URL is a mess. Naming the file “ABC budget draft 3, 3Q2004.htm” would be a better choice, and still includes enough information to be understandable. Even better, you could replace the spaces with underscores, to avoid the need for the %20 encoding entirely: “ABC_budget_draft 3,_3Q2004.htm.”

Parameter Strings In HTTP URLs, the syntax for specifying a query (following a question mark character) is often used to allow a web browser to send various types of information to a web server, especially parameters for interactive queries. These parameter strings can get quite lengthy. For example, I typed in a query to the great web search engine Google to find recipes for potato salad. This is what the URL for one of the recipe files looks like:

```
http://groups.google.com/groups?q=%22potato+salad%22&hl=en&lr=&ie=UTF-8&safe=off&selm=B826FB57.89C0%25sbrooks%40ev1.net&rnum=2
```

Almost all of that consists of parameters that tell the Google server exactly what document I want based on my query. It is necessary, but still cumbersome.

URL Wrapping and Delimiting

For humans, long and complex URLs are hard to remember and use. In addition to the sheer difficulty of remembering all those characters, there is the issue of URL wrapping, which occurs when they are presented in certain forms. Most programs can display only 78 or 80 characters in a single line. If a URL is longer than this, the characters of the URL will wrap onto multiple lines; when you read that Google example of parameter strings, you probably noticed that.

URL wrapping can lead to mistakes when copying a URL from one form to another, such as if you copied it from this document into your web browser. If a URL is 81 characters long, and 80 are on the first line and the last character is on the second line, many users may not realize that the URL has wrapped. I have seen URLs that are hundreds of characters long, requiring several manual copy-and-paste operations to get the URL to work.

Perhaps surprisingly, some software may not handle this wrapping properly either. While this is not a problem when a hyperlink is used in something like an HTML document, it can be troublesome when links are included in an email message or Usenet article.

Another issue is delimiting where a URL starts and ends when it appears. A URL begins with a scheme name that could, in theory, be used in other contexts that are not URLs. Without a clear way of labeling a URL as being a URL, a software program might not recognize it. Consider discussion of a URL in a document like this one. If I say, “Please visit http://www.thissite.com; you will see the information you need there,” we all know the semicolon is part of the sentence and not part of the URL, but a computer program might not be so sure. And again, this problem is worse when a URL is long and complex, and wraps on to multiple lines of text. How does the program recognize the end of the URL?

Explicit URL Delimiting and Redirectors

To resolve both the wrapping and delimiting problems, a special URL super-syntax is sometimes employed, especially when URLs are used in other text. This is done by surrounding the URL in angle brackets, possibly including the label URL: before the scheme name. For example, all of the following are equivalent:

```
http://www.networkingistoodarnedcomplicated.com
<http://www.networkingistoodarnedcomplicated.com>
<URL:http://www.networkingistoodarnedcomplicated.com>
```

The angle brackets indicate clearly where the URL begins and ends, making it easier for both programs and humans to deal with long URLs.

Another solution sometimes used for long URLs are redirection services, provided by many websites. For example, <http://www.tinyurl.com> is a free service that allows someone to create a short URL that automatically loads a resource at a much longer URL.

URL Abbreviation

One final issue I want to discuss isn't related directly to long or complex URLs, but is related indirectly to the matter of length: URL abbreviation. Many people use URLs so often that they become lazy when it comes to specifying URLs. They tend to leave off portions of the full URL syntax to save time and energy. I don't mean by this that they specify relative URLs, but rather, they specify absolute URLs with missing pieces.

For example, rather than type `http://www.sitename.com`, they might type `http:www.sitename.com`, leaving off the two slashes. More commonly, people omit the scheme name entirely, just entering `www.sitename.com`. Technically, this is not a URL—it is just a domain name. However, most web browsers can handle this, assuming by default that the scheme is `http://` if none is provided.

URL Obscuration, Obfuscation, and General Trickery

Most of the time, the owner of a resource wants the URL that refers to the resource to be short, simple and easily understood. Thus, long and complex URLs are usually the result of necessity, accident, or ignorance. Some resources need to have long names for a specific reason, such as the use of the long query string in the Google example earlier; other times, URLs are made long because the owner of the resource doesn't realize that using a long DNS host name or file name will make for a long and unwieldy URL.

Whatever the reasons for these situations, they are not deliberate. Recent years, however, have seen a dramatic rise in the use of intentionally long, complex, confusing and deliberately deceptive URLs. These URLs are either structured so that it is impossible to tell what they are, or worse, they are made to appear as if they point to one resource when they really go to another.

Why would people do this? Because they do not want to be open and honest about their “resources.” And who would these people be? Why, they would be the spammers and con artists who overload our Internet email boxes with offers of every sort imaginable, from making you rich beyond your wildest dreams to inflating the dimensions of certain body parts to unnatural sizes.

They are afraid that if the URL indicated clearly what the “resource” was, you might not click the link, or that if you identify them as spammers you might filter out their email. They also figure that if they can make the URL appear to be something interesting, you’ll load it. Even if it turns out to be something you didn’t expect, maybe you’ll pay attention anyway.

You may be thinking that you are too smart to be tricked into buying a product through a deceptive URL. And you would never support a spammer anyway. What a coincidence—same with me! Yet the spam keeps coming. It must work, or they wouldn’t keep doing it . . . would they?

It is a cruel irony that the complex syntax that was built into URLs to allow them to be so flexible has been subject to exploitation. Tricksters know that most people are used to seeing simple URLs like `http://www.myfavoritesite.com` and do not realize that the full URL syntax allows the same resource to be specified in literally millions of different ways. So, desperate for hits to their websites at any cost, they keep coming up with new tricks for manipulating URLs. These are focused on HTTP scheme URLs, though in theory, the tricks can be applied to several other types as well (though they won’t work with some schemes).

Here are some of the more common gimmicks that have been used (note that if you are trying these out as you read, some examples may not work on certain browsers):

Excessive Length In some cases, a URL is just made really long by the addition of a lot of gibberish as a query string, so that the user’s eyes glaze over just looking at it. This is a relatively unsophisticated technique, however, since you can easily tell what the real host name is by looking at the start of the URL. Most of the better scammers have moved beyond such simple tricks today.

Regular IP Address Hosts Internet users are so accustomed to using DNS names that they don’t realize that you can access a URL using an IP address. So most people don’t realize that The PC Guide can be accessed as easily using `<http://209.68.14.80>` as `<http://www.PCGuide.com>`. (Note that this is not true of all Internet hosts; those that use virtual names cannot be accessed using just an IP address.) This is not really trickery per se. It is quite legitimate, and in some ways, even necessary; for example, for accessing a site that is having DNS problems. The problem here is that usually you cannot tell what a site is from the IP address alone, and many people will just click an IP address link without bothering to find out what it is.

Numeric Domain Names It is possible to register a DNS domain name consisting of just a single number. For example, one could register 114.com. And then you could create subdomains within it such as 42.12.205.114.com. At first glance, this appears to be an IP address specification, so someone might think it would resolve to the address 42.12.205.114, but it’s actually some other address. I believe that DNS name registrars have been cracking down on this sort of trickery, so it may not be as prevalent now as it once was.

Bogus Authentication Information HTTP URLs theoretically support the inclusion of authentication information, by including `<user>:<password>@` before the host in the URL. Yet the vast majority of websites are open, and neither require nor use this type of information. If you specify an authentication string and it is not needed, it is ignored. One way to abuse this is by including “authentication information” that looks like a benign host, to make the user think the URL is for that host. For example, if I wanted to trick you into visiting The PC Guide, I might use this URL to make it look like clicking it would go to CNN: `<http://www.cnn.com@www.PCGuide.com>`. This is still too obvious, however, so this method is often combined with some of the following techniques.

Deceptive Character Encoding The use of the percent sign to encode special characters such as spaces and punctuation can also be abused to obscure the name of a domain. For example, the following is another way of expressing the DNS name for The PC Guide: `<http://%57%57%57.%50%43%47%55%49%44%45.%43%4F%4D>`. Try it!

IP Address Math Trickery Okay, this is where things get really bizarre. Most of the time, we express an IP address as a dotted decimal number. Remember, however, that to computers, the IP address is just a 32-bit binary number. Most browsers support a rather shocking number of methods for expressing these numbers. This is unfortunate, because this flexibility is really not needed and almost never used for legitimate purposes. It can lead to some really bizarre URLs that are unrecognizable or that look like regular IP addresses but are not. Here are some examples, all of which are the same as the IP address form of The PC Guide (`<http://209.68.14.80>`):

- An IP address in dotted octal uses a leading zero to signify where each byte is in octal, as in `<http://0321.0104.016.0120>`.
- An IP address in dotted hexadecimal uses a leading zero followed by an `x` to signify where each byte is in hexadecimal, as in `<http://0xD1.0x44.0x0E.0x50>`.
- We can even take the entire 32-bit number and express it as a single number, and that will work too. In decimal, this would look like `<http://3510898256/>`; in octal, `<http://032121007120/>`; and in hexadecimal, `<http://0xd1440e50/>`.

As if these tricks weren’t bad enough taken individually, we can have some real fun by combining them! For example, start with the regular PC Guide URL:

`<http://www.PCGuide.com>`

And convert it to IP:

`<http://209.68.14.80>`

Then add some bogus authentication gibberish:

`<http://www.cnn.com@209.68.14.80>`

And convert the real URL into a single number, so it looks like a document on the CNN website:

```
<http://www.cnn.com@3510898256>
```

Alternatively, we can use the octal form, and even include a lot of extra leading zeros just for fun:

```
<http://www.cnn.com@000000000000321.0000000104.0000000000016.00000120>
```

Believe it or not, this is just the tip of the iceberg. In some browsers, even the IP address numbers can be expressed using percent sign ASCII encoding!

While quite irritating, I must give these people points for creativity at least—some of the tricks are quite ingenious. At the same time, their inventiveness is potentially hazardous. While these false URLs are usually more a waste of time than anything harmful, there are sometimes good reasons a person would go to great lengths to hide the identity of a resource. Deceptive URLs are just one more danger that network administrators must deal with today.

KEY CONCEPT The syntax of Internet URLs includes many elements that provide great flexibility in how URLs can be constructed. Unfortunately, these capabilities of expression are now often abused by people who create intentionally obfuscated URLs to trick users into accessing their websites and other resources. Some of these can be potentially hazardous, which means that care is required before clicking unknown links or accessing strange URLs.

URNs

“HTTP 404 - NOT FOUND”

Have you ever tried to access a website or other Internet resource, only to see those dreaded words appear? You probably have, and in seeing them, you have experienced firsthand one of the most common problems with URLs.

URLs specify a resource using two key pieces of information: the resource’s location and a method by which the resource may be accessed or retrieved. This focus on the means of access for the resource makes URLs very practical, in that the URL usually contains all the data we need to use the resource. This is why URLs are so widely used today. However, this access orientation also means that URLs have a number of serious limitations.

The Problem with URLs

The main difficulty with URLs is that since they describe a resource based on its location, they tie the resource and its location together inextricably. While this may not seem to be a big deal, it is actually a fairly serious matter in a number of ways, because a resource and its location are not the same thing. It is only because most Internet resources rarely change location that we don’t notice this issue more often with URLs.

Suppose that your name is Joe Xavier Zachariah and you live at 44 Glendale Crescent in Sydney, Australia. If someone asked you who you were, would you say, “Joe Xavier Zachariah,” or “the man living at 44 Glendale Crescent in Sydney, Australia”? Almost certainly, you would supply the former answer. But a URL would be like describing yourself as a “resource” using the latter description.

Since we realize that Mr. Zachariah is obviously not always going to be at 44 Glendale Crescent, we know that describing him using just a location is not sufficient. The same thing occurs with Internet resources when they are identified using only location.

However, the problem with Internet resources and URLs goes beyond just the matter of movement. Consider a situation where a particular resource is very popular and we want to duplicate the same resource in multiple locations. Using URLs, we would need a different identifier for each copy of the resource, even though each copy is the same. Again, the problem is that we are not identifying the resource itself, but rather the place where it can be found.

In recognition of this issue with URLs, an alternative identification mechanism for Internet resources was developed, called *Uniform Resource Names (URNs)*.

Overview of URNs

The basic standard describing URNs is RFC 1737, “Functional Requirements for Uniform Resource Names,” which was published in 1994. In 1997, RFC 2141 was published, which specifies the syntax of URNs.

As you can probably tell from that term, a URN is intended to label a resource based on its actual identity, rather than where it can be found. So, where a URL is like Joe Zachariah’s address, a URN would be his name. Or, as I gave as an example in the overview of URIs at the beginning of this chapter, a URN would be identifying a book based on its ISBN number rather than specifying which bookshelf it is on in a building.

To be useful in identifying a particular resource, it is necessary that a URN be globally unique, and that’s not always as simple as it may at first appear. Consider human names, for example. Even though there is probably only one Charles Marlin Kozierok in the entire world, if your name is John Paul Smith or José Garcia, you likely share that name with thousands of others. This means using common names may not be sufficient for identifying human “resources,” and some other method might need to be devised.

URN Namespaces and Syntax

There are many types of resources that URNs are intended to identify on the Internet, each of which may require a different form of naming. To allow URNs to represent many kinds of resources, numerous *URN namespaces* are defined.

A namespace is referenced using a unique string that tells the person or computer interpreting the URN what type of resource the URN identifies. The namespace also ensures the uniqueness of URNs, when a particular identifier might exist in more than one context. For example, both North American telephone numbers and ISBN numbers consist of ten digits, so a particular number

such as 4167819249 could represent both a telephone number and a book number. The namespace identifier tells us what the number means when it is encountered in a URN.

The general syntax of a URN is as follows:

URN:<namespace-ID>:<resource-identifier>

For example, a book with the ISBN number 0-679-73669-7 could be represented as URN:isbn:0-679-73669-7. This string identifies that particular book uniquely, wherever it might happen to be in the world. Many other namespaces have also been defined to specify the URNs for other types of resources, such as documents on the Internet.

KEY CONCEPT Where URLs specify a resource based on an access method and location, Uniform Resource Names (URNs) identify a resource by name. A URN consists of a namespace identifier that indicates what type of name it contains, and a resource identifier that specifies the individual resource within the context of that namespace.

URN Resolution and Implementation Difficulties

URNs are a more natural way of identifying resources, which gives them intuitive appeal. Despite this, URNs are still not widely used, even though they have been in development for more than a decade. The main reason for this is somewhat ironic: It is because URNs are independent of location! The very characteristic that provides URNs with identification advantages over URLs also makes URNs much harder to use practically, which has led to long delays in workable URN systems.

To understand the problem, consider the example URN:isbn:0-679-73669-7. This uniquely identifies a particular book, and will always refer to it no matter where the book may be, unlike a URL. The problem is that while the URL equivalent tells us how to actually find this book, the URN does not. The same thing goes for our previous human example: Identifying Joe Xavier Zachariah by his name is more sensible than identifying him as the man living at 44 Glendale Crescent in Sydney, Australia, but at least with the latter, we know where Joe is!

In order for URNs to be useful on an internetwork, they require an additional mechanism for translating a simple URN identification string into a particular location and/or access method. In other words, we need to be able to change a URN into the equivalent of a URL, so that the resource can be found. This requirement is analogous to the problem of resolving Internet DNS domain names into IP addresses, and the same term is used to describe it: *URN resolution*.

Ideally, we want to be able to use some sort of technique where we specify the name Joe Xavier Zachariah, and we are told where Joe is so we can find him. Or, we provide the string URN:isbn:0-679-73669-7, and we are provided with a list of libraries or other places where the book can be found. The power of URNs can also be taken advantage of in such a system, by having the resolution system specify the location of a copy of the resource that is closest (in terms of network distance, cost, or other measurements) to the entity making the request.

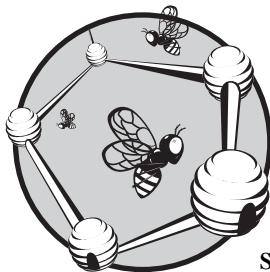
However, setting up URN resolution mechanisms is a nontrivial task. The matter of URN resolution has been the subject of much of the work on URNs over the past decade. RFC 2483, “URI Resolution Services Necessary for URN Resolution,” was published in 1999 and discusses some of the important issues in URN resolution. In October 2002, a series of RFCs, 3401 to 3405, defined a new system called the *Dynamic Delegation Discovery System (DDDS)* that was designed not just to resolve URNs, but to handle the entire class of resolution problems where an identifier is given and the output is information about where to get more information about that identifier. RFC 3406 was published at the same time, providing more information about URN namespaces.

KEY CONCEPT Since URNs identify resources by name rather than location, they are a more natural way of identifying resources than using URLs. Unfortunately, this advantage is also a disadvantage, since URNs don’t, by themselves, provide a user with the necessary information to find the resource so it can be used. A process of URN resolution must be performed to transform the URN into a set of information that allows the resource to be accessed.

Although progress on URNs has been slow, it has been steady. While it may yet be a few years before URNs are widely used, I believe it is likely that they will play an increasingly prominent role in identifying resources on the Internet in the future.

71

FILE AND MESSAGE TRANSFER OVERVIEW AND APPLICATION CATEGORIES



The purpose of networking applications is to allow different types of information to be sent between networked devices. In the world of computers, information is most often arranged into discrete units called *files*. When those files are created specifically for the purpose of communication, they are often called *messages*.

Transferring files and messages between networked computers is the most basic type of network communication. For this reason, it would not be an exaggeration to say that *file and message transfer applications* may be the most important class of internetworking applications. Some of the members of this group are so common that many people use them every day without even thinking about it.

In this brief introductory chapter, I take a general look at the concepts behind file and message transfer, and how different applications treat them. I begin with a discussion of the general concept behind files, then discuss the categories of applications that use them, contrasting message transfer with file transfer.

File Concepts

To understand the file and message transfer applications, let's first take a quick step back to look at the fundamental concept of a *file*. Simply put, a file is just a collection of information that is treated as a single unit by a computer system.

Files are stored in directories or folders in a *file system*. In modern computers, files are normally expressed as a sequence of bytes or characters, and each file is read, written, copied, or otherwise manipulated as an independent object. In addition to the data it contains, each file has associated with it file *attributes* that describe it.

For our purposes, the critical characteristic of a file is that it is a self-contained object carrying arbitrary information. Since files are the building blocks of information in computer systems, it's no surprise that the transfer of information in networking was originally defined in terms of the movement of files. Some of the protocols describing how to transfer files predate all of the modern protocols in the lower levels of TCP/IP, including Internet Protocol version 4 (IPv4), the Transmission Control Protocol (TCP), and the User Datagram Protocol (UDP). It's not the case that file transfer was an early application of internetworking, but that internetworking was invented in large part to permit file transfer!

Application Categories

Files in modern computing systems are inherently designed to be generic; they can contain any type of information. The significance of the contents of a file depends entirely on the user or software program that examines it. The TCP/IP file and message transfer protocols have in common the notion of moving files from one computer to another. Where they differ is in how the files are handled and processed. There are two basic approaches: file transfer and message transfer.

General File Transfer Applications

General transfer applications normally treat files as a “black box,” moving them from place to place and paying little or no attention to what the files contain. The TCP/IP File Transfer Protocol (FTP) and Trivial File Transfer Protocol (TFTP) fall into this category. FTP has been around in one form or another for more than 30 years now and is still widely used.

Message Transfer Applications

Other TCP/IP applications work with particular types of files, processing and interpreting them in various ways. These files are usually designed for the specific purpose of communication, and are thus called *messages*; these applications allow users to construct, send, and receive messages that fit a particular message format. There are several prominent TCP/IP messaging applications we'll examine in this book.

Electronic Mail (Email)

Email is a system that allows users to exchange “letters” (in fact, any type of document) in a manner equivalent to the conventional postal system, but with the advantages of great speed and simplicity. Email has not replaced regular mail entirely, but many people now use it for the vast majority of their correspondence.

Network News (Usenet)

Usenet is an application that is like email in that it allows users to send messages. However, while email is normally used to allow a message to be sent to one user or a small number of recipients, network news is a way for thousands of users to share messages on various topics.

Any user can contribute a message that can be seen by others, any of whom can respond. Unlike the case with email, recipients do not need to be explicitly identified, which makes network news far more suitable to communication among large groups of people who may not even know each other. This was one of the first TCP/IP applications to create something like an electronic bulletin board: an online community.

Hypertext (World Wide Web)

You probably don’t even need me to explain what the World Wide Web is, such is its great significance in modern internetworking. Hypertext moves the idea of messaging beyond the simple exchange of text messages or plain files to the notion of rich messages that can contain a variety of types of information. This includes text, graphics, multimedia, and embedded files.

Most important, hypertext allows one document to be linked to another, forming the web of related documents that led to the name World Wide Web. The Web is almost certainly the single most important TCP/IP application, used daily by millions of people.

The Merging of File and Message Transfer Methods

In recent years, a number of developments have caused the lines between applications that transfer files and applications that transfer messages to become greatly blurred. Email is no longer limited to simple text messages; it can now be used to carry general files by encoding them into text form using special methods, and even to carry hypertext documents. World Wide Web clients (browsers) continue to be enhanced to let them access other types of servers and files, and can also be used for general file transfer.

These developments mean even more functionality and flexibility for the TCP/IP user—and a bit more care required on the part of you, the TCP/IP learner.

KEY CONCEPT One of the most important groups of TCP/IP applications is the one that enables files to be moved between devices on an internetwork: file and message transfer applications. This group contains many of the common applications that TCP/IP users employ every day to communicate. It can be broken into two main categories: general file transfer applications that are used to move any type of file between devices, and message transfer applications, which allow different types of communication using special file types, such as electronic-mail messages or hypertext files.

PART III-6

TCP/IP GENERAL FILE TRANSFER PROTOCOLS

File and message transfer protocols represent the most basic type of network communication: the simple movement of blocks of data. Of the many file and message transfer methods, the most fundamental application is what I call *general file transfer*. General file transfer protocols perform one main function: allowing files to be copied from one computer to another.

Since file transfer protocols move files from place to place without much consideration of their contents, they are relatively unsophisticated compared with certain message-processing applications. However, the idea of being able to move files around is so important that general file transfer protocols were one of the very first applications in internetworking. While many people now use electronic mail or web browsers to perform the functions formerly performed exclusively using general file transfer, these older protocols are still very important and widely used, and important to understand.

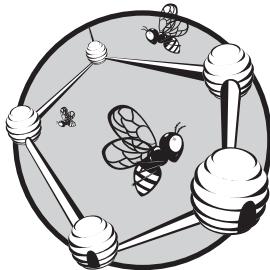
This part covers the two TCP/IP general file transfer protocols: the File Transfer Protocol (FTP) and the Trivial File Transfer Protocol (TFTP). Each is described in its own chapter.

The relationship between FTP and TFTP is similar to that of the two transport protocols, the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP) at layer 4 (discussed in Part II-8). FTP is full-featured, session-oriented, and somewhat complex. It is the more often used of the two protocols, providing a full command interface and taking advantage

of the reliability and stream-transfer functions of TCP, over which it runs. TFTP, like the UDP it uses at the transport layer, is a stripped-down version of FTP. It has far fewer commands and capabilities than FTP, but it is ideal for cases where simplicity and small software program size are important, such as in the case of embedded software in devices.

72

FILE TRANSFER PROTOCOL (FTP)



The primary general file transfer protocol in the TCP/IP suite shows its generality directly through its unqualified name: the

File Transfer Protocol (FTP). FTP is one of the most widely used application protocols in the world. It was designed to allow the efficient transfer of files between any two devices on a TCP/IP internetwork. It automatically takes care of the details of how files are moved, provides a rich command syntax to allow various supporting file operations to be performed (such as navigating the directory structure and deleting files), and operates using the Transmission Control Protocol (TCP) transport service for reliability.

In this chapter, I describe in detail the operation of FTP. I begin with an overview of FTP, a discussion of its long history, and the standards that define it. I then explain the key concepts related to FTP and how it functions. This includes a description of the FTP operational model and a look at how FTP control connections are established, how and when normal and passive data connections are used, and FTP's transmission modes and data representation methods. I then move on to the details of FTP commands and how they work,

including a discussion of FTP command groups, reply codes, and user commands. Finally, I provide a sample illustration of a user FTP session showing the internal commands used for each action.

FTP Overview, History, and Standards

The TCP/IP protocol suite as we know it today was developed in the late 1970s and early 1980s, with the watershed event probably the publishing of the version 4 standards of IP and TCP in 1980. Modern TCP/IP was the result of experimentation and development work that had been underway since the 1960s. This work included both the design and implementation of the protocols that would implement internetworks and also the creation of the first networking applications to allow users to perform different tasks.

FTP Development and Standardization

The developers of early applications conceptually divided methods of network use into two categories: *direct* and *indirect*. Direct network applications let a user access a remote host and use it as if it were local, creating the illusion that the remote network doesn't even exist (or at least, minimizing the importance of distance). Indirect network use meant getting resources from a remote host and using them on the local system, and then transferring them back. These two methods of use became the models for the first two formalized TCP/IP networking applications: Telnet for direct access (see Chapter 87) and the FTP for indirect network use.

The first FTP standard was RFC 114, published in April 1971, before TCP and IP even existed. This standard defined the basic commands of the protocol and the formal means by which devices communicate using it. At this time, the predecessor of TCP (called the *Network Control Protocol* or *NCP*) was used for conveying network traffic. There was no Internet back then. Its precursor, the ARPAnet, was tiny, consisting of only a small group of development computers.

A number of subsequent RFCs refined the operation of this early version of FTP, with revisions published as RFC 172 in June 1971 and RFC 265 in November 1971. The first major revision was RFC 354, published in July 1972, which for the first time contained a description of the overall communication model used by modern TCP and details on many of the current features of the protocol. In subsequent months, many additional RFCs were published that defined features for FTP or raised issues with it. In RFC 542, published in August 1973, the FTP specification looks remarkably similar to the one we use today, more than three decades later, except that it was still defined to run over NCP.

After a number of subsequent RFCs that defined and discussed changes, the formal standard for modern FTP was published in RFC 765, “File Transfer Protocol Specification,” in June 1980. This was the first standard to define FTP operation over modern TCP/IP and was created at around the same time as the other primary defining standards for TCP/IP.

RFC 959, “File Transfer Protocol (FTP),” was published in October 1985 and made some revisions to RFC 765, including the addition of several new commands, and it is now the base specification for FTP. Since that time, a number of other standards have been published that define extensions to FTP, better security measures, and other features.

Overview of FTP Operation

FTP was created with the overall goal of allowing indirect use of computers on a network by making it easy for users to move files from one place to another. Like most TCP/IP protocols, FTP is based on a client/server model, with an FTP client on a user machine creating a connection to an FTP server to send and retrieve files to and from the server. The main objectives of FTP were to make file transfer simple and to shield the user from implementation details of how the files are actually moved from one place to another. To this end, FTP is designed to deal automatically with many of the issues that can potentially arise due to format differences in files stored on differing systems.

To ensure that files are sent and received without loss of data that could corrupt them, FTP uses the reliable TCP at the transport layer. An authentication system is used to ensure that only authorized clients are allowed to access a server. At the same time, a feature sometimes called *anonymous FTP* allows an organization that wishes it to set up a general information server to provide files to anyone who might want to retrieve them.

After a TCP connection is established, an FTP control connection is created. Internal FTP commands are passed over this logical connection based on formatting rules established by the Telnet Protocol. Each command sent by the client receives a reply from the server to indicate whether it succeeded or failed. A data connection is established for each individual data transfer to be performed. FTP supports normal and passive data connections, allowing either the server or client to initiate the data connection. Multiple data types and file types are supported to allow flexibility for various types of transfers.

The interface between an FTP user and the protocol is provided in the form of a set of interactive user commands. After establishing a connection and completing authentication, two basic commands can be used to send or receive files. Additional support commands are provided to manage the FTP connection as well as to perform support functions such as listing the contents of a directory or deleting or renaming files. In recent years, graphical implementations of FTP have been created to allow users to transfer files using mouse clicks instead of having to memorize commands. Also, other applications can use FTP directly to move files from one place to another.

KEY CONCEPT The most important general file transfer protocol in TCP/IP is the simply named *File Transfer Protocol (FTP)*. The need to be able to move files of any type between machines is so fundamental that FTP’s history goes back more than 30 years. FTP runs over TCP to ensure that files are transferred reliably with no data loss. The protocol uses a set of *FTP commands* sent from an FTP client to an FTP server to perform file-transfer operations; the FTP server sends to the client *FTP replies* that indicate the success or failure of commands.

FTP Operational Model, Protocol Components, and Key Terminology

The standards that define FTP describe its overall operation using a simple conceptual tool called the *FTP model*. This model defines the roles of the devices that participate in a file transfer and the two communication channels that are established between them. It also describes the components of FTP that manage these channels and defines the terminology used for the components. This makes it an ideal place for us to see how FTP works in broad terms.

The Server-FTP Process and User-FTP Process

FTP is a classic client/server protocol, as mentioned earlier. However, the client is not called by that name, but rather is called the *user*. The name comes from the fact that the human user that issues FTP commands works on the client machine. The full set of FTP software operating on a device is called a *process*. The FTP software on the server is called the *server-FTP process*, while the software on the client is the *user-FTP process*.

KEY CONCEPT The FTP client is sometimes called the *user device*, since the human user interacts with the client directly. The FTP client software is called the *user-FTP process*; the FTP server software is the *server-FTP process*.

FTP Control Connection and Data Connection

A critical concept in understanding FTP is that, although it uses TCP like many other applications, it does not use just one TCP connection for all communication the way most protocols do. Instead, the FTP model is designed around two logical channels of communication between the server and user FTP processes:

Control Connection This is the main logical TCP connection that is created when an FTP session is established. It is maintained throughout the FTP session and is used only for passing control information, such as FTP commands and replies. It is not used to send files.

Data Connection Each time data is sent from the server to the client or vice versa, a distinct TCP data connection is established between them. Data is transferred over this connection. When the file transfer is complete, the connection is terminated.

Using separate channels provides flexibility in how the protocol is used, but it also adds complexity to FTP.

KEY CONCEPT Unlike most protocols, FTP does not use a single TCP connection. When a session is set up, a permanent *control connection* is established using TCP for passing commands and replies. When files or other data are to be sent, they are passed over separate TCP *data connections* that are created and then dismantled as needed.

FTP Process Components and Terminology

Since the control and data functions are communicated using distinct channels, the FTP model divides the software on each device into two logical protocol components that are responsible for each channel. The *protocol interpreter (PI)* is a piece of software that is charged with managing the control connection, issuing and receiving commands and replies. The *data transfer process (DTP)* is responsible for actually sending and receiving data between the client and server. In addition to these two elements, the user FTP process includes a third component, a *user interface*, that interacts with the human FTP user; it is not present on the server side.

Thus, two server process components and three client (user) process components are included in FTP. These components are referred to in the FTP model by specific names, which are used in the standard to describe the detailed operation of the protocol. I plan to do the same in this chapter, so I will now describe more fully the components in each device of this model, which are illustrated in Figure 72-1.

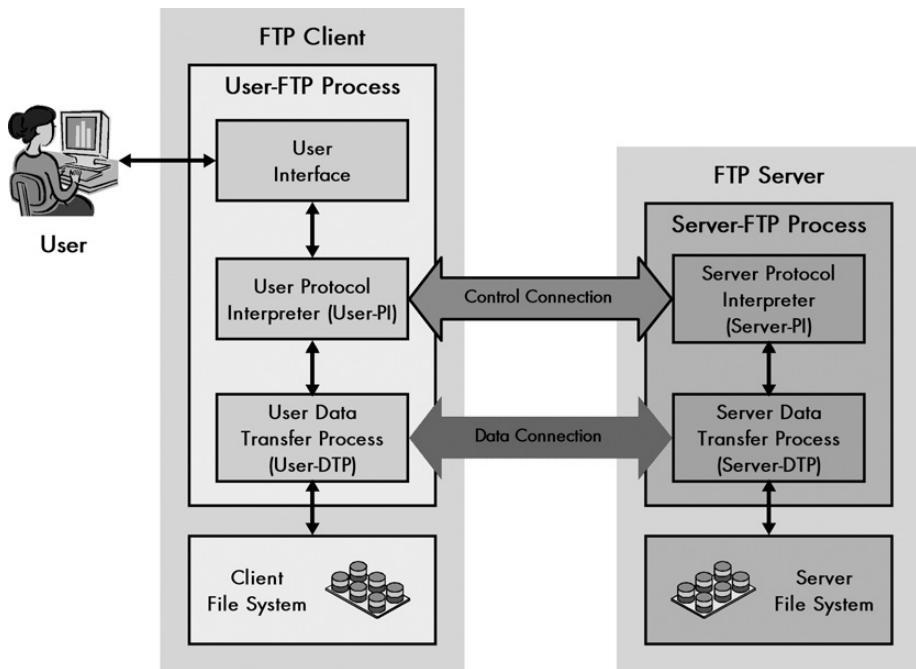


Figure 72-1: FTP operational model FTP is a client/server protocol, with communication taking place between the user-FTP process on the client and the server-FTP process on the server. Commands, replies, and status information are passed between the user-PI and server-PI over the control connection, which is established once and maintained for the session. Data is moved between devices over data connections that are set up for each transfer.

Server-FTP Process Components

The server-FTP process contains two protocol elements:

Server Protocol Interpreter (Server-PI) The protocol interpreter is responsible for managing the control connection on the server. It listens on the main reserved

FTP port for incoming connection requests from users (clients). Once a connection is established, it receives commands from the user-PI, sends back replies, and manages the server data transfer process.

Server Data Transfer Process (Server-DTP) The DTP on the server side is used to send or receive data to or from the user-DTP. The server-DTP may either establish a data connection or listen for a data connection coming from the user. It interacts with the server's local file system to read and write files.

User-FTP Process Components

The User-FTP Process contains three protocol elements:

User Protocol Interpreter (User-PI) This protocol interpreter is responsible for managing the control connection on the client. It initiates the FTP session by issuing a request to the server-PI. Once a connection is established, it processes commands received from the user interface, sends them to the server-PI, and receives replies. It also manages the user data transfer process.

User Data Transfer Process (User-DTP) The DTP on the user side sends or receives data to or from the server-DTP. The user-DTP may either establish a data connection or listen for a data connection coming from the server. It interacts with the client device's local file system.

User Interface The user interface provides a more friendly FTP interface to a human user. It allows simpler user-oriented commands to be used for FTP functions rather than the somewhat cryptic internal FTP commands, and it allows results and information to be conveyed back to the person operating the FTP session.

KEY CONCEPT The server-FTP process and user-FTP process both contain a *protocol interpreter (PI)* element and a *data transfer process (DTP)* element. The *server-PI* and *user-PI* are logically linked by the FTP control connection; the *server-DTP* and *user-DTP* are logically linked by data connections. The user-FTP process includes a third component, the *user interface*, which provides the means for the human user to issue commands and see responses from the FTP software.

Third-Party File Transfer (Proxy FTP)

The FTP standard actually defines a separate model for an alternative way of using the protocol. In this technique, a user on one host performs a file transfer from one server to another. This is done by opening two control connections: one each from the user-PI on the user's machine to the two server-PIs on the two servers. Then, a server-DTP is invoked on each server to send data; the user-DTP is not used.

This method, sometimes called *third-party file transfer* or *proxy FTP*, is not widely used today. A major reason for its lack of use is that it raises security concerns and has been exploited in the past. Thus, while it is worth mentioning, I will not be discussing it further in my coverage of FTP.

FTP Control Connection Establishment, User Authentication, and Anonymous FTP Access

You just saw how FTP uses distinct logical data and control channels that are established between an FTP client (user) and an FTP server. Before the data connection can be used to send actual files, the control connection must be established. A specific process is followed to set up this connection and thereby create the permanent FTP session between devices that can be used for transferring files.

As with other client/server protocols, the FTP server assumes a passive role in the control connection process. The server protocol interpreter (server-PI) listens on the special well-known TCP port reserved for FTP control connections: port 21. The user-PI initiates the connection by opening a TCP connection from the user device to the server on this port. It uses an ephemeral port number as its source port in the TCP connection.

Once TCP has been set up, the control connection between the devices is established, allowing commands to be sent from the user-PI to the server-PI and reply codes to be sent back in response. The first order of business after the channel is operating is *user authentication*, which the FTP standard calls the *login sequence*. This process has two purposes:

Access Control The authentication process allows access to the server to be restricted to only authorized users. It also lets the server control what types of access each user has.

Resource Selection By identifying the user making the connection, the FTP server can make decisions about what resources to make available to the user.

FTP Login Sequence and Authentication

The FTP's regular authentication scheme is quite rudimentary: it is a simple *user name/password* login scheme, shown in Figure 72-2. Most of us are familiar with this type of authentication for various types of access on the Internet and elsewhere. First, the user is identified by sending a user name from the user-PI to the server-PI using the USER command. Then, the user's password is sent using the PASS command.

The server checks the user name and password against its user database to verify that the connecting user has valid authority to access the server. If the information is valid, the server sends back a greeting to the client to indicate that the session is opened. If the user improperly authenticates (by specifying an incorrect user name or password), the server will request that the user attempt authorization again. After a number of invalid authorization tries, the server may time out and terminate the connection.

Assuming that the authentication succeeds, the server then sets up the connection to allow the type of access to which the user is authorized. Some users may have access to only certain files or certain types of files. Some servers may allow particular users to read and write files on the server, while other users may only retrieve files. The administrator can thus tailor FTP access as needed.

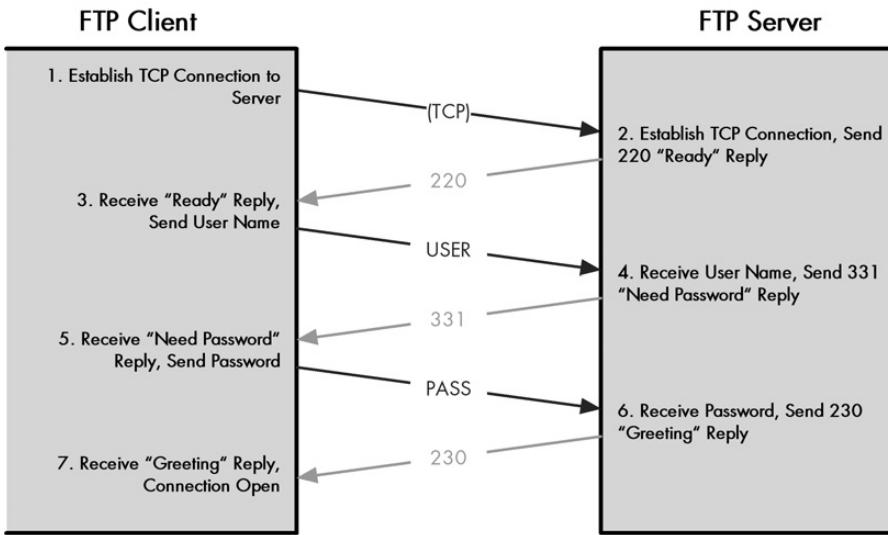


Figure 72-2: FTP connection establishment and user authentication An FTP session begins with the establishment of a TCP connection between the client and server. The client then sends the user name and password to authenticate with the server. Assuming that the information is accepted by the server, it sends a greeting reply to the client and the session is open.

Once the connection is established, the server can also make resource selection decisions based on the user's identity. For example, on a system with multiple users, the administrator can set up FTP so that when any user connects, she automatically is taken to her own home directory. The optional ACCT (account) command also allows a user to select a particular account if she has more than one.

FTP Security Extensions

Like most older protocols, the simple login scheme used by FTP is a legacy of the relatively closed nature of the early Internet. It is not considered secure by today's global Internet standards, because the user name and password are sent across the control connection in clear text. This makes it relatively easy for login information to be intercepted by intermediate systems and accounts to be compromised. RFC 2228, "FTP Security Extensions," defines more sophisticated authentication and encryption options for those who need added security in their FTP software.

KEY CONCEPT An FTP session begins with the establishment of a control connection between an FTP client and server. After the TCP connection is made, the user must authenticate with the server using a simple user name/password exchange between client and server. This provides only rudimentary security, so if more security is required, it must be implemented using FTP security extensions or through other means.

Anonymous FTP

Perhaps surprisingly, many organizations did not see the need for an enhanced level of security. These organizations, in fact, went in the opposite direction: They used FTP without any authentication at all. But why would any business want to allow just anybody access to its FTP server? The answer is pretty simple: Anyone who wants to use the server can do so to provide information to the general public.

Today, most organizations use the World Wide Web to distribute documents, software, and other files to customers and others who want to obtain them. But in the 1980s, before the Web became popular, FTP was often used to distribute such information. For example, today, if you had a 3Com network interface card and wanted to obtain a driver for it, you would go to the web server *www.3com.com*, but several years ago, you might have accessed the 3Com FTP server (*ftp.3com.com*) to download a driver.

Clearly, requiring every customer to have a user name and password on such a server would be ridiculously difficult. For this reason, RFC 1635, published in 1994, defined a use for the protocol called *anonymous FTP*. In this technique, a client connects to a server and provides a default user name to log in as a guest. Usually the names *anonymous* or *ftp* are supported. Seeing this name, the server responds back with a special message, saying something like “Guest login OK, send your complete email address as password.” The password in this case isn’t really a password; it is used simply to allow the server to log who is accessing it.

The guest is then able to access the site, though the server will usually severely restrict the access rights of guests on the system. Many FTP servers support both identified and anonymous access, with authorized users having more permissions (such as being able to traverse the full directory path and having the right to delete or rename files) and anonymous users restricted to only reading files from a particular directory set up for public access.

KEY CONCEPT Many FTP servers support *anonymous FTP*, which allows a guest who has no account on the server to have limited access to server resources. This is often used by organizations that wish to make files available to the public for purposes such as technical support, customer support, or distribution.

FTP Data Connection Management

The control channel created between the server-PI and the user-PI using the FTP connection establishment and authentication process is maintained throughout the FTP session. Over the control channel, the protocol interpreters exchange commands and replies, but not data.

Each time files or other data need to be sent between the server and user FTP processes, a data connection must be created. The data connection links the user-DTP with the server-DTP. This connection is required both for explicit file transfer actions (getting or receiving a file) and for implicit data transfers, such as requesting a list of files from a directory on the server.

The FTP standard specifies two different ways of creating a data connection, though it doesn’t really explain them in a way that is very easy to understand. The

two methods differ primarily in which device—the client or the server—initiates the connection. This may at first seem like a trivial matter, but as you’ll see shortly, it is actually quite important.

Normal (Active) Data Connections

The first method is sometimes called creating a *normal* data connection (because it is the default method) and sometimes an *active* data connection (in contrast with the passive method we will discuss in a moment). In this type of connection, the server-DTP initiates the data channel by opening a TCP connection to the user-DTP. The server uses the special reserved port number 20 (one less than the well-known control FTP port number 21) for the data connection. On the client machine, the default port number used is the same as the ephemeral port number used for the control connection, but as you’ll see shortly, the client will often choose a different port for each transfer.

Let’s use an example to see how this works. Suppose the user-PI established a control connection from its ephemeral port number 1678 to the server’s FTP control port of 21. Then, to create a data connection for data transfer, the server-PI would instruct the server-DTP to initiate a TCP connection from the server’s port 20 to the client’s port 1678. The client would acknowledge this, and then data could be transferred (in either direction—remember that TCP is bidirectional).

In practice, having the client’s control and data connection on the same port is not a good idea; it complicates the operation of FTP and can lead to some tricky problems. For this reason, it is strongly recommended that the client specify a different port number using the PORT command prior to the data transfer. For example, suppose the client specifies port 1742 using PORT. The server-DTP would then create a connection from its port 20 to the client’s port 1742 instead of 1678. This process is shown in Figure 72-3.

Passive Data Connections

The second method is called a *passive* data connection. The client tells the server to be passive—that is, to accept an incoming data connection initiated by the client. The server replies, giving the client the server IP address and port number that it should use. The server-DTP then listens on this port for an incoming TCP connection from the user-DTP. By default, the user machine uses the same port number it used for the control connection, as in the active case. However, here again, the client can choose to use a different port number for the data connection if necessary (typically an ephemeral port number).

Let’s consider our example again, with the control connection from port 1678 on the client to port 21 on the server, but this time consider data transfer using a passive connection, as illustrated in Figure 72-4. The client would issue the PASV command to tell the server it wanted to use passive data control. The server-PI would reply with a port number for the client to use—say port 2223. The server-PI would then instruct the server-DTP to listen on this port 2223. The user-PI would instruct the user-DTP to create a connection from client port 1742 to server port 2223. The server would acknowledge this, and then data could be sent and received, again in either direction.

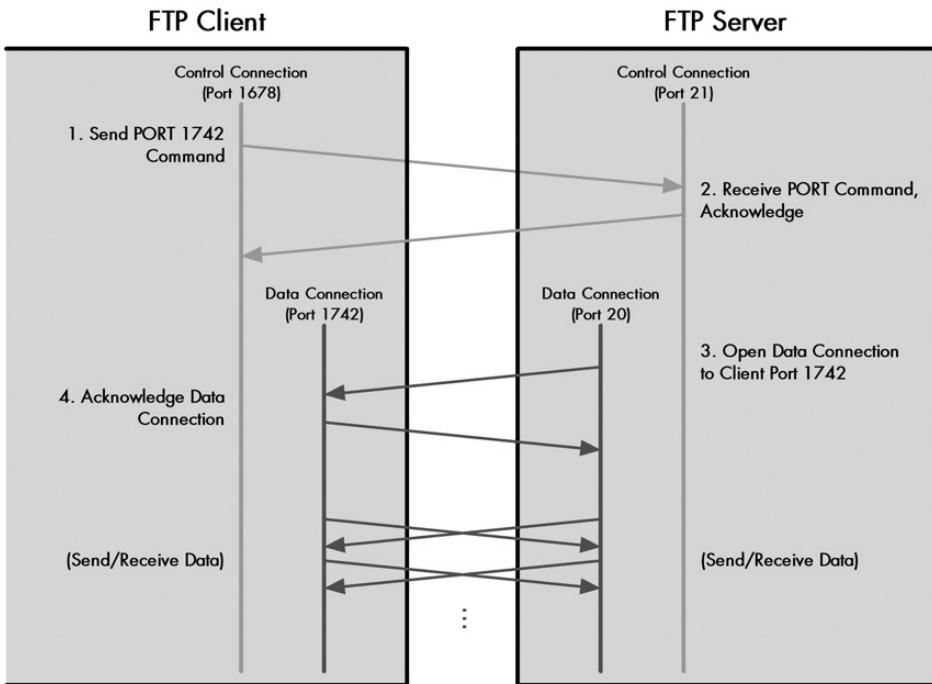


Figure 72-3: FTP active data connection In a conventional, or active, FTP data connection, the server initiates the transfer of data by opening the data connection to the client. In this case, the client first sends a PORT command to tell the server to use port 1742. The server then opens the data connection from its default port number of 20 to client port 1742. Data is then exchanged between the devices using these ports.

Efficiency and Security Issues Related to the Connection Methods

At this point, you may be wondering what the practical difference is between the active and passive connection types. I already said that in either case, the data transfer can go in both directions. So what does it matter who initiates the data connection? Isn't this like arguing over who makes a local telephone call?

The answer is related to the dreaded “S word:” *security*. The fact that FTP uses more than one TCP connection can cause problems for the hardware and software that people use to ensure the security of their systems.

Consider what is happening in the case of an active data connection, as described in Figure 72-3. From the perspective of the client, an established control connection exists from the client’s port 1678 to the server’s port 21. But the data connection is initiated by the server. So the client sees an incoming connection request to port 1678 (or some other port). Many clients are suspicious about receiving such incoming connections, since under normal circumstances, clients *establish* connections—they don’t respond to them. Since incoming TCP connections can potentially be a security risk, many clients are configured to block them using firewall hardware or software.

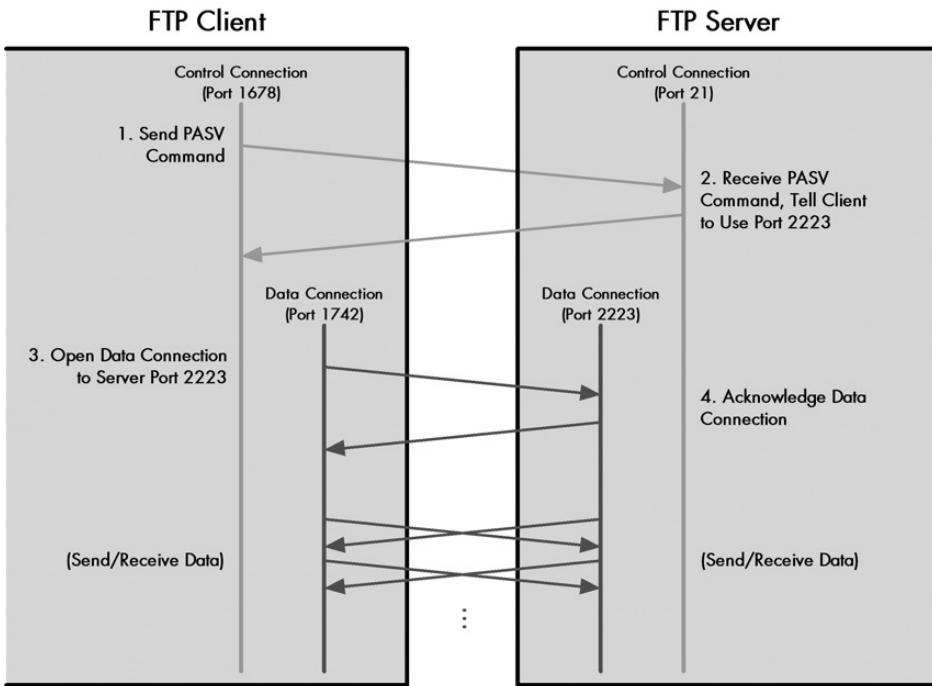


Figure 72-4: FTP passive data connection In a passive FTP data connection, the client uses the PASV command to tell the server to wait for the client to establish the data connection. The server responds, telling the client what port it should use on the server for the data transmission—in this case, port 2223. The client then opens the data connection using that port number on the server and a client port number of its own choosing—in this case, 1742.

Why not just make it so that the client always accepts connections to the port number one above the ephemeral number used for the control connection? The problem here is that clients often use different port numbers for each transfer by using the PORT command. This is done because of the rules of TCP. As I describe in Chapter 47, after a connection is closed, a period of time must elapse before the port can be used again to prevent mixing up consecutive sessions. This would cause delays when sending multiple files one after the other, so to avoid this, clients usually use different port numbers for each transfer. This is more efficient, but it means a firewall protecting the client would be asked to accept incoming connections that appear to be going to many unpredictable port numbers.

The use of passive connections largely eliminates this problem. Most firewalls have a lot more difficulty dealing with incoming connections to odd ports than outgoing connections. RFC 1579, “Firewall-Friendly FTP,” discusses this issue in detail. It recommends that clients use passive data connections by default instead of using normal connections with the PORT command to avoid the port-blocking problem.

Of course, passive data connections don’t really eliminate the problem; they just push it off onto servers. These servers now must face the issue of incoming connections to various ports. Still, it is, generally speaking, easier to deal with security issues on a relatively smaller number of servers than on a large number of clients.

FTP servers must be able to accept passive mode transfers from clients anyway, so the usual approach is to set aside a block of ports for this purpose, which the server's security provisions allow to accept incoming connections, while blocking incoming connection requests on other ports.

KEY CONCEPT FTP supports two different models for establishing data connections between the client and server. In normal, or *active*, data connections, the server initiates the connection when the client requests a transfer, and the client responds; in a *passive* data connection, the client tells the server it will initiate the connection, and the server responds. Since TCP is bidirectional, data can flow either way in both cases; the chief difference between the two modes has to do with security. In particular, passive mode is often used because many modern client devices are not able to accept incoming connections from servers.

Another point worth mentioning is that it is a significant violation of the layering principle of networks to pass IP addresses and port numbers in FTP commands such as PORT and PASV and the replies to them. This isn't just a philosophical issue. Applications aren't supposed to deal with port numbers, and this creates issues when certain lower-layer technologies are used. For example, consider the use of Network Address Translation (NAT; see Chapter 28), which modifies IP addresses and possibly port numbers. To prevent NAT from "breaking" when FTP is used, special provisions must be made to handle the protocol.

FTP General Data Communication and Transmission Modes

Once a data connection has been established between the server-DTP and the user-DTP, data is sent directly from the client to the server, or the server to the client, depending on the specific command issued. Since control information is sent using the distinct control channel, the entire data channel can be used for data communication. (These two logical channels are multiplexed at lower layers along with all other TCP and User Datagram Protocol (UDP) connections on both devices, so this doesn't actually represent a performance improvement over a single channel.)

FTP defines three different *transmission modes* (also called *transfer modes*) that specify exactly how data is sent from one device to another over an open data channel: *stream mode*, *block mode*, and *compressed mode*.

Stream Mode

In stream mode, data is sent simply as a continuous stream of unstructured bytes. The sending device simply starts pushing data across the TCP data connection to the recipient. No message format with distinct header fields is used, making this method quite different from the way many other protocols send information in discrete chunks. It relies strongly on the data streaming and reliable transport services of TCP. Since there is no header structure, the end of the file is indicated simply by the sending device closing the data connection when it is done.

Of the three modes, stream is by far the most widely used in real FTP implementations, for three main reasons:

- It is the default and also the simplest method, so it is the easiest to implement and is required for compatibility.
- It is the most general, because it treats all files as simple streams of bytes without paying attention to their content.
- It is the most efficient method because no bytes are wasted on overhead such as headers.

Block Mode

Block mode is a more conventional data transmission mode, in which data is broken into data blocks and encapsulated into individual FTP blocks, or records. Each record has a three-byte header that indicates its length and contains information about the data blocks being sent. A special algorithm is used to keep track of the transmitted data and to detect and restart an interrupted transfer.

Compressed Mode

Compressed mode is a transmission mode in which a relatively simple compression technique called *run-length encoding* is used to detect repeated patterns in the data being sent, which then represents data in such a way that the overall message takes fewer bytes. The compressed information is sent in a way similar to block mode, using a header-plus-payload record format.

Compressed mode seems on the surface to be useful. In practice, however, compression is often implemented in other places in a typical networking software stack, making it unnecessary in FTP. For example, if you are transferring a file over the Internet using an analog modem, your modem normally performs compression down at layer 1. Large files on FTP servers are also often already compressed using something like the ZIP format, meaning further compression would serve no purpose.

KEY CONCEPT FTP includes three different *transmission modes*: *stream*, *block*, and *compressed*. In stream mode, the most commonly used mode, data is sent as a continuous sequence of bytes. In block mode, data is formatted into blocks with headers. In compressed mode, bytes are compacted using run-length encoding.

FTP Data Representation: Data Types, Format Control, and Data Structures

The most general way of designing FTP would have been to make it treat all files as “black boxes.” A file would be represented as just as a set of bytes. FTP would pay no attention to what the file contained and would simply move the file, one byte at a time, from one place to another. In this scenario, FTP would seem to be very similar to the Copy command that is implemented on most file systems, which likewise creates a copy without looking into the file to see what it contains.

So what would be the problem with that, you may wonder? Well, for some types of files, this is exactly what we want, but for others, it introduces a problem. Certain types of files use different representations on different systems. If you copy a file from one place to another on the same computer using a Copy command, there is no problem, because the same representation for files is used everywhere within that computer. But when you copy it to a computer that uses a different representation, you may encounter difficulties.

The most common example of this is a type of file that may surprise you: simple text files. All ASCII text files use the ASCII character set, but they differ in the control characters used to mark the end of a line of text. On UNIX, a line feed (LF) character is used; on Apple computers, a carriage return (CR) is used; and Windows machines use both (CR+LF).

If you move a text file from one type of system to another using regular FTP, the data will all get moved exactly as it is. Moving a text file from a UNIX system to a PC as just a set of bytes would mean programs would not properly recognize end-of-line markers. To avoid this predicament, FTP moves past the idea that all files are just bytes and incorporates some intelligence to handle different types of files. The FTP standard recognizes this by allowing the specification of certain details about a file's internal representation prior to transfer.

FTP Data Types

The first piece of information that can be provided about a file is its *data type*, which dictates the overall representation of the file. Four different data types are specified in the FTP standard:

ASCII This data type defines an ASCII text file, with lines marked by some sort of end-of-line marker.

EBCDIC Conceptually, EBCDIC is the same as the ASCII type, but it is used for files using IBM's EBCDIC character set.

Image With the image data type, the file has no formal internal structure and is sent one byte at a time without any processing; this is the black box mode mentioned earlier.

Local This data type is used to handle files that may store data in logical bytes containing a number of bits other than eight. Specifying this type along with the way the data is structured allows the data to be stored on the destination system in a manner consistent with its local representation.

NOTE *The term byte conventionally refers to eight bits, but strictly speaking, the term used to describe eight bits is octet. A byte may in fact contain a number of bits other than eight on certain systems. For details, see "Binary Information and Representation: Bits, Bytes, Nibbles, Octets, and Characters" in Chapter 4.*

In practice, the two data types most often used are ASCII and image. The ASCII type is used for text files, and allows them to be moved between systems with line-end codes converted automatically. The Image type is used for generic binary files, such as graphical images, ZIP files, and other data that is represented in a universal manner. It is also often called the *binary* type for that reason.

ASCII Data Type Line-Delimiting Issues

When the ASCII data type is used, differences in internal representations between systems are handled by using a universal external representation that acts as a common language. Lines of the file being transmitted are converted by the sending FTP process from the sender's internal representation to the neutral ASCII representation used by the Telnet Protocol (NETASCII), with each line ending in CR+LF. The receiving device then converts from this neutral representation to the internal format used by the recipient file system.

For example, when using FTP to move a text file from a Macintosh to a UNIX system, each line would have the CR changed to a CR+LF for transmission over the FTP data channel. The receiving UNIX system would change each CR+LF to just LF so UNIX programs could read it properly.

Note that because of these changes, the resulting file can be bigger or smaller than the original if it is transferred between systems using ASCII mode. Also, since FTP works by converting to a neutral representation for universality, sending an ASCII file from a UNIX system to a UNIX system means each LF is changed to CR+LF for transmission, and then it's changed back to LF by the recipient. It's slightly inefficient, but not that big a deal.

It's very important that the correct data type be specified with the appropriate user command. Sending a text file between dissimilar systems without setting the ASCII mode will result in either a file that cannot be properly read on the destination or one that contains stray characters. Conversely, binary files must be sent in binary mode. If you send something like a ZIP file or a JPG graphic in ASCII mode, the FTP software will think it is a text file. It will treat the file as if it were text, and each time it encounters bytes in the file that look like CR, LF, or CR+LF, it will convert them, which you do not want. (Having the wrong data type set is a leading cause of corrupted files when using FTP to move files between PCs and UNIX systems. I know from experience!)

KEY CONCEPT FTP defines four data types: *ASCII*, *EBCDIC*, *image*, and *local*. *ASCII* and *EBCDIC* are used for text files in the ASCII and EBCDIC character sets, respectively. The *image* type is used for files with no specific structure. The *local* type is used for local representation. The *ASCII* type is important because it allows text files to be transferred successfully between file systems that may use different methods of indicating the end of a line of text. The *image* type, also called *binary*, is used for files that must be sent and received byte-for-byte with no transformation, such as executable files, graphics, and files with arbitrary formats.

FTP Format Control

For the ASCII and EBCDIC types, FTP defines an optional parameter called *format control*, which allows a user to specify a particular representation for how vertical formatting is used to describe a file. The format control option was created to handle files transferred from host devices to printers. It is not used today, to my knowledge (or if it is used, it is used only in special applications).

Three options can be used in this control:

Non Print This is the default, indicating no vertical formatting.

Telnet Format The file uses vertical format control characters, as specified in the Telnet Protocol.

Carriage Control/FORTRAN The file uses format control characters given as the first character of each line, as specified for the FORTRAN programming language.

FTP Data Structures

In addition to specifying a file's data type, it is also possible to specify the file's *data structure* in three ways:

File Structure The file is a contiguous stream of bytes with no internal structure. This is the default and is used for most types of files.

Record Structure The file consists of a set of sequential records, each of which is delimited by an end-of-record marker. The record structure can be used for ASCII text files, but these are more commonly sent with the regular file structure using the ASCII data type.

Page Structure The file contains a set of special indexed data pages. This structure is not commonly used; it was initially created for a now archaic type of computer used in the early ARPAnet.

FTP Internal Command Groups and Protocol Commands

Once a connection is established between an FTP server and user, all communication to manage the operation of the protocol takes place over the control channel. The user-PI sends *protocol commands* to the server-PI, which processes them and takes appropriate action. The server-PI responds with *reply codes* to tell the user-PI the result of the commands it issued and convey other important information.

Interestingly, the actual transmission of FTP commands over the control channel is done using specifications based on the Telnet Protocol. You may recall from the “FTP Overview, History, and Standards” section earlier in this chapter that Telnet and FTP are two of the very oldest TCP/IP applications, the former being for direct network use and the latter for indirect resource access. They were developed at around the same time, and setting up the FTP control channel to act as a type of Telnet connection is a good example of how Internet standards try not to reinvent the wheel.

FTP Command Groups and Commands

Each command is identified by a short, three- or four-letter *command code* for convenience, and the command performs a specific task in the overall functionality of FTP. Several dozen of these protocol commands are available, and to help organize them, the FTP standard categorizes them into three groups, based on overall function type:

Access Control Commands Commands that are part of the user login and authentication process, are used for resource access, or are part of general session control. See Table 72-1.

Transfer Parameter Commands Commands that specify parameters for how data transfers should occur. For example, commands in this group specify the data type of a file to be sent, indicate whether passive or active data connections will be used, and so forth. See Table 72-2.

FTP Service Commands Commands that actually perform file operations, such as sending and receiving files, and to implement support functions, such as deleting or renaming files. This is the largest group. See Table 72-3.

KEY CONCEPT FTP operation is controlled through the issuing of *protocol commands* from the FTP client to the FTP server. Each command has a three- or four-letter command code that indicates its function. The commands are organized into three groups: *access control commands* used for login and general session control, *transfer parameter commands* that control how transfers are performed, and *FTP service commands* that are used to perform actual file operations.

Since the commands are based on the Telnet specifications, they are sent as plain text, as specified by Telnet's Network Virtual Terminal (NVT) conventions. Tables 72-1, 72-2, and 72-3 list and describe the FTP internal protocol commands in the access control, transfer parameters, and service command groups. They are shown in the order that they appear in the FTP standard (RFC 959).

Table 72-1: FTP Access Control Commands

Command Code	Command	Description
USER	User name	Identifies the user attempting to establish an FTP session.
PASS	Password	Specifies the password for the user given previously by the USER command during login authentication.
ACCT	Account	Specifies an account for an authenticated user during the FTP session. Used only on systems that require this to be separately identified; most select an account automatically based on the name entered in the USER command.
CWD	Change working directory	Allows the user to specify a different directory for file transfer during an FTP session.
CDUP	Change to parent directory ("change directory up")	A special case of the CWD command that goes to the directory one level up in the server's directory structure. It is implemented separately to abstract out differences in directory structures between file systems; the user can use CDUP instead of knowing the specific syntax for navigating up the directory tree on the server.
SMNT	Structure mount	Allows the user to mount a particular file system for access to different resources.
REIN	Reinitialize	Reinitializes the FTP session, flushing all set parameters and user information. This returns the session to the state when the control connection is just established. It is, in essence, the opposite of the USER command. The next command issued is often USER, to log in a different user.
QUIT	Logout	Terminates the FTP session and closes the control connection. Note that the naming of this command was unfortunate. The REIN command is really most similar to a conventional logout command, as it terminates a logged-in user and allows another user to log in. In contrast, the QUIT command shuts down the entire session.

Table 72-2: FTP Transfer Parameter Commands

Command Code	Command	Description
PORT	Data port	Used to tell the FTP server that the client wants to accept an active data connection on a specific port number.
PASV	Passive	Requests that the FTP server allow the user-DTP to initiate passive data connections.
TYPE	Representation type	Specifies for the file to be transferred the data type (ASCII, EBCDIC, image, or local), and optionally the format control (Non Print, Telnet, or Carriage Control).
STRU	File structure	Specifies the data structure for the file (file, record, or page).
MODE	Transfer mode	Specifies the transmission mode to be used (stream, block, or compressed).

Table 72-3: FTP Protocol Service Commands

Command Code	Command	Description
RETR	Retrieve	Tells the server to send the user a file.
STOR	Store	Sends a file to the server.
STOU	Store unique	Like STOR, but instructs the server to make sure the file has a unique name in the current directory. This is used to prevent overwriting a file that may already exist with the same name. The server replies back with the name used for the file.
APPE	Append (with create)	Like STOR, but if a file with the name specified already exists, the data being sent is appended to it instead of replacing it.
ALLO	Allocate	An optional command used to reserve storage on the server before a file is sent.
REST	Restart	Restarts a file transfer at a particular server marker. Used only for block or compressed transfer modes.
RNFR	Rename from	Specifies the old name of a file to be renamed. See the RNTO command.
RNTO	Rename to	Specifies the new name of a file to be renamed. Used with the RNFR command.
ABOR	Abort	Tells the server to abort the last FTP command and/or the current data transfer.
DELE	Delete	Deletes a specified file on the server.
RMD	Remove directory	Deletes a directory on the server.
MKD	Make directory	Creates a directory.
PWD	Print working directory	Displays the current server working directory for the FTP session; shows the users where they are in the server's file system.
LIST	List	Requests a list of the contents of the current directory from the server, including both names and other information. Similar in concept to the DIR command in DOS/Windows or the ls command in UNIX.
NLST	Name list	Like LIST, but returns only the names in a directory.
SITE	Site parameters	Used to implement site-specific functions.
SYST	System	Requests that the server send to the client information about the server's operating system.
STAT	Status	Prompts the server to send an indication of the status of a file or the transfer currently in progress.

(continued)

Table 72-3: FTP Protocol Service Commands (continued)

Command Code	Command	Description
HELP	Help	Asks the server for any help information that might be useful in allowing the user to determine how the server should be used.
NOOP	No operation	Does nothing, other than prompting the server to send an "OK" response to verify that the control channel is alive.

NOTE *FTP commands are not case-sensitive, but they have been shown in uppercase for clarity in Tables 72-1, 72-2, and 72-3.*

FTP commands are all sent between FTP elements; they are not usually issued directly by users. Instead, a special set of user commands is employed for this purpose. The FTP user interface implements the link between the user and the user-FTP process, including the translation of user commands into FTP commands. We'll explore these commands later in this chapter.

FTP Replies

Each time the user-PI sends a command to the server-PI over the control connection, the server sends back a reply. FTP replies serve three main purposes:

- They serve as confirmation that the server received a command.
- They tell the user device whether or not the command was accepted, and if an error occurred, what it was.
- They communicate various types of information to the user of the session, such as the status of a transfer.

Advantages of Using Both Text and Numeric Replies

For a human user, a string of reply text would be sufficient to satisfy the requirements just mentioned, and FTP replies do include descriptive text. But having only a text string would make it difficult or impossible for FTP software on the client side to interpret results coming from the server. FTP was designed to allow software applications to interact with each other over the FTP command link. For this reason, the protocol's reply system uses *reply codes*.

FTP reply codes are three-digit numeric responses that can be easily interpreted by a computer program. They are also useful for human users who are familiar with FTP, because they communicate at a glance the results of various operations. While each FTP server implementation may differ in the text sent for each type of reply, the reply codes are used in a consistent manner based on the specifications of the FTP standard. It is, therefore, the codes that are examined to determine the results of a command; the text is just descriptive.

Reply Code Structure and Digit Interpretation

To make reply codes even more useful, they are not just assigned in a linear or random order. Rather, a special encoding scheme is used, in which each code has

three digits that each communicate a particular type of information and categorize replies. A code can be considered to be of the form xyz , where x is the first digit, y is the second, and z is the third.

The first digit indicates the success or failure of the command in general terms, whether a successful command is complete or incomplete, and whether or not an unsuccessful command should be retried. Table 72-4 shows the possible values.

Table 72-4: FTP Reply Code Format: First Digit Interpretation

Reply Code Format	Meaning	Description
1yz	Positive preliminary reply	An initial response indicating that the command has been accepted and processing is still in progress. The user should expect another reply before a new command may be sent.
2yz	Positive completion reply	The command has been successfully processed and completed.
3yz	Positive intermediate reply	The command was accepted, but processing has been delayed, pending receipt of additional information. This type of reply is used in the middle of command sequences. For example, it is used as part of the authentication sequence after receiving a USER command but before the matching PASS command is sent.
4yz	Transient negative completion reply	The command was not accepted and no action was taken, but the error is temporary and the command may be tried again. This is used for errors that may be a result of temporary glitches or conditions that may change—for example, a file being busy due to another resource accessing it at the time a request was made for it.
5yz	Permanent negative completion reply	The command was not accepted and no action was taken. Trying the same command again is likely to result in another error. For example, a request for a file that is not found on the server, or sending an invalid command like BUGU, would fall into this category.

The second digit of the reply code is used to categorize messages into functional groups. These groups are shown in Table 72-5.

Table 72-5: FTP Reply Code Format: Second Digit Interpretation

Reply Code Format	Meaning	Description
x0z	Syntax	Syntax errors or miscellaneous messages
x1z	Information	Replies to requests for information, such as status requests
x2z	Connections	Replies related to the control connection or data connection
x3z	Authentication and accounting	Replies related to login procedures and accounting
x4z	Unspecified	Not defined
x5z	File system	Replies related to the server's file system

The third digit indicates a specific type of message within each of the functional groups described by the second digit. The third digit allows each functional group to have ten different reply codes for each reply type given by the first code digit (preliminary success, transient failure, and so on).

These x , y , and z digit meanings are combined to make specific reply codes. For example, consider reply code 530, diagrammed in Figure 72-5. The first digit tells you that this is a permanent negative reply, and the second indicates that it is related to login or accounting. (It is, in fact, an error message received when a login fails.) The third digit tells you the specific type of error that has occurred.

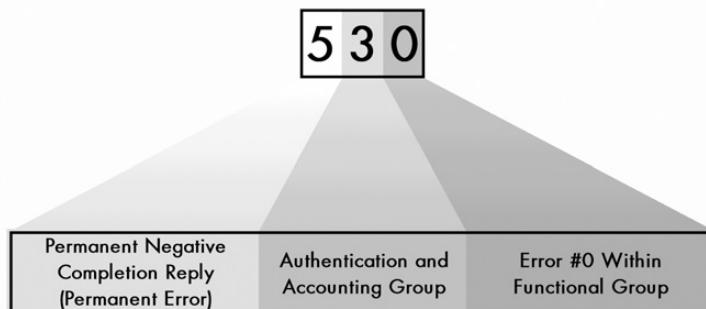


Figure 72-5: FTP reply code format This diagram shows how the three-digit FTP reply code format is interpreted. In reply code 530, the 5 indicates a permanent error, the 3 specifies that the error is related to authentication or accounting, and the 0 is the specific error type. A similar method is used for reply codes in many other TCP/IP application protocols, including the Simple Mail Transfer Protocol (SMTP) and Hypertext Transfer Protocol (HTTP).

Using encoded reply codes allows the code itself to communicate information immediately and provides a way of keeping different types of responses organized. This idea was adapted for use by several other application protocols, including the Simple Mail Transfer Protocol (SMTP) for email, the Network News Transfer Protocol (NNTP) for network news, and the Hypertext Transfer Protocol (HTTP) for the World Wide Web.

Table 72-6 contains a list of some of the more common FTP reply codes taken from RFC 959. They are shown in numerical order, along with the reply text presented as typical in that document and additional descriptive information as needed.

Table 72-6: FTP Reply Codes

Reply Code	Typical Reply Text	Description
110	Restart marker reply.	Used as part of the marker restart feature when transferring in block mode.
120	Service ready in nnn minutes.	nnn indicates the number of minutes until the service will be available.
125	Data connection already open; transfer starting.	
150	File status okay; about to open data connection.	
200	Command okay.	Sometimes the text indicates the name of the command that was successful.

(continued)

Table 72-6: FTP Reply Codes (continued)

Reply Code	Typical Reply Text	Description
202	Command not implemented, or superfluous at this site.	
211	System status, or system help reply.	Will contain system-specific status or help information.
212	Directory status.	
213	File status.	
214	Help message.	Includes help information of use to a human user of this server.
215	NAME system type.	NAME is the name of a type of operating system. Often sent as a reply to the SYST command.
220	Service ready for new user.	Sent when the command channel is established before the USER command is sent.
221	Service closing control connection.	A "goodbye" message is sent when the session is closed.
225	Data connection open; no transfer in progress.	
226	Closing data connection.	Sent after a successful file transfer or a file abort.
227	Entering Passive Mode (h1,h2,h3,h4,p1,p2).	Sent in reply to the PASV command, indicates the IP address and port to use for the data connection.
230	User logged in, proceed.	Sent after successful USER and PASS authentication. Systems often include additional greeting or other information with this code after a login.
250	Requested file action okay, completed.	The text description will provide more details about what was successfully done, such as confirming a change of directory or deleted file.
257	PATHNAME created.	PATHNAME is replaced by the path created.
331	User name okay, need password.	Intermediate result after sending USER but before sending PASS.
332	Need account for login.	
350	Requested file action pending further information.	
421	Service not available, closing control connection.	Sometimes sent if the FTP server is in the process of shutting down.
425	Can't open data connection.	
426	Connection closed; transfer aborted.	
450	Requested file action not taken. File unavailable.	The file is not available; for example, it may be locked by another user. Contrast to reply code 550.
451	Requested action aborted: local error in processing.	
452	Requested action not taken. Insufficient storage space in system.	The file system is full.
500	Syntax error, command unrecognized.	Bad or excessively long command line was sent.

(continued)

Table 72-6: FTP Reply Codes (continued)

Reply Code	Typical Reply Text	Description
501	Syntax error in parameters or arguments.	
502	Command not implemented.	
503	Bad sequence of commands.	
504	Command not implemented for that parameter.	
530	Not logged in.	Sent if authentication fails due to a bad user name or incorrect password.
550	Requested action not taken. File unavailable.	File was not found or user does not have access to it. This error code may be sent in reply to any file transfer command if the user has not successfully logged in yet. Contrast to reply code 450.
551	Requested action aborted: page type unknown.	
552	Requested file action aborted. Exceeded storage allocation.	
553	Requested action not taken. File name not allowed.	

KEY CONCEPT Each command sent by the FTP client results in a reply sent by the FTP server. FTP replies consist of a three-digit numeric *reply code*, along with a line of descriptive text. The reply code serves to standardize FTP replies, both so they can be interpreted by client software and so experienced users can see at a glance the results of a command. The reply code is structured so that the first two digits indicate the type of reply and to what category it belongs.

FTP Multiple-Line Text Replies

It is possible for a reply to contain more than one line of text. In this case, each line starts with the reply code, and all lines but the last have a hyphen between the reply code and the reply text, to indicate that the reply continues. The last line includes a space between the reply code and reply text, just like a single-line reply. This facility is often used to provide additional response information after a user logs in, via the 230 reply code. Listing 72-1 contains an example.

```
230-Welcome user to FTP server jabberwockynocky.  
230-  
230-You are user #17 of 100 simultaneous users allowed.  
230-  
230-  
230-Please see the file "faq.txt" for help using this server.  
230-  
230 Logged in.
```

Listing 72-1: FTP multiple-line text reply example

As I mentioned, the actual text string for each reply code is implementation-specific. You can sometimes find some rather humorous text strings associated with some of these error messages. For example, I tried some commands using the FreeBSD FTP client on one of my Internet accounts. I tried to send or receive a file before I was logged in, and it didn't return an error like "Requested action not taken. File unavailable." Instead, it told me this: "Login first, then I might let you do that."

FTP User Interface and User Commands

The FTP command set provides a rich, complete set of instructions for implementing FTP. A human user could employ these commands to perform file-transfer functions directly with an FTP server. But to do this, the user must have an intimate knowledge of how FTP works. The user must know exactly which commands to send at which time, and in what order.

Gaining knowledge of internal FTP commands might be a reasonable assignment for an internetworking expert, but not for a typical TCP/IP application user. For this reason, the FTP protocol defines an additional protocol component as part of the user-FTP process: the *FTP user interface*. It provides three main benefits to the FTP user:

User Friendliness The FTP user interface presents FTP to the human user in a way that is easier and simpler to use than issuing protocol commands. Instead of requiring the knowledge of all those four-letter codes, the user interface can allow functions to be performed with more intuitive human-language commands. For example, we can say get a file instead of having to use the command RETR.

Customization The command used to perform a particular function can be customized based on common parlance in the networking industry, without requiring changes to be made to FTP itself. For example, the image transfer mode is now also commonly called binary mode, so a user command called binary has been created to set this mode.

Detail Abstraction and Command Sequence Simplification A single user command can be made to issue multiple FTP protocol commands, hiding internal FTP details and making the protocol easier to use. In particular, commands that are related to the maintenance of the connection and other overhead issues that users don't want to deal with can be automated. For example, an FTP client normally issues a PASV or PORT command prior to each data transfer. The user interface can take care of issuing this command automatically prior to a RETR or STOR command when a user tells FTP to get or send a file.

Command-Line and Graphical FTP Interfaces

Traditionally, FTP clients have used a *command-line interface*. In this familiar arrangement, an FTP client is invoked and the user is automatically asked for a user name and password to establish an FTP session. Then the user is presented with a command prompt, where the user can type various FTP commands to perform different functions. Text responses from the server are displayed to the user to indicate the results of various commands. Normally, the internal protocol

commands (such as PASV and STOR) sent by the client are suppressed to avoid screen clutter, but their display can be enabled in a debug mode.

Command-line utilities are efficient, but some folks don't care for them. They are rather "old school" in the context of modern graphical operating systems and applications. Thus, many modern FTP clients are graphical in nature. They allow actions to be performed by the user clicking buttons instead of typing commands. Some FTP clients allow files to be transferred by dragging and dropping from a local file system display to one on a remote server. These make FTP even easier to use.

KEY CONCEPT The FTP *user interface* is the component on the FTP client that acts as an intermediary between the human user and the FTP software. The existence of the user interface allows FTP to be used in a friendly manner without requiring knowledge of FTP's internal protocol commands. Most FTP software uses either a *command-line interface* that understands English-like user commands or a *graphical interface*, where mouse clicks and other graphical operations are translated into FTP commands.

Typical FTP User Commands

To discover the specific commands supported by an FTP client, consult its documentation. In a command-line client, you can enter the command ? to see a list of supported commands. Table 72-7 shows some of the common commands encountered in typical FTP command-line clients, along with the typical parameters they require.

Note how many of these commands are actually synonyms, such as bye, exit, and quit. Similarly, you can use the command type ascii to set the ASCII data type or use the ascii command. This is all done for the user's convenience and is one of the benefits of having a flexible user interface that is distinct from the FTP command set.

Finally, an alternative way of using FTP is through the specification of an FTP Uniform Resource Locator (URL). While FTP is at its heart an interactive system, FTP URLs allow simple functions, such as retrieving a single file, to be done quickly and easily. They also allow FTP file references to be integrated with hypertext (World Wide Web) documents. See "URL Schemes and Scheme-Specific Syntaxes" in Chapter 70 for more on how FTP uses URLs.

Table 72-7: Common FTP User Commands

User Command	Description
account <account-name>	Sends the ACCT command to access a particular account on the server.
append <file-name>	Appends data to a file using APPE.
ascii	Sets the ASCII data type for subsequent transfers.
binary	Sets the image data type for subsequent transfers. Same as the image command.
bye	Terminates FTP session and exits the FTP client (same as exit and quit).
cd <directory-path>	Changes the remote server working directory (using CWD protocol command).
cdup	Goes to parent of current working directory.

(continued)

Table 72-7: Common FTP User Commands (continued)

User Command	Description
chmod <file-name>	On UNIX systems, changes file permissions of a file.
close	Closes a particular FTP session but user stays at FTP command line.
debug	Sets debug mode.
delete <file-name>	Deletes a file on the FTP server.
dir [<optional-file-specification>]	Lists contents of current working directory (or files matching the specification).
exit	Another synonym for bye and quit.
form <format>	Sets the transfer format.
ftp <ftp-server>	Opens session to the FTP server.
get <file-name> [<dest-file-name>]	Gets a file. If the <dest-file-name> parameter is specified, it is used for the name of the file retrieved; otherwise, the source filename is used.
help [<optional-command-name>]	Displays FTP client help information. Same as ?.
image	Sets the image data type, like the binary command.
ls [<optional-file-specification>]	Lists contents of current working directory (or files matching the specification). Same as dir.
mget <file-specification>	Gets multiple files from the server.
mkdir <directory-name>	Creates a directory on the remote server.
mode <transfer-mode>	Sets the file transfer mode.
mput <file-specification>	Sends (puts) multiple files to the server.
msend <file-specification>	Same as mput.
open <ftp-server>	Opens a session to the FTP server (same as ftp).
passive	Turns passive transfer mode on and off.
put <file-name> [<dest-file-name>]	Sends a file to the server. If the <dest-file-name> parameter is specified, it is used as the name for the file on the destination host; otherwise, the source filename is used.
pwd	Prints current working directory.
quit	Terminates FTP session and exits FTP client (same as bye and exit).
recv <file-name> [<dest-file-name>]	Receives file (same as get). If the <dest-file-name> parameter is specified, it is used for the name of the file retrieved; otherwise, the source filename is used.
rename <old-file-name> <new-file-name>	Renames a file.
rhelp	Displays remote help information, obtained using FTP HELP command.
rmdir <directory-name>	Removes a directory.
send <file-name> [<dest-file-name>]	Sends a file (same as put). If the <dest-file-name> parameter is specified, it is used as the name for the file on the destination host; otherwise, the source file name is used.
site	Sends a site-specific command to the server.
size <file-name>	Shows the size of a remote file.
status	Displays current session status.

(continued)

Table 72-7: Common FTP User Commands (continued)

User Command	Description
struct <structure-type>	Sets the file structure.
system	Shows the server's operating system type.
type <data-type>	Sets the data type for transfers.
user <user-name>	Logs in to server as a new user. Server will prompt for a password.
? [<optional-command-name>]	Displays FTP client help information. Same as help.

Sample FTP Session

Having now seen all the details of how FTP works, let's tie everything together by looking at a sample FTP session between an FTP client and server, to see FTP commands and replies in action. In this example, I will invoke FTP from a client to retrieve a text file from an FTP server, and then I'll delete the file from the server and the directory that contained it. In the process, I will issue some additional commands to illustrate more of how FTP works. I will enable debug mode in the FTP client so that for each user command, you can see the actual FTP commands generated.

Table 72-8 shows the sample FTP session, slightly simplified. The first column contains commands entered by the user (that's me, of course) on the FTP client. The second shows the actual protocol command(s) sent to the FTP server in highlighted text and the reply returned from the server to the client in plain text. The third column contains descriptive comments.

Table 72-8: Sample FTP Session

User Command	FTP Protocol Command/FTP Server Reply	Comments
ftp -d pcguide.com	Connected to pcguide.com. 220 ftp199.pair.com NcFTPd Server (licensed copy) ready. Name (pcguide.com:ixl):	This is the command to start up FTP. The -d enables debug mode. In this initial step, the TCP control connection is made and the server replies with a 220 reply code indicating that it's ready for user identification. The FTP client automatically prompts for the user name.
ixl	USER ixl 331 User ixl okay, need password.	I use ixl for user names commonly. Here, the FTP client sends the user name and the server responds, asking for the password.
****	PASS XXXX 230-You are user #1 of 300 simultaneous users allowed. 230- 230-Welcome to (<system name>) 230- 230 Logged in. SYST 215 UNIX Type: L8 Remote system type is UNIX. Using binary mode to transfer files.	I enter my password, which is sent to the FTP server, and the server authenticates me and sends back a 230 message. This tells me the login was successful. It also provides additional information. The FTP client then automatically sends a SYST command to tell me what type of system the server is using, which is UNIX in this case. The client tells me that binary mode has been selected by default; this is often the default when doing FTP from UNIX to UNIX (as I am doing here), since there is no need for ASCII mode when moving text files between similar systems.

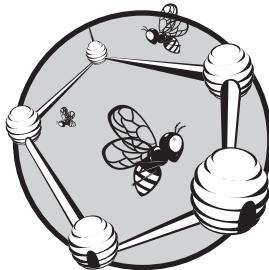
(continued)

Table 72-8: Sample FTP Session (continued)

User Command	FTP Protocol Command/FTP Server Reply	Comments
pwd	PWD 257 "/usr/home/ixl" is cwd.	I check the current working directory (cwd), which the server tells me is my own home directory on this system.
cd ftptest	CWD ftptest 550 No such directory.	I try to go to a directory called ftptest, but that was the wrong name, so I get a 550 error for my trouble. (I wasn't trying to do this; I forgot the directory name but figured I might as well show it to you anyway!)
cd ftpdemo	CWD ftpdemo 250 "/usr/home/ixl/ftpdemo" is new cwd.	I got the name right this time, and the server confirms the new working directory.
dir	PASV 227 Entering Passive Mode (ip1,ip2,ip3,ip4,193,224) LIST 150 Data connection accepted from ip5.ip6.ip7.ip8:4279; transfer starting. -rw-r--r- 1 ix1 users 16 May 22 17:47 testfile.txt 226 Listing completed.	I request a list of files from the server. The FTP client automatically issues a PASV command, and the server responds with a port number and IP address for it to use. (I have not shown the IP here for security reasons.) The directory listing is then transferred from the server to the client.
asc	TYPE A 200 Type okay.	I set ASCII mode, although I didn't really need to do that. Note that this client allowed me to abbreviate the ascii command as asc.
get testfile.txt	PASV 227 Entering Passive Mode (ip1,ip2,ip3,ip4,193,226) RETR testfile.txt 150 Data connection accepted from ip5.ip6.ip7.ip8:4283; transfer starting for testfile.txt (16 bytes). 226 Transfer completed. 17 bytes received in 0.10 seconds (0.17 KB/s)	I get the file in this demo directory using a get command. The server accepts the PASV command and sends the file. It initially sends a 150 reply as the transfer starts (initial positive reply), and then sends 226 when it is done. Note that the port numbers used here are different (for both server and client) than they were for the directory listing I did earlier.
del testfile.txt	DELE testfile.txt 250 Deleted.	I delete the original file on the server.
cdup	CDUP 250 "/usr/home/ixl" is new cwd.	I go up to the parent directory.
rmdir ftpdemo	RMD ftpdemo 250 Directory removed.	I remove the directory that the file was in.
quit	QUIT 221 Goodbye.	I end the FTP session. The quit command also automatically closes the FTP client and returns me to the UNIX shell.

73

TRIVIAL FILE TRANSFER PROTOCOL (TFTP)



In Chapter 72, you saw how the File Transfer Protocol (FTP) implements a full set of commands and reply functionalities that enables a user to perform a wide range of file movement and manipulation tasks. Although FTP is ideal as a general-purpose protocol for file transfer between computers, on certain types of hardware, it is too complex to implement easily and provides more capabilities than are really needed. In cases where only the most basic file transfer functions are required while simplicity and small program size is of paramount importance, a companion to FTP called the *Trivial File Transfer Protocol (TFTP)* can be used.

This chapter provides a description of the operation of TFTP, beginning with an overview description of the protocol, its history and motivation, and the relevant standards that describe it. I discuss its operation in general terms, cover how TFTP clients and servers communicate, and explain TFTP messaging in detail. I then discuss TFTP options and the TFTP option negotiation mechanism. The chapter concludes by showing the various TFTP message formats.

BACKGROUND INFORMATION While TFTP is a distinct protocol from FTP, explaining the former is easier when the reader is familiar with the latter. I assume that the reader has some understanding of FTP, since it is the more commonly used protocol. If you have come to this chapter prior to reading Chapter 72, I recommend at least reading the section “FTP Overview, History, and Standards” in that chapter before proceeding here.

TFTP Overview, History, and Standards

FTP is the main protocol used for the majority of general file transfers in TCP/IP internetworks. One of the objectives of the designers of FTP was to keep the protocol relatively simple, but that was possible only to a limited extent. To enable the protocol to be useful in a variety of cases and between many kinds of devices, FTP needed a fairly large set of features and capabilities. As a result, while FTP is not as complex as certain other protocols, it is still fairly complicated in a number of respects.

Why TFTP Was Needed

The complexity of FTP is partly due to the protocol itself, with its dozens of commands and reply codes, and partly due to the need of using TCP for connections and data transport. The reliance on TCP means that any device wanting to use FTP needs not only the FTP program but a full TCP implementation as well. It must handle FTP’s need for simultaneous data and control channel connections and other requirements.

For a conventional computer, such as a regular PC, Macintosh, or UNIX workstation, none of this is really an issue, especially with today’s large hard disks and fast, cheap memory. But remember that FTP was developed more than three decades ago, when hardware was slow and memory was expensive. Furthermore, even today, regular computers are not the only devices used on networks. Some networked devices do not have the capabilities of true computers, but they still need to be able to perform file transfers. For these devices, a full FTP and TCP implementation is a nontrivial matter.

One of the most notable examples of such devices are *diskless workstations*—computers that have no permanent storage, so when they start up, they cannot read a whole TCP/IP implementation from a hard disk like most computers easily do. They start with only a small amount of built-in software and must obtain configuration information from a server and then download the rest of their software from another network device. The same issue arises for certain other hardware devices with no hard disks.

The process of starting up these devices is commonly called *bootstrapping* and occurs in two phases. First, the workstation is provided with an IP address and other parameters through the use of a host configuration protocol such as the Bootstrap Protocol (BOOTP; see Chapter 60) or the Dynamic Host Control Protocol (DHCP; see Chapters 61 to 64). Second, the client downloads software, such as an operating system and drivers, that let it function on the network like any other device. This requires that the device have the ability to transfer files quickly and easily. The instructions to perform this bootstrapping must fit onto a read-only memory (ROM) chip, and this makes the size of the software an important issue—again, especially many years ago.

The solution to this need was to create a “light” version of FTP that would emphasize small program size and simplicity over functionality. This new protocol, TFTP, was initially developed in the late 1970s and first standardized in 1980. The modern version, *TFTP version 2*, was documented in RFC 783 in 1981, which was revised and published as RFC 1350, “The TFTP Protocol (Revision 2),” in 1992. This is the current version of the standard.

Comparing FTP and TFTP

Probably the best way to understand the relationship between TFTP and FTP is to compare it to the relationship between the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) at the transport layer. UDP is a simplified, stripped-down alternative to TCP that is used when simplicity is more important than rich functionality. Similarly, TFTP is a greatly simplified version of FTP that allows only basic operations and lacks some of FTP’s fancy capabilities in order to keep its implementation easy (even trivial) and its program size small.

Due to its limitations, TFTP is a complement to FTP, not a replacement for it. TFTP is used only when its simplicity is important and its lack of features is not. Its most common application is bootstrapping, as described above, though it can be used for other purposes. One specific application that the TFTP standard describes for the protocol is the transport of electronic mail (email). While the protocol supports this explicitly, TFTP is not generally used for this purpose today.

FTP and TFTP have significant differences in at least four significant areas:

Transport The comparison to TCP and UDP is apt not only based on the features/simplicity trade-off, but because FTP uses TCP for transport while TFTP uses UDP. Like TFTP, UDP is simple, and this makes the two ideal for embedding together as a hardware program set in a network device.

Limited Command Set FTP includes a rich set of commands to allow files to be sent, received, renamed, deleted, and so forth. TFTP allows files only to be sent and received.

Limited Data Representations TFTP does not include some of FTP’s fancy data representation options; it allows only simple ASCII or binary file transfers.

Lack of Authentication UDP uses no login mechanism or other means of authentication. This is again a simplification, though it means the operators of TFTP servers must severely restrict the files they make available for access. (It is also part of why TFTP specifically does not allow the client to perform dangerous file operations such as deletion.)

Overview of TFTP Operation

Communication and messaging in TFTP is very different from FTP because of the different transport layer protocols used by each. FTP makes use of TCP’s rich functionality, including its stream data orientation, to allow it to send bytes of data directly over the FTP data connection. TCP also takes care of reliable delivery of data for FTP, ensuring that files are received correctly. In contrast, since TFTP uses UDP,

it must package data into individual messages for both protocol control and data communication. TFTP must also take care of timing transmissions to detect lost datagrams and then retransmitting as needed.

TFTP servers allow connections from TFTP clients to perform file send and receive operations. Many hosts that run FTP servers will also run a separate TFTP server module. TFTP users initiate connections by starting a TFTP client program, which generally uses a command-line interface similar to that of many FTP clients; the main difference is the much smaller number of commands in TFTP.

KEY CONCEPT For situations in which the full FTP is either unnecessary or impractical, the simpler *Trivial File Transfer Protocol (TFTP)* was developed. TFTP is like FTP in that it is used for general file transfer between a client and server device, but it is stripped down in its capabilities. Rather than including a full command set and using TCP for communication, like FTP, TFTP can be used only for reading or writing a single file, and it uses the fast but unreliable UDP for transport. It is often preferred in situations where small files must be transferred quickly and simply, such as for bootstrapping diskless workstations.

The basic operation of TFTP has not changed since RFC 1350 was published, but a new feature was added to the protocol in 1995. RFC 1782, “TFTP Option Extension,” defines a mechanism by which a TFTP client and TFTP server can negotiate certain parameters that will control a file transfer prior to the transfer commencing. This allows more flexibility in how TFTP is used, adding a slight amount of complexity to TFTP, but not a great deal.

The option extension is backward-compatible with regular TFTP and is used only if both server and client support it. Two subsequent RFCs define the actual options that can be negotiated: RFC 1783, “TFTP Blocksize Option,” and RFC 1784, “TFTP Timeout Interval and Transfer Size Options.” This set of three RFCs (1782, 1783, and 1784) was replaced in 1998 by updated versions in RFCs 2347, 2348, and 2349.

TFTP General Operation, Connection Establishment, and Client/Server Communication

Since the *T* in *TFTP* stands for *Trivial*, and the protocol was specifically designed to be simple, you would think that describing how it works would, in fact, be simple, wouldn’t you? And, actually, that’s pretty much true. TFTP communication is client/server based, as discussed in the overview. The process of transferring a file consists of three main phases:

Initial Connection The TFTP client establishes the connection by sending an initial request to the server. The server responds back to the client, and the connection is effectively opened.

Data Transfer Once the connection is established, the client and server exchange TFTP messages. One device sends data, and the other sends acknowledgments.

Connection Termination When the last TFTP message containing data has been sent and acknowledged, the connection is terminated.

Connection Establishment and Identification

The matter of a connection is somewhat different in TFTP than it is with a protocol like FTP that uses TCP. FTP must establish a connection at the TCP level before anything can be done by FTP itself. TFTP, however, uses the connectionless UDP for transport, so there is no connection in the sense that one exists in TCP. In TFTP, the connection is more in a *logical* sense, meaning that the client and server are participating in the protocol and exchanging TFTP messages.

The TFTP server listens continuously for requests on well-known UDP port number 69, which is reserved for TFTP. The client chooses for its initial communication an ephemeral port number, as is usually the case in TCP/IP. This port number actually identifies the data transfer and is called a *transfer identifier (TID)*.

What's different about TFTP, however, is that the server also selects a pseudo-random TID that it uses for sending responses back to the client; it doesn't send them from port number 69. This is done because by using a unique client port number and source port number, multiple TFTP exchanges can be conducted simultaneously by a server. Each transfer is identified automatically by the source and destination port number, so there is no need to identify in data messages the transfer to which each block data belongs. This keeps the TFTP header size down, allowing more of each UDP message to contain actual data.

For example, suppose the TFTP client selects a TID of 3145 for its initial message. It would send a UDP transmission from its port 3145 to the server's port 69. Say the server selects a TID of 1114. It would send its reply from its port 1114 to the client's port 3145. From then on, the client would send messages back to server port 1114 until the TFTP session was completed.

Lock-Step Client/Server Messaging

After the initial exchange, the client and server exchange data and acknowledgment messages in *lock-step* fashion. Each device sends a message for each message it receives: one device sends data messages and waits for acknowledgments; the other sends acknowledgments and waits for data messages. This form of rigid communication is less efficient than allowing the transmitter to fire away with one data message after another, but it is important because it keeps TFTP simple when it comes to an important issue: retransmissions.

Like all protocols using the unreliable UDP, TFTP has no assurances that any messages sent will actually arrive at their destination, so it must use timers to detect lost transmissions and resend them. What is different about TFTP is that both clients and servers perform retransmission. The device that is sending data messages will resend the data message if it doesn't receive an acknowledgment in a reasonable period of time; the device sending the acknowledgments will resend the acknowledgment if it doesn't receive the next data message promptly. The lock-step communication greatly simplifies this process, since each device needs to keep track of only one outstanding message at a time. It also eliminates the need to deal with complications such as reorganizing blocks received out of order (which protocols like FTP rely on TCP to manage).

KEY CONCEPT Since TFTP uses UDP rather than TCP, no explicit concept of a connection exists as in FTP. A TFTP session instead uses the concept of a logical connection, which is opened when a client sends a request to a server to read or write a file. Communication between the client and server is performed in lock-step fashion: one device sends data messages and receives acknowledgments so it knows the data messages were received; the other sends acknowledgments and receives data messages so it knows the acknowledgments were received.

Difficulties with TFTP's Simplified Messaging Mechanism

One of the most important drawbacks with this technique is that while it simplifies communication, it does so at the cost of performance. Since only one message can be in transit at a time, this limits throughput to a maximum of 512 bytes for exchange of messages between the client and server. In contrast, when using FTP, large amounts of data can be pipelined; there is no need to wait for an acknowledgment for the first piece of data before sending the second.

Another complication is that if a data or an acknowledgment message is resent and the original was not lost but rather just delayed, two copies will show up. The original TFTP rules stated that upon receipt of a duplicate datagram, the device receiving it may resend the current datagram. So, receipt of a duplicate block 2 by a client doing a read would result in the client sending a duplicate acknowledgment for block 2. This would result in two acknowledgments being received by the server, which would in turn send block 3 twice. Then there would be two acknowledgments for block 3, and so on.

The end result of this is that once the initial duplication occurs, every message thereafter is sent twice. This has been affectionately dubbed the *Sorcerer's Apprentice bug*, after the story used as the basis of the famous scene in the movie *Fantasia*, where Mickey Mouse cuts animated brooms in half only to find that each half comes to life. This problem was fixed by changing the rules so that only the device receiving a duplicate data message may send a duplicate acknowledgment. Receipt of a duplicate acknowledgment does not result in sending a duplicate data message. Since only one of the two devices can send duplicates, this fixes the problem.

It's also worth emphasizing that TFTP includes absolutely no security, so no login or authentication process is in place. As mentioned earlier, administrators must use caution in deciding what files to make available via TFTP and in allowing write access to TFTP servers.

TFTP Detailed Operation and Messaging

You saw earlier that TFTP operation consists of three general steps: initial connection, data transfer, and connection termination. All operations are performed through the exchange of specific TFTP messages. Let's take a more detailed look now at these three phases of operation and the specifics of TFTP messaging.

Initial Message Exchange

The first message sent by the client to initiate TFTP is either a read request (RRQ) message or a write request (WRQ) message. This message serves implicitly to establish the logical TFTP connection and to indicate whether the file is to be sent from

the server to the client (read request) or the client to the server (write request). The message also specifies the type of file transfer to be performed. TFTP supports two transfer modes: *netascii* mode (ASCII text files as used by the Telnet Protocol) and *octet* mode (binary files).

NOTE Originally, a third file type option existed, called *mail mode*, but TFTP was never really designed for transmitting mail and this option is now obsolete.

Assuming no problem occurred with the request (such as a server problem, inability to find the file, and so on), the server will respond with a positive reply. In the case of a read request, the server will immediately send the first data message back to the client. In the case of a write request, the server will send an acknowledgment message to the client, telling it that it may proceed to send the first data message.

After the initial exchange, the client and server exchange data and acknowledgment messages in lock-step fashion as described earlier. For a read, the server sends one data message and waits for an acknowledgment from the client before sending the next one. For a write, the client sends one data message and the server sends an acknowledgment for it, before the client sends the next data message.

Data Block Numbering

Each data message contains a block of between 0 and 512 bytes of data. The blocks are numbered sequentially, starting with 1. The number of each block is placed in the header of the data message carrying that block and then used in the acknowledgment for that block so the original sender knows it was received. The device sending the data will always send 512 bytes of data at a time for as long as it has enough data to fill the message. When it gets to the end of the file and has fewer than 512 bytes to send, it will send only as many bytes as remain. (Interestingly, this means that if the size of the file is an exact multiple of 512, the last message sent will have zero bytes of data!)

The receipt of a data message with between 0 and 511 bytes of data signals that this is the last data message. Once this is acknowledged, it automatically signals the end of the data transfer. There is no need to terminate the connection explicitly, just as it was not necessary to establish it explicitly.

TFTP Read Process Steps

Let's look at an example that shows how TFTP messaging works. Suppose the client wants to read a particular file that is 1200 bytes long. Here are the steps in simplified form (also displayed in Figure 73-1):

1. The client sends a read request to the server, specifying the name of the file.
2. The server sends back a data message containing block 1, carrying 512 bytes of data.
3. The client receives the data and sends back an acknowledgment for block 1.
4. The server sends block 2, with 512 bytes of data.
5. The client receives block 2 and sends back an acknowledgment for it.

- The server sends block 3, containing 176 bytes of data. It waits for an acknowledgment before terminating the logical connection.
- The client receives the data and sends an acknowledgment for block 3. Since this data message had fewer than 512 bytes, it knows the file is complete.
- The server receives the acknowledgment and knows the file was received successfully.

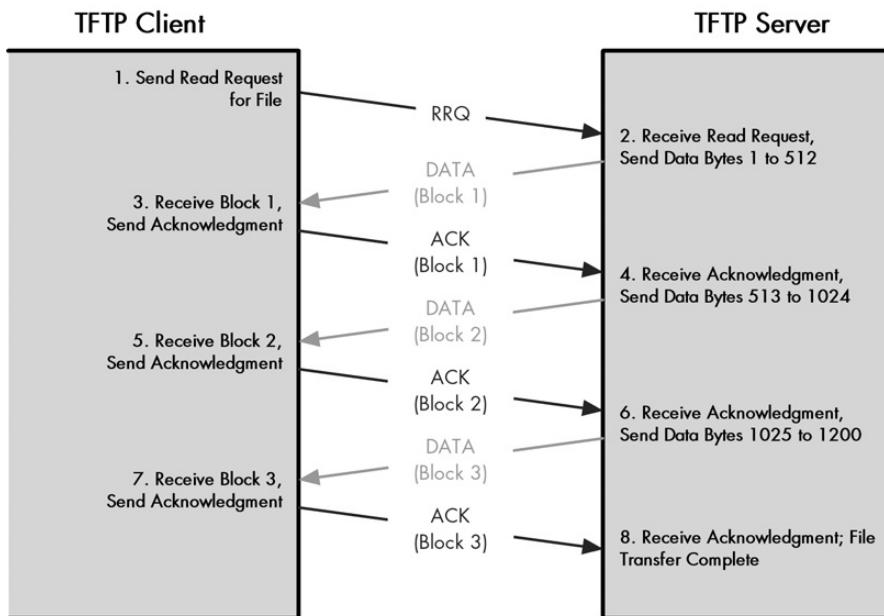


Figure 73-1: TFTP read process In this example, the client starts the process of reading a file by sending a request for it to the server. The server acknowledges this request by immediately sending a DATA message carrying block 1, containing the first 512 bytes of the file. The client acknowledges this with an ACK message for block 1. The server then sends block 2, containing bytes 513 to 1024, which the client acknowledges. When the client receives block 3, it realizes it has only 176 bytes, which marks it as the last block of the file.

TFTP Write Process Steps

Here are the steps in the same process, but where the client is writing the file (see Figure 73-2):

- The client sends a write request to the server, specifying the name of the file.
- The server sends back an acknowledgment. Since this acknowledgment is prior to the receipt of any data, it uses block 0 in the acknowledgment.
- The client sends a data message containing block 1, with 512 bytes of data.
- The server receives the data and sends back an acknowledgment for block 1.
- The client sends block 2, containing 512 bytes of data.
- The server receives the data and sends back an acknowledgment for block 2.

7. The client sends block 3, containing 176 bytes of data. It waits for an acknowledgement before terminating the logical connection.
8. The server receives block 3 and sends an acknowledgment for it. Since this data message had fewer than 512 bytes, the transfer is done.
9. The client receives the acknowledgment for block 3 and knows the file write was completed successfully.

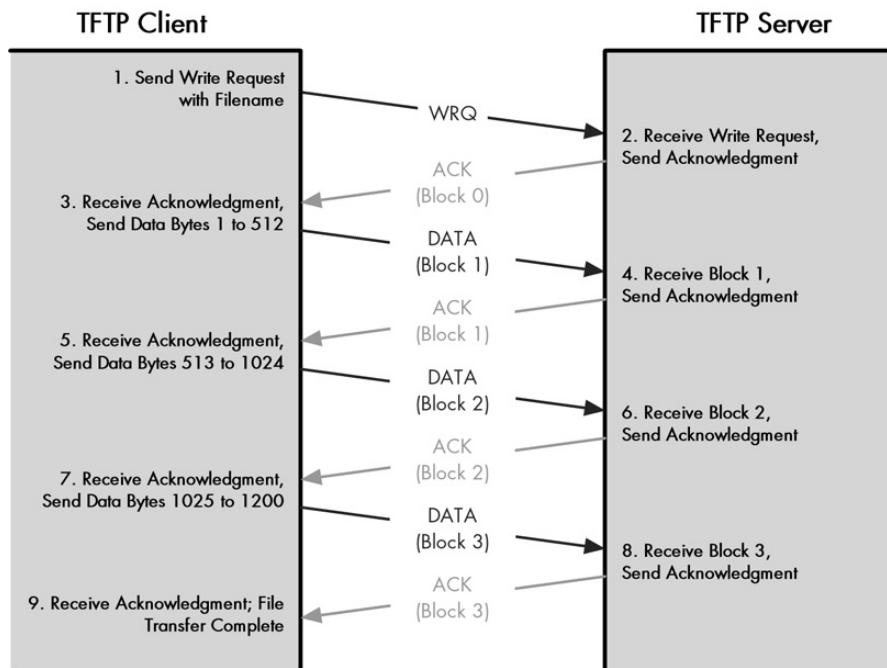


Figure 73-2: TFTP write process This example shows the client sending the same 1200-byte file to the server that it read in Figure 73-1. The client sends a write request to the server, which acknowledges it; it uses block 0 to represent acknowledgement of the request prior to receipt of any data. The client then sends blocks of data one at a time, each of which is acknowledged by the server. When the server receives block 3 containing fewer than 512 bytes of data, it knows it has received the whole file.

KEY CONCEPT A TFTP *read* operation begins with the client sending a read request message to the TFTP server; the server then sends the file in 512-byte data messages, waiting after each one for the client to acknowledge receipt before sending the next. A TFTP *write* operation starts with a write request sent by the client to the server, which the server acknowledges. The client then sends the file in 512-byte data blocks, waiting after each for the server to acknowledge receipt. In both cases, there is no explicit means by which the end of a transfer is marked; the device receiving the file simply knows the transfer is complete when it receives a data message containing fewer than 512 bytes.

If a problem is encountered at any stage of the connection establishment or transfer process, a device may reply with an error message instead of a data or acknowledgment message, as appropriate. An error message normally results in the failure of the data transfer; this is one of the prices paid for the simplicity of TFTP.

Each TFTP file transfer proceeds using the process described, which transfers a single file. If another file needs to be sent or received, a new logical communication is established, in a manner analogous to how FTP creates data connections. The main difference is that TFTP has no persistent control connection, as FTP does.

TFTP Options and Option Negotiation

One of the difficulties that designers of simple protocols and applications seem to have is keeping them simple. Many protocols start out small, but over time well-intentioned users suggest improvements that are added slowly but surely. Eventually, the program that was once lean and mean has become, shall we say, “well-marbled.” In the software industry, this is called *feature creep* and has happened to many protocols and applications.

The temptation to add features is especially strong when the program or protocol has few to begin with. Given this, the maintainers of TFTP have done a good job over the years of avoiding this pitfall. However, they did allow one new feature to be added to the protocol in 1995: the “TFTP Option Extension,” which describes how a TFTP client and server can negotiate *options* before transferring a file.

The reason for adding this capability is that the original TFTP provided no way at all for the client and server to exchange important control information prior to sending a file. This limited the flexibility of the protocol to deal with special cases, such as the transfer of data over unusual network types. The TFTP option negotiation feature allows additional parameters to be exchanged between the client and server that govern how data is transferred. It does this without significantly complicating the protocol and is backward-compatible with normal TFTP. It is used only if both client and server support it, and one device trying to use the feature will not cause problems if the other doesn’t support it.

TFTP Option Negotiation Process

The client begins the negotiation by sending a modified TFTP read request or write request message. In addition to the normal information that appears in this message (described in the “TFTP Message Formats” section later in this chapter), a list of options may also be included. Each is specified with an option code and an option value. The names and values are expressed as ASCII strings, terminated by a null character (0 byte). Multiple options may be specified in the request message.

The server receives the request containing the options, and if it supports the option extension, it processes them. It then returns a *special option acknowledgment* (*OACK*) message to the client, where it lists all the options that the client specified that the server recognizes and accepts. Any options that the client requested but the server rejects are not included in the OACK. The client may use only the options that the server accepts. If the client rejects the server’s response, it may send back an error message (with error code 8) upon receipt of the unacceptable OACK message.

The server may specify an alternative value in its response for certain options, if it recognizes the option but doesn't like the client's suggested value. Obviously, if the server doesn't support options at all, it will ignore the client's option requests and respond with a data message (for a read) or a regular acknowledgment (for a write) as in normal TFTP.

If the server did send an OACK, the client proceeds to send messages using the regular messaging exchange described in the previous section. In the case of a write, the OACK replaces the regular acknowledgment in the message dialog. In the case of a read, the OACK is the server's first message instead of the first data block that it would normally send. TFTP doesn't allow the same device to send two datagrams in a row, so a reply from the client must be received before that first block can be sent. The client does this by sending a regular acknowledgment with a block number of 0 in it—the same form of acknowledgment a server normally sends for a write.

KEY CONCEPT TFTP is supposed to be a small and simple protocol, so it includes few extra features. One that it does support is *option negotiation*, where a TFTP client and server attempt to come to agreement on additional parameters that they will use in transferring a file. The TFTP client includes one or more options in its read request or write request message; the TFTP server then sends an option acknowledgment (OACK) message listing each option the server agrees to use. The use of options when reading a file means that an extra acknowledgment must be sent by the client—to acknowledge the OACK—before the server sends the first block of the file.

For review, let's take a look at each of the four possible cases: read and write, with and without options.

The initial message exchange for a normal read (without option negotiation), as shown in Figure 73-1, is as follows:

1. Client sends read request.
2. Server sends data block 1.
3. Client acknowledges data block 1.

And so on . . .

With option negotiation, a read is as follows (see Figure 73-3):

1. Client sends read request with options.
2. Server sends OACK.
3. Client sends regular acknowledgment for block 0; that is, it acknowledges the OACK.
4. Server sends data block 1.
5. Client acknowledges data block 1.

And so on . . .

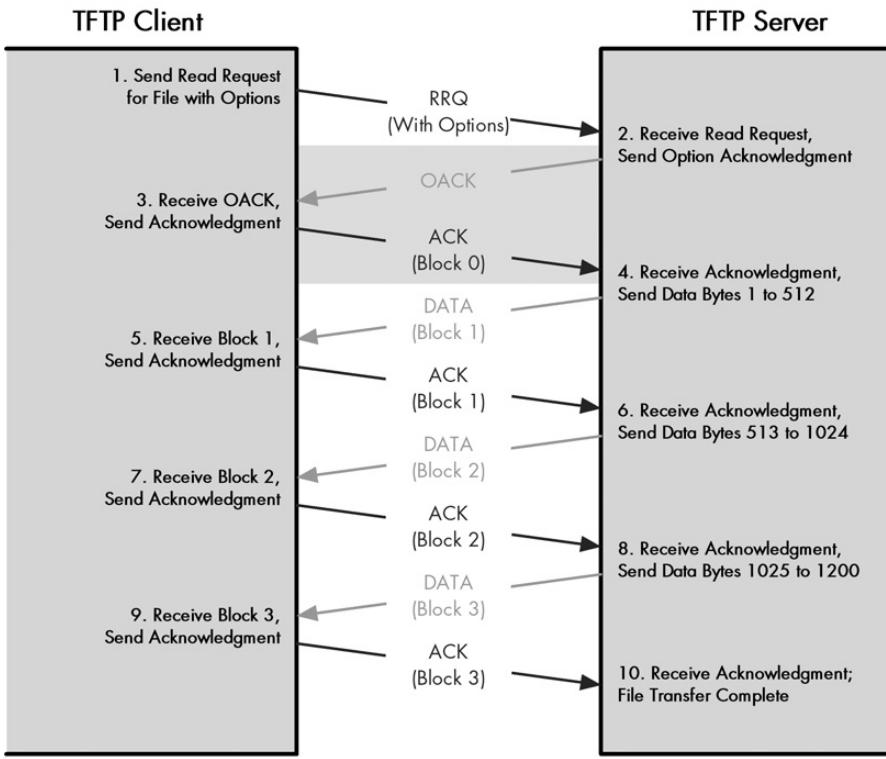


Figure 73-3: TFTP read process with option negotiation This diagram shows the same example illustrated in Figure 73-1, but with one added message exchange used for option negotiation. The client's initial read request here includes options that it wants to use for this transfer. The server responds not immediately with the first data block, but with an OACK. The client indicates receipt of the OACK by sending an acknowledgment using block 0. The server sends data block 1, and the rest of the exchange proceeds as normal.

The initial message exchange for a normal write (without option negotiation) is as follows:

1. Client sends write request.
 2. Server sends acknowledgment.
 3. Client sends data block 1.
 4. Server acknowledges data block 1.
- And so on . . .

And here's a write with option negotiation:

1. Client sends write request with options.
 2. Server sends option acknowledgment (instead of regular acknowledgment).
 3. Client sends data block 1.
 4. Server acknowledges data block 1.
- And so on . . .

TFTP Options

Table 73-1 contains a summary of the three TFTP options currently defined.

Table 73-1: TFTP Options

TFTP Option Name	TFTP Option Code (Used in Request Messages)	Defining RFC	Description
Block Size	blksize	2348	Allows the client and server to send data blocks of a size other than 512 bytes to improve efficiency or address limitations of a particular type of network.
Timeout Interval	interval	2349	Permits the client and server to agree on a specified number of seconds to use for their retransmission timers. Again, may be of value on certain networks with high latency or other special requirements.
Transfer Size	tsize	2349	Lets the device sending the file (client on a write, server on a read) tell the other device the size of the file before the transfer commences. This allows the receiving device to allocate space for it in advance.

TFTP Message Formats

Unlike FTP, all communication in TFTP is accomplished in the form of discrete messages that follow a particular message format. The reason why TFTP and FTP are so different in this regard is because of the different transport protocols they use. FTP uses TCP, which allows data to be streamed a byte at a time; FTP also makes use of a dedicated channel for commands. TFTP runs on UDP, which uses a conventional header/data formatting scheme.

The original TFTP standard defines five different types of messages: read request (RRQ), write request (WRQ), data (DATA), acknowledgment (ACK), and error (ERROR). The TFTP option extension feature defines a sixth message: option acknowledgment (OACK). Of these six messages, the first two share the same message format. The only common field in every TFTP message is the operation code (Opcode), which tells the recipient of the message what type it is.

TFTP's message formats are different than those used for certain other protocols, because many of the fields in TFTP are variable in length. Usually, variable-length fields in messages are expressed using a preceding length field that specifies the length of the variable-sized field. Instead, TFTP sends such fields as strings of ASCII characters using netascii, the Telnet version of ASCII. The end of the string is marked by a zero byte. The exception to this is the data field in data messages, the content of which depends on the transfer mode.

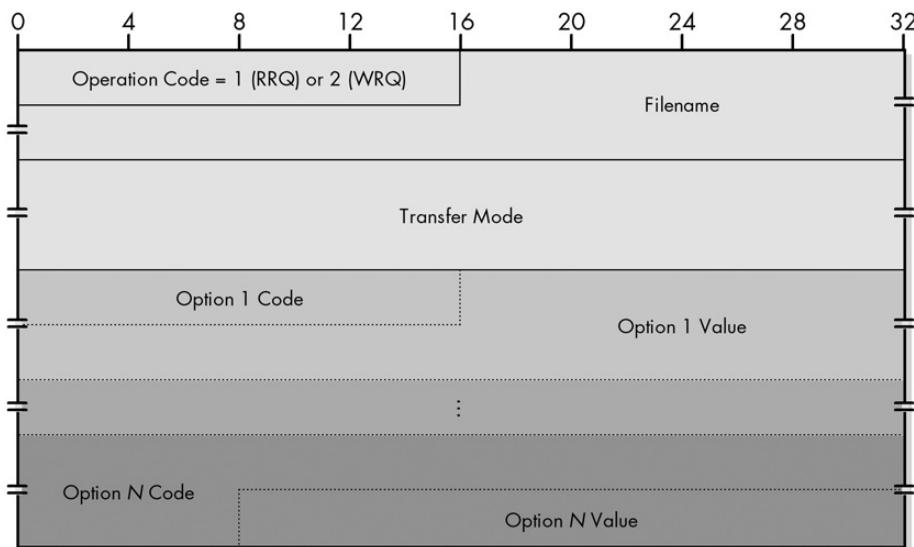
The remainder of the chapter contains the details on each of the TFTP messages.

Read Request and Write Request Messages

These messages use a common message format, described in Table 73-2 and shown graphically in Figure 73-4.

Table 73-2: TFTP RRQ/WRQ Message Format

Field Name	Size (Bytes)	Description
Opcode	2	Operation Code: Specifies the TFTP message type. A value of 1 indicates a RRQ message, while a value of 2 is a WRQ message.
Filename	Variable	The name of the file to be read or written.
Mode	Variable	Transfer Mode: The string netascii or octet, zero-terminated.
Options	Variable	When the client supports TFTP options, it will encode them in sequence following the Mode field. Each option consists of two variable-length subfields. The optN subfield is the option code for option N, containing a string specifying the name of the option; currently, blksize, interval, and tsize are supported. The valueN subfield is the option value for option N, containing the value the client is requesting for this option. (Note that this is a zero-terminated string just like other TFTP variable-length fields, even for a numeric value.)

**Figure 73-4:** TFTP RRQ/WRQ message format

Data Messages

Data blocks are sent using the simplified format shown in Table 73-3 and Figure 73-5.

Table 73-3: TFTP Data Message Format

Field Name	Size (Bytes)	Description
Opcode	2	Operation Code: Specifies the TFTP message type. A value of 3 indicates a data message.
Block #	2	Block Number: The number of the data block being sent.
Data	Variable	Data: 0 to 512 bytes of data.

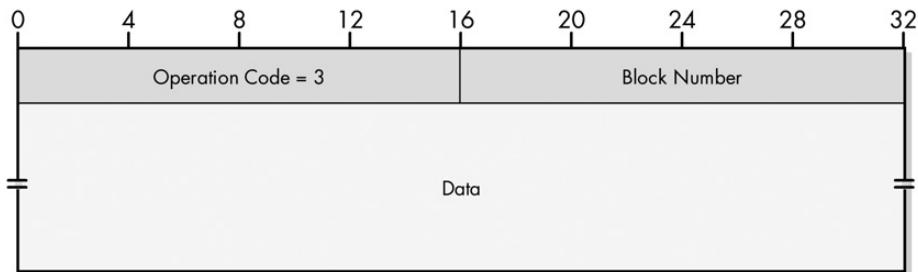


Figure 73-5: TFTP data message format

Acknowledgment Messages

Acknowledgments have the simplest format of any TFTP message, as you can see in Table 73-4 and Figure 73-6.

Table 73-4: TFTP Acknowledgment Message Format

Field Name	Size (Bytes)	Description
Opcode	2	Operation Code: Specifies the TFTP message type. A value of 4 indicates an ACK message.
Block #	2	Block Number: The number of the data block being acknowledged; a value of 0 is used to acknowledge receipt of a write request without options or to acknowledge receipt of an OACK.

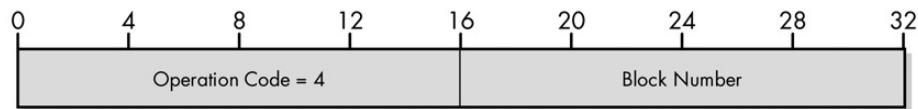


Figure 73-6: TFTP acknowledgment message format

Error Messages

Error messages can be sent by either the client or server in cases where a problem is detected in the communication. They have the format indicated in Table 73-5 and Figure 73-7.

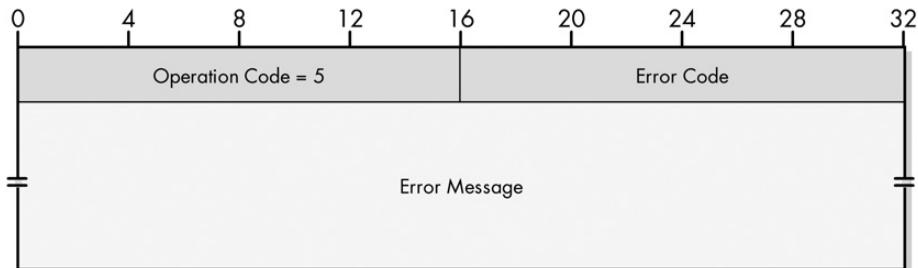


Figure 73-7: TFTP error message format

Table 73-5: TFTP Error Message Format

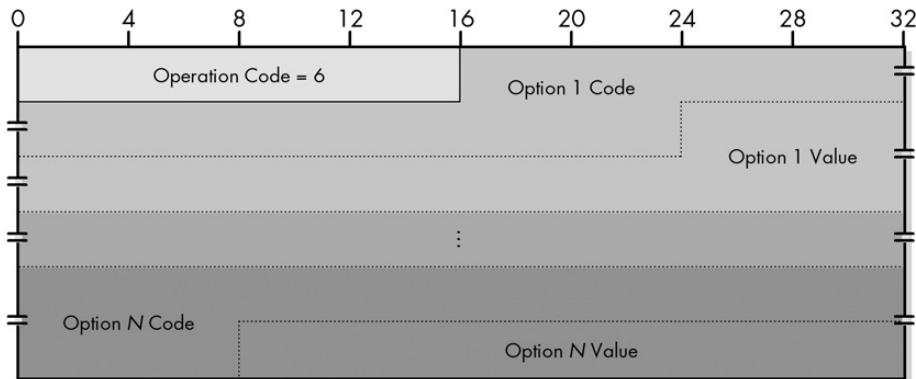
Field Name	Size (Bytes)	Description
Opcode	2	Operation Code: Specifies the TFTP message type. A value of 5 indicates an error message.
Error Code	2	A numeric code indicating the type of message being communicated. Values 0 to 7 are defined by the TFTP standard, while value 8 was added by the TFTP option extension: 0 = Not defined; see error message field for details 1 = File not found 2 = Access violation 3 = Disk full or allocation exceeded 4 = Illegal TFTP operation 5 = Unknown transfer ID 6 = File already exists 7 = No such user 8 = Client transfer termination due to unacceptable option negotiation
Error Msg	Variable	Error Message: A descriptive text error message string, intended for “human consumption,” as the standard puts it.

Option Acknowledgment Messages

OACK messages are used to acknowledge receipt of TFTP options. They are structured as shown in Table 73-6 and Figure 73-8.

Table 73-6: TFTP OACK Message Format

Field Name	Size (bytes)	Description
Opcode	2	Operation Code: Specifies the TFTP message type. A value of 6 indicates an OACK message.
Options	Variable	A list of options being acknowledged by the server. Each option consists of two variable-length subfields. The optN subfield is the option code for option N, containing a string specifying the name of the option, copied from the RRQ or WRQ message. The valueN subfield is the option value for option N, containing the acknowledged value for the option, which may be the value that the client specified or an alternative value, depending on the type of option.

**Figure 73-8:** TFTP OACK message format

PART III-7

TCP/IP ELECTRONIC MAIL SYSTEM: CONCEPTS AND PROTOCOLS

It is common for human beings to create systems that are reminiscent of ones with which they are already familiar. We are all accustomed to using the regular mail system to send letters and other documents from our location to recipients anywhere that the postal system serves. Naturally, one of the first applications of internetworks was to create an electronic version of this conventional mail system that would allow messages to be sent in a similar manner, but more quickly and easily. Over the course of many years, an *electronic mail system* for TCP/IP was created and refined. It is now the most widely used means of electronic messaging in the world.

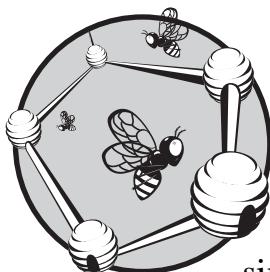
In this part, I describe TCP/IP electronic mail (email) in detail, in five chapters that discuss electronic mail concepts and the various components and protocols that comprise the overall TCP/IP email system. The first chapter provides an overview of TCP/IP email and discusses the way that it is used and the different protocols and methods that make up the system. The second chapter discusses how email messages are addressed, and the third chapter covers standard and special formats for email messages. The fourth and fifth chapters describe the TCP/IP protocols that implement email functionality. These include the Simple Mail Transfer Protocol (SMTP), which is responsible

for the delivery of email, and several protocols and methods used for mailbox access and mail retrieval, including the Post Office Protocol version 3 (POP3) and the Internet Message Access Protocol (IMAP).

This discussion focuses primarily on the mechanisms used for email composition, delivery, and access in modern internetworks. In the email overview in Chapter 74, I mention some techniques used in the past for TCP/IP email, but only briefly for historical completeness and to contrast these methods to the ones presently used.

74

TCP/IP ELECTRONIC MAIL SYSTEM OVERVIEW AND CONCEPTS



Electronic mail (email) in the TCP/IP protocol suite is not implemented as just a single protocol or technology. Rather, it is a complete system that contains a number of related components that work together. These include standards defining methods for addressing and message formatting and a number of protocols that play different functions in implementing email messaging. Before proceeding to examine each of these pieces, it makes sense to start with an overview of the system as a whole.

In this chapter, I provide an introductory look at TCP/IP email to help you understand the system, how it works, and how different components fit into it. I begin with an overview and history of email and its implementation in TCP/IP. I provide a general overview of the steps involved in the email communication process, concluding with a more specific discussion of the communication model used in TCP/IP and the roles played by various TCP/IP devices and protocols in the sending and receiving of email.

TCP/IP Electronic Mail System Overview and History

The need to communicate is as old as humanity itself. Thousands of years ago, communication was of necessity almost exclusively local. Messages were primarily oral, and even when in writing, they were rarely delivered a great distance. Most people never traveled far from their homes and rarely communicated with those distant from themselves. But even in ancient times, leaders used messengers to send short pieces of critical information from place to place. It was slow and unreliable, but some messages were important enough that an effort to communicate often had to be made despite the difficulties.

Advances in transportation led to advances in communication capability, eventually resulting in the creation of physical mail systems. Today, these systems have evolved to the point at which anyone in the developed world can send a letter or package to just about anyone else. Reliability has vastly improved, despite all the jokes people make about the postal service. Speed is also much better than it used to be, with messages now taking days to reach their destination instead of weeks or months.

Waiting even days for a message to get from one place to another is pretty slow by the standards of our modern world. For this reason, one of the most natural applications of networks was to use them as a replacement for the physical transportation of messages from one place to another. Transforming mail from a physical process to an electronic one yields enormous benefits, including greatly increased communication speed, the ability to send one message to multiple recipients instantly, and the ability to get nearly instantaneous feedback upon receipt of a message.

The Early Days of Email

The idea behind email is not only as old as computer networks, but it actually predates internetworking. The first email systems were implemented on traditional mainframe computers. These were single large computers accessed by many users simultaneously through connected terminals. An email system on a mainframe consisted of a set of software running on the mainframe that implemented the entire email system. Each user simply had a mailbox that resided on this machine, and mail was delivered by moving messages from one mailbox to the next. Users sent and received mail through a user-interface program.

Such an early email system was useful for local communication but not for sending messages to a person in another organization. Mainframe email is somewhat analogous to local mail being sent by one resident of a town to another. There is no way to send mail to a person in a distant town without the infrastructure in place for delivery.

The power of internetworking is what really enables email to become a universal method of communication. Internetworks link together systems the way the postal service's fleet of airplanes and vehicles link together post offices. Mail is sent from user to user over the underlying technology of the internetwork. Since TCP/IP is the most commonly used internetworking protocol suite, and the modern Internet uses TCP/IP to tie together systems across the globe, it is the vehicle used for sending email.

History of TCP/IP Email

As with some other file and message transfer protocols, email on TCP/IP actually goes back to before TCP/IP and the Internet formally existed. The first protocols for email were developed during the days of the ARPAnet. Prior to the creation of email, several Internet RFCs, such as RFC 95 (yes 95, two digits—we are going back a long way here) and RFC 155, describe physical mailing lists that were used for distributing documents in the early 1970s. It was this need to send documents that likely made TCP/IP pioneers realize the usefulness of an electronic messaging system, using the technology they were themselves creating.

The first Internet document describing email was probably RFC 196, published in 1971. It describes the *Mail Box Protocol*, a very rudimentary message transfer method using the predecessors of TCP/IP. This protocol was designed for the specific purpose of sending documents for remote printing. In those days, it was not as common for people to use computers at interactive terminals as it is today, but the idea of electronically mailing documents was the same. The Mail Box Protocol was revised several times in 1971.

In the mid-1970s, developers began working on a more comprehensive method of implementing email on the fledgling Internet. The technique was originally described using a number of existing application layer transfer protocols, including the File Transfer Protocol (FTP). In 1980 the “Mail Transfer Protocol (MTP)” was published in RFC 772. This was the first precursor of today’s TCP/IP email and was defined using principles from the Telnet Protocol as well as FTP.

During the time that email protocols were being developed in the 1970s, mail was being exchanged between host systems using a variety of techniques. One of the most common used was the *Unix-to-Unix Copy Protocol (UUCP)*, which was designed to allow files to be transferred between UNIX systems, moving them from one connected system to the next. UUCP was also used for communicating Usenet newsgroup articles and other files.

In 1981, the modern TCP/IP email era came into being with the definition of the *Simple Mail Transfer Protocol (SMTP)*. SMTP described in detail how mail could be moved directly or indirectly from one TCP/IP host to another without the need to use FTP or another file transfer method. (SMTP has its own detailed history and discussion in Chapter 77.) Other complementary specifications were created at around the same time, which formalized or defined other components and elements of the system. We’ll explore these pieces of the puzzle throughout the rest of this chapter.

Overview of the TCP/IP Email System

One of the most important general concepts in the modern email system is that a distinction is made between protocols that deliver email between SMTP hosts on the internetwork and those that let users access received mail on their local hosts. To continue the postal mail analogy, different protocols are used for sending mail between post offices and for home delivery. As you’ll see, this was done intentionally to make it possible to send mail to users, even if they are not connected to the

Internet when the mail is sent. This decoupling is critical, as it enables delayed communication, where mail can be sent when the sender wants to transmit it and received when the recipient wants to read it.

KEY CONCEPT One of the most important TCP/IP applications is the internetworking equivalent of the real-world postal delivery system, commonly called *electronic mail* or *email*. The history of email goes back to the very earliest days of TCP/IP's development. Today, it is used by millions of people every day to send both simple and complex messages around the world. TCP/IP email is not a single application, but rather a complete system that includes several protocols, software elements, and components.

Over the years, the basic components defined in the early 1980s have not changed substantially, but how they are used has evolved and improved. Early email delivery involved the use of route specifications by one SMTP host to dictate how mail was to be delivered through intermediate systems; today, the Domain Name System (DNS) makes much of that obsolete, facilitating nearly immediate direct mail delivery in most cases. Early email supported only simple text, but we can now send graphical images, programs, and other file attachments in email. Modern high-speed Internet connections and updated access protocols allow email to be the realization of the ultimate goal of nearly instantaneous communication, even across continents.

TCP/IP Email Communication Overview

You've just seen that TCP/IP email is implemented as a complete system, with a number of elements that perform different portions of the complete job of email communication. These included a standard message format, a specific syntax for recipient addressing, and protocols to both deliver mail and allow access to mailboxes from intermittently connected TCP/IP clients.

To help set the groundwork for examining these components, here, I provide an overview of the complete end-to-end process of email communication, so you can see how everything works. I will show the basic steps in simplified form and continue the analogy to the regular mail system for comparison.

The modern TCP/IP email communication process consists of five basic steps:

1. **Mail Composition** A user begins the email journey by creating an email message. The message contains two sections: the *body* and the *header*. The body of the message is the actual information to be communicated. The header contains data that describes the message and controls how it is delivered and processed. The message must be created so that it matches the standard message format for the email system so that it can be processed (see Chapter 76). It must also specify the email addresses of the intended recipients for the message (see Chapter 75). By way of analogy to "snail mail," the body of the message is like a letter, and the header is like the addressed and stamped envelope into which the letter is placed.

2. **Mail Submission** Email is different from many other internetworking applications in that the sender and receiver of a message do not necessarily need to be connected to the network simultaneously, nor even continuously, to use it. The system is designed so that after composing the message, the user decides when to submit it to the email system so it can be delivered. This is done using SMTP (see Chapter 77). This is analogous to dropping off an envelope at the post office or to a postal worker picking up an envelope from a mailbox and carrying it to the local post office to insert into the mail delivery stream.
3. **Mail Delivery** The email message is accepted by the sender's local SMTP system for delivery through the mail system to the destination user. Today, this is accomplished by performing a DNS lookup of the intended recipient's host system and establishing an SMTP connection to that system. SMTP also supports the ability to specify a sequence of SMTP servers through which a message must be passed to reach a destination. Eventually, the message arrives at the recipient's local SMTP system. This is like the transportation of the envelope through the postal system's internal "internetwork" of trucks, airplanes, and other equipment to the intended recipient's local post office.
4. **Mail Receipt and Processing** The local SMTP server accepts the email message and processes it. It places the mail into the intended recipient's mailbox, where it waits for the user to retrieve it. In our physical analogy, this is the step at which the recipient's local post office sorts mail coming in from the postal delivery system and puts the mail into individual post office boxes or bins for delivery.
5. **Mail Access and Retrieval** The intended recipient periodically checks with its local SMTP server to determine whether any mail has arrived. If so, the recipient retrieves the mail, opens it, and reads its content. This is done using a special mail access protocol or method (see Chapter 78). To save time, the access protocol and client email software may allow the user to scan the headers of received mail (such as the subject and sender's identity) to decide which mail messages to download. This is analogous to the step where mail is physically picked up at the post office or delivered to the home.

KEY CONCEPT TCP/IP email communication normally involves a sequence of five steps, each of which is analogous to a portion of the journey taken by a regular letter through the postal system. First, email is *composed* (written); second, it is *submitted* to the email system; third, it is *delivered* to recipient's server; fourth, it is *received and processed*; and fifth, it is *accessed and retrieved* by its recipient.

In some cases, not all of these steps are performed. If a user is sending email from a device that is already an SMTP server, then step 2 can be omitted. If the recipient is logged in to a device that is also an SMTP server, step 5 will be skipped, as the user can read mail directly on the server. Thus, in the simplest case, all that occurs is composition, delivery, and receipt; this occurs when one user of a dial-up UNIX host sends mail to another. In most cases today, however, all five steps occur.

TCP/IP Email Message Communication Model

The purpose of the email system as a whole is to accomplish the transmission of messages from a user of a TCP/IP internetwork to one or more recipients. To accomplish this, a special method of communication is required that makes the email system quite different from that used by most other protocols. To understand what I mean by this, just consider the difference in communication between sending a letter and making a phone call.

Most TCP/IP protocols are analogous to making a phone call in this respect: The sender and the receiver must both be on the network at the same time. You can't call someone and talk to him if he isn't around to answer the phone. (I'm ignoring answering machines and voice mail of course!) Most TCP/IP protocols are like this. To send a file using FTP, for example, you must make a direct connection from the sender's machine to the recipient's machine. If the recipient's machine is not on the network at the exact time that the sender's machine is, no communication is possible. For email, immediate communication of this sort is simply unacceptable.

As with real-world snail mail, Joe wants to be able to put a message into the system at a time that is convenient for him, and Ellen wants to be able to receive Joe's mail at a time that works for her. For this to work, email must use a "send and forget" model, just like real mail, where Joe drops the "envelope" into the email system and it eventually arrives at its destination.

This *decoupling* of the sender and receiver is critical to the design of the email system. This is especially true because many of the users of Internet email are not on the Internet all the time. Just as you wouldn't want real mail to be rejected if it arrived when you are not home, you wouldn't want email to not be delivered if you are not on the Internet when it arrives. Similarly, you may not want to be connected to the Internet for the entire time it takes to write a message, especially if you have access to the Internet for only a limited amount of time each day.

Also critical to the entire email system is that idea that communication is between specific *users*, not between particular machines. This makes email inherently different from many other types of communication on TCP/IP internetworks. You'll see more of why this is important when we look at email addressing in Chapter 75.

To allow the type of communication needed for email, the entire system is designed to facilitate the *delayed delivery* of email messages from one user to another. To see how this works, let's look again at the example communication we discussed earlier—but this time, consider the roles of the different devices in the exchange (as shown in Figure 74-1):

Sender's Client Host The sender composes an email message, generally using a mail client program on her local machine. The mail, once composed, is not immediately sent out over the Internet; it is held in a buffer area called a *spool*. This allows the user to be "unattached" for the entire time that a number of outgoing messages are created. When the user is done, all of the messages can be sent at once.

Sender's Local SMTP Server When the user's mail is ready to be sent, she connects to the internetwork. The messages are then communicated to the user's designated local SMTP server, normally run by the user's Internet service provider

(ISP). The mail is sent from the client machine to the local SMTP server using SMTP. (It is possible for the sender to be working directly on a device with a local SMTP server, in which case sending is simplified.)

Recipient's Local SMTP Server The sender's SMTP server sends the email using SMTP to the recipient's local SMTP server over the internetwork. There, the email is placed into the recipient's incoming mailbox (or inbox). This is comparable to the outgoing spool that existed on the sender's client machine. It allows the recipient to accumulate mail from many sources over a period of time and retrieve them when it is convenient.

Recipient's Client Host In certain cases, the recipient may access her mailbox directly on the local SMTP server. More often, however, a mail access and retrieval protocol, such as Post Office Protocol (POP3) or Internet Message Access Protocol (IMAP), is used to read the mail from the SMTP server and display it on the recipient's local machine. There, it is displayed using an email client program, similar to the one the sender used to compose the message in the first place.

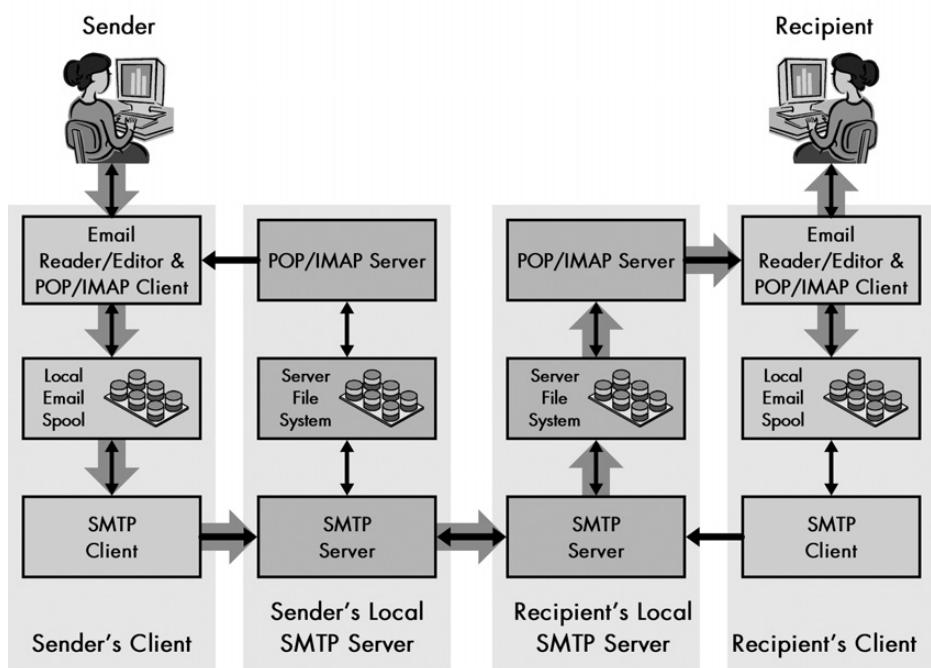


Figure 74-1: Email communication model This diagram shows the four devices that are involved in a typical email communication between two users. Each device consists of a number of different elements, which communicate as indicated by the black arrows. Note the inherent asymmetry, because the method used to send an email from a user is not the same as that used to retrieve it from the server. The large, shaded arrows show a typical transaction: the sender composes mail and it goes to her local email spool. It is sent to the sender's local SMTP server using SMTP, and then to the recipient's SMTP server, where it goes into that user's inbox. It is then retrieved, usually using a protocol such as POP or IMAP.

Protocol Roles in Email Communication

You may have noticed that SMTP is used for most of this communication process. In fact, if the recipient uses a machine that runs SMTP software, which is common for those using dial-up UNIX shell Internet access, the process of sending email uses SMTP exclusively. SMTP servers must, however, always be available on the Internet and ready to accept mail. Most people access the Internet using devices that aren't always online or that don't run SMTP software. That is why the last step, email access and retrieval, is usually required.

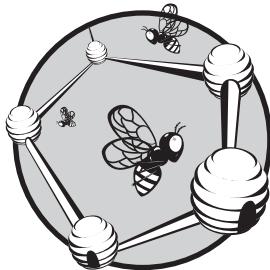
It might have been possible to define the email system so that this last step of communication was carried out using SMTP as well, which would mean the entire system used the same protocol. However, SMTP was tailored for the specific purpose of transporting and delivering email, not for remote mailbox access. It made more sense to leave the function of mailbox access to dedicated, separate protocols. This not only allows these protocols to be tailored to the needs of email recipients, but it also provides flexibility by giving users more than one option for how email is retrieved. I discuss email access protocols and methods in Chapter 78, highlighting the two most common protocols: POP and IMAP.

The three protocols discussed here—SMTP, POP3, and IMAP—get lead billing on the TCP/IP email stage, but they rely on two other elements to play supporting roles. The first is a method of addressing email messages to ensure that they arrive at their destinations. The second is the set of message formats used to encode messages and control how they are delivered and used. These elements don't usually get as much attention as they deserve, but they do here, as I have devoted the next two chapters to them.

KEY CONCEPT One of the critical requirements of an email system is that the sender and receiver of a message are not required to be on the system at the time mail is sent. TCP/IP therefore uses a communication model with several devices that allow the sender and recipient to be *decoupled*. The sender's client device spools mail and moves it to the sender's local SMTP server when it is ready for transmission; the email is then transmitted to the receiver's SMTP server using SMTP. The email can remain on the recipient's server for an indefinite period of time. When the recipient is ready to read it, he retrieves it using one or more of a set of mail access protocols and methods, the two most popular of which are POP and IMAP.

75

TCP/IP ELECTRONIC MAIL ADDRESSES AND ADDRESSING



The entire concept of electronic mail (email) is based on an analogy: sending electronic messages is like sending paper messages. The analogy works well, because email was indeed intended to be like regular mail, only with the advantages of the technological era: speed and flexibility.

One of the many similarities between email and regular mail is the need for *addressing*. For a message to be delivered, it is necessary for the sender to specify the recipient and provide a reasonable amount of information to indicate how and where the recipient can be reached. In TCP/IP email, a standard *electronic mail address* format is used for this, and support is also provided for alternative addressing schemes that may be used in special cases.

In this chapter, I describe how email messages are addressed. I begin with a discussion of standard email addressing in TCP/IP and how those addresses are used to determine where email should be sent. I then provide a brief discussion of historical and special email addresses that you may encounter from time to time. I also discuss the use of email address books

(aliases) and how multiple recipients may be addressed, and I provide an overview of electronic mailing lists, one of the earliest ways in which electronic group communication was implemented.

TCP/IP Email Addressing and Address Resolution

All communication on an internetwork requires some way of specifying the identity of the intended recipient of the communication. Most application protocols, such as the File Transfer Protocol (FTP) and Hypertext Transfer Protocol (HTTP), use conventional TCP/IP constructs—IP addresses and port numbers—to specify the destination of information to be sent. The IP address normally identifies a particular host computer, and the port number indicates a software process or application running on that computer.

Email, however, uses a model for communication that differs from most applications. As you saw in the discussion of the email model in the previous chapter, one element that sets email apart from many other systems is that communication is *user-oriented*. Email is not sent from one machine to another, as a file is transferred using FTP. Instead, it is sent from one user to another. This is critical to the operation of the entire system. For one thing, it allows someone to retrieve email that has been sent from any number of different client computers. This allows the recipient to receive email even when traveling, for example.

Since email messaging is user-based, the addressing scheme must also be user-based. We cannot use conventional IP addresses and ports, so we need a distinct system that specifies two primary pieces of information: who the user is and where the user is located. These are, of course, analogous to a name and address on a regular mail envelope.

The idea of a user name is relatively straightforward, but identifying the location of the user is not. In regular mail, an address refers to a physical place. It would have been possible to define email addresses in the same way; that is, to have an email address refer to the user's client machine. However, recall the other important characteristic of email delivery: it is indirect and based on the concept of a user's local Simple Mail Transfer Protocol (SMTP) server holding received messages until they can be retrieved. The machine that the user employs to access his email may not even routinely be connected to the Internet, and it may thus not be easy to identify. And we also want a user to be able to access email from multiple machines.

For all of these reasons, we want addresses to identify not the user's specific location at any particular time, but the place where the user's permanent mailbox lives—on the user's SMTP server, which is permanently connected to the Internet.

Standard DNS-Based Email Addresses

In TCP/IP, the system used for identifying servers (and other machines) is the *Domain Name System (DNS)*. DNS is a big system and is described in Part III-1 of this book, which you should read if you want to learn more. For now, it is important that you realize that in DNS, all devices on the Internet are arranged into a device-naming hierarchy, and any device can be identified using a *domain name* consisting of a series of text labels separated by dots.

The complete TCP/IP address consists of two components: a user name specification and a domain name specification. The two are connected together using the at symbol (@) to form the TCP/IP email address syntax that most of us are familiar with today: <username>@<domainname>.

The format of <domainname> follows the syntax rules of DNS (see Chapter 53), which specify that it can contain only numbers and digits for each label, and periods to separate the labels. The format of <username> is slightly less restrictive, allowing special characters such as the underscore (_). Other special characters and spaces are also allowed in the <username> if they are surrounded by quotation marks (or otherwise marked as being part of the name, such as through the use of an escape character). Domain names are case-insensitive; user names may be case-sensitive, depending on the system.

An example of a valid email address is cmk@athena.mit.edu (an address I used when I was in school many years ago). Here, *cmk* is my user name (my initials); *athena.mit.edu* is the name of the host where I was receiving mail; and *athena* is a particular system at Massachusetts Institute of Technology (*mit*), an educational institution that uses the *.edu* top-level domain.

It is also possible to specify an email address using an Internet-standard Uniform Resource Locator (URL). This allows a link to be embedded in a hypertext (Web) document; when clicked, it invokes an email client to send mail to a user. Email URLs are created by preceding the address by the special URL scheme string *mailto:*, like this: *mailto:cmk@athena.mit.edu*.

Special Requirements of Email Addresses

Having an email address refer to a user's local SMTP server provides a great deal of flexibility compared to having the address mention a specific client computer. But this doesn't provide enough flexibility to handle the following situations:

- An organization may want to use generic addresses that do not specify the name of the SMTP server to handle email, to make it easier for senders or clients to remember an email address. For example, if someone knew my real name and that I was at MIT, it would be easier for him to remember my email address as cmk@mit.edu than to remember cmk@athena.mit.edu.
- An administrator may change which machines handle mail over a period of time. This would mean all the users' email addresses would have to be changed, too—and most of us know what a pain that is. For example, if I moved from the athena machine to the jabberwocky machine, my old address would need to be changed to cmk@jabberwocky.mit.edu. But if the address were just cmk@mit.edu, a server change would not affect the address.
- In larger organizations, it might be desirable to have multiple servers share the load of handling incoming email.

To address all of these requirements, the DNS system includes a feature that was specifically designed to support email addressing. A special *mail exchange (MX)* record can be set up that specifies which SMTP server should be used for mail arriving at a particular domain name. If properly configured, this allows considerable

flexibility to handle the cases described above, and more. For more details, please see the description of the MX record and DNS electronic mail support in Chapter 56.

KEY CONCEPT Some form of addressing is required for all network communication; since email is *user-oriented*, email addresses are also based on users. In modern TCP/IP email, standard addresses consist of a *user name*, which specifies the recipient, and a *domain name*, which specifies the DNS domain where the user is located. A special DNS *mail exchange (MX)* record is set up for each domain that accepts email, so a sending SMTP server can determine what SMTP server it should use to send mail to a particular recipient.

Suppose, for example, that I am the owner of the pcguide.com domain name. Email can be sent to me at pcguide.com, but the mail is not actually stored on any server by that name. Instead, it is redirected to the real server where my inbox is located. This allows me to handle all incoming mail to pcguide.com, regardless of where my mailbox is actually located.

DNS is also significant in that its MX resource records eliminate the need to relay email from one SMTP server to the next to deliver it. In modern TCP/IP, it is possible to send email directly from the sender's SMTP server to the recipient's server, making communication faster and more efficient. This is also discussed in Chapter 56.

TCP/IP Historical and Special Email Addressing

TCP/IP email has been so successful that it is arguably the most important worldwide standard for electronic messaging. The widespread acceptance of email is tied inextricably to that of TCP/IP and the Internet as a whole. Since most organizations want to be part of the Internet, they connect to it and use its technologies, including DNS, which is the basis for TCP/IP email addresses. In turn, the use of simple DNS-style email addresses (*user@domain*) encourages further use of email because people find it conceptually easy to decide how to send messages.

TCP/IP is not, however, the only email system around. Over the years, several other networks have developed email systems. Due to the fact that the Internet is the largest internetwork in the world, TCP/IP email has often been used as a clearinghouse of sorts to link together some of these different email mechanisms. This is called *gatewaying*, and it allows someone using a non-SMTP email system to interact with someone using TCP/IP, and vice versa. Gatewaying is complex, in part because email systems use different ways of addressing mail. Let's take a look at a couple of these systems and how they interact with TCP/IP.

FidoNet Addressing

One of the earliest independent email systems was the *FidoNet*, which has been around for a long time and is still in use today. FidoNet is a worldwide network connected using modems and proprietary protocols; it is, in essence, a “competitor” to the global TCP/IP Internet. I put *competitor* in quotes because FidoNet and the

TCP/IP Internet are not really comparable in terms of number of users and the kinds of applications they support, but they are similar in overall objectives: worldwide electronic communication.

FidoNet users are identified using four numbers that specify the FidoNet *zone*, *net*, *node*, and *point* (*connection point*). These addressing elements are used for sending mail on this system, which again is completely distinct from TCP/IP. However, to allow communication between TCP/IP and FidoNet, the FidoNet administrators have set up a gateway system that allows mail to be sent to FidoNet using TCP/IP-style domain names. This style of mapping was also used by other systems with proprietary mail address formats to allow them to interface with the Internet.

For example, if a user was on machine 4, node 205, net 141, zone 1 (North America), the FidoNet address would be 1:141/205:4. The equivalent domain name would be p4.f205.n141.z1.fidonet.org and could be used for TCP/IP-style *user@domain* addressing.

UUCP-Style Addressing

An older address style commonly associated with email was the UUCP-style address. The *Unix-to-Unix Copy Protocol (UUCP)* was commonly used years ago to route mail before SMTP became widely deployed (again, it is still used, just not as much as before). The addresses in this system are specified as a path of hosts separated by exclamation marks (!). The path dictates the route that mail takes to get to a particular user, passing through a series of intermediate machines running UUCP. For example, if mail to joe at the host joesplace had to go through three hosts—host1, host2, and host3, the address would be host1!host2!host3!joesplace!joe. Since the slang term for an exclamation mark is *bang*, this came to be called *bang path* notation.

The use of UUCP-style notation was sometimes mixed with TCP/IP-style domain name address notation when DNS came into use. So you might have seen something like host1!user@domain. There was some confusion in how exactly to interpret such an address: Does it mean to send mail first to host1 and then to user@domain? Or does it mean to first send it to the domain, which then goes to user at host1? There was no universal answer to this. The problem was mostly resolved both by the decrease in use of UUCP and the move on the part of UUCP systems to TCP/IP-style domain name addressing.

Addressing for Gateways

You may encounter email addresses that appear as if multiple TCP/IP addresses have been nested using unusual punctuation. For example, you may see something like this: user%domain1.com@subdomain.domain2.edu. This is a way of addressing sometimes seen when email gateways are used; it will cause the mail to be sent to user%domain1.com at subdomain.domain2.edu. The address then is interpreted as user@domain1.com. However, again, not all systems are guaranteed to interpret this the same way.

Email gatewaying is not a simple matter in general, and as you can see, one reason is the use of different email address styles and the problems of consistency in how complex hybrid addresses are interpreted. However, as the Internet expands

and TCP/IP becomes more widespread, it is becoming less and less common to see these older special address formats in use. They are becoming more and more a historical curiosity (unless you happen to use one of them).

TCP/IP Email Aliases and Address Books

Email is analogous to regular mail but superior to it due to two main advantages of digital and electronic communication. One advantage is *speed*, which is why modern Internet users have come up with the slang term *snail mail* to refer to the regular postal service. But the other advantage, *flexibility*, is also essential. Email allows you to send messages easily in ways that would be cumbersome with regular mail. And one of the ways this flexibility can be seen is in addressing.

The first way that email addressing is flexible is that most email clients support advanced features that allow users to specify the identity of recipients in convenient ways. While TCP/IP addressing is fairly straightforward, remembering the addresses of everyone you know is difficult. In the real world, we use address books to help us remember addresses. With email, we can do the same by allowing email software to associate a name with an email address.

This is usually done in one of two ways. In old-fashioned, text-based email such as that used on many UNIX systems, name and address association is performed using *aliases*. These are short forms for email addresses that save typing. For example, I often send email to my wife, Robyn, but I'm too lazy to type in her complete address all the time. So I have defined an alias for her in my email program called simply *r*. I enter the mail command and specify the alias *r* as the intended recipient, and it expands her email address for me.

In modern graphical email systems, aliases aren't used. Instead, an *electronic address book* is usually implemented, which is the equivalent of the paper address book. The difference is that there is no manual copying; you just choose the name from the list using your mouse.

Multiple Recipient Addressing

Another advantage of email addressing is that it allows the easy specification of multiple recipients. With paper mail, sending a message to ten people means you need ten copies of the message, ten envelopes, and ten stamps. With email, you just list the recipient addresses separated by a comma in the recipient list: <*user1@domain1*>,<*user2@domain2*>,<*user3@domain3*>. A separate copy is mailed to each recipient. Of course, aliases and/or address books can be used to specify each recipient here as well, making this even simpler.

Since email makes it so easy for one person to send information to a set of others, so-called *one-to-many* messaging, it was also one of the first ways in which electronic group communication was implemented. Prior to email, sharing information in a group setting required either a face-to-face meeting or a telephone conference call. In both cases, all parties must be present simultaneously, and a cost is involved, especially when the parties are geographically distant.

With email, a group of individuals can share information without needing to meet or even be available at the same time. Suppose a group comprises four individuals: Ellen, Joe, Jane, and Tom. Ellen has a proposal that she wants to discuss. She sends it to Joe, Jane, and Tom. Each recipient will read it at a time convenient for him or her. Each person can then reply back to the group. For example, Tom might have a comment on the proposal, so he just sends it to Ellen, Joe, and Jane. Most email clients include a *group reply* feature for this purpose.

Mailing Lists

In larger groups, communication by typing the addresses of each recipient becomes cumbersome. Instead, a *mailing list* is used. The list is created by an individual termed the *list owner* and contains the email addresses of all the members of the group. A special *list address* is created, which functions just like a regular email address. However, when anyone sends mail to this special address, it is not simply deposited into a mailbox. It is instead intercepted by special software that processes the message and sends it out automatically to all recipients on the list. Any recipient can reply to the list address, and all members will receive the reply.

Many other ways can be used by groups to share information today, such as using World Wide Web bulletin boards, Usenet newsgroups, Internet Relay Chat (IRC), and so forth. Some of these have a lot of features that make mailing lists seem unsophisticated by comparison. Despite this, electronic mailing lists are still very popular, largely because email is the most universal Internet communication method and one of the easiest methods to use.

Many thousands of mailing lists are in use on the Internet, covering every subject imaginable. Each list differs in a number of regards, including the following five aspects:

Implementation Usually, some sort of special software is used to allow the list owner to manage it, add and remove users, and set parameters that control how the list operates. These programs are commonly called *robots* or *listservs* (*list servers*). One of the more common listservs is named *Majordomo*. Some mailing lists are actually implemented and managed using the Web. (The line between Internet applications continues to get more and more blurry.)

Subscription Rules and Technique Some mailing lists are open to anyone who wishes to join; others are by invitation only. Most allow a new subscriber to join automatically using software; others require the list owner to add new members.

Management Method and Style The list owner decides what is acceptable for discussion on the list. Some lists are *moderated*, meaning that all submissions to the list must be approved by the list owner before they are sent to list members. Some lists allow mail to the list from nonmembers, and some do not.

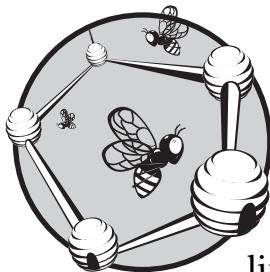
Culture Like all groups, groups of people on mailing lists have their own culture, interesting personalities, and other unique traits. New members of a list are often encouraged to read the list and not send to it for a while until they become accustomed to it and how it works. This is similar to the acclimation process for Usenet newbies (Usenet is covered in Chapter 85).

Special Features Some lists support special features, such as the ability to subscribe in *digest mode* (where messages are collected into large digests to reduce the number of individual messages sent) or to access messages on the Web.

KEY CONCEPT One of the many benefits of email is that it is easy to send a message to many people at once, simply by specifying several recipient addresses. This permits easy and simple *group communication*, because each recipient can then send a group reply to respond to each of the people who were sent the original message. Electronic *mailing lists* provide a more formalized way for groups to exchange ideas and information; many thousands of such lists are in use on the Internet.

76

TCP/IP ELECTRONIC MAIL MESSAGE FORMATS AND MESSAGE PROCESSING: RFC 822 AND MIME



The advantages of using computers for communication are obvious, but some limitations are also imposed by the use of computer technology. When I compare electronic mail (email) to regular mail, I always point out that email is much faster and more flexible in how it can be delivered, and this is true. An email message can reach its destination in seconds, while a conventional letter can take days.

However, one significant drawback of using computers to communicate is that they are not very flexible in figuring out how to understand messages. Consider that anyone can put any type of letter, memorandum, or other communication in an envelope and send it to you, and assuming you know the language in which it is written, you can open the envelope and probably understand it. You can figure out how to deal with a date that appears in an unusual place in the letter, or your name appearing at the top compared to

the bottom, or the body of the message being structured in different ways. You can read notes that are typed or handwritten in pen, pencil, or crayon—as long as the letters are decipherable, you can understand what is being said.

Computers are not good at deciphering such subtleties. It is for that reason that email systems must rely on standard message formats to ensure that all messages have the same form and structure. This then makes it possible for all devices in the email system to read and understand one another's messages, to enable TCP/IP email to work on many different types of computers.

In this chapter, I describe the two formats used for TCP/IP email messages: the main TCP/IP email standard, which is called the RFC 822 format after the standard that defines it, and the Multipurpose Internet Mail Extensions (MIME) standard, which greatly expands the ability of email to support the communication of different types of information by defining methods of encoding various media and non-English-language text into the standard RFC 822 format.

TCP/IP Email RFC 822 Standard Message Format Overview

One of the most astute observations I have read about internetworking applications asserts that their usefulness is proportional to the number of people who use them. TCP/IP email is a great example. It is a powerful communication method in large part because almost everyone with a computer today participates in the system. The more people who sign on to use email, the more powerful it becomes.

The creators of TCP/IP email realized that people who use the system would employ many different types of hardware and software. To ensure that everyone was able to understand all email messages, regardless of who sent them, they specified a common message format for email messages. This format doesn't have an official fancy name; it is simply known by the name of the standard that defines it: the RFC 822 message format.

RELATED INFORMATION *This discussion will make certain references to the discussion of the Simple Mail Transfer Protocol (SMTP; see Chapter 77) but was designed so that you could read it prior to learning about SMTP without getting confused.*

The primary protocol for delivering email is Simple Mail Transfer Protocol (SMTP). For this reason, the message format used for TCP/IP email could be considered SMTP's protocol message format, not unlike the special message formats discussed for other protocols, such as the Internet Protocol (IP) and the Transmission Control Protocol (TCP). However, the TCP/IP email message format is used not only by SMTP, but by all protocols and applications that deal with email. This includes the mail-access protocols Post Office Protocol (POP3) and Internet Message Access Protocol (IMAP), as well as others. It was also intended to be potentially usable by other non-TCP/IP mail delivery protocols. Perhaps for this reason, the TCP/IP email format was not specified as part of the SMTP itself, RFC 821, but was specified in a companion document, RFC 822. Both were published in 1982.

Development of the RFC 822 Message Format Standard

The history of the message format used in TCP/IP starts long before 1982. It was originally defined as the format for passing text messages on the Internet's precursor, the ARPAnet, in the early 1970s. The format was refined several times, leading to the publication in 1977 of the important email standard RFC 733, "Standard for the Format of ARPA Network Text Messages." RFC 822 later streamlined the contents of RFC 733, removing some of the features described in the earlier standard that failed to gain acceptance and simplifying the specification.

In 2001, both SMTP and the RFC 822 message format were revised; SMTP is now described in RFC 2821 and the message format in RFC 2822. This newer standard makes relatively small changes to the RFC 822 message format to reflect modern use of TCP/IP email. Even though RFC 2822 is the current standard, the original name is still the one most commonly used. I will respect that convention in this discussion, describing the message format based on RFC 2822 while still calling it the RFC 822 message format.

The RFC 822 format describes the form, structure, and content of TCP/IP email messages. It is, as I said, analogous to the message formats used for other protocols in TCP/IP. Like those other formats, the RFC 822 format can be logically divided into two main sections: the *message header*, which contains important control and descriptive information, and the *message body* or *payload*, which carries the data.

Overview of RFC 822 Messages

Where RFC 822 differs from the field formats of other protocols is in expression. Most TCP/IP protocols encode header information into a compact set of bytes that are read and understood based on their location in the message and the semantic meaning assigned to them. Consider IP, for example. The ninth byte of every IP datagram is the Time to Live (TTL) field, which is encoded as a value from 0 to 255. A device reading an IP datagram simply knows that byte number 9 contains the TTL value. If it sees the binary value 00010011 there, it knows the TTL value for this datagram is the decimal value 19.

In contrast, RFC 822 messages do not use a binary format. They are composed entirely of lines of regular ASCII text (as used in the United States, called *US-ASCII* by the standard), even the headers. Each line ends with an ASCII *carriage return* (*CR*) character, followed by a *line feed* (*LF*) character; the combination is collectively termed *CRLF*. Each line of text should be 78 or fewer characters (not including the terminating CRLF) and must not be more than 998 characters (again, excluding the CRLF). Also, the CR and LF characters must not appear by themselves within the text.

The RFC 822 message begins with a set of lines of text that collectively make up the message header. Each *header field* is expressed in the following form, in text: *<header name>: <header value>*. So, for example, if a *TTL* field were in an RFC 822 message (which it isn't, as that concept is not particular to email) and a value of 19 needed to be expressed, the header field would appear like this: *Time to Live: 19*.

This expressing of all fields as simple text means each header takes up more space in each message; the string *Time To Live: 19* takes up 18 bytes including the terminating CRLF, whereas the binary-encoded TTL field in the IP header takes only a single byte. What we gain from this are two important benefits:

- Any user can easily check the headers and immediately understand what headers are present and what their values are, which makes RFC 822 messages very readable.
- Since each header is explicitly labeled, RFC 822 messages can vary in terms of the number of headers they contain and even in what order they appear, making them flexible.

General RFC 822 Message Structure

The RFC 822 message always starts with a set of header fields (as described in the next section). After all the headers, an empty line must occur. This consists simply of the characters CRLF alone, immediately following the CRLF at the end of the final header field line. Seeing two CRLF character pairs in sequence tells the device reading the message that the end of the headers has been reached. All the remaining lines are considered the body of the message. Like the header lines, body lines are composed of ASCII text and must be no more than 998 characters, with 78 characters or fewer recommended (for easier reading on standard 80-character terminal displays).

KEY CONCEPT To ensure that every device on a TCP/IP internetwork can read email sent by every other device, all messages are required to adhere to a specific structure. The standard that first specified the form of modern TCP/IP email messages was RFC 822, and as a result, this is now called the *RFC 822 message format*. An RFC 822 message consists of a set of *message headers* and a *message body*, which are separated by a blank line. RFC 822 messages must contain only plain ASCII text characters. Each line must be no more than 1,000 characters in length, and the last two characters must be the ASCII CR and LF characters to mark the end of the line.

Since both the header and body of email messages are simply ASCII text, the entire message is just a text file, so these messages are very readable and also easy to create. You can use a simple text editor to create a complete email message, including headers, and it can be read with a simple text display utility. This contributes to email's universal appeal.

The drawback is that the decision to make messages entirely ASCII means that no native support is available in RFC 822 messages for anything that requires more complex structuring or that cannot be expressed using the limited number of ASCII characters. So, you cannot express pictures, binary files, spreadsheets, sound clips, and similar types of files directly using ASCII. Also, the use of ASCII makes RFC 822 well suited to expressing messages in English but not in many other languages that use characters that ASCII cannot represent. All of these limitations eventually prompted the creation of the enhanced MIME message format, which we will explore in detail later in this chapter.

TCP/IP Email RFC 822 Standard Message Format Header Fields and Groups

The RFC 822 message format describes the structure and content of TCP/IP email messages. The structure is intentionally designed to be simple and easy to create and understand. Each message begins with a set of headers that describe the message and its contents. An empty line marks the end of the headers, and then the message body follows.

The message body contains the actual text that the sender is communicating to the recipient(s), while the message header contains information that serves various purposes. The header helps control how the message is processed by specifying who the recipients are, describing the contents of the message, and providing information to a recipient of a message about processing that occurred on the message as it was delivered.

Header Field Structure

As mentioned earlier, the *<header name>* field is the name of the header, and the *<header value>* is the value associated with that header, which depends on the header type. Like all RFC 822 lines, headers must be no more than 998 characters long and are recommended to be no more than 78 characters in length, for easier readability. The RFC 822 and 2822 standards support a special syntax for allowing headers to be folded onto multiple lines if they are very lengthy. This is done by continuing a header value onto a new line, which must begin with at least one white-space character, such as white space or a tab character, like this:

```
<header name>: <header value part 1>
<white space> <header value part 2>
<white space> <header value part 3>
```

The tab character is most often used for this purpose. So, for example, if we wanted to specify a large number of recipients for a message, we could do it as follows:

```
To:<tab>person1@domain1.org, person2@domain2.com,
<tab>person3@domain3.net, person4@domain4.edu
```

Header Field Groups

The RFC 822 message format specifies many types of headers that can be included in email messages. A small number of headers are mandatory, meaning they must be included in all messages. Some are not mandatory but are usually present, because they are fundamental to describing the message. Other optional headers are included only when needed.

To help organize the many headers, the RFC 2822 standard categorizes them into header field groups (as did RFC 822, though the groups are a little different in the older standard):

Origination Date Field Specifies the date and time that the message was made ready for delivery; see the next section for details. (This field is in its own group for reasons that are unclear to me; perhaps just because it is so important.)

Originator Fields Contain information about the sender of the message.

Destination Address Fields Specify the recipient(s) of the message, which may be in one of three different recipient classes: the primary recipients (“To”), copied recipients (“Cc”), and blind-copied recipients (“Bcc”).

Identification Fields Contain information to help identify the message.

Informational Fields Contain optional information to help make clear to the recipient what the message is about.

Resent Fields Preserve the original originator, destination, and other fields when a message is resent.

Trace Fields Show the path taken by mail as it was transported.

In addition, the format allows other user-defined fields to be specified, as long as they correspond to the standard `<header name>: <header value>` syntax. This can be used to provide additional information of various sorts. For example, sometimes the email client software will include a header line indicating the name and version of the software used to compose and send the message. As you’ll see later in this chapter, MIME uses new header lines to encode information about MIME messages.

KEY CONCEPT Each RFC 822 message begins with a set of *headers* that carry essential information about the message. These headers are used to manage how the message is processed and interpreted, and they also describe the contents of the message body. Each header consists of a *header name* and a *header value*. More than a dozen different standard RFC 822 headers are available for use and organized into groups. It is also possible to define custom user headers.

Common Header Field Groups and Header Fields

Table 76-1 describes the header fields in TCP/IP email messages and how they are used.

Table 76-1: RFC 822 Email Header Field Groups and Fields

Field Group	Field Name	Appearance	Number of Occurrences Per Message	Description
Origination Date	Date:	Mandatory	1	Indicates date and time that the message was made available for delivery by the mail transport system. This is commonly the date/time that the user tells her email client to send the message.
Originator Fields	From:	Mandatory	1	Email address of the user sending the message, who should be the person who is the source of the message.
	Sender:	Optional	1	Email address of the person sending the email, if different from the message originator. For example, if person B is sending an email containing a message from person A on A's behalf, person A's address goes in the From: header and person B's in the Sender: header. If the originator and the sender are the same (commonly the case), this field is not present.
	Reply-To:	Optional	1	Tells the recipient of the message the address the originator would like the recipient to use for replies. If absent, replies are normally sent back to the From: address.
Destination Address Field	To:	Normally present	1	A list of primary recipients of the message.
	Cc:	Optional	1	A list of recipients to receive a copy of the message (cc stands for carbon copy, as used in old typewriters). There is no technical difference between how a message is sent to someone listed in the Cc: header and someone in the To: header. The difference is only in how the recipient interprets the message. The person in the To: list is usually the main recipient of the message, while the person in the Cc: list is being copied on the message for informational purposes.
	Bcc:	Optional	1	Contains a list of recipients to receive a "blind" copy of the message without other recipients knowing they have received it. For example, if person X is specified in the To: line, person Y is in the Cc: line, and person Z is in the Bcc: line, all three would get a copy of the message, but X and Y would not know Z had received a copy. This is done by either removing the Bcc: line before message delivery or altering its contents.

(continued)

Table 76-1: RFC 822 Email Header Field Groups and Fields (continued)

Field Group	Field Name	Appearance	Number of Occurrences Per Message	Description
Identification Fields	Message-ID:	Should be present	1	Provides a unique code for identifying a message; normally generated when a message is sent.
	In-Reply-To:	Optional, normally present for replies	1	When a message is sent in reply to another, the Message-ID: field of the original message is specified in this field, to tell the recipient of the reply to what original message the reply pertains.
	References:	Optional	1	Identifies other documents related to this message, such as other email messages.
Informational Fields	Subject:	Normally present	1	Describes the subject or topic of the message.
	Comments:	Optional	Unlimited	Contains summarized comments about the message.
	Keywords:	Optional	Unlimited	Contains a list of comma-separated keywords that may be useful to the recipient. May be used optionally when searching for messages on a particular subject matter.
Resent Fields	Resent-Date: Resent-From: Resent-Sender: Resent-To: Resent-Cc: Resent-Bcc: Resent-Message-ID:	Each time a message is resent, a resent block is required	For each resent block, Resent-Date: and Resent-Sender: are required; others are optional	Special fields used only when a message is resent by the original recipient to someone else, called forwarding. For example, person X may send a message to Y, who forwards it to Z. In that case, the original Date:, From:, and other headers are as they were when person X sent the message. The Resent-Date:, Resent-From:, and other resent headers are used to indicate the date, originator, recipient, and other characteristics of the resent message.
Trace Fields	Received: Return-Path:	Inserted by email system	Unlimited	Inserted by computers as they process a message and transport it from the originator to the recipient. Can be used to trace the path a message took through the email system.

TCP/IP Email RFC 822 Standard Message Format Processing and Interpretation

The standards that define SMTP describe the protocol as being responsible for transporting *mail objects*. A mail object is described as consisting of two components: a *message* and an *envelope*. The message is everything in the email message, including both message header and body; the envelope contains all the information necessary to accomplish transport of the message.

The distinction between these objects is important technically. Just as the postal service looks only at the envelope and not its contents in determining what to do with a letter, SMTP likewise looks only at the envelope in deciding how to send a message. It does not rely on the information in the actual message itself for basic transport purposes.

So the envelope is not the same as the message headers. However, as you can tell by looking at the list of email headers, each message includes the recipients and other information needed for mail transport. For this reason, it is typical for an email message to be specified with enough header information to accomplish its own delivery. Email software can process and interpret the message to construct the necessary envelope for SMTP to transport the message to its destination mailbox. The distinction between an email message and its envelope is discussed in more detail in the section describing SMTP mail transfers, in Chapter 77.

The processing of RFC 822 messages is relatively straightforward, due again to the simple RFC 822 message format. The creation of the complete email message begins with the creation of a message body and certain headers by the user creating the message. Whenever a message is “handled” by a software program, the headers are examined so the program can determine what to do with it. Additional headers are also added and changed as needed.

The following is the sequence of events that occur during the lifetime of a message’s headers.

Composition The human composer of the message writes the message body and tells the email client program the values to use for certain important header fields. These include the intended recipients, the message subject, other informational fields, and certain optional headers such as the Reply-To field.

Sender Client Processing The email client processes the message, puts the information the human provided into the appropriate header form, and creates the initial email message. At this time, it inserts certain headers into the message, such as the origination date. The client also parses the intended recipient list to create the envelope for transmission of the message using SMTP.

SMTP Server Processing SMTP servers do not pay attention to most of the fields in a message as they forward it. They will, however, add certain headers, especially trace headers such as Received and Return-Path, as they transport the message. These are generally prepended to the beginning of the message to ensure that existing headers are not rearranged or modified. Note, however, that when gatewaying occurs between email systems (as described in Chapter 75), certain headers must actually be changed to ensure that the message is compatible with non-TCP/IP email software.

Recipient Client Processing When the message arrives at its destination, the recipient's SMTP server may add headers to indicate the date and time the message was received.

Recipient Access When the recipient of a message uses client software, optionally via an email access protocol such as POP3 or IMAP, the software analyzes each message in the mailbox. This enables the software to display the messages in a way that's meaningful to the human user and may also permit the selection of particular messages to be retrieved. For example, most of us like to see a summary list of newly received mail, showing the originator, message subject, and the date and time the message was received, so we can decide what mail we want to read first, what mail to defer to a later time, and what to delete without reading (such as spam).

MIME Overview

The RFC 822 email message format is the standard for the exchange of email in TCP/IP internetworks. Its use of simple ASCII text makes it easy to create, process, and read email messages, which has contributed to the success of email as a worldwide communication method.

Unfortunately, while ASCII text is great for writing simple memorandums and other short messages, it provides no flexibility to support other types of communication. To allow email to carry multimedia information, arbitrary files, and messages in languages using character sets other than ASCII, the MIME standard was created.

NOTE While MIME was developed specifically for email, its encoding and data representation methods have proven so useful that it has been adopted by other application protocols as well. One of the best known of these is the Hypertext Transfer Protocol (HTTP), which uses MIME headers for indicating the characteristics of data being transferred. Some elements of MIME were in fact developed not for email but for use by HTTP or other protocols, and I indicate this where appropriate. Be aware that HTTP only uses some elements of MIME; HTTP messages are not MIME-compliant.

Most protocols become successful specifically because they are based on open standards that are widely accepted. The RFC 822 email message format standard is an excellent example; it is used by millions of people every day to send and receive TCP/IP email.

However, success of standards comes at a price: *reliance* on those standards. Once a standard is in wide use, it is very difficult to modify it, even when times change and the standard is no longer sufficient for the requirements of modern computing. Again, unfortunately, the RFC 822 email message format is an excellent example.

The Motivation for MIME

TCP/IP email was developed in the 1960s and 1970s. Compared to the way the world of computers and networking is today, almost everything back then was *small*. The networks were small; the number of users was small; the computing capabilities

of networked hosts was small; the capacity of network connections was small; the number of network applications was small. (The only thing that wasn't small back then was the size of the computers themselves!)

As a result of this, the requirements for electronic mail messaging were also rather . . . small. Most computer input and output back then was text-based, and it was therefore natural that the creators of SMTP and the RFC 822 standard would have envisioned email as being strictly a text medium. Accordingly, they specified RFC 822 to carry text messages.

The fledgling Internet was also developed within the United States, and at first, the entire internetwork was within American borders. Most people in the United States speak English, a language that as you may know uses a relatively small number of characters that is well-represented using the ASCII character set. Defining the email message format to support United States ASCII (US-ASCII) also made sense at the time.

However, as computers developed, they moved away from a strict text model toward graphical operating systems. And predictably, users became interested in sending more than just text. They wanted to be able to transmit diagrams, non-ASCII text documents (such as Microsoft Word files), binary program files, and eventually multimedia information: digital photographs, MP3 audio clips, slide presentations, movie files and much more. Also, as the Internet grew and became global, other countries came "online," some of which used languages that simply could not be expressed with the US-ASCII character set.

Unfortunately, by this point, the die was cast. RFC 822 was in wide use and changing it would have also meant changes to how protocols such as SMTP, POP and IMAP worked, protocols that ran on millions of machines. Yet by the late 1980s, it was quite clear that the limitations of plain ASCII email were a big problem that had to be resolved. A solution was needed, and it came in the form of the Multi-purpose Internet Mail Extensions (MIME).

NOTE *MIME is usually referred to in the singular, as I will do from here forward, even though it is an abbreviation of a plural term.*

MIME Capabilities

The idea behind MIME is both clever and elegant: RFC 822 restricts email messages to ASCII text, but that doesn't mean that we can't define a more specific structure for how that ASCII text is created. Instead of just letting the user type an ASCII text message, we can use ASCII text characters to encode nontext data parcels (commonly called *attachments*). Using this technique, MIME allows regular RFC 822 email messages to carry the following:

Nontext Information Includes graphics files, multimedia clips, and all the other nontext data examples listed earlier.

Arbitrary Binary Files Includes executable programs and files stored in proprietary formats (for example, AutoCAD files, Adobe Acrobat PDF files, and so forth).

Text Messages That Use Character Sets Other Than ASCII Includes the ability to use non-ASCII characters in the headers of RFC 822 email messages.

MIME even goes one step beyond this, by actually defining a structure that allows multiple files to be encoded into a single email message, including files of different types. For example, someone working on a budget analysis could send one email message that includes a text message, a PowerPoint presentation, and a spreadsheet containing the budget figures. This capability has greatly expanded email's usefulness in TCP/IP.

All of this is accomplished through special encoding rules that transform non-ASCII files and information into an ASCII form. Headers are added to the message to indicate how the information is encoded. The encoded message can then be sent through the system like any other message. SMTP and the other protocols that handle mail pay no attention to the message body, so they don't even know MIME has been used.

The only change required to the email software is adding support for MIME to email client programs. Both the sender and receiver must support MIME to encode and decode the messages. Support for MIME was not widespread when MIME was first developed, but the value of the technique is so significant that it is present in nearly all email client software today. Furthermore, most clients today can also use the information in MIME headers to not only decode nontext information but pass it to the appropriate application for presentation to the user.

KEY CONCEPT The use of the RFC 822 message format ensures that all devices are able to read one another's email messages, but it has a critical limitation: It supports only plain ASCII text. This is insufficient for the needs of modern internetworks, yet reliance on the RFC 822 standard would have made replacing it difficult. *MIME* specifies several methods that allow email messages to contain multimedia content, binary files, and text files using non-ASCII character sets, all while still adhering to the RFC 822 message format. *MIME* also further expands email's flexibility by allowing multiple files or pieces of content to be sent in a single message.

MIME Standards

MIME was first described in a set of two standards, RFC 1341 and RFC 1342, published in June 1992. These were updated by RFCs 1521 and 1522 in September 1993. In March 1994, a supplemental standard was published, RFC 1590, which specified the procedure for defining new MIME media types.

Work continued on MIME through the mid-1990s, and in November 1996, the standards were revised again. This time, the documents were completely restructured to improve the readability of the information and published as a set of five individual standards. These standards are shown in Table 76-2.

Since the time that these five primary MIME standards were released, numerous additional RFCs have been published that have defined various extensions to MIME itself, including additional MIME header types and new media types. Notable examples are RFCs 2183 and 2557, which define the MIME Content-Disposition and Content-Location headers, respectively. Some other MIME capabilities are actually defined as part of other technologies that use MIME; for example, the first HTTP standard, RFC 1945 defines the Content-Length header. Other RFCs define new media types and subtypes (too many to list here).

Table 76-2: MIME Standards

RFC Number	RFC Name	Description
2045	Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies	Describes the fundamental concepts behind MIME and the structure of MIME messages.
2046	Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types	Explains the concept of MIME media types and subtypes and describes some of the kinds of media whose encoding is defined in the MIME standards.
2047	MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text	Describes how RFC 822 headers can be modified to carry non-ASCII text.
2048	Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures	Discusses how organizations can register additional media types for use with MIME.
2049	Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples	Provides additional implementation information and examples of how MIME can be used.

MIME Basic Structures and Headers

The creators of the MIME standard had a difficult challenge on their hands: how to bring flexibility in the types of data contained in email messages, when RFC 822 said that messages could contain only ASCII text. To accomplish this, MIME creators had to exploit the areas of flexibility that had already been put into the existing RFC 822.

Two such opportunities were available: The first was the fact that RFC 822 message bodies are allowed to contain any type of ASCII text, as long as lines don't exceed 998 text characters and each line ends with a CRLF control code combination. Even though the creators of RFC 822 naturally assumed this ASCII text would be human-readable, there was nothing stopping it from being machine-readable code. The second opportunity was the facility built into RFC 822 (and the protocols that use it, such as SMTP) that allowed custom user-defined header fields to be added to any email message.

The nonspecific nature of RFC 822 message bodies forms the basis for how MIME itself works. An email client that supports the MIME standard uses special encoding algorithms that transform non-ASCII information into ASCII form. It then places this set of encoded ASCII characters into the body of the message, as if it had been typed by a user, using one of two special structures.

The ability to add new headers to RFC 822 is used to communicate information about the use of MIME from the sender to the recipient. The devices transporting a MIME message don't care that MIME was used, because they don't pay attention to the contents of the message body. However, when the message reaches its destination, the recipient's email client program must have some way of knowing that MIME was used and must also be told how the information in the message was encoded. Otherwise, it might just present the encoded non-ASCII data to the user as ASCII text (which would look like random gibberish).

Basic Structures

The exact method by which data is encoded in the message body and MIME headers are included depends on the overall structure of the MIME message. Two basic structure types are described, based on the kind of media the message carries:

Simple Structure (Discrete Media) MIME messages carrying a single discrete media type, such as a text message or a graphical image, use a simple structure. Only one encoding of information is present in the body of the message.

Complex Structure (Composite Media) Some MIME messages carry a composite media type, which allows multiple different media to be contained in a single message, such as a text message and a graphical image, or which allows the email to encapsulate another email message in its entirety. Many of these messages use a more complex structure, where the body of the message contains several MIME body parts.

MIME Entities

Collectively, both whole MIME messages and individual body parts are called *MIME entities*. Each set of MIME headers provides information about either type of MIME entity: a MIME message as a whole or a body part in a composite message. When a MIME message is received, the recipient first examines the headers in the message as a whole (the RFC 822 headers) to determine the overall message type. This then indicates whether the message uses a simple or complex structure. If the latter is used, the body of the message is parsed and each individual body part is individually interpreted, including its individualized headers. The section “MIME Composite Media Types,” later in this chapter, provides more details on how these body parts are formatted.

Primary MIME Headers

The first of the five main MIME standards, RFC 2045, describes a set of five primary MIME headers that communicate basic information about the content of each MIME entity (message or body part).

MIME-Version Each MIME message is required to have a MIME-Version header, which serves two purposes. First, it identifies the email message as being MIME-encoded. Second, even though only one version of MIME has been defined so far, having a version number header provides future proofing in case a new version is created later that may have some incompatibilities with the present one. Currently, all MIME messages use version 1.0. This is the only MIME header that applies to an entire message; it is not used to label individual MIME body parts. This is easy to remember, as it is the only header whose name does not begin with Content-.

Content-Type Describes the nature of the data that is encoded in the MIME entity. This header specifies a content type and a content subtype, which are separated by a slash character. It may optionally also contain certain parameters that convey additional information about the type and subtype. In a message body, this

header tells the recipient of the email message what sort of media it contains and whether the body uses a simple or complex structure. In a body part, it describes the media type the body part contains. For example, a message containing an HTML document might have a Content-Type header of `text/html`, where a message containing a JPEG graphical file might be specified as `image/jpeg`. For a composite MIME type, the Content-Type header of the whole message will contain something like `multipart/mixed` or `multipart/alternative`, and each body part will contain individual Content-Type headers such as `text/html` or `image/jpeg`. These are all discussed in detail in the next two sections. This header is optional. When not present, the default of a regular US-ASCII text message is assumed (the media type of regular RFC 822 messages).

Content-Transfer-Encoding For a message using simple structure, specifies the method used to encode the data in the message body; for a composite message, identifies the encoding method for each MIME body part. For data that is already in ASCII form, no special encoding is needed, but other types of data must be converted to ASCII for transmission. This header tells the recipient how to decode the data back into its normal representation. (MIME encoding methods are described later in this chapter.) This header is optional; the default value, if it is not present, is 7-bit encoding, which again is the encoding of regular ASCII.

Content-ID Allows the MIME content to be assigned a specific identification code. This header is analogous to the RFC 822 Message-ID header field but is specific to the MIME content itself. It is optional and is most often used for body parts in multipart MIME messages.

Content-Description This optional header allows an arbitrary additional text description to be associated with the MIME entity. In a multipart message, each body part might be given a description header to make clear to the recipient what the parts represent.

KEY CONCEPT MIME provides flexibility in the information that can be carried in email messages, by encoding non-ASCII data in ASCII form, and by adding special headers that describe this data and how it is to be interpreted. The most important MIME headers are `Content-Type`, which describes what sort of data is in the message, and `Content-Transfer-Encoding`, which specifies how the data is encoded. MIME supports two basic overall formats: *simple structure*, in which a single type of *discrete media* is encoded in a message, and *complex structure*, which encodes a *composite media* type that can carry multiple kinds of information.

Additional MIME Headers

In addition to the five basic headers, the MIME standard allows additional headers to be defined. The only restriction is that they all must start with the word `Content-`, which clearly labels them as describing content of a MIME entity (message or body part). Both the sender and recipient must support a custom header for it to be useful.

Several new MIME headers have in fact been created and documented in various Internet RFCs. Some are actually designed not specifically for use by email messages, but for use by other protocols that make use of MIME technology, such as HTTP. Three are notable:

Content-Disposition In multipart MIME messages, this header may be given to MIME body parts to control how information is presented to the user. The two most common values are *inline*, which says the content is intended to be displayed automatically along with other body parts, and *attachment*, which indicates that the content is separate from the main document. This header is defined in RFC 2183.

Content-Location Allows the location of a MIME body part to be identified using a Uniform Resource Locator (URL). This is sometimes used when encoding HTML and other multimedia-enabled document formats into email using MIME multipart messages. It is defined in RFC 2557.

Content-Length Specifies the length of a MIME entity in bytes. This header is not commonly used in email applications of MIME but is an important header in HTTP. It is described in the HTTP standards, first appearing in RFC 1945.

MIME Content-Type Header and Discrete Media

MIME uses special techniques to encode various kinds of information into ASCII text form, such as graphical images, sound files, video clips, application programs, compressed data files, and many others. We commonly refer to these as different types of *media*, and MIME uses the same term to describe them.

Since MIME supports so many kinds of media, it is necessary that each message contain information that describes what it contains to permit accurate decoding of message contents. This is the function of the important MIME Content-Type header.

Content-Type Header Syntax

The syntax of the Content-Type header is as follows:

Content-Type: <type>/<subtype> [; parameter1 ; parameter2 .. ; parameterN]

The purpose of these different elements is to describe the media in the MIME entity in a way that proceeds from the general to the specific. The first element, *<type>*, is called the *top-level media type* and describes the overall form of the data. For example, it indicates whether the MIME entity contains text, an image, audio, and so forth. The second element, *<subtype>*, provides specific information about the form or format of the data. For example, a JPEG image and a GIF image are both images, but they are in a different format. Both *<type>* and *<subtype>* are mandatory in the Content-Type header.

Following these elements may appear one or more *parameters*, which are usually optional but may be required for some media types. These provide still more details about the nature of the data, when it is required. Each parameter is preceded by a semicolon and is expressed as an attribute/value pair, separated by an equal (=) sign, like this: ; *attribute*=*value*.

One example of how parameters may be used is in specifying the character set in a text message. The representation of regular RFC 822 ASCII text is as follows:

Content-type: text/plain; charset="us-ascii"

The top-level media type is text, and the subtype is plain, so this indicates a plain-text message. The parameter charset specifies that the message uses the US-ASCII character set. Another common use for parameters is to specify the name of an attached file, like this:

Content-type: image/jpeg; name="ryanpicture.jpg"

Discrete Media Types and Subtypes

As I mentioned earlier, MIME supports two basic structures: simple and complex. A simple message carries only one media type, such as a piece of text, a picture, or an executable file. These are called *discrete media types* in MIME. A complex message carries a *composite media type*, which may incorporate multiple body parts. Each body part in turn carries data corresponding to one of the discrete media types. The top-level media type indicates whether the whole message carries a discrete media type or a composite type.

KEY CONCEPT The MIME Content-Type header specifies what sort of data is encoded in a MIME message. The header indicates the general form of the message's content through a *top-level media type*, and the more specific nature of the data through the specification of a *subtype*. It may also contain optional *parameters* that provide still more information about the content.

The RFC 2046 standard (part two of the set of five standards that describes MIME) defines five discrete top-level media types: text, image, audio, video, and application. They each represent one of the major classes of data commonly transmitted over TCP/IP. Each of these has one or more subtypes, and some also have parameters that are used to provide more information about them.

The creators of MIME recognized that the standard could not describe every media type and that new ones would be created in the future. RFC 2048 (part four of the MIME set) describes the process by which new media types, subtypes, and parameters can be described and registered with the Internet Assigned Numbers Authority (IANA).

Thus far, only one new top-level media type has been created; this is the model top-level type, defined for CAD modeling files and similar uses, as described in RFC 2077. However, many dozens of new subtypes have been created over the

years, some specified in RFCs and others just registered directly with IANA. This includes many vendor-specific subtypes, which are usually identified by either the prefix *x-* or *vnd.* in the subtype name.

Literally hundreds of type/subtype combinations now exist, and I will not list them all. You can find a complete list of MIME media organized by top-level media type on IANA's website: <http://www.iana.org/assignments/media-types/index.html>.

Here, I will briefly describe the six MIME discrete top-level media types. For each, I've provided a table showing some of the more commonly encountered MIME subtypes to give you an idea of what is out there.

Text Media Type (text)

The text media type is used for sending data that is primarily in textual form. Table 76-3 describes shows the subtypes.

Table 76-3: MIME text Media Type Subtypes

Type/ Subtype	Description	Defining Source
text/plain	Plain text, used for regular messages such as those corresponding to the initial RFC 822 standard	RFC 2046
text/enriched	Text that includes formatting information or other enrichment that makes it no longer plain	RFC 1896
text/html	A document expressed in HTML, commonly used for the World Wide Web	RFC 2854
text/css	Cascading style sheet information for the World Wide Web	RFC 2318

Image Media Type (image)

The image media type indicates graphical image files, such as pictures. The subtype normally indicates the specific format to allow the recipient to decode and present the file properly. Some of the more common subtypes are shown in Table 76-4.

Table 76-4: MIME image Media Type Subtypes

Type/Subtype	Description	Defining Source
image/jpeg	An image in JPEG format	RFC 2046
image/gif	A Graphical Interchange Format (GIF) image	IANA says RFC 2046, but it's not there.
image/tiff	Tagged Image File Format (TIFF) image	RFC 2302
image/vnd.dwg, image/vnd.dxf, image/vnd.svf	Vector images used in AutoCAD	Registration with IANA

Audio Media Type (audio)

The audio media type is used for sending audio information. The subtype normally indicates the specific format. Table 76-5 shows a couple of common values.

Table 76-5: MIME audio Media Type Subtypes

Type/Subtype	Description	Defining Source
audio/basic	A basic audio type defined in the main MIME standards that describes single-channel audio encoded using 8-bit ISDN mu-law pulse code modulation at 8,000 Hz	RFC 2046
audio/mpeg	MPEG standard audio (including the popular MP3 file format)	RFC 3003

Video Media Type (video)

The video media type is used for sending video information. Again, the subtype normally indicates the specific format, as shown in Table 76-6.

Table 76-6: MIME video Media Type Subtypes

Type/Subtype	Description	Defining Source
video/mpeg	Video encoded to the MPEG digital video standard	RFC 2046
video/dv	Digital video corresponding to several popular standards including SD-VCR, HD-VCR, and DVB, as used by various types of video equipment	RFC 3189
video/quicktime	Apple's QuickTime movie format	Registration with IANA

Model Media Type (model)

The model media type describes a model representation, such as a two-dimensional or three-dimension physical model. Its subtypes are described in Table 76-7.

Table 76-7: MIME model Media Type Subtypes

Type/Subtype	Description	Defining Source
model/mesh	A mesh, as used in modeling	RFC 2077
model/vrml	A Virtual Reality Modeling Language (VRML) model	RFC 2077
model/iges	A model file corresponding to the Initial Graphics Exchange Specification (IGES)	Registration with IANA

Application Media Type (application)

The application media type is a catchall for any kind of data that doesn't fit into one of the preceding categories or that is inherently application-specific. The subtype describes the data by indicating the kind of application that uses it. This can be used to guide the recipient's email program in choosing an appropriate application program to display it, just as a file extension in Windows tells the operating system how to open different kinds of files.

For example, if you have Microsoft Excel installed on your PC, clicking a filename ending with .XLS will launch Excel automatically. Similarly, an Excel spreadsheet will normally be sent using MIME with a media type of application/vnd.ms-excel. This tells the recipient's email program to launch Excel to read this file.

Since so many applications are out there, more than 100 different subtypes exist within this top-level type. Table 76-8 contains a few representative samples.

Table 76-8: MIME application Media Type Subtypes

Type/Subtype	Description	Defining Source
application/octet-stream	An arbitrary set of binary data octets (see the discussion following this table for more details)	RFC 2046
application/postscript	A PostScript file, used for printing and for generating Adobe Acrobat (PDF) files	RFC 2046
application/applefile	Resource file information for representing Apple Macintosh files	Registration with IANA
application/msword	Microsoft Word document (note that this does not have the vnd prefix like most other Microsoft file types)	Registration with IANA
application/pdf	A Portable Document Format (PDF) file, as created by Adobe Acrobat	Registration with IANA
application/vnd.framemaker	An Adobe FrameMaker file	Registration with IANA
application/vnd.lotus-1-2-3	A Lotus 1-2-3 file	Registration with IANA
application/vnd.lotus-notes	A Lotus Notes file	Registration with IANA
application/vnd.ms-excel	A Microsoft Excel spreadsheet file	Registration with IANA
application/vnd.ms-powerpoint	A Microsoft PowerPoint presentation file	Registration with IANA
application/vnd.ms-project	A Microsoft Project file	Registration with IANA
application/zip	A compressed archive file containing one or more other files, using the ZIP/PKZIP compression format	Registration with IANA

Of these application subtypes, a special one is worth further mention: the application/octet-stream subtype. This is the catchall within the catchall of the application type, which just means the file is a sequence of arbitrary binary data. It is usually used when the sender is unsure of what form the data takes or cannot identify it as belonging to a particular application. When this type is used, the recipient will usually be prompted to save the data to a file. He must then figure out what application to use to read it.

The application/octet-stream MIME type/subtype may even be used for images, audio, or video in unknown formats. If you try to send a multimedia document that your sending program does not understand, it will generally encode it as application/octet-stream for transmission. This is your email program's way of saying to the recipient, "I am sending you this file as-is; you figure out what to do with it."

This application/octet-stream type is also often used for transmitting executable files (programs) especially on Windows systems. Unfortunately, while convenient, this can be a serious security hazard. In recent years, the Internet has been subject to a steady stream of viruses and worms that spread by sending themselves to other users through executable file attachments in email. This makes opening and running any unknown application/octet-stream attachment potentially dangerous.

MIME Composite Media Types: Multipart and Encapsulated Message Structures

MIME discrete media types allow MIME to represent hundreds of different kinds of data in email messages. This alone would make MIME an incredibly useful technology, but the MIME standard goes one step further by defining *composite* media types. These allow MIME to perform even more spectacular feats, such as sending many types of data at once or encapsulating other messages or information into email.

The use of a MIME composite media type is indicated via the Content-Type header of an RFC 822 message. Instead of one of the six discrete media types (text, image, audio, video, model, or application), one of these two composite media types is used: `multipart`, which allows one or more sets of data to be sent in a single MIME message, and `message`, which allows a message to encapsulate another message.

KEY CONCEPT Two MIME composite media types exist: `message`, which allows one message to encapsulate another, and `multipart`, which allows multiple individual media types to be encoded into a single email message.

MIME Multipart Message Type

The `multipart` media type is the more common of the two types, and for good reason: It is an *incredibly* powerful mechanism. It allows one message to contain many different kinds of information that can be used in different ways. Each piece of data is encoded separately as a MIME body part, and the parts are combined into a single email message. How these parts are used depends on the semantics of the message, indicated by the MIME subtype. RFC 2046 describes several of these, and a few new ones have also been defined by the IANA registration scheme described earlier.

MIME Multipart Message Subtypes

Table 76-9 shows the most common multipart media subtypes and how they are used. The first four are defined in RFC 2046.

Table 76-9: Common MIME multipart Media Type Subtypes

Type/Subtype	Description	Defining Source
<code>multipart/mixed</code>	Indicates that the body parts are not really related, but they have been bundled for transport in a single message for convenience. For example, this might be used by someone to send an office memo along with a vacation snapshot just for fun. This subtype is also sometimes used when the parts are related but the relationship is communicated to the recipient in some other way (such as via a description in a distinct body part).	RFC 2046

(continued)

Table 76-9: Common MIME multipart Media Type Subtypes (continued)

Type/Subtype	Description	Defining Source
multipart/alternative	Specifies that the body parts are alternative representations of the same information. The recipient decodes the parts and chooses the one that is best suited to her needs. A common use of this is in sending Hypertext Markup Language (HTML)-encoded email. Some email clients can't display HTML, so it is courteous to send a <i>multipart/alternative</i> message containing the message in both HTML and plain text forms. The alternatives should be placed in the message in increasing order of preference, meaning that the preferred format goes last. In the case of a document that includes plain text and rich text alternatives—such as the preceding example with plain text and HTML versions of a document—the plainest format should go first and the fanciest last.	RFC 2046
multipart/parallel	Tells the recipient that the body parts should all be displayed at the same time (in parallel). For example, someone sends an audio clip along with explanatory text to be displayed alongside it as it plays.	RFC 2046
multipart/digest	Allows a message to carry a digest, such as a collection of other email messages.	RFC 2046
multipart/related	Indicates specifically that the body parts are related to each other. Special parameters are used to provide more information on how they are to be interpreted.	RFC 2387
multipart/encrypted	Used for encrypted data. The first body part contains information on how the data is to be decrypted, and the second contains the data itself.	RFC 1847

Multipart Message Encoding

You can see just from the different subtypes shown in Table 78-9 how much flexibility the multipart type provides to MIME, and there are other subtypes. In all cases, the same syntax is used to encode the constituent body parts into a single message. The basic process is as follows:

1. Each individual piece of data is processed as if it were to be transmitted as the body of a discrete media type MIME message. This includes the specification of appropriate headers, such as Content-Type, Content-ID, and Content-Transfer-Encoding, as needed.
2. A special *boundary delimiter* is chosen to separate the body parts. It must be selected so that it will not appear in any of the body parts; a random string is sometimes used. It is prepended with two dashes (--) when placed in the message to reduce the chance of it being mistaken for data.
3. The multipart message is assembled. It consists of a *preamble* text area, then a boundary line, followed by the first body part. Each subsequent body part is separated from the previous one with another boundary line. After the last body part, another boundary line appears, followed by an *epilogue* text area.
4. The special parameter *boundary* is included in the Content-Type header of the message as a whole, to tell the recipient what pattern separates the body parts.

KEY CONCEPT MIME multipart messages are formed by first processing each individual data component to create a MIME *body part*. Each can have a distinct encoding method and set of headers, as if it were a separate MIME message. These body parts are then combined into a single multipart message and separated with a *boundary delimiter*. The identity of the delimiter is inserted into the *boundary* parameter of the *Content-Type* header, so the recipient can easily separate the individual body parts upon receipt of the message.

These rules may seem rather complicated, but once you've seen a couple of multipart messages, the structure will make sense. To help clarify multipart message encoding, Figure 76-1 shows graphically the overall structure of a multipart MIME message.

Listing 76-1 contains a specific example of a multipart message (with portions abbreviated to keep the length down), so you can see what one looks like in text form. (If you want to see more, you probably have several in your own email inbox right now!)

```
From: Joe Sender <joe@someplace.org>
To: Jane Receiver <jane@somewhereelse.com>
Date: Sun, 1 Jun 2003 13:28:19 -0800
Subject: Photo and discussion
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary="exampledelimtext123"
```

This is a multipart message in MIME format

```
-exampledelimtext123
Content-Type: text/plain
```

Jane, here is the photo you wanted from me for the new client.
Here are some notes on how it was processed.
(Blah blah blah...)
Talk to you soon,
Joe.

```
-exampledelimtext123
```

```
Content-Type: image/jpeg; name="clientphoto.jpg"
Content-Transfer-Encoding: base64
```

```
SDc9Pjv/2wBDAQoLCw4NDhwQEbw7KCIo0zs70zs70zs
```

```
...
```

```
zv/wAARCADIARoDASIAhEBAxEB/8QAHAAAAQUBA
```

```
-exampledelimtext123
```

```
(Epilogue)
```

Listing 76-1: Example of a MIME multipart message

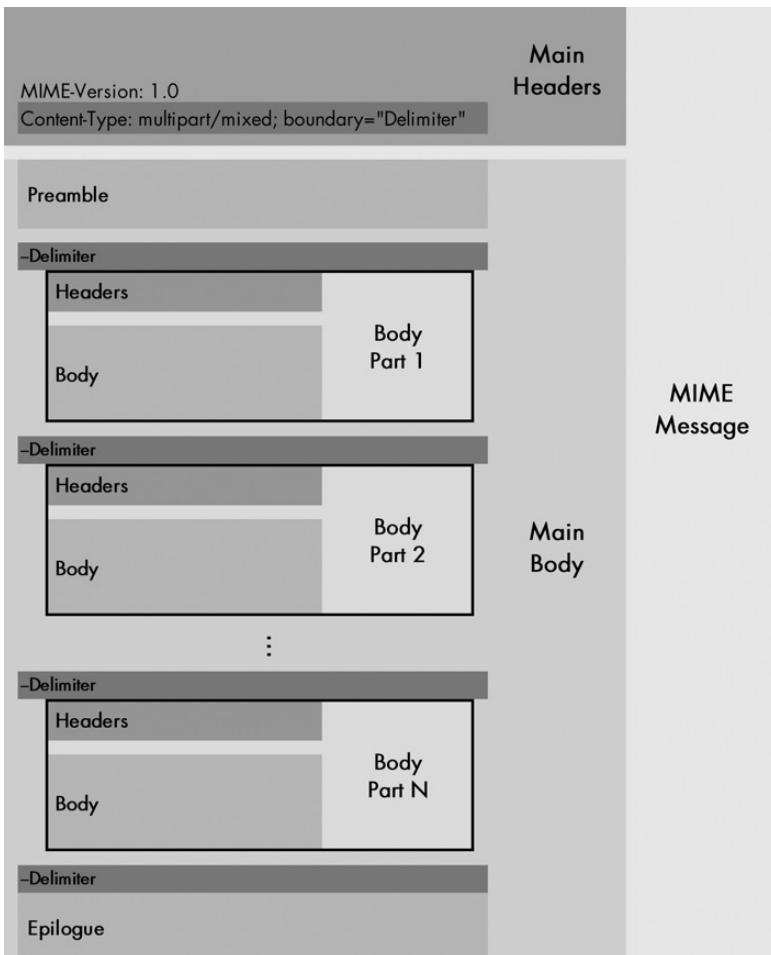


Figure 76-1: MIME multipart message structure A MIME multipart message consists of a set of main headers and main body portion, like all messages. Within the main body are one or more body parts, each of which has its own body-part-specific headers followed by the body part itself; each body part is shown in a black box. The Content-Type header of the message as a whole indicates that the message type is *multipart*, and the boundary parameter specifies the name of the delimiter, in this case just called "Delimiter." This delimiter is used to separate the body parts from each other and from the preamble and epilogue that begin and end the message body, respectively.

In this example, Joe is sending Jane a multipart message containing a JPEG photograph and some explanatory text. The main header of the message specifies the `multipart/mixed` type and a boundary string of `exampledelimtext123`. The message begins with the preamble, which is ignored by the recipient email client but can be seen by the human reader. It is common to put a string here such as the one given in this example. That way, if a person using a client that does not support MIME receives the message, the recipient will know what it is.

The first delimiter string is then placed in the message, followed by the first body part, the text Joe is sending Jane. This is preceded by whatever headers are needed by the body part, in this case `Content-Type: text/plain`. (Note, however, that this is the default in MIME, so it could be omitted here.) After the text message is

another delimiter, and then the encoded JPEG photo in the second body part, with its own headers. Finally, there is one more delimiter, and then a space for the epilogue. This is ignored if present and is often not used at all.

It is possible to send a multipart message that has only a single body part. This is sometimes done to take advantage of the preamble area to provide information about how to decode a nontext media type. Of course, this can also be done by including such text decoding instructions as a body part.

MIME Encapsulated Message Type

The other composite media type is the message type, which is devoted to the special purpose of encapsulating existing email messages within the body of a new message, or encapsulating other types of messages. This may be another email message previously sent or a message of some other kind. This media type also provides flexibility for sending partial messages and other special types of communication. Table 76-10 shows the three subtypes defined in RFC 2046.

Table 76-10: Common MIME message Media Type Subtypes

Type/Subtype	Description	Defining Source
message/rfc822	Indicates that the body contains an encapsulated email, itself formatted according to the RFC 822 standard. Note that this doesn't necessarily mean it is a plain text email message; it could be a MIME message (though encapsulating MIME within MIME must be done carefully).	RFC 2046
message/partial	Allows the fragmentation of larger messages into pieces that can later be reassembled.	RFC 2046
message/external-body	Indicates that the body of the message is not actually contained in the message itself; instead, a reference is provided to where the body is located. Sufficient information to locate the real message body must be provided.	RFC 2046

MIME Content-Transfer-Encoding Header and Encoding Methods

One of the main reasons why MIME was created was the significant restrictions that the RFC 822 standard places on how data in email messages must be formatted. To follow the rules, messages must be encoded in US-ASCII, a 7-bit data representation. This means that even though each byte can theoretically have any of 256 values, in ASCII only 128 values are valid. Furthermore, lines can be no longer than 1,000 characters including the carriage return and line feed (CRLF) characters at the end, and those two characters cannot appear elsewhere.

For some types of data, such as text files, this is no big deal; but for others it is a serious problem. This is especially the case with binary data. If you look at the data in a video clip, MP3 file, or executable program, it will appear to be random gibberish. In fact, such data is not random; it is represented using specific rules, but the data is expressed in raw binary form, where any 8-bit byte can contain any value from 0 to 255, which is why it looks like junk to humans. More important, this means that this data does not follow the rules for RFC 822 files and cannot be sent directly in this form.

To send non-ASCII data in MIME, it must be encoded. The Content-Transfer-Encoding header is used to specify how a MIME message or body part has been encoded, so that it can be decoded by its recipient. Four types of encoding are defined: `7bit`, `8bit/binary`, `quoted-printable`, and `base64`. The `quoted-printable` and `base64` encodings are the most interesting ones, because they are what allow non-RFC-822 data to be sent using RFC 822.

KEY CONCEPT MIME supports four encoding methods: `7bit`, `8bit (binary)`, `quoted-printable`, and `base64`. `7bit` encoding is standard ASCII and is used for text. `quoted-printable` encoding is for output that is mostly text but has some special characters that must be encoded. `base64` is used for arbitrary binary files. The 8-bit encoding method is defined in MIME but not used for RFC 822 messages.

7-Bit and 8-Bit Encoding

`7bit` encoding indicates that the message is already in ASCII form compatible with RFC 822. It is the default and is what is assumed if no Content-Transfer-Encoding header is present.

The `8bit` and `binary` values are synonymous. They mean the message has been encoded directly in 8-bit binary form. Yes, I did just say that this would violate the rules of RFC 822. These options appear to have been included to support future mechanisms for transporting binary data directly. RFC 1652 describes an SMTP extension that discusses this in part: “*SMTP Service Extension for 8bit-MIMEtransport*” (there is no space between *MIME* and *transport*). However, the standard clearly states that this still does not allow the transfer of raw binary data using SMTP and RFC 822.

Quoted-Printable Encoding

`Quotable-printable` encoding is a special type that is used when most of the data is ASCII text, but it contains certain violations of the rules of RFC 822. These illegal sections are converted using special encoding rules so the data as a whole is consistent with RFC 822; only the problem bytes are encoded. The result is that RFC 822 compatibility is achieved while maintaining most of the data as regular text so it can still be easily understood by a human.

An example would be letters with tildes or accents, such as those used in French or Spanish. Another would be a text message formed using an editor that inserts carriage return characters in the middle of a line. Most of the message is still text. The `quoted-printable` encoding can be used here, with the carriage return characters represented as `=0D` (the hexadecimal value of the character prepended by an equal sign). RFC 2046 contains more details on how this is done.

Base64 Encoding

In contrast, `base64` encoding is more often used for raw binary data that is not in human-readable form anyway, such as graphical image, audio, video, and application files. This encoding is used to allow arbitrary binary data to be represented in ASCII

form. The data is then sent as ASCII and decoded back into binary form by the recipient. The idea behind this type of encoding is simple: The data that needs to be sent can have any value for each 8-bit byte, which is not allowed. So why not rearrange the bits so the data fits into the 7-bit ASCII limits of RFC 822?

This is done by processing the data to be sent three bytes at a time. There are 24 bits in each three-byte block, which are carved into four sets of 6 bits each. Each 6-bit group has a value from 0 to 63 and is represented by a single ASCII character, as presented in Table 76-11.

Table 76-11: MIME base64 Encoding Groups

6-Bit Value	Encoding						
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

For example, suppose the first three bytes of the data to be sent were the decimal values 212, 39, and 247. These cannot all be expressed in 7-bit ASCII. In binary form, they are expressed like so:

11010100 00100111 11110111

We can divide these into four 6-bit groups:

110101 - 00 0010 - 0111 11 - 110111

Which yields the four values 53, 2, 31, and 55. Thus, the values 214, 39, and 247 would be encoded as the three ASCII characters 1Cf3. The conceptual steps of this process are shown in Figure 76-2.

NOTE The sequence of steps for the encoding are intended to help you understand the process. Computers inherently deal directly with bits and would not bother with converting to decimal before encoding the 6-bit groups into ASCII characters.

Data Bytes in Decimal Form	212	39	247
Data Bytes in Binary Form	11010100	00100111	11110111
Data Rearranged into 6-Bit Groups	110101	000010	011111
6-Bit Groups in Decimal Form	53	2	31
Groups Converted to ASCII Characters	1	C	f
			3

Figure 76-2: MIME base64 encoding In this simplified example, three binary data bytes are encoded as four ASCII characters using MIME base64 encoding. Instead of transmitting those three bytes, two of which would not be valid in RFC 822, the four ASCII characters 1Cf3 are sent.

This 3-to-4 encoding is done for all the data. The converted ASCII characters are then placed into the body of the entity instead of the raw binary data, 76 characters to a line. I showed how this is done in the second body part in the example in Listing 76-1 (except I didn't use 76 characters per line, to keep the line lengths short). One final character is involved in this scheme, the equal sign (=), which is used as a padding character when needed.

Since base64 characters are regular ASCII, they appear to SMTP like a regular text message. Of course, the data looks like gibberish to us, but that's not a problem since it will be converted back to its regular form and displayed to the recipient as an image, movie, audio, or whatever.

KEY CONCEPT MIME uses base64 encoding to transform arbitrary 8-bit files into a form that is acceptable for communication in email. Each set of three 8-bit bytes is divided into four 6-bit groups, and each 6-bit group is represented by an ASCII character. Since the data is ASCII, it conforms to the RFC 822 message format standard, even if it is not human-readable. The receiving device reverses the encoding, changing each four-character block back into three 8-bit bytes.

The main drawback of the base64 method is that it is about 33 percent less efficient than sending binary data directly, using a protocol like the File Transfer Protocol (FTP). The reason is that three 8-bit bytes of binary data are sent as four ASCII characters, but of course, each ASCII character is represented using 8 bits itself. So there is one-third more overhead when using base64. In most cases, this is not a big deal, but it can be significant if downloading very large email files over a slow Internet connection.

Note that RFC 2046 also defines two other encodings: ietf-token and x-token. These are included to allow new encoding types to be defined in the future.

MIME Extension for non-ASCII Mail Message Headers

All of the MIME mechanisms discussed up to this point deal with ways of encoding different kinds of ASCII and non-ASCII data into the *body* of an RFC 822 message. In addition to these capabilities, MIME also includes a way in which non-ASCII data can be encoded into *headers* of an RFC 822 message.

At this point, you might be wondering why anyone would want to do this. Sure, it makes sense to be able to use MIME to encode binary data such as an image into an email, but why do it in a header? Well, if you can't see the need for this, chances are that you are a native English speaker. ASCII does a great job of representing English, but isn't so good with many other languages. With RFC 822, speakers of languages that use non-ASCII characters were unable to use descriptive headers fully, such as the Subject and Comments headers. Some could not even properly express their own names!

The solution to this problem is the subject of RFC 2047, the third of the five main MIME standards. It describes how to encode non-ASCII text into ASCII RFC 822 message headers. The idea is straightforward: As with message bodies, the non-ASCII text is replaced with ASCII, and information is provided to describe how this was done.

With this technique, the value of a regular header is replaced by a MIME *encoded-word* that has the following syntax:

=?< charset >?< encoding >?< encoded-text >?=

The strings =? and ?= are used to *brace* the non-ASCII header, which flags it as a MIME encoded header to the recipient's email client. The other elements, separated by ?, indicate how the non-ASCII text is encoded, as follows:

<*charset*> The character set used, such as iso-8859-1.

<*encoding*> Two different encoding types are defined, each represented by a single letter for brevity: B indicates base64 encoding, and Q indicates quoted-printable encoding (these encoding types are discussed in the previous section).

<*encoded-text*> The non-ASCII text that has been encoded as ASCII using the encoding type indicated.

As you can see, this method is analogous to how a non-ASCII message body or body part would be encoded, but the information about the encoding has been condensed so everything can fit in a single header line. The <*charset*> parameter is somewhat analogous to the Content-Type header for a message body, but since headers can contain only text, it specifies what kind of text it is. The <*encoding*> parameter is clearly equivalent to the Content-Transfer-Encoding header.

KEY CONCEPT In addition to its many functions for encoding a variety of data in email message bodies, MIME provides a feature that allows non-ASCII information to be placed into email headers. This is done by encoding the data using either *quoted-printable* or *base64* encoding, and then using a special format for the header value that specifies its encoding and character set. This technique is especially useful for email sent in languages that cannot be represented easily in standard ASCII, such as many Asian languages.

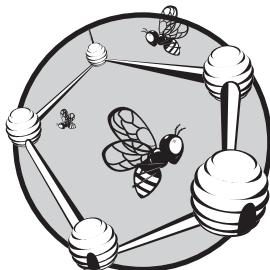
Here's an example of a non-ASCII header, using the GB2312 character set (for Chinese characters) and base64 encoding:

Subject: =?GB2312?B?u7bTrbL0vNPDwLn61bm740==?=

I hope that doesn't say anything inappropriate; I took it from a piece of spam email I received once!

77

TCP/IP ELECTRONIC MAIL DELIVERY PROTOCOL: THE SIMPLE MAIL TRANSFER PROTOCOL (SMTP)



I emphasized in my overall description of TCP/IP email that communication using email requires the interaction of various protocols and elements. One mistake that some people make is to equate the method used for delivering email with the entire system. This is, however, an understandable mistake—just as the postal service is only a part of the whole system of mailing a letter, it is nonetheless a very big part. Likewise, the delivery of email from sender to recipient is arguably the most important part of email as a whole. In modern TCP/IP, this task is the responsibility of the *Simple Mail Transfer Protocol (SMTP)*.

In this chapter, I describe in detail the operation of SMTP. I begin with an overview and history of the protocol and a discussion of the standards that define it. I then examine the way that SMTP client/server communication and message transport work. I explain the way that SMTP servers establish connections and transaction sessions, and then the process by which mail is transferred from one server to another. I describe some of the special features implemented in SMTP and discuss SMTP security issues. I conclude with a reference summary of SMTP commands and replies.

BACKGROUND INFORMATION My discussion of SMTP assumes that you already have a basic understanding of the general concepts of TCP/IP email, as well as familiarity with TCP/IP email addressing and message formatting. These topics are discussed in Chapters 74, 75, and 76, respectively.

SMTP Overview, History, and Standards

The overview and history of the TCP/IP email system in Chapter 74 describes how TCP/IP evolved from its early beginnings to its current form. Since the mechanism used to deliver email is such a big part of the system as a whole, any overview of the system must of necessity discuss how delivery mechanisms have changed as well. In the case of TCP/IP, the delivery of mail evolved through many forms during the 1970s, as developers sought to find effective ways of communicating email messages between systems. Most of these efforts involved attempts to transmit mail using existing protocols; this makes sense, since it is easier to adapt a technology than design one from scratch.

SMTP Standards

One important achievement in the development of a mail system was the publishing of the *Mail Transfer Protocol (MTP)*, which was first defined in RFC 772 in September 1980, and then updated in RFC 780 in May 1981. MTP describes a set of commands and procedures by which two devices can connect using TCP to exchange email messages. Its operation is described largely using elements borrowed from two early TCP/IP application protocols that were already in use at that time: Telnet and the File Transfer Protocol (FTP). The commands of MTP are actually based directly on those of FTP.

Although there was nothing inherently wrong with basing email delivery on FTP, defining it this way made MTP somewhat of a hack. It was also restricted to the capabilities defined by FTP, a general file transfer protocol, so it was not possible to include features in MTP that were specific to sending and receiving mail. Due to the importance of email, a specific protocol designed for the purpose of delivering email was warranted. SMTP was first defined in RFC 788 and published in November 1981.

The name suggests that SMTP is simpler than the protocol that it replaced. Whether or not this is true is somewhat a matter of opinion; I do note that RFC 788 is 61 pages long, while the earlier RFC 780 was only 43 pages. What SMTP definitely has over MTP is *elegance*; the protocol is designed specifically for the transport of email. While it retains certain similarities to FTP, it is an independent protocol running over the Transmission Control Protocol (TCP). So, from a conceptual standpoint, it can be considered simpler than MTP. In terms of mechanics, the process SMTP uses to transfer an email message is indeed rather simple, especially compared to some other protocols.

RFC 788 described the operation of SMTP carrying email messages corresponding to the ARPAnet text message standard as described in RFC 733. Development of both email messages and SMTP continued, and in August 1982, a milestone in TCP/IP email was achieved when RFCs 821 and 822 were published. RFC 821 revised SMTP and became the defining standard for the protocol for the next two decades. RFC 822, its companion standard, became the standard for TCP/IP email messages carried by SMTP.

KEY CONCEPT The most important component of the TCP/IP email system is the *Simple Mail Transfer Protocol (SMTP)*. SMTP was derived from the earlier Mail Transfer Protocol (MTP) and is the mechanism used for the delivery of mail between TCP/IP systems and users. The only part of the email system for which SMTP is not used is the final retrieval step by an email recipient.

As the 1980s progressed, and TCP/IP and the Internet both grew in popularity, SMTP gradually overtook other methods to become the dominant method of email message delivery. For a number of years, the protocol was used mostly as is, with no new RFCs published to define new versions or formally change its behavior. This changed in February 1993, when RFC 1425, “SMTP Service Extensions,” was published. As the name suggests, this standard describes a process for adding new capabilities to extend how SMTP works, while maintaining backward compatibility with existing systems. SMTP with these extensions is sometimes called *Extended SMTP* or *ESMTP* (though use of this term seems not to be entirely universal).

As development of SMTP continued, RFC 1425 was revised in RFC 1651 in July 1994, and then in RFC 1869 in November 1995. Along with these revisions, a number of other RFCs defining particular SMTP extensions, such as pipelining and message size declaration, were defined.

In April 2001, another major milestone in TCP/IP email was reached when revisions of RFC 821 and RFC 822 were published, as RFCs 2821 and 2822, respectively. Both documents are consolidations of updates and changes that had been made to RFCs 821 and 822 between 1982 and 2001. And, no, I don’t think it is a coincidence that the old and new RFC numbers are exactly 2,000 apart. RFCs 2820 and 2823 were both published in May 2000, so it looks like 2821 and 2822 were reserved for the email standards. I think this naming was a great idea, as it makes it clear that the new RFCs are revisions of the old ones.

RFC 2821 is the current base standard for SMTP. It incorporates the base protocol description from RFC 821 and the latest SMTP extensions as defined in RFC 1869. It updates the description of the email communication model to reflect the realities of modern TCP/IP networks, especially the email features built into the Domain Name System (DNS). We’ll examine this in more detail in the next section.

SMTP Communication and Message Transport Methods

The TCP/IP email communication model describes the way email messages are conveyed from the sender to the recipient. In most cases, this involves the sender’s client machine sending the email to its local SMTP server, which sends it to the recipient’s local SMTP server, which then sends it to the recipient’s local host. SMTP handles the transport between SMTP servers. In fact, the overall email communication model is largely described by the RFC 821 and 2821 SMTP standards.

The initial communication takes place between the sender’s client machine and a local SMTP server that the sender is allowed to access. After submission of the email message, that SMTP server becomes responsible for delivering the message to the SMTP server responsible for the recipient’s mailbox.

Early Email Communication Using Relaying

In the early days of email, when RFC 821 and its predecessors were first defined, the Internet was very different from what it is today. There was no DNS, and this made email delivery complex, because there was no way to map a mailbox address to the IP address of the SMTP server that managed that mailbox. Also, many proprietary networks were connected to the Internet, which meant that it was not always possible for any particular system to communicate with any other.

Given this, how could email be delivered? The most common way in the early days of SMTP was through a process called *relaying*. SMTP routing information was included along with the email address, to specify a sequence of SMTP servers that the mail should be relayed through to get to its destination. For example, if a sender using SMTP Server A wanted to send email to someone whose mailbox was on SMTP Server Z, the sender might have needed to specify that the mail be sent through intermediate SMTP Servers D, P, and U to get there. An SMTP connection would be established from Server A to Server D to send the message on one leg of its journey; then it would go from Server D to P, Server P to U, and then Server U to Z. The process is analogous to how Internet Protocol (IP) routing works, but at the application layer (actually using IP routing at a lower level).

You can probably see the problems with this quite easily: It's cumbersome, requires many devices to handle the mail, results in delays in communication, and requires the communication of source routes between SMTP servers. It was certainly functional, but it was far from ideal.

Modern Email Communication Using DNS and Direct Delivery

The creation of DNS radically changed how email delivery worked. DNS includes support for a special *mail exchange (MX)* record that allows easy mapping from the domain name in an email address to the IP address of the SMTP server that handles mail for that domain. I explain this in the description of the regular email address format in Chapter 76, as well as the section about DNS email support in Chapter 56.

In the new system, SMTP communication is much simpler and more direct. The sending SMTP server uses DNS to find the MX record of the domain to which the email is addressed. This gives the sender the DNS name of the recipient's SMTP server. This is resolved to an IP address, and a connection can be made directly from the sender's SMTP server to the recipient's server to deliver the email. While SMTP still supports relaying, direct email delivery using MX records is faster and more efficient, and RFC 2821 makes clear that this is now the preferred method.

In this new system, SMTP is generally used only for two transfers: first, from the sender's client machine to the sender's local SMTP server, and then from that server to the recipient's local SMTP server, as shown in Figure 74-1 in Chapter 74. (A distinct mail access protocol or method is used by the recipient for the last leg of the journey.) Each transfer of an email message between SMTP servers involves the establishment of a TCP connection, and then the transfer of the email headers and body using the SMTP mail transfer process. The following sections describe in detail how this occurs.

KEY CONCEPT In the early days of SMTP, mail was delivered using the relatively inefficient process of relaying from server to server across the internetwork. Today, when an SMTP server has mail to deliver to a user, it determines the server that handles the user's mail using the Domain Name System (DNS) and sends the mail directly to that server.

Terminology: Client/Server and Sender/Receiver

The original RFC 821 standard referred to the device that initiates an SMTP email transfer as the *sender* and the device that responds to it as the *receiver*. These terms were changed to *client* and *server* in RFC 2821 to “reflect current industry terminology.” Strictly speaking, this is correct, but in some ways, the more current terminology is significantly *less* clear.

As I explained in the general discussion of TCP/IP client/server operation in Chapter 8, the terms *client* and *server* are used in many different senses in networking, which often leads to confusion. In common parlance, the computers that handle email on the Internet are usually all called *SMTP servers*. This is because they run SMTP server software to provide SMTP services to client machines, such as end-user PCs. In addition, these devices are usually dedicated hardware servers running in network centers, typically managed by Internet service providers (ISPs).

However, the terms *client* and *server* are now used to refer to the roles in a particular SMTP communication as well. Since all SMTP servers both send and receive email, they all act as both clients and servers at different times. An SMTP server that is relaying an email will act as both server and client for that message, receiving it as a server, and then sending it to the next server as a client. Adding to this potential confusion is the fact that the initial stage in sending an email is from the sender's client machine to the sender's local SMTP server. Thus, the client role in an SMTP transaction may not be an actual SMTP server, but the server role will always be a server.

For all of these reasons, the old terms *sender* and *receiver* are still used in places in RFC 2821, where needed for clarity. I consider them much more straightforward and use them in the rest of this chapter.

KEY CONCEPT SMTP servers both send and receive email. The device sending mail acts as a client for that transaction, and the one receiving it acts as a server. To avoid confusion, it is easier to refer to the device sending email as the *SMTP sender* and the one receiving as the *SMTP receiver*; these terms were used when SMTP was originally created.

SMTP Connection and Session Establishment and Termination

The delivery of email using SMTP involves the regular exchange of email messages among SMTP servers. SMTP servers are responsible for sending email that users of the server submit for delivery. They also receive email intended for local recipients, or for forwarding or relaying to other servers.

Overview of Connection Establishment and Termination

All SMTP communication is done using TCP. This allows SMTP servers to make use of TCP's many features that ensure efficient and reliable communication. SMTP servers generally must be kept running and connected to the Internet 24 hours a day, seven days a week, to ensure that mail can be delivered at any time. (This is a big reason why most end users employ access protocols such as the Post Office Protocol to access their received email rather than running their own SMTP servers.) The server listens continuously on the SMTP server port, well-known port number 25, for any TCP connection requests from other SMTP servers.

An SMTP server that wishes to send email normally begins with a DNS lookup of the MX record corresponding to the domain name of the intended recipient's email address to get the name of the appropriate SMTP server. This name is then resolved to an IP address; for efficiency, this IP address is often included as an *additional* record in the response to the MX request to save the sending server from needing to perform two explicit DNS resolutions.

The SMTP sender then establishes an SMTP session with the SMTP receiver. Once the session is established, mail transactions can be performed to allow mail to be sent between the devices. When the SMTP sender is finished sending mail, it terminates the connection. All of these processes involve specific exchanges of commands and replies, which are illustrated in Figure 77-1.

Let's take a look at these processes in more detail, starting with SMTP session establishment.

Connection Establishment and Greeting Exchange

The SMTP sender begins by initiating a TCP connection to the SMTP receiver. The sending SMTP server uses an ephemeral port number, since it is playing the role of the client in the transaction. Assuming that the server is willing to accept a connection, it will indicate that it is ready to receive instructions from the client by sending reply code 220. This is called the *greeting* or *service ready* response. It commonly includes the full domain name of the server machine, the version of the SMTP server software it is running, and possibly other information.

Now, it would be rude for the server acting as a client to start sending commands to the responding server without saying hello first, wouldn't it? So that's exactly what comes next: the client says, "Hello." In the original SMTP protocol, this is done by issuing a HELO command, which includes the domain name of the sending (client) SMTP server as a courtesy. The receiving device then responds back with a return hello message using an SMTP reply code 250.

For example, if the SMTP server smtp.sendersite.org was making a connection to the SMTP server mail.receiversplace.com, it would say:

```
HELO smtp.sendersite.org.
```

After receiving this greeting, mail.receiversplace.com would respond back with a hello message of its own, something like this:

```
250 mail.receiversplace.com Hello smtp.sendersite.org, nice to meet you.
```

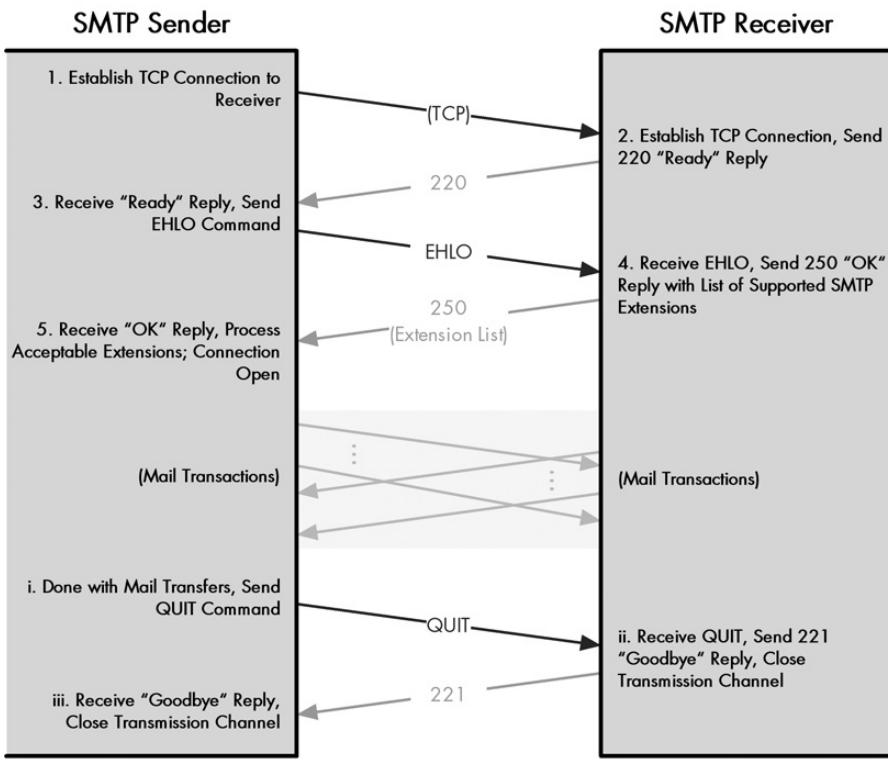


Figure 77-1: SMTP transaction session establishment and termination An SMTP session begins with the SMTP sender establishing a TCP connection to the SMTP receiver. The receiver sends a ready message; the sender sends a HELO or EHLO command, to which the receiver responds. Assuming no difficulties are encountered, the session is established and mail transactions take place. When the sender is finished, it sends a QUIT command; the receiver responds with a 221 reply and closes the session.

(The chatty text is of course purely optional; most of the time, SMTP communication is between software programs, so the pleasantries are usually written by programmers who have a sense of humor.)

Connection Establishment Using SMTP Extensions

The SMTP extensions first defined in RFC 1425, and then in subsequent standards up to RFC 2821, define an alternative hello message for the client to use: EHLO (extended hello). An SMTP sender supporting SMTP extensions (and most do) uses EHLO instead of HELO in response to the 220 greeting. This serves both to say hello to the SMTP receiver and to tell it that the sender supports SMTP extensions.

If the SMTP receiver supports the extensions, it replies with the usual 250 reply, as well as a series of extra 250 responses. Each of these lists an EHLO keyword that indicates a particular SMTP extension the receiver supports. If the receiving server doesn't support the extensions, it will reject the EHLO command with a 500 reply code ("syntax error, command not recognized"). This tells the SMTP sender that it

cannot use extensions. It will then issue a conventional HELO command, or it will QUIT the connection if it requires the SMTP extension to be present. (In practice, it is rare for a server to *require* the use of SMTP extensions.)

Here's the same example used earlier, but using EHLO. The sender says:

EHLO smtp.sendersite.org.

Assuming mail.receiversplace.com supports the SMTP extensions, a typical reply might look like this:

250-mail.receiversplace.com Hello smtp.sendersite.org, nice to meet you.
250-SIZE
250-DSN
250 PIPELINING

Each of these additional replies identifies a particular SMTP extension supported by mail.receiversplace.com; in this case, message size declaration (SIZE), delivery status notification (DSN), and command pipelining. (The dashes after the 250 indicate a multiple-line response to a command; this is discussed in the “SMTP Multiple-Line Text Replies” section later in the chapter.)

Once the HELO or EHLO command has been sent and the receiving device has responded, the session is initiated. Further commands can be sent by the sending SMTP server to the responding server. These usually take the form of email message transfer transactions using the process described in the upcoming “SMTP Mail Transaction Process” section, and other command/reply exchanges as needed.

Connection Termination

When the sending device is finished sending all the email it has to transfer to the receiving device, and it has completed all its other activities, it terminates the session by issuing the QUIT command. This normally results in a 221 “goodbye” message from the SMTP receiver, which says something like “closing transmission channel.” The TCP connection is then terminated.

KEY CONCEPT An SMTP session consists of three basic phases. The session is first established through the creation of a TCP connection and the exchange of identity information between the SMTP sender and receiver using the HELO command. Once established, *mail transactions* can be performed. When the SMTP sender is finished with the session, it terminates it using the QUIT command. If *SMTP extensions* are supported, the SMTP sender uses the *EHLO* (*extended hello*) command instead of *HELO*, and the SMTP receiver replies with a list of extensions it will allow the SMTP sender to use.

A server may also terminate prematurely in special cases. If it is given a local command to shut down (for example, due to imminent rebooting of the hardware server on which it is running), it may respond to any routine command with a 421 response (“Service not available, closing transmission channel”). A server is not

supposed to terminate a session simply due to receipt of an invalid command, however; this should happen only in special cases where session termination cannot be avoided.

SMTP Mail Transaction Process

As described in the previous section, the delivery of an email message begins with the establishment of an SMTP session between the devices sending and receiving the message. The SMTP sender initiates a TCP connection to the SMTP receiver and then sends a HELO or an EHLO command, to which the receiver responds. Assuming no problems ensue, the session is then established and ready for actual email message transactions.

Overview of SMTP Mail Transaction

The SMTP mail transaction process itself consists of three steps:

1. **Transaction Initiation and Sender Identification** The SMTP sender tells the SMTP receiver that it wants to start sending a message and gives the receiver the email address of the message's originator.
2. **Recipient Identification** The sender tells the receiver the email address(es) of the intended recipients of the message.
3. **Mail Transfer** The sender transfers the email message to the receiver. This is a complete email message meeting the RFC 822 specification (which may be in MIME format as well).

That's it! So you can see that the word *Simple* in *Simple Mail Transfer Protocol* definitely has at least *some* merit. In fact, one question that sometimes comes up when examining SMTP is "Why couldn't this process be even simpler?" The first two steps identify the sender of the email and the intended recipient(s). But all of this information is already contained in headers in the message itself. Why doesn't SMTP just read that information from the message, which would make the mail transaction a *one-step* process?

The explanation isn't specifically addressed in the SMTP standards, but I believe there are several reasons for this:

- Specifying the sender and recipients separately is more efficient, as it gives the SMTP receiver the information it needs up front before the message itself is transmitted. In fact, the SMTP receiver can decide whether or not to accept the message based on the source and destination email addresses.
- Having this information specified separately gives greater control on how email is distributed. For example, an email message may be addressed to two recipients, but they may be on totally different systems; the SMTP sender might wish to deliver the mail using two separate SMTP sessions to two different SMTP receivers.

- In a similar vein, there is the matter of delivering blind carbon copies. Someone who is BCC’ed a message must receive it without being mentioned in the message itself.
- Having this information separate makes implementing security on SMTP much easier.

For these reasons, SMTP draws a distinction between the message itself, which it calls the *content*, and the sender and recipient identification, which it calls the *envelope*. This is consistent with our running analogy between regular mail and email. Just as the postal service delivers a piece of mail using only the information written on the envelope, SMTP delivers email using the envelope information, not the content of the message. It’s not quite the case that the SMTP server doesn’t look at the message itself, just that this is not the information it uses to manage delivery.

NOTE *It is possible for the sender of a message to generate envelope information based on the contents of the message, but this is somewhat external to SMTP itself. It is described in the standard, but caution is urged in exactly how this is implemented.*

SMTP Mail Transaction Details

Let’s take a more detailed look at the SMTP mail transaction process, using as aids the process diagram in Figure 77-2 and the sample transaction of Listing 77-1 (which has commands highlighted in bold and replies in italics).

```

MAIL FROM:<joe@someplace.org>
250 <joe@someplace.org> . . . Sender ok
RCPT TO:<jane@somewhereelse.com>
250 <jane@somewhereelse.com> . . . Recipient ok
DATA
354 Enter mail, end with "." on a line by itself
From: Joe Sender <joe@someplace.org>
To: Jane Receiver <jane@somewhereelse.com>
Date: Sun, 1 Jun 2003 14:17:31 -0800
Subject: Lunch tomorrow

```

Hey Jane,

It's my turn for lunch tomorrow. I was thinking we could
[rest of message]
Hope you are free. Send me a reply back when you get a chance.
Joe.
.
250 OK

Listing 77-1: Example of an SMTP mail transaction

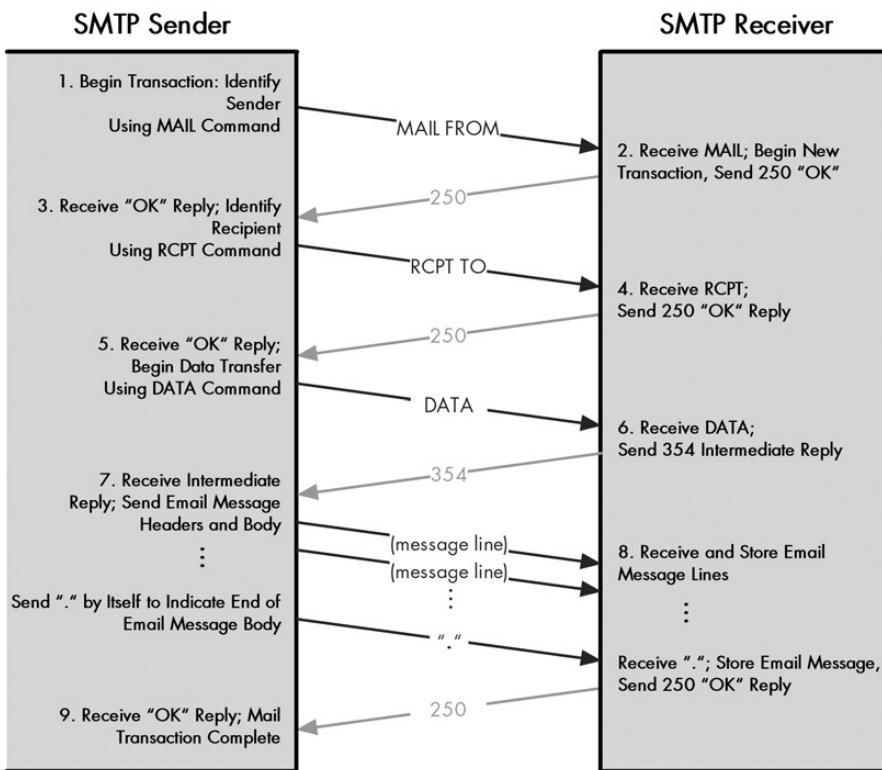


Figure 77-2: SMTP mail transaction process Once an SMTP session is established between a sender and receiver, each mail transaction consists of a set of three command/reply sequences. The sender is first identified using the `MAIL` command and the recipients are specified using one or more `RCPT` commands. The actual mail message is then transferred using the `DATA` command, which involves a preliminary reply before the actual message is sent and a completion reply when it has been fully received.

The first two steps in the mail transaction are responsible for providing the receiving SMTP server with the envelope information just discussed. The transaction begins by the SMTP sender issuing a `MAIL` command. This serves to inform the receiver that a new transaction is commencing and also to tell it the *from* information on the envelope. Here's an example:

```
MAIL FROM:<joe@someplace.org>
```

The email address of the originator is always enclosed in angle brackets (< and >). The SMTP receiver acknowledges the command with a `250 ("OK")` reply message, sometimes sending back the address as a confirmation. Here's an example:

```
250 <joe@someplace.org> . . . Sender ok
```

Next, the SMTP sender uses `RCPT` commands to specify the intended recipients of the email that is being sent. Each `RCPT` line can contain only one recipient, so if multiple recipients are indicated, two or more `RCPT` commands must be issued.

Each one normally specifies an email address, but if relaying is being used, the command may contain routing information as well. (As described earlier in the “SMTP Communication and Message Transport Methods” section, this is not as commonly done as it was in the past.) Here’s an example:

```
RCPT TO:<jane@somewhereelse.com>
```

Assuming the server accepts the email, it will give a 250 “OK” reply again, like so:

```
250 <jane@somewhereelse.com> . . . Recipient ok
```

The SMTP sender then issues the DATA command, which tells the SMTP receiver that the message is coming:

```
DATA
```

The SMTP receiver responds with a 354 “intermediate” reply message, such as this:

```
354 Enter mail, end with "." on a line by itself
```

The SMTP sender then sends the email message, one line at a time, with a single dot (.) on a line to terminate it. The server confirms the receipt of the message with another 250 “OK” reply, and the transaction is finished.

KEY CONCEPT After an SMTP session is established, email messages are sent using the SMTP *mail transaction process*. The SMTP sender starts the transaction by identifying the sender of the email and then specifying one or more recipients. The email message itself is then transmitted to the SMTP receiver. Each email to be sent is a separate transaction.

While this indeed is quite simple, notice that I have shown an email transfer from a sender to one recipient, one in which there were no problems or complications in the transaction. Due either to command syntax or server issues, it is possible for various types of errors to occur at different stages of the process, which may result in the transaction failing. As you’ll see shortly, security concerns may come also into play, leading to restrictions in what transactions a server may allow.

SMTP Special Features, Capabilities, and Extensions

The primary job of the SMTP is to implement the TCP/IP email delivery system. Whenever the user of an SMTP server gives it an email message addressed to a remote mailbox, the server will attempt to transfer it to the appropriate destination server, using the SMTP mail transaction process. Many billions of such transfers are performed every day on the Internet, allowing email to reach its destination quickly anywhere around the world.

SMTP Special Features and Capabilities

In addition to this basic transfer mechanism, SMTP includes a number of other features and capabilities. These allow SMTP to support special requirements and auxiliary needs of the mail system, as described in detail in RFC 2821. It would take many pages to describe them all in detail, so I will provide a quick summary of the more important ones here so you know a bit about them.

The following are some of SMTP's special features:

Mail Relaying As discussed in the “SMTP Communication and Message Transport Methods” section earlier in this chapter, the protocol was once widely used in a relaying mode, where email was routed from one SMTP server to another to reach its destination. Today, the more efficient, normal method of email transfer on the Internet is directly from the sender’s SMTP server to the recipient’s server, using DNS MX records to determine the recipient SMTP server address. SMTP still includes the ability to relay mail from one server to another, provided certain conditions are met. Note that many servers won’t relay mail because this feature has been abused for spamming and malicious hacking.

Mail Forwarding Under certain conditions, an SMTP server may agree to accept email for a remote mailbox and forward it to the appropriate destination. This sounds similar to relaying but is used in a different way. A common example is when users change their email address. For example, if you have worked at XYZ Industries for years and then retire, the company may no longer wish to let you receive email at the company’s SMTP server. As a courtesy, however, they may forward email sent to you there, so that you receive it at your new company.

Mail Gateways Certain SMTP servers may be configured as email gateways. These devices translate TCP/IP email into a form suitable for another email system, and vice versa. Gateways are a complex topic because email systems can be so different. One of the more important problems is the inconsistency of addressing methods of different email systems.

Address Debugging SMTP includes a VRFY (verify) command that can be used to check the validity of an email address without actually sending mail to it.

Mailing List Expansion The SMTP command EXPN (expand) can be used to determine the individual email addresses associated with a mailing list. (Note, however, that this has nothing directly to do with mailing list software like *Majordomo*.)

Turning The original SMTP included a command that allows the SMTP sender and SMTP receiver to change roles. This could be used to allow SMTP Server A to send email to Server B, and then have Server B send email it has queued for Server A in the same session. In practice, this capability was not widely used for a variety of reasons, including security considerations. It is now officially not recommended but may still be implemented in some SMTP software.

These are just a few of the features that are mentioned in the SMTP standards. In addition, developers of a particular type of SMTP server software may give it other features as well. The HELP command is one way of determining what commands are supported by a particular SMTP server.

SMTP servers also must perform a great deal of background processing that doesn't get a great deal of attention. This includes managing connections, checking for errors in commands and email messages, and reacting accordingly. They must also be on the lookout for problem conditions, such as looping that may result in an email message being passed back and forth between two SMTP servers, each thinking the other is the intended recipient. In the event of an initial failure to deliver mail, an SMTP server is also required to retry communication periodically with the destination device and return a failure message to the sender if it cannot deliver the message after a certain period of time. RFC 2821 contains more details.

SMTP Extensions

As discussed earlier in this chapter, during the 1990s, many extensions to the basic operation of SMTP were defined. These are enabled when two SMTP servers supporting the extension set up a session using the EHLO command and appropriate extension response codes. Table 77-1 summarizes some of the more interesting SMTP extensions that have been defined and gives the RFC number where each is described. You can find the full current set of SMTP extensions at <http://www.iana.org/assignments/mail-parameters>.

Table 77-1: SMTP Extensions

Extension Keyword	Extension	Defining Document	Description
8BITMIME	8-bit MIME support	RFC 1652	Theoretically defines support for the 8-bit content transfer encoding type in MIME, but complications associated with this. See the discussion of content encoding in Chapter 76 for details.
AUTH	Authorization	RFC 2554	Used to implement an authorization mechanism for servers requiring enhanced security.
DSN	Delivery status notification	RFC 1891	Allows an SMTP sender to request that the SMTP receiver notify it if a problem occurs in delivering a message.
ENHANCEDSTATUSCODES	Enhanced status codes	RFC 2034, RFC 1893	Extends the traditional three-digit SMTP reply code format with extra codes that provide more information. See the "SMTP Replies and Reply Codes" section later in this chapter for more information.
PIPELINING	Command pipelining	RFC 2920	Allows multiple commands to be transmitted in batches from the SMTP sender to the receiver, rather than sending one command at a time and waiting for a response code.
SIZE	Message size declaration	RFC 1870	Allows information about the size of a message to be declared by an SMTP sender prior to transmitting it, so the SMTP receiver can decide if it wants the message or not.

NOTE Certain commands in the basic SMTP description that are considered optional are also sometimes considered extensions, such as the EXPN and HELP commands; I have not listed these here, since they are not true SMTP extensions.

SMTP Security Issues

When it comes to security and SMTP, the theme is a common one in TCP/IP: A lack of security in how the protocol is implemented, because it was developed when the Internet was just a small group of machines controlled by individuals who mostly knew and trusted each other or who were able to use physical security. Developers never imagined TCP/IP being used by millions of anonymous average Joes around the world, which necessitates far more attention to security than a small research internetwork like the ARPAnet.

With SMTP, security matters are, if anything, *worse* than they are with some of the other protocols. Not only does SMTP not have any real security mechanism, the original relaying model of SMTP communication is entirely designed around the idea of cooperation and trust among servers. Since most SMTP servers would be asked to handle a certain number of intermediate transfers, each server was required to accept mail from any originator to be delivered to any destination.

The basic assumption in this model is that users of SMTP servers would all be well behaved and not abuse the system by flooding intermediate servers with a lot of mail to be delivered or sending bogus messages to cause problems. This all changed as the Internet exploded in popularity in the 1990s. Con artists, malicious hackers, and disreputable salespeople discovered that email could be used for free delivery of messages simply by submitting them to an SMTP server for delivery. The result was overloaded servers, primarily due to the sending of large quantities of unwanted email, which Internet users commonly call *spam*.

NOTE The term *spam*, in this context, has nothing directly to do with the Hormel processed meat product. Its use in reference to massive amounts of email comes from a Monty Python comedy sketch in which that word is repeated in phrases over and over again.

It is actually very easy to impersonate an SMTP server. You can use the Telnet Protocol to connect directly to an SMTP server on port 25. SMTP commands are all sent as text, and so are SMTP replies, so you can have a conversation with a server, and even manually perform a mail transaction. This is useful for debugging, but it also makes abuse of a wide-open SMTP server trivially easy. Since spammers often don't want to be identified, they employ spoofing techniques to make it more difficult to identify them, so resolving these problems is even more difficult.

Despite this obvious dilemma, efforts to implement a general security mechanism in SMTP have been resisted for two main reasons. First, there is no foolproof way to retrofit a new security mechanism onto something as widely used as SMTP without creating incompatibilities between newer and older systems. Second, many administrators were reluctant to do away completely with the general notion of cooperation among sites that has helped make the Internet so successful.

Still, something had to be done. The compromise was for system administrators to tighten up their SMTP servers through the imposition of both technical and policy changes. Naturally, these vary from one organization to another. Some of the more common SMTP security provisions include the following:

- Checking the IP address of a device attempting connection and refusing even to start an SMTP session unless it is in a list of authorized client devices.
- Restricting certain commands or features, such as email relaying, to authorized users or client servers. This is sometimes done by requiring authentication via the SMTP extension AUTH before the command will be accepted.
- Limiting the use of commands such as EXPN to prevent unauthorized users from determining the email addresses of users on mailing lists.
- Checking the validity of envelope information before accepting a message for delivery. Some servers will first verify that the originator's email address is valid before agreeing to accept the MAIL command. Many will check the recipient's address and refuse the message if delivery is not to a local mailbox. Others use even more advanced techniques.
- Limiting the size of email messages that may be sent or the number that may be sent in a given period of time.
- Logging all access to the server to keep records of server use and check for abuse.

Because of all the abuse in recent years, you will find that most SMTP servers implement these or other features, even though most of those features are not formally defined by the SMTP standards. Rather, they are enhancements built into individual SMTP server software packages.

SMTP was designed during an era in which Internet security was not much of an issue; as a result, the base protocol includes no security mechanism at all. Since email is so often abused today, most modern SMTP servers incorporate one or more security features to avoid problems.

Some of these measures can actually be quite sophisticated. For example, the SMTP server run by pair Networks, the great web-hosting company I have used for years, uses *POP-before-SMTP authentication*. This means that before the server will accept outgoing mail from the user via SMTP, the user must first log in to check incoming mail using the Post Office Protocol (POP). Since POP includes authentication, a successful POP login tells the server the user is authorized. This “flips a switch” in the server that allows the user to access the SMTP service after that login for a limited period of time. If this seems convoluted, then you’re starting to get an idea of the hassle that spammers and malicious hackers have created for ISPs today.

It’s also worth noting that SMTP does not include any mechanism for encryption to ensure the privacy of email transmissions. Users requiring security to control who sees their messages must use a separate encryption scheme to encode the body of the message prior to submission.

SMTP Commands

Early TCP/IP email mechanisms were developed by borrowing techniques and elements from existing application protocols, especially Telnet and FTP. SMTP is an independent protocol, but its heritage can still be seen clearly in a few areas. One of the more obvious of these is in the method by which commands are issued by an SMTP sender and replies returned by an SMTP receiver.

Like FTP, all SMTP commands are sent as plain ASCII text over the TCP connection established between the client and server in an SMTP connection. These commands must end with the two-character CRLF sequence that normally terminates ASCII text as required for the Telnet Network Virtual Terminal (NVT; see Chapter 87). In fact, you can check the function of an SMTP server and even issue commands to it yourself simply by using Telnet to connect to it on port 25.

All SMTP commands are specified using a four-letter command code. Some commands also either allow or require parameters to be specified. The basic syntax of a command is

<command-code> <parameters>

When parameters are used, they follow the command code and are separated from it by one or more space characters. For example, the HELO and EHLO commands are specified with the command code, a space character, and then the domain name of the SMTP sender, as you saw earlier in the discussion of SMTP connection establishment.

Table 77-2 lists the commands currently used in modern SMTP in the order they are described in RFC 2821, with a brief description of each.

Table 77-2: SMTP Commands

Command Code	Command	Parameters	Description
HELO	Hello	The domain name of the sender	The conventional instruction sent by an SMTP sender to an SMTP receiver to initiate the SMTP session.
EHLO	Extended Hello	The domain name of the sender	Sent by an SMTP sender that supports SMTP extensions to greet an SMTP receiver and ask it to return a list of SMTP extensions the receiver supports. The domain name of the sender is supplied as a parameter.
MAIL	Initiate Mail Transaction	Must include a FROM: parameter specifying the originator of the message, and may contain other parameters as well	Begins a mail transaction from the sender to the receiver.
RCPT	Recipient	Must include a TO: parameter specifying the recipient mailbox, and may also incorporate other optional parameters	Specifies one recipient of the email message being conveyed in the current transaction.

(continued)

Table 77-2: SMTP Commands (continued)

Command Code	Command	Parameters	Description
DATA	Mail Message Data	None	Tells the SMTP receiver that the SMTP sender is ready to transmit the email message. The receiver normally replies with an intermediate “go ahead” message, and the sender then transmits the message one line at a time, indicating the end of the message by a single period on a line by itself.
RSET	Reset	None	Aborts a mail transaction in progress. This may be used if an error is received upon issuing a MAIL or RCPT command, if the SMTP sender cannot continue the transfer as a result.
VRFY	Verify	Email address of mailbox to be verified	Asks the SMTP receiver to verify the validity of a mailbox.
EXPN	Expand	Email address of mailing list	Requests that the SMTP server confirm that the address specifies a mailing list, and return a list of the addresses on the list.
HELP	Help	Optional command name	Requests general help information if no parameter is supplied; otherwise, information specific to the command code supplied.
NOOP	No Operation	None	Does nothing except for verifying communication with the SMTP receiver.
QUIT	Quit	None	Terminates the SMTP session.

Like FTP commands, SMTP commands are not case-sensitive.

KEY CONCEPT The SMTP sender performs operations using a set of *SMTP commands*. Each command is identified using a four-letter code. Since SMTP supports only a limited number of functions, it has a small command set.

The commands in Table 77-2 are those most commonly used in SMTP today. Certain other commands were also originally defined in RFC 821 but have since become obsolete. These include the following:

SEND, SAML (Send and Mail), and SOML (Send or Mail) RFC 821 defined a distinct mechanism for delivering mail directly to a user’s terminal as opposed to a mailbox, optionally in combination with conventional email delivery. These were rarely implemented and obsoleted in RFC 2821.

TURN Reverses the role of the SMTP sender and receiver as described earlier in the SMTP special features discussion. This had a number of implementation and security issues and was removed from the standard in RFC 2821.

Finally, note that certain SMTP extensions make changes to the basic SMTP command set. For example, the *AUTH* extension specifies a new command (also called *AUTH*) that specifies an authentication method the SMTP client wants to use. Other extensions define new parameters for existing commands. For example, the *SIZE* extension defines a *SIZE* parameter that can be added to a *MAIL* command to tell the SMTP receiver the size of the message to be transferred.

SMTP Replies and Reply Codes

All SMTP protocol operations consist of the plain ASCII text SMTP commands you saw in Table 77-2, issued by the sender to the receiver. The receiver analyzes each command, carries out the instruction requested by the sender if possible, and then sends a reply to the sender. The reply serves several functions: confirming command receipt, indicating whether or not the command was accepted, and communicating the result of processing the command.

Just as SMTP commands are sent in a manner reminiscent of how FTP internal commands work, SMTP replies are formatted and interpreted in a way almost identical to that of FTP replies. As with FTP, the reply consists of not just a string of reply text, but a combination of reply text and a numerical *reply code*. And as with FTP, these reply codes use three digits to encode various information about the reply, with each digit having a particular significance. The reply code is really the key part of the reply, with the reply text being merely descriptive.

NOTE *The discussion of FTP reply codes in Chapter 72 contains a thorough explanation of the benefits of using these structured numeric reply codes.*

Reply Code Structure and Digit Interpretation

SMTP reply codes can be considered to be of the form *xyz*, where *x* is the first digit, *y* is the second, and *z* is the third.

The first reply code digit (*x*) indicates the success or failure of the command in general terms, whether a successful command is complete or incomplete, and whether an unsuccessful command should be tried again or not. This particular digit is interpreted in exactly the same way as it is in FTP, as shown in Table 77-3.

Table 77-3: SMTP Reply Code Format: First Digit Interpretation

Reply Code Format	Meaning	Description
1yz	Positive Preliminary Reply	An initial response indicating that the command has been accepted and processing of it is still in progress. The SMTP sender should expect another reply before a new command may be sent. Note that while this first digit type is formally defined in the SMTP specification for completeness, it is not currently used by any of the SMTP commands; that is, no reply codes between 100 and 199 exist in SMTP.
2yz	Positive Completion Reply	The command has been successfully processed and completed.

(continued)

Table 77-3: SMTP Reply Code Format: First Digit Interpretation (continued)

Reply Code Format	Meaning	Description
3yz	Positive Intermediate Reply	The command was accepted but processing it has been delayed, pending receipt of additional information. For example, this type of reply is often made after receipt of a DATA command to prompt the SMTP sender to send the actual email message to be transferred.
4yz	Transient Negative Completion Reply	The command was not accepted and no action was taken, but the error is temporary and the command may be tried again. This is used for errors that may be a result of temporary glitches or conditions that may change, such as a resource on the SMTP server being temporarily busy.
5yz	Permanent Negative Completion Reply	The command was not accepted and no action was taken. Trying the same command again is likely to result in another error. An example would be sending an invalid command.

The second reply code digit (*y*) is used to categorize messages into functional groups. This digit is used in the same general way as in FTP, but some of the functional groups are different in SMTP, as you can see in Table 77-4.

Table 77-4: SMTP Reply Code Format: Second Digit Interpretation

Reply Code Format	Meaning	Description
x0z	Syntax	Syntax errors or miscellaneous messages.
x1z	Information	Replies to requests for information, such as status requests.
x2z	Connections	Replies related to the connection between the SMTP sender and SMTP receiver.
x3z	Unspecified	Not defined.
x4z	Unspecified	Not defined.
x5z	Mail System	Replies related to the SMTP mail service itself.

The third reply code digit (*z*) indicates a specific type of message within each of the functional groups described by the second digit. The third digit allows each functional group to have ten different reply codes for each reply type given by the first code digit (preliminary success, transient failure, and so on).

Again, as in FTP, these *x*, *y*, and *z* digit meanings are combined to make specific reply codes. For example, the reply code 250 is a positive reply indicating command completion, related to the mail system. It is usually used to indicate that a requested mail command was completed successfully.

Table 77-5 contains a list of some of the more common SMTP reply codes taken from RFC 2821, in numerical order. For each, I have shown the typical reply text specified in the standard and provided additional descriptive information when needed.

As mentioned earlier, the actual text string for each reply code is implementation-specific. While the standard specifies dry response text such as “Requested action completed” for a 250 message, some servers will customize this code or even give different replies to different 250 messages, depending on the context.

Table 77-5: SMTP Reply Codes

Reply Code	Reply Text	Description
211	System status or system help reply.	
214	<Help message...>	Used for text sent in reply to the HELP command.
220	<servername> Service ready.	Greeting message sent when TCP connection is first established to an SMTP server.
221	<servername> closing transmission channel.	Goodbye message sent in response to a QUIT message.
250	Requested mail action ok, completed	Indicates successful execution of a variety of commands.
251	User not local; will forward to <forward-path>	Used when the SMTP receiver agrees to forward a message to a remote user.
252	Cannot VRFY user, but will accept message and attempt delivery	Indicates that a server tried to verify an email address, but was not able to do so completely. Usually means the address appears to be valid but it was not possible to ascertain this to be positively true.
354	Start mail input; end with <CRLF>.<CRLF>	Intermediate reply to a DATA command.
421	<servername> Service not available, closing transmission channel	Sent in response to any command when the SMTP receiver prematurely terminates the connection. A common reason for this is receipt of a local shutdown command, due to a hardware reboot, for example.
450	Requested mail action not taken: mailbox unavailable	Sent when a mailbox is busy due to another process accessing it.
451	Requested action aborted: local error in processing	Local processing problem on the server.
452	Requested action not taken: insufficient system storage.	Time to clean out the server's hard disk!
500	Syntax error, command unrecognized	Response to a bad command or one that was too long.
501	Syntax error in parameters or arguments	
502	Command not implemented	Command is valid for SMTP in general but not supported by this particular server.
503	Bad sequence of commands	Commands were not sent in the correct order, such as sending the DATA command before the MAIL command.
504	Command parameter not implemented.	
550	Requested action not taken: mailbox unavailable	Generic response given due to a problem with a specified mailbox. This includes trying to send mail to an invalid address, refusal to relay to a remote mailbox, and so forth.
551	User not local; please try <forward-path>	Tells the SMTP sender to try a different path; may be used to support mailbox forwarding.
552	Requested mail action aborted: exceeded storage allocation	User's mailbox is full.
553	Requested action not taken: mailbox name not allowed	Specification of an invalid mailbox address.
554	Transaction failed.	General failure of a transaction.

SMTP Multiple-Line Text Replies

As in FTP, it is possible for an SMTP reply to contain more than one line of text. In this case, each line starts with the reply code, and all lines but the last have a hyphen between the reply code and the reply text to indicate that the reply continues. The last line has a space between the reply code and reply text, just like a single-line reply. See the “Connection Establishment Using SMTP Extensions” section earlier in this chapter for an example of a multiple-line response to an EHLO command.

Enhanced Status Code Replies

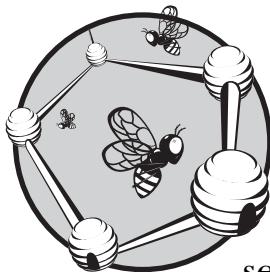
When the ENHANCEDSTATUSCODES SMTP extension is enabled, this causes supplemental reply codes to be issued by the SMTP receiver in response to each command. These codes are similar in some respects to the standard reply codes; they also use three digits, but the digits are separated by periods. These enhanced codes provide more information about the results of operations, especially errors.

For example, if you try to issue a RCPT command specifying a remote mailbox on a server that does not support this feature, it will send back a 550 reply, which is a generic error meaning “requested action not taken: mailbox unavailable.” When enhanced status codes are active, the response will be 550 5.7.1, which is the more specific message “delivery not authorized, request refused.” A full description of these enhanced codes can be found in RFC 1893.

KEY CONCEPT Each time the SMTP sender issues a command, it receives a *reply* from the SMTP receiver. SMTP replies are similar to FTP replies, using both a three-digit reply code and a descriptive text line. A special *enhanced status codes* SMTP extension is also defined; when enabled, this causes the SMTP receiver to return more detailed result information after processing a command.

78

TCP/IP ELECTRONIC MAIL ACCESS AND RETRIEVAL PROTOCOLS AND METHODS



The Simple Mail Transfer Protocol (SMTP) is responsible for most of the process of sending an electronic mail (email) message from the originator to the recipient. SMTP's job ends when the message has been successfully deposited into the recipient's mailbox on his local SMTP server.

In some cases, this mailbox is the end of the message's travels through cyberspace. More often, however, it is only a "rest stop"—the last step of the journey is for the message to be accessed and read by the user to whom it was sent. This may require that it be retrieved from the mailbox and transferred to another client machine. For a variety of reasons, SMTP is not used for the process of accessing a mailbox; instead, a special set of protocols and methods is designed specifically for email access and retrieval.

In this chapter, I describe some of the more common techniques used for TCP/IP email access and retrieval. I begin with an overview of the subject that describes in general the different paradigms used for email access and gives an overview of the protocols.

I then describe the operation of the very popular Post Office Protocol (POP), focusing on version 3 (POP3). I look at the protocol in general terms, discussing its history, the various versions of the protocol, and the standards that define them. I describe POP3's general operation and the communication between a client and server, concentrating on the three main states through which the session transitions. I then describe each of these states in sequence: the *Authorization* state, *Transaction* state, and *Update* state.

Following this, I discuss the other common mail access protocol: the Internet Message Access Protocol (IMAP). This includes a description of its benefits compared to the simpler POP3, a discussion of its operation, and a look at how client and server devices communicate, showing how the IMAP session moves through a series of four states.

Finally, I conclude with a discussion of two alternative methods of email access and retrieval. The first I call *direct server access*, which describes several ways that mailboxes are accessed without the use of special remote-access protocols such as POP and IMAP. The second is email access using a web browser. This is the newest email access method and is growing in popularity every year.

TCP/IP Email Mailbox Access Model, Method, and Protocol Overview

In an ideal world, every device on the Internet would run SMTP server software, and that one protocol would be sufficient to implement the entire TCP/IP email system. You would compose email on your machine, your SMTP software would send it to a recipient's machine, and she would read it.

Here in the real world, however, this is not possible in general terms. An SMTP server must be connected to the Internet and available around the clock to receive email sent at any time by any of the millions of other computers in the world. Most of us either cannot or do not want to run machines continuously connected to the Internet, nor do we want to configure and maintain potentially complex SMTP software. For these reasons, a complete email exchange normally involves not two devices but four: A message is composed on the sender's client machine, and then transferred to the sender's SMTP server, then to the recipient's SMTP server, and finally to the recipient's machine.

The communication between SMTP servers is done with SMTP; so is the initial step of sending the email from the sender's machine to the sender's SMTP server. However, SMTP is not used for the last part of the process, which is accessing the recipient's mailbox. Instead, specific mailbox access and retrieval protocols and methods were devised.

Why not simply have mail wait on the recipient's SMTP server, and then have the mail sent to the recipient client device when it comes online, using SMTP? This isn't possible for two main reasons. First, SMTP was designed for the specific purpose of transporting only email. Having it responsible for client mailbox access would require adding more functionality, making it difficult to keep SMTP *simple*. In the same vein, SMTP works on a *push* model, with transactions being initiated by the sender. It would need changes to allow it to respond to requests from a client device that is only online intermittently.

The second reason is probably more important, because the current protocol configuration allows *flexibility* in how email is accessed. If we used SMTP, all we would be able to do is transfer email to the recipient's client machine. This would be functional, but it would greatly limit the capabilities of how email is used, especially, for example, for users who wish to access mail directly on the server and manipulate it there. Also consider the problem of people with special requirements, such as those who travel and may need to access email from a number of different client devices. There is thus an advantage to providing more than one way to access a mailbox.

Email Access and Retrieval Models

RFC 1733, "Distributed Electronic Mail Models in IMAP4," describes three different paradigms, or models, for mail access and retrieval:

Online Access Model We would all be using this mode of access in my ideal world scenario, where every machine was always connected to the Internet running an SMTP server. We would have constant, direct online access to our mailboxes. In the real world, this model is still used by some Internet users, especially those who have UNIX accounts or run their own SMTP servers. I call this *direct server access*.

Offline Access Model In this paradigm, a user establishes a connection to a server where his mailbox is located. The user downloads received messages to the client device and then deletes them from the server mailbox. All reading and other activity performed on the mail can be done offline once the mail has been retrieved.

Disconnected Access Model This is a hybrid of online and offline access. The user downloads messages from the server, so she can read or otherwise manipulate them without requiring a continuous connection to the server. However, the mail is not deleted from the server, as in the offline model. At some time in the future, the user connects back with the server and synchronizes any changes made on the local device with the mailbox on the server. What sort of changes can be made? Examples include marking whether or not a message has been read to keep track of read and unread mail, and marking messages to which the user has already replied. These are important tools to help those with busy mailboxes keep track of what they need to do.

None of the three models is entirely better than the others. Each has advantages and disadvantages, which is why it is good that we have these options rather than the single SMTP protocol for mail access.

Direct server access has the main benefits of instant speed and universal access from any location. As for disadvantages, you must be online to read mail, and it usually requires that you use UNIX email clients, which with most people are not familiar. However, IMAP can also be used for online access.

Offline access has the main advantages of simplicity and short connection time requirements; you can easily connect to the mailbox, download messages, and then read them locally. But that makes this method somewhat inflexible and poorly suited to access from different machines. Still, it is currently the most popular access method because simplicity is important; it is best typified by POP.

Disconnected access attempts to combine the advantages of offline and online access without combining their disadvantages, and it does a pretty good job. The advantages are significant: the ability to access mail quickly and use it offline, while retaining and updating the mailbox on the server to allow access from different client machines. IMAP is popularly used for disconnected access. In the IMAP overview later in this chapter, I explore its advantages over offline access as well as its main disadvantages, which are complexity and far less universal support than POP (though acceptance of IMAP is slowly increasing).

Finally, in recent years, a somewhat new mailbox access method has become popular: email access using the World Wide Web. This technique allows a user to access his mailbox from any computer with an Internet connection and a web browser. It is a good example of line blurring, not only between the access models discussed here, but between TCP/IP applications—in this case, the Web and email.

KEY CONCEPT For flexibility, TCP/IP uses a variety of mailbox access and retrieval protocols and methods to allow users to read email. Three different models describe how these different methods work: the *online model*, in which email is accessed and read on the server; the *offline model*, in which mail is transferred to the client device and used there; and the *disconnected model*, in which mail is retrieved and read offline but remains on the server with changes synchronized for consistency.

TCP/IP Post Office Protocol (POP/POP3)

The overall communication model used for TCP/IP email provides many options to an email user for accessing her electronic mailbox. The most popular access method today is the simple offline access model, in which a client device accesses a server, retrieves mail, and deletes it from the server. POP was designed for quick, simple, and efficient mail access; it is used by millions of people to access billions of email messages every day.

POP Overview, History, Versions, and Standards

Of the three mailbox access paradigms—online, offline, and disconnected—the offline model is probably the least capable in terms of features. And it is also the most popular. This may seem counterintuitive, but it is in fact a pattern that repeats itself over and over in the worlds of computing and networking. The reason is that *simplicity* and *ease of implementation* are keys to the success of any technology, and the offline mail access model beats the other two in these areas.

The history of offline email access goes back farther than one might expect—to the early 1980s. Two decades ago, not everyone and his brother were accessing the Internet to check email the way we do today. In fact, only a relatively small number of machines were connected using TCP/IP, and most users of these machines could access their email on a server, using the online access model.

However, even back then, developers recognized the advantages of being able to retrieve email from a server directly to a client computer, rather than accessing the mailbox on the server using Telnet or Network File System (NFS). In 1984, RFC

918 was published, defining POP. This protocol provided a simple way for a client computer to retrieve email from a mailbox on an SMTP server so it could be used locally.

The emphasis was on *simple*. The RFC for this first version of POP is only five pages long, and the standard it defined is extremely rudimentary. It describes a simple sequence of operations in which a user provides a name and password for authentication and then downloads the entire contents of a mailbox. Simple is good, but simple has limits.

RFC 937, “Post Office Protocol - Version 2” was published in February 1985. POP2 expanded the capabilities of POP by defining a much richer set of commands and replies. This included the ability to read only certain messages, rather than dumping a whole mailbox. Of course, this came at the cost of a slight increase in protocol complexity, but POP2 was still quite simple as protocols go.

These two early versions of POP were used in the mid-1980s, but not very widely. Again, this is simply because the need for an offline email access protocol was limited at that time; most people were not using the Internet before the 1990s.

In 1988, RFC 1081 was published, describing POP3. By this time, the personal computer (PC) was transitioning from a curiosity to a place of importance in the worlds of computing and networking. POP3 was based closely on POP2, but the new version was refined and enhanced with the idea of providing a simple and efficient way for PCs and other clients not normally connected to the Internet to access and retrieve email.

Development on POP3 continued through the 1990s, with several new RFCs published every couple of years. RFC 1081 was made obsolete by, in turn, RFCs 1225, 1460, 1725, and 1939. Despite the large number of revisions, the protocol itself has not changed a great deal since 1988; these RFCs contain only relatively minor tweaks to the original description of the protocol. RFC 1939 was published in 1996, and POP3 has not been revised since that time, though a few subsequent RFCs define optional extensions and additions to the basic protocol, such as alternative authentication mechanisms.

While POP3 has been enhanced and refined, its developers have remained true to the basic idea of a very simple protocol for quick and efficient email transfer. POP3 is a straightforward state-based protocol, with a client and server proceeding through three stages during a session. A very small number of commands is defined to perform simple tasks, and even after all its changes and revisions, the protocol has a minimum of fluff.

For reasons that are unclear to me, almost everyone refers to POP with its version number—that is, they say *POP3* instead of *POP*. This is true despite most people not using version numbers with many other protocols, and almost no one using any other version of POP. But it is the convention, and I will follow it in the rest of this discussion.

KEY CONCEPT POP is currently the most popular TCP/IP email access and retrieval protocol. It implements the offline access model, allowing users to retrieve mail from their SMTP server and use it on their local client computers. It is specifically designed to be a simple protocol and has only a small number of commands. The current revision of POP is version 3, and the protocol is usually abbreviated *POP3*.

NOTE Some implementations of POP attempt to implement the disconnected access model, with limited success. More often, however, IMAP is used for this purpose, since it is better suited to that access model. See the overview of IMAP later in this chapter for more details.

POP3 General Operation

POP3 is a regular TCP/IP client/server protocol. In order to provide access to mailboxes, POP3 server software must be installed and continuously running on the server on which the mailboxes are located. This does not necessarily have to be the same physical hardware device that runs the SMTP server software that receives mail for those boxes—a mechanism such as NFS may be used to allow both the POP3 and SMTP servers to “see” mailboxes locally. POP3 clients are regular end-user email programs that make connections to POP3 servers to get mail; examples include Microsoft Outlook and Eudora Email.

POP3 uses TCP for communication, to ensure the reliable transfer of commands, responses, and message data. POP3 servers listen on well-known port number 110 for incoming connection requests from POP3 clients. After a TCP connection is established, the POP3 session is activated. The client sends commands to the server, which replies with responses and/or email message contents.

POP3 commands are three or four letters long and are case-insensitive. They are all sent in plain ASCII text and terminated with a CRLF sequence, just as with FTP and SMTP commands. POP3 replies are also textual, but the protocol does not use the complex three-digit reply code mechanism of FTP (and SMTP). In fact, it defines only two basic responses:

+OK A positive response, sent when a command or action is successful

-ERR A negative response, sent to indicate that an error has occurred

These messages may be accompanied by explanatory text, especially in the case of an ERR response, to provide more information about the nature of the error.

POP3 Session States

POP3 is described in terms of a *finite state machine (FSM)*, with a session transitioning through three states during the course of its lifetime, as shown in Figure 78-1. (I describe the concepts behind using FSM as a descriptive tool in Chapter 47.) Fortunately, unlike the FSMs of protocols like TCP, this one really is simple, because it is *linear*. The session goes through each state once and only once, in the following sequence:

1. **Authorization State** The server provides a greeting to the client to indicate that it is ready for commands. The client then provides authentication information to allow access to the user’s mailbox.
2. **Transaction State** The client is allowed to perform various operations on the mailbox. These include listing and retrieving messages and marking retrieved messages for deletion.

3. **Update State** When the client is finished with all of its tasks and issues the QUIT command, the session enters this state automatically, where the server actually deletes the messages marked for deletion in the Transaction state. The session is then concluded, and the TCP connection between the two is terminated.

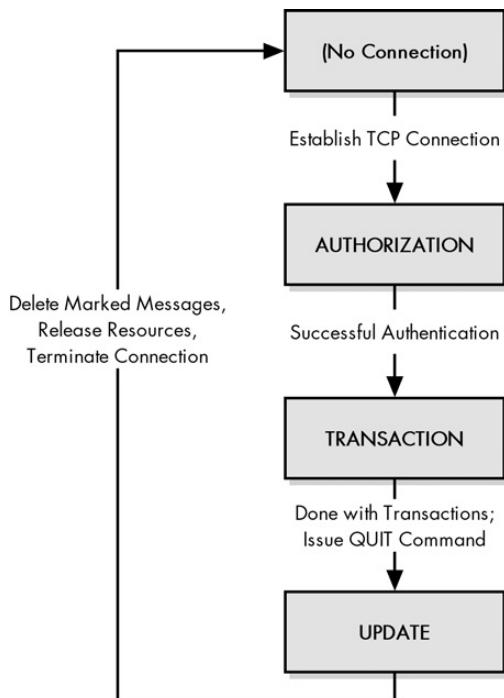


Figure 78-1: POP3 finite state machine
 POP uses a finite state machine (FSM) to describe its operation, but it is very simple because it is linear. Once a TCP connection is established between a POP3 client and POP3 server, the session proceeds through three states in sequence, after which the connection is terminated.

POP3 is designed so that only certain commands may be sent in each of these states. Here, I will describe the activities that take place in these three states, including the commands that are issued by the client in each.

KEY CONCEPT POP3 is a client/server protocol that is described using a simple linear sequence of states. A POP3 session begins with a POP3 client making a TCP connection to a POP3 server, at which point the session is in the *Authorization* state. After successful authentication, the session moves to the *Transaction* state, where the client can perform mail access transactions. When it is finished, the client ends the session and the *Update* state is entered automatically, where cleanup functions are performed and the POP3 session ended.

POP3 Authorization State: User Authentication Process and Commands

A session between a POP3 client and a POP3 server begins when the client sends a TCP connection request to the server. The connection is established using the standard TCP three-way handshake, and the POP3 session commences. The first of the three states of a POP3 session, the Authorization state, is responsible for authenticating the POP3 client with the server.

When the session first enters this state, the server sends a greeting message to the client. This tells the client that the connection is alive and ready for the client to send the first command. An example of such a greeting follows:

```
+OK POP3 server ready
```

The client is now required to authenticate the user who is trying to access a mailbox. This proves that the user has the right to access the server and identifies the user so the server knows which mailbox is being requested.

The normal method of authorization in POP3 is a standard user name/password login. This is pretty much identical to how a login is performed in FTP; even the commands are the same. First the client issues a **USER** command along with the user's mailbox name (his user name or email address). The server responds with an intermediate acknowledgment. The client then uses the **PASS** command to send the user's password. Assuming the login is valid, the server responds to the client with an acknowledgment that indicates successful authentication. The response will also typically specify the number of messages waiting for the user in the mailbox. This process is illustrated in Figure 78-2.

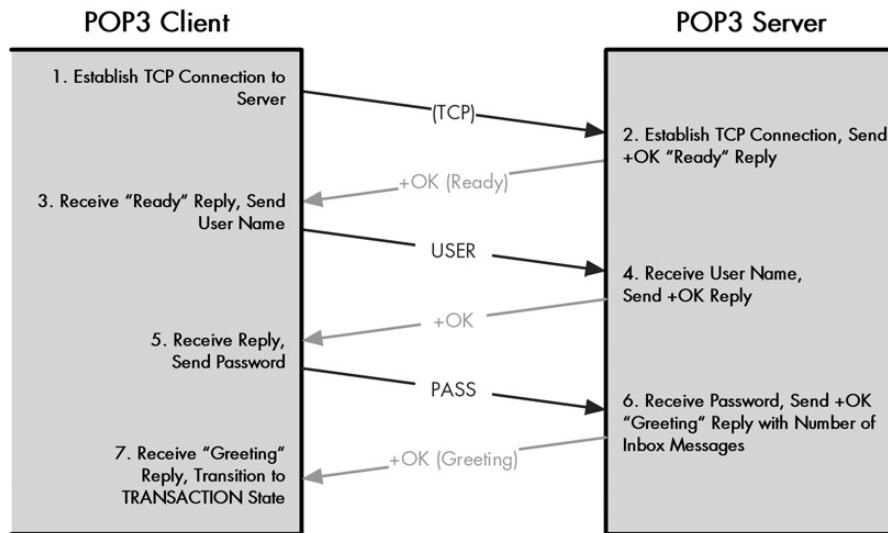


Figure 78-2: POP3 user authentication process Once the TCP connection is established from the client to the server, the server responds with a greeting message, and the simple POP3 authentication process begins. The client sends a user name and password to the server using the **USER** and **PASS** commands, and the server evaluates the information to determine whether or not it will allow the client access.

Listing 78-1 shows an example POP3 authorization, with the client's commands in boldface and the server's responses in italics.

NOTE Some servers may require only the name of the user (**jane**), while others require the full email address, as shown in Listing 78-1.

```
+OK POP3 server ready
USER jane@somewhereelse.com
+OK
PASS *****
+OK jane@somewhereelse.com has 3 messages
```

Listing 78-1: Example of POP3 authorization

If authorization is successful, the POP3 session transitions to the Transaction state, where mail-access commands can be performed. If the user name or password is incorrect, an error response is given, and the session cannot proceed. The authorization may also fail due to technical problems, such as an inability by the server to lock the mailbox (perhaps due to new mail arriving via SMTP).

Since user name/password authorization is considered by many people to be insufficient for the security needs of modern internetworks, the POP3 standard also defines an alternative authentication method using the APOP command. This is a more sophisticated technique based on the Message Digest 5 (MD5) encryption algorithm. If the server supports this technique, in its opening greeting it provides a string indicating a *timestamp* that is unique for each POP3 session. The client then performs an MD5 calculation using this timestamp value and a shared secret known by the server and client. The result of this calculation is included in the client's APOP command. If it matches the server's calculation, authentication is successful; otherwise, the session remains in the Authorization state.

POP was also designed to be extendable through the addition of other authentication mechanisms. This process is based on the use of the optional AUTH command, as described in RFC 1734.

KEY CONCEPT A POP3 session begins in the Authorization state, where the client device is expected to authenticate with the server. By default, POP3 uses only a simple user name/password authentication method. Optional authentication methods are also defined for applications requiring more security.

POP3 Transaction State: Mail and Information Exchange Process and Commands

Once the POP3 client has successfully authenticated the user who is performing mailbox access, the session transitions from the Authorization state to the Transaction state. There's no real mystery as to what this phase of the connection is all about: The POP3 client issues the commands that perform mailbox access and message retrieval transactions.

Most of the commands defined in POP3 are valid only in the Transaction state. Table 78-1 lists each of them, in the order in which they appear in RFC 1939.

Table 78-1: POP3 Transaction Commands

Command Code	Command	Parameters	Description
STAT	Status	None	Requests status information for the mailbox. The server will normally respond, telling the client the number of messages in the mailbox and the number of bytes of data it contains. Optionally, more information may also be returned.
LIST	List Messages	Optional message number	Lists information for the messages in a mailbox; generally this means showing the message number and its size. If a message number is given, only that message's information is provided; otherwise, the full contents of the mailbox are described, one line at a time, with a line containing just a single period at the end.
RETR	Retrieve	Message number	Retrieves a particular message from the mailbox. The server responds with a standard +OK message and then immediately sends the message in RFC 822 format, one line at a time. A line with a single period is sent after the last line.
DELE	Delete	Message number	Marks a message as deleted. Once deleted, any further attempt to access a message (using LIST or RETR, for example) results in an error.
NOOP	No Operation	None	Does nothing; the server just returns an +OK reply.
RSET	Reset	None	Resets the session to the state it was in upon entry to the Transaction state. This includes undeleting any messages already marked for deletion.
TOP	Retrieve Message Top	Message number and number of lines	Allows a client to retrieve only the beginning of a message. The server returns the headers of the message and only the first N lines, where N is the number of lines specified. This command is optional and may not be supported by all servers.
UIDL	Unique ID Listing	Optional message number	If a message number was specified, returns a unique identification code for that message; otherwise, returns an identification code for each message in the mailbox. This command is optional and may not be supported by all servers.

The Transaction state is relatively unstructured in that commands do not need to be issued in any particular order to meet the requirements of the standard. However, there is a natural progression to how a mailbox is retrieved, and that means the commands are usually used in the following order:

1. The client issues a STAT command to see the number of messages in the mailbox.
2. The client issues a LIST command, and the server tells it the number of each message to be retrieved.
3. The client issues a RETR command to get the first message and, if successful, marks it for deletion with DELE. The client uses RETR/DELE for each successive message.

Listing 78-2 and Figure 78-3 show a sample access sequence for a mailbox containing two messages that total 574 bytes; the client's commands are in boldface and the server's responses are in italics.

```
STAT
+OK 2 574
LIST
+OK
1 414
2 160
.
RETR 1
+OK
(Message 1 is sent)
.
DELE 1
+OK message 1 deleted
RETR 2
+OK
(Message 2 is sent)
.
DELE 2
+OK message 2 deleted
QUIT
```

Listing 78-2: Example of the POP3 mail exchange process

The exact message sent in reply to each command is server-dependent; some say +OK, while others provide more descriptive text, as I have done here for the responses to the DELE command.

KEY CONCEPT After successful authorization, the POP3 session transitions to the Transaction state, where the client actually accesses email messages on the server. The client normally begins by first retrieving statistics about the mailbox from the server and obtaining a list of the messages in the mailbox. The client then retrieves each message one at a time, marking each retrieved message for deletion on the server.

In some cases, a POP3 client may be configured to *not* delete messages after retrieving them. This is useful, for example, when Web-based access is being combined with a conventional email client program.

POP3 Update State: Mailbox Update and Session Termination Process and Commands

Once the POP3 client has completed all the email message access and retrieval transactions that it needs to perform, it isn't quite finished yet. The POP3 standard defines a final session state, the Update state, to perform various housekeeping functions, after which both the POP3 session and the underlying TCP connection are terminated.

The transition from the Transaction state to the Update state occurs when the POP3 client issues the QUIT command. This command has no parameters and serves to tell the POP3 server that the client is finished and wishes to end the session. The POP3 standard lists this command as part of its description of the Update state, though it is actually issued from the Transaction state.

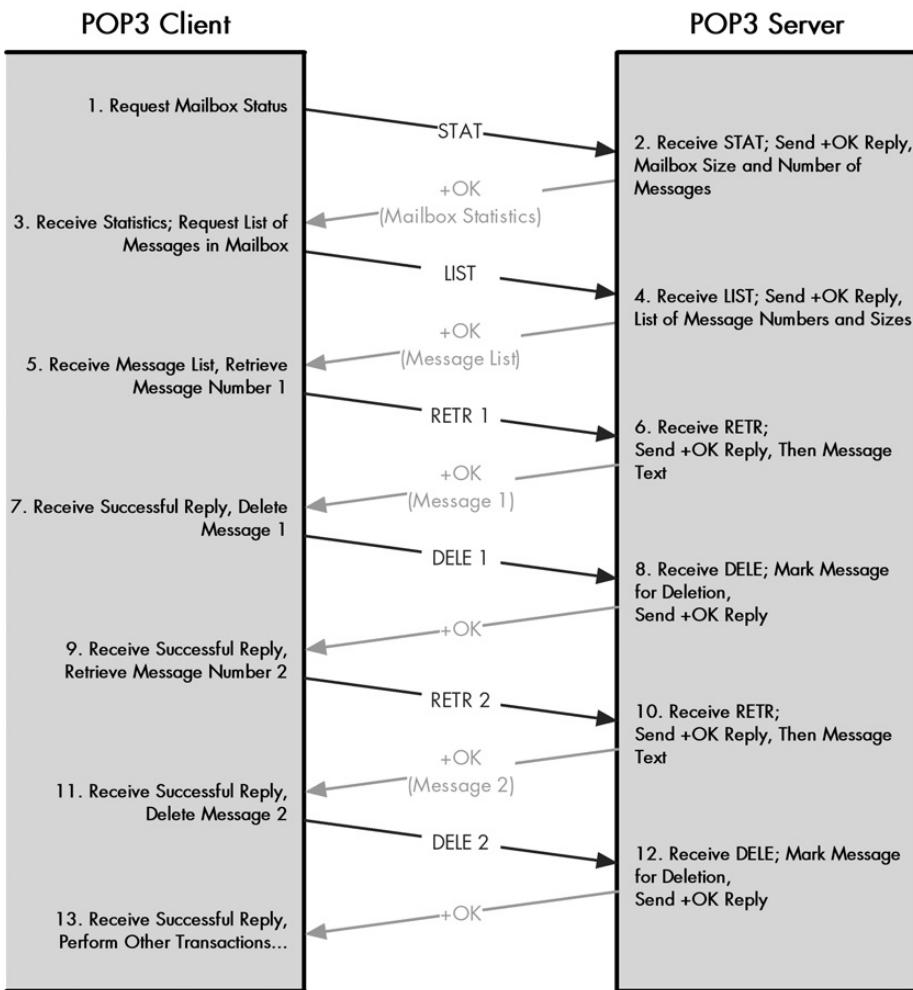


Figure 78-3: POP3 mail exchange process This diagram shows the typical exchange of commands and replies employed by a POP3 client to retrieve email from a POP3 server. The STAT command is used to get mailbox statistics, followed by the LIST command to obtain a list of message numbers. Each message in turn is then retrieved using RETR and marked for deletion by DELE. (Messages are not actually deleted until the Update state is entered.)

After the POP3 server receives the QUIT command, it deletes any messages that were previously marked for deletion by the DELE command in the Transaction state. It's interesting to note that POP chose to implement this two-stage deletion process. The standard doesn't describe specifically why this was done, but it seems likely that it is a precaution to insure against accidental deletion and loss of mail.

By delaying actual deletion until the Update state, the server can verify that it has received and processed all commands prior to the move to the Update state. This also allows the deletion of messages to be undone if necessary, using the RSET command, if the user changes her mind about the deletion prior to exiting the Transaction state. Finally, if any problem occurs with communication between the

client and server that causes the TCP connection to be interrupted prematurely before the QUIT command is issued, no messages will be removed from the mailbox, giving the client a second chance to retrieve them in case they were not received properly.

Once the deleted messages have been removed, the server returns an acknowledgment to the client: +OK if the update was successful, or -ERR if there was a problem removing one or more of the deleted messages. Assuming no problems occurred, the +OK response will also contain a goodbye message of some sort, indicating that the session is about to be closed. The TCP connection between the client and server is then torn down and the session is done.

KEY CONCEPT When the POP3 client is done with its email transactions, it issues the QUIT command. This causes the Update state to be entered automatically, where the server performs necessary cleanup operations, including deleting any messages marked for deletion in the Transaction state.

A POP3 mail-retrieval session normally lasts a few seconds or minutes, but it can take many minutes if the mailbox is large and the connection between the client and server is slow. There is no limit on how long the client and server can be connected, as long as commands continue to be sent by the client. A POP3 server will normally implement an inactivity timer, however, which is customizable but must have a duration of no less than ten minutes. If the connection is idle for the full duration of the inactivity timer, the server assumes that the client has experienced some sort of a problem and shuts down the connection. If this occurs, the server does not delete any messages marked for deletion—again, this is to give the client another chance to retrieve those messages if a problem occurred getting them the first time.

TCP/IP Internet Message Access Protocol (IMAP/IMAP4)

The offline mailbox access model provides the basic mail access functions that most users need. Using the popular POP3, a user can access her mailbox and retrieve messages so she can read them on her local machine. This model has the advantage of simplicity, but it does not provide many features that are increasingly in demand today, such as keeping track of the status of messages and allowing access from many client devices simultaneously. To provide better control over how mail is accessed and managed, we must use either the online or disconnected access models. IMAP was created to allow these access models to be used; it provides rich functionality and flexibility for the TCP/IP email user.

RELATED INFORMATION *The main price that IMAP pays for having a much richer set of functionality than POP is much more complexity. In this section, I have described IMAP in approximately the same level of detail that I did earlier for POP. Please see the appropriate RFC documents for the full description of the protocol and more discussion of some of its nuances, particularly the syntax of the many commands and parameters, which would take dozens of pages to cover fully here.*

IMAP Overview, History, Versions, and Standards

POP3 has become the most popular protocol for accessing TCP/IP mailboxes, not because of its rich functionality, but in spite of its lack of functionality. POP implements the offline mail access model, where mail is retrieved and then deleted from the server where the mailbox resides, so it can be used on a local machine. Millions of people use POP3 every day to access incoming mail. Unfortunately, due to the way the offline access model works, POP3 cannot be used for much else.

The online model is the one we would use in an ideal world, in which we all would be always connected to the Internet all the time. Offline access is a necessity, however, because most user client machines are connected to the Internet only periodically. The transfer of mail from the server to a client machine removes the requirement that we be online to perform mail functions, but it costs us the benefits of central mail storage on the server.

This may seem counterintuitive: how can it be better to have mail stored on some remote server rather than on our local computer? The main reason for this is flexibility of access. One of the biggest problems with offline access using POP3 is that mail is transferred permanently from a central server to one client machine. This is fine as long as an individual uses only that one machine, but what if the person has separate work and home computers or travels a great deal? And what about a mailbox shared by many users? These concerns have become more and more important in recent years.

Another issue is data security and safety. Mail servers run by Internet service providers (ISPs) are usually located in professionally managed data centers. They are carefully controlled and monitored, and backups occur on a routine basis. Most people do not take this sort of care with their own PCs and Macs, nor do they back up their data routinely. So, it's less likely that people will lose mail that on the server.

Of course, we still have the problem of not wanting to force users to be online all the time to access their mail. The solution is the disconnected mailbox access model, which marries the benefits of online and offline access. Mail is retrieved for local use as in the offline model, so the user does not need to be connected to the server continuously. However, changes made to the mailbox are synchronized between the client and the server. The mail remains on the server, where it can be accessed from a different client in the future, and the server acts as a permanent home base for the user's mail.

Recognizing these benefits, developers made some attempts to implement POP using the disconnected access model. Typically, this was done by using POP commands to retrieve mail but still leave it on the server, which is an option in many client programs. This works, but only to a limited extent; for example, keeping track of which messages are new or old becomes an issue when they are both retrieved and left on the server. POP simply lacks the features required for proper disconnected access because it was not designed for it.

In the mid-1980s, development began at Stanford University on a new protocol that would provide a more capable way of accessing user mailboxes. The result was the Interactive Mail Access Protocol, later renamed the Internet Message Access Protocol (IMAP).

IMAP Features

IMAP was designed for the specific purpose of providing flexibility in how users access email messages. It, in fact, can operate in all three of the access modes: online, offline, and disconnected access. Of these, the online and disconnected access modes are of interest to most users of the protocol; offline access is similar to how POP works.

IMAP allows a user to do all of the following:

- Access and retrieve mail from a remote server so it can be used locally while retaining it on the server.
- Set message flags so that the user can keep track of which messages he has already seen, already answered, and so on.
- Manage multiple mailboxes and transfer messages from one mailbox to another. You can organize mail into categories, which is useful for those working on multiple projects or those who are on various mailing lists.
- Determine information about a message prior to downloading it, to decide whether or not to retrieve it.
- Download only portions of a message, such as one body part from a MIME multipart message. This can be quite helpful in cases where large multimedia files are combined with short text elements in a single message.
- Manage documents other than email. For example, IMAP can be used to access Usenet messages.

Of course, there are some disadvantages to IMAP, but not many. One disadvantage is that it is more complex, but it's really not that complex, and the protocol has been around for enough years that this is not a big issue. The most important sticking point with IMAP is simply that it is used less commonly than POP, so providers that support it are not as easy to find as those that support POP. This is changing, however, as more people discover IMAP's benefits.

KEY CONCEPT POP is popular because of its simplicity and long history, but it has few features and normally supports only the rather limited offline mail access method. To provide more flexibility for users in how they access, retrieve, and work with email messages, IMAP was developed. IMAP is used primarily in the online and disconnected access models. It allows users to access mail from many different devices, manage multiple mailboxes, select only certain messages for downloading, and much more. Due to its many capabilities, it is growing in popularity.

IMAP History and Standards

IMAP has had a rather interesting history—interesting in the sense that the normal orderly development process that is used for most TCP/IP protocols broke down. The result wasn't quite as bad as the chaos that occurred in the development of SNMP version 2 (see Chapter 65), but it was still unusual.

The first version of IMAP formally documented as an Internet standard was IMAP version 2 (IMAP2) in RFC 1064, published in July 1988. This was updated in RFC 1176, August 1990, retaining the same version number. However, it seems that

some of the people involved with IMAP were not pleased with RFC 1176, so they created a new document defining version 3 of IMAP (IMAP3): RFC 1203, published in February 1991. This is described by its authors as a “counter proposal.”

For whatever reason, however, IMAP3 was never accepted by the marketplace. Instead, people kept using IMAP2 for a while. An extension to the protocol was later created, called IMAP2bis, which added support for Multipurpose Internet Mail Extensions (MIME) to IMAP. This was an important development due to the usefulness of MIME, and many implementations of IMAP2bis were created. Despite this, for some reason IMAP2bis was never published as an RFC. This may have been due to the problems associated with the publishing of IMAP3.

NOTE *bis* is a Latin word meaning again. It is sometimes used to differentiate changed technical documents from their previous versions when no official new version number is allocated.

In December 1994, IMAP version 4 (IMAP4) was published in two RFCs: RFC 1730 describing the main protocol, and RFC 1731 describing authentication mechanisms for IMAP4. IMAP4 is the current version of IMAP that is widely used today. It continues to be refined; the latest specific version is actually called version 4rev1 (IMAP4rev1), defined in RFC 2060, and then most recently by RFC 3501. Most people still just call this *IMAP4*, and that’s what I will do in the rest of this section.

IMAP General Operation

IMAP4 is a standard client/server protocol like POP3 and most other TCP/IP application protocols. For the protocol to function, an IMAP4 server must be operating on the server where user mailboxes are located. Again, as with POP3, this does not necessarily need to be the same physical server that provides SMTP service. The mailbox must in some way be made accessible to both SMTP for incoming mail, and to IMAP4 for message retrieval and modification. A mechanism for ensuring exclusive access to avoid interference between the various protocols is also needed.

IMAP4 uses the Transmission Control Protocol (TCP) for communication. This ensures that all commands and data are sent reliably and received in the correct order. IMAP4 servers listen on well-known port number 143 for incoming connection requests from IMAP4 clients. After a TCP connection is established, the IMAP4 session begins.

IMAP Session States

The session between an IMAP4 client and server is described in the IMAP standards using an FSM. Again, this is similar to how POP3 operates, except that IMAP4 is a bit more complex. Its FSM defines four states instead of three, and where a POP3 session is linear (going through each state only once) in IMAP4 the session is not. However, the state flow is still fairly straightforward, mostly following a logical sequence from one state to the next. The IMAP FSM is illustrated in Figure 78-4.

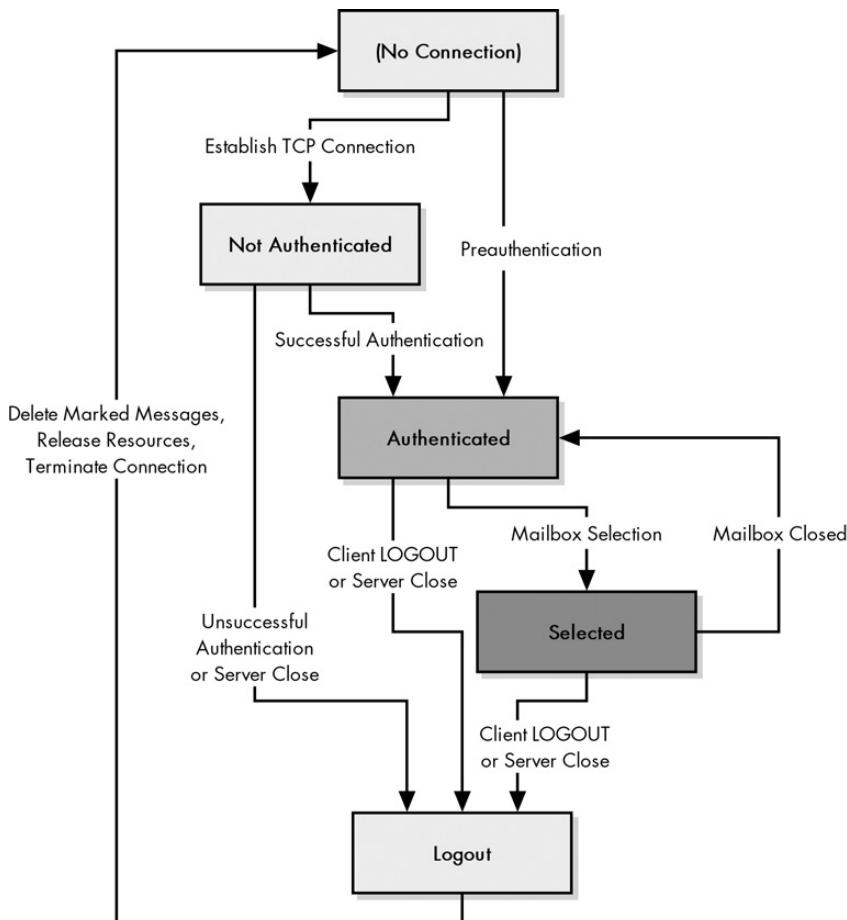


Figure 78-4: IMAP FSM The IMAP FSM is slightly more complex than that of POP (shown in Figure 78-1) but it's still rather straightforward. Once the TCP connection is made between client and server, the Not Authenticated state is entered; after successful authorization, the session moves to the Authenticated state. The session may move between Authenticated and Selected several times, as different mailboxes are selected for use and then closed when no longer needed. From any state the session may be terminated, entering the Logout state.

The following are the IMAP states, in the usual sequence in which they occur for a session:

1. **Not Authenticated State** The session normally begins in this state after a TCP connection is established, unless the special IMAP *preatheautication* feature has been used (we'll get to this feature shortly). At this point, the client cannot really do much aside from providing authentication information so it can move to the next state.
2. **Authenticated State** The client has completed authentication, either through an authentication process in the prior state or through preauthentication. The client is now allowed to perform operations on whole mailboxes. The client must select a mailbox before individual message operations are permitted.

3. **Selected State** After a mailbox has been chosen, the client is allowed to access and manipulate individual messages within the mailbox. When the client is finished with the current mailbox, it can close it and return to the Authenticated state to select a new one to work with, or it can log out to end the session.
4. **Logout State** The client may issue a Logout command from any of the other states to request that the IMAP session be ended. The session may also enter this state if the session inactivity timer expires. The server sends a response, and the connection is terminated.

KEY CONCEPT IMAP is a client/server application, and an IMAP session begins with the client making a TCP connection to the server. The session then normally starts in the Not Authenticated state and remains there until successful authentication. In the Authenticated state, the client may perform operations on whole mailboxes, but a mailbox must be *selected* to transition to the Selected state, where individual messages can be manipulated. The client can work with many mailboxes by selecting each one in turn; it then logs out from the server.

Of the four IMAP states, only the first three are *interactive*, meaning states in which commands are actively issued by the client and responses provided by the server. Some IMAP commands can be used while the session is in any state; others are state-specific.

Session Establishment and Greeting

The server determines in which state the IMAP session begins and sends a *greeting* message to tell the client the session is established and indicate which state it is in. Normally, the server will begin the session in the Not Authenticated state. This is conveyed to the client with the normal OK greeting message, such as this:

* OK <server-name> server ready

Preattentification

In certain circumstances, a server may already know the identity of the client, perhaps as a result of some external authentication mechanism not part of the IMAP protocol. In this case, a special greeting is used:

* PREAUTH <server-name> server ready, logged in as <user-name>

This tells the client that it is already in the Authenticated state.

If the server decides for whatever reason not to accept a new session from the client, it can respond with a BYE response, instead of OK or PREAUTH, and close the TCP connection.

IMAP Commands, Results, and Responses

Once an IMAP session is established, all communication between the client and server takes place in the form of *commands* sent by the client and *responses* returned by the server. Like POP3, commands and responses are sent as strings of ASCII text

and terminated with a CRLF sequence, making them compatible with the way data is sent using the Telnet Protocol. However, IMAP has a few differences from POP and many other TCP/IP application protocols.

The first interesting thing about IMAP commands is that most are not abbreviated into codes of three or four letters—they are spelled out in full. So where POP3 has a STAT command, the command in IMAP is called STATUS. Commands are normally shown in uppercase, as I do in this book, but they are case-insensitive.

IMAP also uses an interesting system of *command tagging* to match client commands explicitly with certain server responses. Each time a client sends a command, it prefixes it with a tag that is unique for the particular session. The tags are usually short strings with a monotonically increasing number in them; the examples in the IMAP standards have the first command tagged a0001, the second a0002, and so on. That said, as long as each command is uniquely labeled, it doesn't matter what tagging scheme is used. When the server needs to send a response that is specific to a command, it tags the reply with the appropriate command tag. Not all replies are tagged, however.

The standard doesn't state explicitly why this tagging scheme is needed, but I believe it is probably related to IMAP's multiple command feature. IMAP clients are allowed to send a sequence of commands to the server to be processed, rather than sending commands only one at a time. This can improve performance when certain commands would take a long time to complete. The only restriction is that the commands must be independent enough that the result of executing them all would be the same, regardless of the order in which they were processed. For example, sending a command to read a particular entity in combination with a command to store a value into the same entity is not allowed.

KEY CONCEPT IMAP tags its commands with a unique identifier. These tags can then be used in replies by the server to match replies with the commands to which they correspond. This enables multiple commands to be sent to an IMAP server in succession.

Command Groups

IMAP commands are organized into groups based on which session states the IMAP session may be in when they are used:

“Any State” Commands A small number of commands that can be used at any time during an IMAP session.

Not Authenticated State Commands Commands that can be used only in the Not Authenticated state. They are usually used for authentication, of course.

Authenticated State Commands Commands used to perform various actions on mailboxes. (Note that despite the name, these commands can also be used in the Selected state.)

Selected State A set of commands for accessing and manipulating individual messages that can be used only in the Selected state.

The reason for having the distinct Authenticated and Selected states and command groups is that IMAP is designed specifically to enable the manipulation of multiple mailboxes. After the session starts and the client is authenticated, the client is allowed to issue commands that work with entire mailboxes. However, it may not issue commands that manipulate individual messages until it tells the server which mailbox it wants to work with, which puts it in the Selected state. The client can also issue mailbox commands from the Selected state.

NOTE *In addition to these four state groups, the standard also defines an extension mechanism that allows new commands to be defined. These must begin with the letter X.*

“Any State” Commands

Table 78-2 describes the IMAP “any state” commands, which can be used whenever needed.

Table 78-2: IMAP “Any State” Commands

Command	Parameters	Description
CAPABILITY	None	Asks the server to tell the client what capabilities and features it supports.
NOOP (No Operation)	None	Does nothing. May be used to reset the inactivity timer or to prompt the server periodically to send notification if new messages arrive.
LOGOUT	None	Tells the server that the client is done and ready to end the session, which transitions to the Logout state for termination.

Results and Responses

Each command sent by the IMAP client elicits some sort of reaction from the IMAP server. The server takes action based on what the client requested and then returns one or more text strings to indicate what occurred. The server can send two types of replies after a command is received:

Result A reply usually indicating the status or disposition of a command. It may be tagged with the command tag of the command whose result it is communicating, or it may be a general message that is not tagged.

Response Any type of information that is being sent by the server to the client. It is usually not tagged with a command tag and is not specifically intended to indicate server status.

NOTE *The IMAP standards sometimes use the terms result, response, and reply in a manner that I find to be inconsistent. Watch out for this if you examine the IMAP RFCs.*

KEY CONCEPT IMAP servers issue two basic types of replies to client commands: *results* are replies that indicate the success, failure, or status of a command; *responses* are general replies containing many different types of information that the server needs to send to the client.

Result Codes

Three main result codes are sent in reply to a command, and two special ones are used in certain circumstances:

OK A positive result to a command, usually sent with the tag of the command that was successful. May be sent untagged in the server's initial greeting when a session starts.

NO A negative result to a command. When tagged, indicates the command failed; when untagged, serves as a general warning message about some situation on the server.

BAD Indicates an error message. It is tagged when the error is directly related to a command that has been sent and otherwise is untagged.

PREAUTH An untagged message sent at the start of a session to indicate that no authentication is required; the session goes directly to the Authenticated state.

BYE Sent when the server is about to close the connection. It is always untagged and is sent in reply to a Logout command or when the connection is to be closed for any other reason.

Response Codes

In contrast to results, responses are used to communicate a wide variety of information to the client device. Responses normally include descriptive text that provides details about what is being communicated. They may be sent either directly in reply to a command or incidentally to one. An example of the latter case would be if a new message arrives in a mailbox during a session. In this case, the server will convey this information unilaterally at its first opportunity, regardless of what command was recently sent.

The following are the response codes defined by the IMAP standard:

ALERT An alert message to be sent to the human user of the IMAP client to inform him of something important.

BADCHARSET Sent when a search fails due to use of an unsupported character set.

CAPABILITY A list of server capabilities may be sent as part of the initial server greeting so the CAPABILITY command does not need to be used.

PARSE Sent when an error occurs parsing the headers or MIME content of an email message.

PERMANENTFLAGS Communicates a list of message status flags that the client is allowed to manipulate.

READ-ONLY Tells the client that the mailbox is accessible only in a read-only mode.

READ-WRITE Tells the client that the mailbox is accessible in read-write mode.

TRYCREATE Sent when an APPEND or COPY command fails due to the target mailbox not existing, to suggest to the client that it try creating the mailbox first.

UIDNEXT Sent with a decimal number that specifies the next unique identifier value to use in an operation. These identifiers allow each message to be uniquely identified.

UIDVALIDITY Sent with a decimal number that specifies the unique identifier validity value, used to confirm unique message identification.

UNSEEN Sent with a decimal number that tells the client the message that is flagged as not yet seen (a new message).

IMAP Not Authenticated State: User Authentication Process and Commands

An IMAP4 session begins with an IMAP4 client establishing a TCP connection with an IMAP4 server. Under normal circumstances, the IMAP4 server has no idea who the client is, and therefore starts the session in the Not Authenticated state. For security reasons, the client is not allowed to do anything until it is authenticated. Thus, the only purpose of this state is to allow the client to present valid credentials so the session can move on to the Authenticated state.

IMAP Authentication Methods

The IMAP4 standard defines three different mechanisms by which a client may authenticate itself. These are implemented using one or more of the three different commands allowed only in the Not Authenticated state, which are shown in Table 78-3.

Table 78-3: IMAP Not Authenticated State Commands

Command	Parameters	Description
LOGIN	User name and password	Specifies a user name and password to use for authentication.
AUTHENTICATE	Authentication mechanism name	Tells the server that the client wants to use a particular authentication mechanism and prompts the client and server to exchange authentication information appropriate for that mechanism.
STARTTLS	None	Tells the IMAP4 server to use the Transport Layer Security (TLS) protocol for authentication, and prompts TLS negotiation to begin.

In response to a LOGIN or AUTHENTICATE command, the server will send an OK message if the authentication was successful, and then transition to the Authenticated state. It will send a NO response if authentication failed due to incorrect information. The client can then try another method of authenticating or terminate the session with the LOGOUT command.

The three authentication methods are as follows:

Plain Login This is the typical user name/password technique, using the LOGIN command by itself. This is similar to the simple scheme used in POP3, except that in IMAP4 one command is used to send both the user name and password. Since

the command and parameters are sent in plain text, this is by far the least secure method of authentication and is not recommended by the standard unless some other means is used in conjunction.

TLS Login This is a secure login where the Transport Layer Security (TLS) protocol is first enabled with the STARTTLS command, and then the LOGIN command can be used securely. Note that STARTTLS only causes the TLS negotiation to begin and does not itself cause the IMAP client to be authenticated. Either LOGIN or AUTHENTICATE must still be used.

Negotiated Authentication Method The AUTHENTICATE command allows the client and server to use any authentication scheme that they both support. The server may indicate which schemes it supports in response to a CAPABILITY command. After specifying the authentication mechanism to be used, the server and client exchange authentication information as required by the mechanism specified. This may require one or more additional lines of data to be sent.

KEY CONCEPT IMAP supports three basic types of authentication: a plain user name/password login, authentication using the Transport Layer Security (TLS) protocol, or the negotiation of some other authentication method between the client and server. In some cases, the IMAP server may choose to preauthenticate clients that it is able to identify reliably; in which case, the Not Authenticated state is skipped entirely.

IMAP Authenticated State: Mailbox Manipulation/Selection Process and Commands

In the normal progression of an IMAP session, the Authenticated state is the first state in which the IMAP client is able to perform useful work on behalf of its user. This state will normally be reached from the Not Authenticated state after successful authentication using the LOGIN or AUTHENTICATE command. Alternately, a server may preauthenticate a client and begin the session in this state directly.

Once in the Authenticated state, the client is considered authorized to issue commands to the server. However, it may issue only commands that deal with *whole mailboxes*. As mentioned in the general operation overview, IMAP was created to allow access to, and manipulation of, multiple mailboxes. For this reason, the client must specify dynamically which mailbox it wants to use before commands dealing with individual messages may be given. This is done in this state using the SELECT or EXAMINE command, which both cause a transition to the Selected state.

It is also possible that the Authenticated state can be reentered during the course of a session. If the CLOSE command is used from the Selected state to close a particular mailbox, the server will consider that mailbox deselected, and the session will transition back to the Authenticated state until a new selection is made. The same can occur if a new SELECT or EXAMINE command is given from the Selected state but fails.

Authenticated State Commands

Table 78-4 provides a brief description of the mailbox-manipulation commands that can be used in the Authenticated state.

Table 78-4: IMAP Authenticated State Commands

Command	Parameters	Description
SELECT	Mailbox name	Selects a particular mailbox so that messages within it can be accessed. If the command is successful, the session transitions to the Selected state. The server will also normally respond with information for the client about the selected mailbox, as described after this table.
EXAMINE	Mailbox name	The same as the SELECT command, except that the mailbox is opened read-only; no changes are allowed.
CREATE	Mailbox name	Creates a mailbox with the given name.
DELETE	Mailbox name	Deletes the specified mailbox.
RENAME	Current and new mailbox names	Renames a mailbox.
SUBSCRIBE	Mailbox name	Adds the mailbox to the server's set of active mailboxes. This is sometimes used when IMAP4 is employed for Usenet message access.
UNSUBSCRIBE	Mailbox name	Removes the mailbox from the active list.
LIST	Mailbox name or reference string	Requests a partial list of available mailbox names, based on the parameter provided.
LSUB	Mailbox name or reference string	The same as LIST but returns only names from the active list.
STATUS	Mailbox name	Requests the status of the specified mailbox. The server responds providing information such as the number of messages in the box and the number of recently arrived and unseen messages.
APPEND	Mailbox name, message, optional flags, and date/time	Adds a message to a mailbox.

NOTE All of the commands in Table 78-4 may also be used in the Selected state; they should really be called Authenticated+Selected state commands.

When either the SELECT or EXAMINE command is successfully issued, the server will return to the client a set of useful information about the mailbox, which can be used to guide commands issued from the Selected state. This information includes the following three mandatory responses:

<n> EXISTS Tells the client the number of messages in the mailbox.

<n> RECENT Tells the client the number of recently arrived (new) messages.

FLAGS (<flag-list>) Tells the client which flags are supported in the mailbox. These include the following: \Seen, \Answered, \Flagged (marked for special attention), \Deleted, \Draft, and \Recent. (The backslashes are part of the flag names.)

The reply from the server may also contain these optional replies:

UNSEEN <n> The message number of the first unseen message.

PERMANENTFLAGS (<flag-list>) A list of flags (as for the FLAGS response above) that the client is allowed to change.

UIDNEXT <n> The next unique identifier value. This is used to check for changes made to the mailbox since the client last accessed it.

UIDVALIDITY <n> The unique identifier validity value, used to confirm valid UID values.

KEY CONCEPT In the Authenticated state, the IMAP client can perform operations on whole mailboxes, such as creating, renaming, or deleting mailboxes, or listing mailbox contents. The SELECT and EXAMINE commands are used to tell the IMAP server which mailbox the client wants to open for message-specific access. Successful execution of either command causes the server to provide the client with several pieces of important information about the mailbox, after which the session transitions to the Selected state.

IMAP Selected State: Message Manipulation Process and Commands

Once the IMAP client has been authorized to access the server, it enters the Authenticated state, where it is allowed to execute tasks on whole mailboxes. Since IMAP allows multiple mailboxes to be manipulated, message-specific commands cannot be used until the client tells the server which mailbox it wants to work with. Only one mailbox can be accessed at a time in a given session.

After the SELECT or EXAMINE command is successfully issued, the session enters the Selected state. In this state, the full palette of message and mailbox commands is available to the client. This includes the message-specific commands in Table 78-5 as the mailbox commands defined for the Authenticated state. Most of IMAP's message-specific commands do not include a mailbox name as a parameter, since the server knows automatically that the commands apply to whatever mailbox was selected in the Authenticated state.

The session remains in the Selected state for as long as the client continues to have work to do with the particular selected (or examined) mailbox. Three different actions can cause a transition out of the Selected state:

- If the client has nothing more to do when it is done with the current mailbox, it can use the LOGOUT command to end the session.
- The client can use the CLOSE command to tell the server it is finished with the current mailbox but keep the session active. The server will close the mailbox, and the session will go back to the Authenticated state.
- The client can issue a new SELECT or EXAMINE command, which will implicitly close the current mailbox and then open the new one. The transition in this case is from the Selected state back to the Selected state, but with a new current mailbox.

Selected State Commands

Table 78-5 lists the message-specific commands that can be used only in the Selected state.

Table 78-5: IMAP Selected State Commands

Command	Parameters	Description
CHECK	None	Sets a checkpoint for the current mailbox. This is used to mark when a certain sequence of operations has been completed.
CLOSE	None	Explicitly closes the current mailbox and returns the session to the Authenticated state. When this command is issued, the server will also implicitly perform an EXPUNGE operation on the mailbox.
EXPUNGE	None	Permanently removes any messages that were flagged for deletion by the client. This is done automatically when a mailbox is closed.
SEARCH	Search criteria and an optional character set specification	Searches the current mailbox for messages matching the specified search criteria. The server response lists the message numbers meeting the criteria.
FETCH	Sequence of message numbers and a list of message data items (or a macro)	Retrieves information about a message or set of messages from the current mailbox.
STORE	Sequence of message numbers, message data item name, and value	Stores a value for a particular message data item for a set of messages.
COPY	Sequence of message numbers and a mailbox name	Copies the set of messages specified to the end of the specified mailbox.
UID	Command name and arguments	Used to allow one of the other commands above to be performed using unique identifier numbers for specifying the messages to be operated on, rather than the usual message sequence numbers.

The list in Table 78-5 might seem surprisingly short. You might wonder, for example, where the specific commands are to read a message header or body, delete a message, mark a message as read, and so forth. The answer is that these (and much more) are all implemented as part of the powerful and flexible FETCH and STORE commands.

The FETCH command can be used to read a number of specific elements from either one message or a sequence of messages. The list of message data items specifies what information is to be read. The data items that can be read include the headers of the message, the message body, flags that are set for the message, the date of the message, and much more. The FETCH command can even be used to retrieve part of a message, such as one body part of a MIME multipart message, making it very useful indeed. Special macros are also defined for convenience. For example, the client can specify the message data item FULL to get all the data associated with a message.

The complement to FETCH, the STORE command, is used to make changes to a message. However, this command does not modify the basic message information such as the content of headers and the message body. Rather, it exists for changing the message's status flags. For example, after replying to a particular message, the client may set the \Answered flag for that message using the STORE command.

Message deletion in IMAP is done in two stages for safety, as in POP and many other protocols. The client sets the \Deleted flag for whichever messages are to be removed, using the STORE command. The messages are deleted only when the mailbox is expunged, typically when it is closed.

The search facility in IMAP4 is also surprisingly quite sophisticated, allowing the client to look for messages based on multiple criteria simultaneously. For example, with the appropriate syntax, you could search for “all posts that are flagged as having been answered that were sent by Jane Jones before April 1, 2004.” Users of IMAP clients can thus easily locate specific messages even in very large mailboxes without needing to download and hunt through hundreds of messages.

KEY CONCEPT After the client opens a specific mailbox, the IMAP session enters the Selected state, where operations such as reading and copying individual email messages may be performed. The two most important commands used in this state are FETCH, which can be used to retrieve a whole message, part of a message, or only certain message headers or flags; and STORE, which sets a message’s status information. IMAP also includes a powerful search facility, providing users with great flexibility in finding messages in a mailbox. When the client is finished working with a particular mailbox, it may choose a different one and reenter the Selected state, close the mailbox and return to the Authenticated state, or log out, automatically entering the Logout state.

TCP/IP Direct Server Email Access

This final portion of the journey of a TCP/IP email message is usually the job of an email access and retrieval protocol like POP3 or IMAP4. These are *customized* protocols, by which I mean that they were created specifically for the last step of the email communication process. However, there are also several *generic* methods by which an email client can gain access to a mailbox, without the use of a special protocol.

These methods are all variations of the online email access model. They generally work by establishing *direct access* to the server where the mailbox is located. The mailbox itself is just a file on a server somewhere, so if that file can be made available, it can be viewed and manipulated like any other file using an email client program that reads and writes the mailbox file. The following are some of the ways in which this can be done:

Using the SMTP Server Directly The simplest method for gaining access to the mailbox is to log on to the server itself. This is not an option for most people, and even in years gone by, it was not often done, for security and other reasons. However, some people do run their own SMTP servers, giving them considerable control over access to their email.

File Sharing Access Using a protocol such as NFS, it is possible to have a mailbox mounted on a user’s client machine where it can be accessed as if it were a local file. The mail is still on the server and not the client machine, but the communication between the client and the server occurs transparently to both the user and the email client software.

Dial-Up Remote Server Access A user on a client machine dials up a server where her mailbox is located and logs in to it. The user then can issue commands to access mail on that server as if she were logged in to it directly.

Telnet Remote Server Access Instead of dialing in to the server, a user can connect to it for remote access using the Telnet Protocol.

These techniques are much more commonly associated with timesharing systems, which commonly use the UNIX family of operating systems more than others. They are also often combined; for example, remote access is often provided for UNIX users, but most companies don't want users logging in directly to the SMTP server. Instead, an ISP might run an SMTP server on one machine called *mail.companyname.com* and also operate a different server that is designed for client access called *users.companyname.com*. A user could access email by dialing into the users machine, which would employ NFS to access user mailboxes on the mail machine.

Direct server access is a method that has been around for decades. At one time, this was how the majority of people accessed email, for two main reasons. First, if you go back far enough, protocols like POP or IMAP had not yet been developed; the TCP/IP email system as a whole predates them by many years, and direct access was the only option back then. Second, the general way that email and networks were used years ago was different from what it is today. Most individuals did not have PCs at home, and no Internet as we know it existed. Remotely accessing a UNIX server using a modem or Telnet for email and other services was just the way it was done.

I got started using direct server access for email more than ten years ago, and I still use it today. I Telnet in to a client machine and use a UNIX email program called *elm* to access and manipulate my mailbox. To me, this provides numerous advantages:

- Most important, I can access my email using Telnet from any machine on the Internet, anywhere around the world.
- Since I am logged in directly, I get immediate notification when new mail arrives, without needing to check routinely for new mail.
- My mailbox is always accessible, and all my mail is always on a secure server in a professionally managed data center.
- I have complete control over my mailbox and can edit it, split it into folders, write custom spam filters, or do anything else I need to do.

This probably sounds good, but most people today do not use direct server access because of the disadvantages of this method. One big issue is that you must be logged in to the Internet to access your email. Another one, perhaps even larger, is the need to be familiar with UNIX and a UNIX email program. UNIX is simply not as user-friendly as a graphical operating systems such as Windows or the Mac. For example, my UNIX email program doesn't support color and cannot show me attached graphic images. I must extract images and other files from MIME messages and transfer them to my own PC for viewing.

Most ordinary computer users today don't know UNIX and don't want to know it. They are much happier using a fancy graphical email program based on POP3 or IMAP4. However, a number of us old UNIX dinosaurs are still around and believe the benefits of direct access outweigh the drawbacks. (Oh, one other benefit that I forgot to mention is that it's very hard to get a computer virus in email when you use UNIX!)

KEY CONCEPT Instead of using a dedicated protocol like POP3 or IMAP4 to retrieve mail, on some systems it is possible for a user to have direct server access to email. This is most commonly done on UNIX systems, where protocols like Telnet or NFS can give a user shared access to mailboxes on a server. This is the oldest method of email access. It provides the user with the most control over his mailbox and is well suited to those who must access mail from many locations. The main drawback is that it means the user must be on the Internet to read email, and it also usually requires familiarity with the UNIX operating system, which few people use today.

TCP/IP World Wide Web Email Access

Most email users like the advantages of online access, especially the ability to read mail from a variety of different machines. What they don't care for is direct server access using protocols like Telnet, UNIX, and nonintuitive, character-based email programs. They want online access, but they want it to be simple and easy to use.

In the 1990s, the World Wide Web was developed and grew in popularity very rapidly, due in large part to its ease of use. Millions of people became accustomed to firing up a web browser to perform a variety of tasks, to the point at which using the Web has become almost second nature. It didn't take very long before someone figured out that using the Web would be a natural way of providing easy access to email on a server.

This technique is straightforward. It exploits the flexibility of the Hypertext Transfer Protocol (HTTP) to tunnel email from a mailbox server to the client. A web browser (client) is opened and given a URL for a special web server document that accesses the user's mailbox. The web server reads information from the mailbox and sends it to the web browser, where it is displayed to the user.

This method uses the online access model like direct server access, because requests must be sent to the web server, and this requires the user to be online. The mail also remains on the server, as when NFS or Telnet are used. The big difference between Web-based mail and the UNIX methods is that the former is much easier for nonexperts to use.

Since the idea was first developed, many companies have jumped on the Web-mail bandwagon, and the number of people using this technique has exploded into the millions in just a few years. Many free services even popped up in the late 1990s as part of the dot-com bubble, allowing any Internet user to send and receive email using the Web at no charge (except perhaps for tolerating advertising). Many ISPs now offer Web access as an option in addition to conventional POP/IMAP access, which is useful for those who travel.

There are drawbacks to the technique, however, which as you might imagine are directly related to its advantages. Web-based mail is easy to use, but inflexible; the user does not have direct access to her mailbox and can use only whatever features the provider's website implements. For example, suppose the user wants to search for a particular string in her mailbox; this requires that the Web interface provide this function. If it doesn't, the user is out of luck.

Web-based mail also has a disadvantage that is an issue for some people: performance. Using conventional UNIX direct access, it is quick and easy to read through a mailbox; the same is true of access using POP3, once the mail is downloaded.

In contrast, Web-based mail services mean each request requires another HTTP request/response cycle. The fact that many Web-based services are free often means server overload that exacerbates the speed issue.

Note that when Web-based mail is combined with other methods such as POP3, care must be taken to avoid strange results. If the Web interface doesn't provide all the features of the conventional email client, certain changes made by the client may not show up when Web-based access is used. Also, mail retrieval using POP3 by default removes the mail from the server. If you use POP3 to read your mailbox and then later try to use the Web to access those messages from elsewhere, you will find that the mail is gone—it's on the client machine where you used the POP3 client. Many email client programs now allow you to specify that you want the mail left on the server after retrieving it using POP3.

KEY CONCEPT In the past few years, a new method has been developed to allow email access using the World Wide Web. This technique is rapidly growing in popularity, because it provides many of the benefits of direct server access, such as the ability to receive email anywhere around the world, while being much simpler and easier than the older methods of direct access such as making a Telnet connection to a server. In some cases, Web-based email can be used in combination with other methods or protocols, such as POP3, giving users great flexibility in how they read their mail.

PART III-8

TCP/IP WORLD WIDE WEB AND THE HYPERTEXT TRANSFER PROTOCOL (HTTP)

In my overview of file and message transfer protocols in Chapter 71, I said that the World Wide Web was “almost certainly” the most important TCP/IP application. If anything, I was probably understating the case. The Web is not only quite clearly the most important TCP/IP application today, it is arguably the single most important application in the history of networking, and perhaps even computing as a whole.

This may sound a little melodramatic, but consider what the Web has done in the decade or so that it has been around. It has transformed not only how internetworks are used, but in many ways, it has also changed society itself. The Web put the Internet on the map, so to speak, moving it from the realm of technicians and academics to the mainstream world.

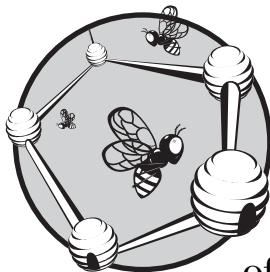
This part contains six chapters that describe the World Wide Web and the all-important *Hypertext Transfer Protocol (HTTP)*, the TCP/IP application layer protocol that makes the Web work. The first chapter discusses the Web and the concepts behind hypertext and hypertext documents in general terms. The second chapter provides an overview of HTTP and describes its operation in general terms, focusing on how connections are established and maintained. The third chapter outlines HTTP messages and how they are formatted, and describes HTTP methods (commands) and status codes. The fourth chapter details the many HTTP headers, which are critically

important because they are the primary way that information is communicated between HTTP servers and clients. The fifth chapter provides information about how resources, called *entities*, are encoded and transferred in HTTP. The sixth and final chapter explores special features and capabilities of the modern HTTP protocol.

Like so many TCP/IP protocols, when HTTP was designed, its creators borrowed elements from other application protocols. In this case, HTTP uses certain elements from email, especially the Multipurpose Internet Mail Extensions (MIME). I would recommend familiarity with both the RFC 822 email message format and MIME, especially MIME headers and media types, before reading this part (both topics are covered in Chapter 76). The relationship between HTTP and MIME is covered more fully in Chapter 83, which discusses HTTP entities and media types.

79

WORLD WIDE WEB AND HYPertext OVERVIEW AND CONCEPTS



The World Wide Web (the Web) expands the concepts of messaging beyond the limits of simple text file transfer of electronic mail (email), File Transfer Protocol (FTP), and Usenet. Its power is in its combination of *hypertext*, a system that allows related documents to be linked together, its rich document format that supports not just text but graphics and multimedia, and the special protocol that allows efficient movement of those media. The result is a powerful system that, once introduced, caught on almost immediately among everyone from large company users to individuals. In a few short years, the Web came to dominate all other applications on the Internet.

In this chapter, I take a high-level, summarized look at the concepts behind the Web. I begin with a short overview and history of the Web and hypertext and a discussion of the components that make up the Web system. I briefly describe the documents and media used on the Web and explain the importance of the Hypertext Markup Language (HTML). I conclude with an overview of how documents are addressed on the Web using Uniform Resource Locators (URLs).

World Wide Web and Hypertext Overview and History

The World Wide Web is one of the members of the class of Internet messaging applications. But for some reason, it just doesn't seem like a message transfer protocol to me. This led me to wonder, what is so special about the Web that caused it to become popular in a way that no prior messaging applications ever had?

There is no truly accurate one-word answer to this question. However, if I had to give one anyway, it would be this: *hypertext*. Sure, applications like email and Usenet allow users to send and receive information, and FTP lets a user access a set of files on a server. But what these methods lack is any way of easily representing the *relationship* between documents or providing a way of moving from one to another. Highly simplified, hypertext does exactly that: It allows the creator of a document to include links to related information, either elsewhere in that document or in other documents. With the appropriate software, a user can easily move from one location to another. So why is this a big deal? In fact, this is more important than it may initially seem.

Without some way of linking documents together, they remain in unconnected islands. In some ways, hypertext-linked documents are to unlinked documents what networked computers are to those that are not networked.

History of Hypertext

The ideas behind hypertext actually go back far beyond the Web and even electronic computers. Vannevar Bush (1890–1974) is generally credited with introducing the idea in his 1945 description of a theoretical device called the *Memex*, which was intended to be used to store and retrieve documents. He described the concept of a *trail* that would link together related information to make it easier to organize and access the information in the device.

Bush's ideas were used as the basis of the work of several researchers who followed. One of these was Ted Nelson, who coined the term *hypertext* and, in 1960, first described a system called *Xanadu*, which is considered one of the original hypertext software models.

The history of the Web itself goes back to 1989 at CERN, the European Organization for Nuclear Research, in Geneva. (The acronym stands for *Conseil Européen pour la Recherche Nucléaire*, the French name of the organization.) Many of the projects undertaken at CERN were large and complex, and they took many years to complete. They also involved many scientists who had to work with and share related documents.

A researcher at CERN, Tim Berners-Lee, proposed the idea of creating a “web” of electronically linked documents. The rapidly growing Internet was the obvious conduit for this project. He designed the first (very crude and simple) version of HTTP for TCP/IP in 1990. He was also responsible for developing or co-developing several of the other key concepts and components behind the Web, such as Uniform Resource Identifiers (URIs) and HTML.

The ability to link documents and files had tremendous appeal, and it took little time before creative individuals found many different uses for this new technology. The early 1990s saw a flurry of development activity. Web server and client software was developed and refined, and the first graphical web browser, *Mosaic*,

was created by the National Center for Supercomputer Applications (NCSA) in 1993. (The developer of this program, Marc Andreessen, eventually formed Netscape Communications.)

Once the Web started to form, it grew very quickly indeed. In fact, to call the growth of the Web anything but *explosive* would not do it justice. In early 1993, only 50 active HTTP web servers existed. By late 1993, more than 1,000 were in service. By late 1995, thousands of new websites were coming online every day, and HTTP requests and responses had overtaken all other TCP/IP application traffic. By the end of the decade, *millions* of websites and more than a billion documents were available on the Web.

The World Wide Web Today

While the rapid growth in the size of the Web is amazing, what is even more fascinating is its growth in *scope*. Since you are reading a book about networking, you are most likely a Web user who is familiar with the incredible array of different types of information you can find on the Web today. Early hypertext systems were based on the use of only text documents; today the Web is a world of many media including pictures, sounds, and movies. The term *hypertext* has in many contexts been replaced with the more generic *hypermedia*—functionally, if not officially.

The Web has also moved beyond providing simple document retrieval to providing a myriad of services. A website can serve up much more than just documents, allowing users to run thousands of kinds of programs to do everything from shop to play music or games online. Websites are also blurring the lines between different types of applications, offering Web-based email, Web-based Usenet access, bulletin boards, and other interactive forums for discussion.

The Web has had an impact on both networking and society as a whole that even its most enthusiastic early fans could never have anticipated. In fact, the Web was the ultimate “killer application” for the Internet as a whole. In the early 1990s, big corporations viewed the Web as an amusing curiosity; by the end of the decade, it was for many a business necessity. Millions of individuals and families discovered the wealth of information at their fingertips, and Internet access became for many another necessary utility, like telephone service. In fact, the huge increase in Web traffic volume spawned the spending of billions of dollars on Internet infrastructure.

The dot-com collapse of the early twenty-first century took some of the wind out of the Web’s sails. The incredible growth of the Web could not continue at its original pace and has slowed somewhat. But the Web as a whole continues to expand and mature, and it will likely be the most important information and service resource on the Internet for some time to come.

KEY CONCEPT The World Wide Web (the Web or WWW) began in 1989 as a project designed to facilitate the representation of relationships between documents and the sharing of information between researchers. The main feature of the Web that makes it so powerful is hypertext, which allows links to be made from one document to another. The many benefits of the Web caused it to grow in only a few short years from a small application to the largest and arguably most important application in the world of networking. It is largely responsible for bringing the Internet into the mainstream of society.

World Wide Web System Concepts and Components

Hypertext is the main concept that makes the Web more than just another message transfer system. However, the idea behind hypertext had been around for decades before the Web was born, as had certain software products based on that idea. Obviously, more than just a concept is needed for an idea to be developed into a successful system.

The Web became a phenomenon because it combined the basic idea of hypertext with several other concepts and technologies to create a rich, comprehensive mechanism for interactive communication. This system today encompasses so many different concepts and software elements, and is so integrated with other technologies, that it's difficult to find any two people who agree on what exactly the Web comprises, and which parts are most critical.

For example, one of the keys to the success of the Web is undeniably the combination of the TCP/IP internetworking protocol suite and the Internet infrastructure that connects together the computers of the world. Is the Internet then an essential component of the Web? In many ways, it is; and, in fact, due to how popular the Web is today, it is common to hear people refer to the Web as *the Internet*. We know that this is not a precise use of terms, of course, but it shows how important the Web has become and how closely it is tied to the Internet.

Major Functional Components of the Web

While the Internet and TCP/IP are obviously important parts of the Web's success, they are generic in nature. When it comes to defining the Web system itself more specifically, three particular components are usually considered most essential (see Figure 79-1):

Hypertext Markup Language HTML is a text language used to define hypertext documents. The idea behind HTML was to add simple constructs, called *tags*, to regular text documents, to enable the linking of one document to another, as well as to allow special data formatting and the combining of different types of media. HTML has become the standard language for implementing information in hypertext and has spawned the creation of numerous related languages.

Hypertext Transfer Protocol HTTP is the TCP/IP application layer protocol that implements the Web, by enabling the transfer of hypertext documents and other files between a client and server. HTTP began as a very crude protocol for transferring HTML documents between computers, and it has evolved to a full-featured and sophisticated messaging protocol. It supports transfers of many different kinds of documents, streaming of multiple files on a connection, and various advanced features including caching, proxying, and authentication.

Uniform Resource Identifiers URIs are used to define labels that identify resources on an internetwork so that they can be easily found and referenced. URIs were originally developed to provide a means by which the users of the Web could locate hypertext documents so they could be retrieved. URIs are actually not specific to the Web, though they are most often associated with the Web and HTTP.

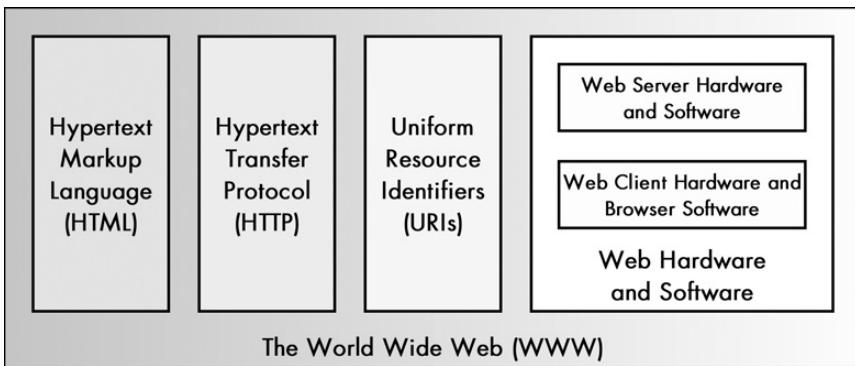


Figure 79-1: Major functional components of the World Wide Web

NOTE Uniform Resource Locators (URLs) are actually a subset of Uniform Resource Identifiers (URIs). The terms are often used interchangeably in World Wide Web discussions.

All three of these components were created and developed at around the same time, and taken together they represent the key technologies that define the Web. In this chapter, I'll describe HTML and the use of URIs in the context of the Web. HTTP is really the heart of the Web and is covered in the remaining five chapters of this part of the book.

Web Servers and Web Browsers

These three main Web components are supplemented by a number of other elements that play supporting roles in rounding out the system as a whole. Chief among these are the hardware and software used to implement client/server communication that makes the Web work, also illustrated in Figure 79-1: *web servers* and *web browsers*.

Web servers are computers that run special server software that allows them to provide hypertext documents and other files to clients who request them. Millions of such machines around the world now serve as a virtual distributed repository of the enormous wealth of information that the Web represents.

Web browsers are HTTP client software programs that run on TCP/IP client computers to access web documents on web servers. These browser programs retrieve hypertext documents and display them, and they also implement many of the Web's advanced features, such as caching. Today's browsers support a wide variety of media, allowing the Web to implement many different functions aside from hypertext document transfer. Examples include displaying images, playing sounds, and implementing interactive programs.

Last, but certainly not least, the *users* of the Web are perhaps its most important component. User involvement has had more of a role in shaping the development of Web technology than any other networking application. The Web began as a simple means of exchanging documents; today, it has grown to encompass thousands of different applications and services, largely as a result of the creativity of its users. Content providers have pushed the boundaries of what the Web can do by creating new ideas for information and services to satisfy the insatiable demands of the end-user community.

KEY CONCEPT The World Wide Web is a complete system comprising a number of related components, of which three are most essential. Hypertext Markup Language (HTML) describes how hypertext documents are constructed. HTML allows links between documents to be represented. The Hypertext Transfer Protocol (HTTP) is the application layer protocol that moves hypertext and other documents over the Web. The Uniform Resource Identifier (URI) mechanism provides a consistent means of identifying resources, both on the Web and more generally on the Internet as a whole.

World Wide Web Media and the Hypertext Markup Language

I've said the Web is based around the central concept of *hypertext*. The prefix *hyper* usually means above or beyond, and thus *hypertext* is like text but goes beyond it in terms of functionality. Documents written in hypertext are similar to regular text files but include information that implements hypertext functions. These are usually called *hypertext documents* or *hypertext files*.

The extra information in a hypertext document is used to tell the computer program that displays the file how to format it. This information takes the form of special instructions that are interspersed with the actual text of the document itself, which are written according to the syntax of a defining language. This addition of extra elements to the content of a document is commonly called *marking up* the document.

Overview of HTML

HTML is one of the three primary system components of the Web and was invented in 1990 by the creator of the Web, Tim Berners-Lee. It was not created in a vacuum; rather, it is a specific application of the general concept of a markup language that is described in ISO standard 8879:1986—the *Standard Generalized Markup Language (SGML)*.

A markup language defines special items that provide information to the software displaying the document about how it should be presented. For the purposes of hypertext, the most basic type of information in a document is a special instruction that specifies how one document can be linked to another—after all, this linking process is the defining attribute of hypertext.

However, HTML goes far beyond just this; it defines a full set of text codes used for describing nearly every aspect of how a document is shown to a user. This includes instructions for formatting text (such as defining its color, size, and alignment), interactive forms, methods for displaying tabular data, specifications for how to present images and other media along with the document, and much more. In theory, the language is only supposed to define the document and leave how it should be displayed up to the browser, but in practice, modern HTML documents also usually contain rather specific instructions for how their information should be presented.

To do justice to HTML, I would need to devote several dozen pages to the subject. I have decided not to do this, because even though HTML is an important part of the Web, it is actually not that important in understanding how the Web works. Knowing HTML is essential if you are writing Web content, and it is also critical if

you want to understand how to write Web software. Perhaps ironically, though, to the actual mechanisms that make the Web work, such as HTTP, a document is a document. HTTP is not designed under the assumption that it will transfer HTML, and in most cases, servers do not even look at the contents of an HTML file—they just transfer it.

That said, a basic understanding of HTML is important, and it just wouldn't seem right not to provide at least an overview of the language, so I will do that here. I encourage you to seek out one of the many good HTML resources if you want to learn more—you'll find dozens of them on the Web (where else?).

HTML Elements and Tags

In simplest terms, an HTML document is a plain ASCII text file, like an email message or other text document. The biggest difference between HTML and regular text is that HTML documents are *structured*; that is, the document is logically organized into a series of *elements* that are arranged according to the rules of the language. Each element defines one part of the document as a whole. The title of a document, a paragraph, a table, and a hyperlink to another document are all examples of elements.

Each element is described using special text *tags* that follow a particular syntax. Each tag begins with the < symbol, which is then followed by the (case-insensitive) element name, and optionally, additional parameters that describe the element. The tag ends with the > symbol. Here's how a tag looks generally:

```
<element parameter1="value1" parameter2="value2" . . .>
```

Some elements are entirely described by the presence of a tag, and in such cases, that tag is the entire element. More often, tags occur in pairs surrounding the actual content of the element; the *start tag* begins with the name of the element, and the *end tag* begins with a slash symbol followed by the name of the element. For example, the title of a document is an element that can be defined as follows:

```
<title>This Is A Great Story</title>
```

The content of each element can contain other elements, which causes tags to be nested within each other. For example, if we wanted to highlight the word *Great* in our title by displaying it in bold letters, we can add the **** tag as follows:

```
<title>This Is A <b>Great</b> Story</title>
```

Each whole HTML document is defined as a single element called `html`; the whole document is enclosed in `<html>` and `</html>` tags. Within this element, the document is divided into two standard subelements that must be present in each document: the `head` and the `body`. The `head` of the document contains information that describes the document and how it is to be processed; it most commonly

contains the title of the document. The body contains the actual content of the document. These three elements define the basic HTML document structure, as follows:

```
<html>
<head>
  (head elements go here...)
</head>
<body>
  (body elements go here...)
</body>
</html>
```

The bulk of the document consists of the body elements that are placed between the `<body>` and `</body>` tags. HTML documents can range from very simple bodies containing only elements such as text paragraphs and perhaps a few links, to very sophisticated documents that are computer-generated and contain hundreds or even thousands of nested tags of various sorts.

Common HTML Elements

Table 79-1 provides a brief description of some of the more common elements used in the body of an HTML message and the tags that define them, to give you a feel for how the language works.

Table 79-1: Common HTML Elements

Element	Example Element and Tags	Description
Paragraph	<code><p>Jack and Jill went up the hill to fetch a pail of water...</p></code>	Delineates a paragraph of text. Note that everything between the start and end tags will be considered one paragraph, even if split onto multiple lines as I have done here. Line breaks are not significant in HTML formatting; only tags are recognized.
Line Break	<code>George W. Bush
The White House
1600 Pennsylvania Ave., NW
Washington, DC 20500</code>	Forces a line break. Used instead of the paragraph tag to present lines close together, such as addresses.
Heading	<code><h1>First Topic</h1><h2>Subtopic</h2></code>	Defines section headings to allow information in a long document to be displayed in hierarchical form. Six sets of tags are defined, from <code><h1></code> and <code></h1></code> to <code><h6></code> and <code></h6></code> . Browsers will automatically display the higher-level headings in more prominent ways, by using larger fonts, underlining the text, or similar treatment.
List	<code><p>Shopping list:MilkEggsSushi</p></code>	Allows information to be presented as a list. The tag <code></code> means unnumbered list and causes the list items to be shown usually as bullet points. Alternatively, <code></code> (ordered list) can be used to show the items preceded by 1, 2, 3, and so on.

(continued)

Table 79-1: Common HTML Elements (continued)

Element	Example Element and Tags	Description
Horizontal Rule	<code>...end of this part of the story.</p><hr size= "3"><p>Start of next part of story...</code>	Draws a horizontal line across the page; the size parameter controls its thickness. Used to separate logical sections in a document.
Image	<code></code>	Displays an inline image in the appropriate section of the text. The src parameter is a relative or absolute URL for the image, and numerous other parameters can be included to define the image's alignment, size, alternate text to display if the browser is nongraphical (as shown here with the alt parameter), and much more.
Link	<code>Click here to visit The PC Guide</code>	Hyperlinks to another document. The a in the tag stands for anchor, which is the formal name for a hyperlink. The href parameter specifies the URL of the link. Most browsers will underline or otherwise highlight text between the start and end tags to make it clear that the text represents a hyperlink. It is also possible to give a hyperlink to an image by combining the and <a> tags.
Bookmark	<code>Step 4: Remove paint using scrubbing tool.</code>	Creates a bookmark that can be used to hyperlink to a particular section in a document. For example, if the bookmark in this example was in a document at URL http://www.homefixitup.com/repainting.htm , the URL http://www.homefixitup.com/repainting.htm#Step4 refers to this particular place in the document. See the discussion of URLs later in this chapter for more details.
Table	<code><table><tr><td>1st row, 1st column.</td><td>1st row, 2nd column.</td></tr><tr><td>2nd row, 1st column.</td><td>2nd row, 2nd column.</td></tr></table></code>	Displays information in tabular form. Each <tr> and </tr> tag set defines one row of the table; within each row, each <td> and </td> pair defines one table data element. Many different parameters can be provided for each of these tags to control table size and appearance.
Form	<code><form method="POST" action="https://www.myfavesite.com/order.php"><input type="hidden" name="PRODUCT" value="widget"><input type="text" name="QUANTITY" size="3"><input type="submit" value="Click Here to Proceed to the Secure Processing Site"></form></code>	Defines an HTML form, allowing various sorts of information to be submitted by a client to a program on a website designed to process forms. The form consists of the initial <form> tag that describes what action to be taken when the submission button is pressed, and other form items such as predefined variables, text-entry fields, and buttons. One example of each of these items is shown here.
Script	<code><script language="javascript">(JavaScript code)</script></code>	Allows instructions in a scripting language to be included in an HTML document. It is most often used for JavaScript.

Common Text Formatting Tags

Numerous tags are used to format the appearance of text within a document; here are some of the more common ones:

text Present the enclosed text in boldface.

<i>text</i> Present the enclosed text in italics.

<u>text</u> Present the enclosed text underlined.

text Present the enclosed text using the indicated font type, size, or color.

This is just the tip of the iceberg when it comes to HTML. If you are not familiar with HTML, however, knowing these basic tags should help you interpret basic HTML documents and learn how HTTP works.

KEY CONCEPT The language used by World Wide Web hypertext documents is called HTML. HTML documents are like ASCII text files, but they are arranged using a special structure of HTML elements that define the different parts of the document and how they should be displayed to the user. Each element is described using special text tags that define it and its characteristics.

World Wide Web Addressing: HTTP Uniform Resource Locators

The main reason that hypertext is so powerful and useful is that it allows related documents to be linked together. In the case of the Web, this is done using a special set of HTML tags that specifies in one document the name of another document that is related in some important way. A user can move from one document to the next using a simple mouse click. The Web has succeeded largely on the basis of this simple and elegant method of referral.

The notion of hyperlinking has some important implications on how Web documents and other resources are addressed. Even though the Web is at its heart a message transfer protocol similar to FTP, the need to be able to define hyperlinks meant that the traditional FTP model of using a set of commands to specify how to retrieve a resource had to be abandoned. Instead, a system was needed whereby a resource could be uniquely specified using a simple, compact string.

The result of this need was the definition of one of the three primary elements of the Web: the *URI*. URIs are divided into two categories: *Uniform Resource Locators (URLs)* and *Uniform Resource Names (URNs)*. While URIs, URLs, and URNs grew out of the development of the Web, they have now been generalized to provide an addressing mechanism for a wide assortment of TCP/IP application layer protocols. They are described in detail in Chapter 70. Here, we will look at how they are used specifically for the Web.

Currently, the Web uses URLs almost exclusively; URNs are still in development. Web URLs specify the use of HTTP for resource retrieval and are thus normally

called *HTTP URLs*. These URLs allow a resource such as a document, graphical image, or multimedia file to be uniquely addressed by specifying the host name, directory path, and filename where it is located.

KEY CONCEPT Uniform Resource Identifiers (URIs) were developed to allow World Wide Web resources to be easily and consistently identified; they are also now used for other protocols and applications. The type of URI currently used on the Web is the Uniform Resource Locator (URL), which identifies the use of HTTP to retrieve a resource, and provides information on where and how it can be found and retrieved.

HTTP URL Syntax

HTTP URLs may be absolute or relative (see “URL Relative Syntax and Base URLs” in Chapter 70 for details on the difference between them). Absolute URLs are usually used for hyperlinks from one website to another or by users requesting a new document without any prior context. Absolute HTTP URLs are based on the following common Internet URL syntax:

```
<scheme>://<user>:<password>@<host>:<port>/<urlpath>;<params>?<query>#<fragment>
```

For the Web, the scheme is `http:`, and the semantics of the different URL elements are defined to have meanings that are relevant to the Web. The general structure of an HTTP URL looks like this:

```
http://<user>:<password>@<host>:<port>/<url-path>?<query>#<bookmark>
```

These syntactic elements are specifically defined for HTTP absolute URLs as follows:

<user> and <password> Optional authentication information, for resources located on password-protected servers. This construct is rarely used in practice, so most people don’t realize it is an option. It has thus become a target of abuse by con artists who use it to obscure undesirable URLs.

<host> The host name of the web server where the resource is located. This is usually a fully qualified Domain Name System (DNS) domain name, but it may also be an IP address.

<port> The TCP port number to use for connecting to the web server. This defaults to port 80 for HTTP and is usually omitted. In rare cases, you may see some other port number used, sometimes to allow two copies of web server software devoted to different uses on the same IP address. Port 8080 is especially common as an alternative.

<url-path> The path pointing to the specific resource to be retrieved using HTTP. This is usually a full directory path expressing the sequence of directories to be traversed from the root directory to the place where the resource is located, and then the resource’s name. It’s important to remember that the path is case-sensitive, even though DNS domain names are not.

<query> An optional query or other information to be passed to the web server. This feature is commonly used to implement interactive functions, because the query value can be specified by the user and then be passed from the web browser to the web server. The alternative method is by using the HTTP POST method.

<bookmark> Identifies a particular location within an HTML document. This is commonly used in very large HTML documents to allow a user to click a hyperlink and scroll to a particular place in the document. See the example near the end of Table 79-1.

Although the URL syntax for the Web is quite rich and potentially complex, most Web URLs are actually quite short. The vast majority of these components are omitted, especially the user, password, port, and bookmark elements. Queries are used only for special purposes. This leaves the more simplified form you will usually encounter for URLs:

`http://<host>/<url-path>`

Resource Paths and Directory Listings

The `<url-path>` used to reference a particular document can also be omitted. This provides a convenient way for a user to see what content is offered on a website without needing to know what particular document to request. For example, a user who wants to see the current headlines on CNN would go to `http://www.cnn.com`. In this case, the request is sent to the web server for the null document (represented by `/`, which is implied if it is not specified; technically, you are supposed to specify `http://www.cnn.com/`).

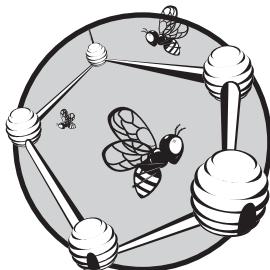
How a `/` request is handled depends on the server. Technically, such a request is actually asking the server, “Please show me the contents of the root directory of the server.” However, this is both ugly (a listing of filenames is not the best way to make a first impression) and a potential security issue (as anyone can see the name of every file on the server). Instead, most HTTP servers are set up to recognize such requests automatically and return a default document, often named something like `index.html` or `default.html`. Many servers will similarly return a default document of some sort if any other directory is specified in a URL; for example, typing `http://www.pcguide.com/ref` in the URL address bar of a web browser actually returns `http://www.pcguide.com/ref/index.htm`.

NOTE While it is technically incorrect to leave the `http://` off an HTTP URL, most web browsers will add it automatically if it's omitted. As a result, many Web users are in the habit of entering URLs that are simply a host name, such as `www.tcpipguide.com`.

The forms shown here apply to absolute HTTP URLs. URLs may also be relative, which is the norm for links between closely related documents, such as graphics that go with a document, or between documents in a set or project. In this case, usually only a fractional portion of a URL path is specified. This is described fully in Chapter 70.

80

HTTP GENERAL OPERATION AND CONNECTIONS



The Hypertext Transfer Protocol (HTTP) began as an extremely basic protocol, designed to do just one thing: allow a client to send a simple request for a hypertext file and receive it back from the server. Modern HTTP remains at its heart a straightforward request/reply protocol, but now includes many new features and capabilities to support the growing size of the World Wide Web (the Web) and the ever-increasing variety of ways that people have found to use the Web. Therefore, the best place to start explaining HTTP is by looking at its operation as a whole and how communication takes place between a web server and a web client.

In this chapter, I introduce HTTP by describing its operation in general terms. I start with an overview of HTTP, discussing its versions and the standards that define them. I then discuss its operational model, which is important to understanding how HTTP works. I explain the two types of connections that are supported between HTTP clients and servers, and the

method by which requests can be pipelined in the current version of HTTP, HTTP/1.1. I then provide more information about how persistent connections are established, managed, and terminated in HTTP/1.1.

HTTP Versions and Standards

The World Wide Web had humble beginnings as a research project at the Swiss research institute, CERN, the European Organization for Nuclear Research. The primary goal of the project was to allow hypertext documents to be electronically linked, so selecting a reference in one document to a second document would cause the reference document to be retrieved. To implement this system, the researchers needed some sort of mechanism to allow a client computer to tell a server to send it a document. To fill this function, the early developers of the Web created a new TCP/IP application layer protocol: the *Hypertext Transfer Protocol (HTTP)*.

This first version is now known as HTTP/0.9. Subsequent versions are HTTP/1.0 and HTTP/1.1.

HTTP/0.9

The original version of HTTP was intended only for the transfer of hypertext documents, and it was designed to be very simple to make implementation of the fledgling Web easier. This early HTTP specifies that an HTTP client establishes a connection to an HTTP server using the Transmission Control Protocol (TCP). The client then issues a single GET request specifying a resource to be retrieved. The server responds by sending the file as a stream of text bytes, and the connection is terminated. The entire document defining this version of HTTP is only a couple of pages long!

This first version of HTTP was functional but extremely limited in its capabilities. It didn't support the transfer of any types of data other than hypertext, and it didn't provide any mechanism for any sort of intelligent communication between the client and server. This early HTTP prototype was not up to the task of providing the basis for data transfer for the future of the Web. It was never made an official RFC standard, and, in fact, never even had a formal version number; it is known today as HTTP version 0.9, or HTTP/0.9, using the HTTP version format. I believe this number has no particular significance, other than being a bit smaller than the number of the first official version of the protocol.

HTTP/1.0

HTTP/0.9's skeleton of functionality formed the basis for a rapid evolution of HTTP in the early 1990s. As the Web grew in size and acceptance, many new ideas and features were incorporated into HTTP. The result of a great deal of development effort was the formalization of the first HTTP standard: version 1.0. The standard for this much enhanced HTTP was published in May 1996 as RFC 1945, "Hypertext Transfer Protocol—HTTP/1.0." It had been in use for several years prior to that formal publication date, however.

HTTP/1.0 transformed HTTP from a trivial request/response application to a true messaging protocol. It described a complete message format for HTTP, and explained how it should be used for client requests and server responses. One of

the most important changes in HTTP/1.0 was the generalization of the protocol to handle many types of different media, as opposed to strictly hypertext documents. To broaden HTTP's scope, its developers borrowed concepts and header constructs from the Multipurpose Internet Mail Extensions (MIME) standard defined for email (discussed in Chapter 76). At the same time that it defined much more capable web servers and web clients, HTTP/1.0 retained backward-compatibility with servers and clients still using HTTP/0.9.

HTTP/1.0 was the version of HTTP that was widely implemented in the mid-1990s as the Web exploded in popularity. After only a couple of years, HTTP accounted for the majority of the traffic on the burgeoning Internet. The popularity of HTTP was so great that it single-handedly prompted the installation of a lot of new hardware to handle the load of browser requests and web server replies.

Unfortunately, much of this huge load of traffic was due to some limitations in HTTP itself. These only became apparent due to the tremendous growth in the use of the protocol, which, combined with the normal growing pains of the Internet, led to many frustrated Web users. The inefficiencies of HTTP/1.0 were a result of design limitations, such as the following:

- The need for each site to be hosted on a different server.
- The fact that each HTTP session handled only one client request.
- A general lack of support for necessary performance-enhancing features such as caching, proxying, and partial resource retrieval.

HTTP/1.1

While impatient pundits coined sarcastic terms such as the “World Wide Wait,” the Internet Engineering Task Force (IETF) continued to work to improve HTTP. In January 1997, the first draft version of HTTP/1.1 appeared, in RFC 2068. This document was later revised and published as RFC 2616, “Hypertext Transfer Protocol—HTTP/1.1,” in June 1999. HTTP/1.1 retains backward-compatibility with both HTTP/1.0 and HTTP/0.9. It is accompanied by RFC 2617, “HTTP Authentication: Basic and Digest Access Authentication,” which deals with security and authentication issues.

HTTP/1.1 introduces several significant improvements over version 1.0 of the protocol, most of which specifically address the performance problems I just described. Some of the more important improvements in HTTP/1.1 include the following:

Multiple Host Name Support In HTTP/1.0, there was no way to specify the host name of the server to which the client needed to connect. As a result, the web server at a particular IP address could support only one domain name. This was not only inefficient, but it also was exacerbating the depletion of IP addresses in the 1990s, because each new web server to come online required a new IP address. HTTP/1.1 allows one web server to handle requests for dozens or even hundreds of different virtual hosts.

Persistent Connections HTTP/1.1 allows a client to send multiple requests for related documents to a server in a single TCP session. This greatly improves performance over HTTP/1.0, where each request required a new connection to the server.

Partial Resource Selection In HTTP/1.1, a client can ask for only part of a resource, rather than needing the get the entire document, which reduces the load on the server and saves transfer bandwidth.

Better Caching and Proxying Support HTTP/1.1 includes many provisions to make caching and proxying more efficient and effective than they were in HTTP/1.0. These techniques can improve performance by providing clients with faster replies to their requests while reducing the load on servers, as well as enhancing security and implementing other functionality.

Content Negotiation HTTP/1.1 has an additional negotiation feature that allows the client and server to exchange information to help select the best resource or version of a resource when multiple variants are available.

Better Security HTTP/1.1 defines authentication methods and is generally more security-aware than HTTP/1.0 was.

In addition to these notable improvements, many other minor enhancements were made in HTTP/1.1. Several of these take the form of new headers that can be included in client requests to better control under what circumstances resources are retrieved from the server, and headers in server responses to provide additional information to the client.

Future HTTP Versions

HTTP/1.1 continues to be the current version of HTTP, even though it is now several years old. This may seem somewhat surprising, given how widely used HTTP is. Then again, it may because so many millions of servers and clients implement HTTP/1.1 that no new version has been created. For a while, there was speculation that version 1.2 of HTTP would be developed, but this has not happened yet.

In the late 1990s, work began on a method of expanding HTTP through extensions to the existing version 1.1. Development of the *HTTP Extension Framework* proceeded for a number of years, and in 1998, a proposed draft for a new Internet standard was created. However, HTTP/1.1 is so widely deployed and so important that it was very difficult to achieve consensus on any proposal to modify it. As a result, when the HTTP Extension Framework was finally published in February 2000 as RFC 2774, the universal acceptance required for a new standard did not exist. The framework was given experimental status and never became a formal standard.

KEY CONCEPT The engine of the World Wide Web (the Web) is the application protocol that defines how web servers and clients exchange information: the *Hypertext Transfer Protocol (HTTP)*. The first version of HTTP, HTTP/0.9, was part of the early Web and was a very simple request/response protocol with limited capabilities that could transfer only text files. The first widely used version was HTTP/1.0, which is a more complete protocol that allows the transport of many types of files and resources. The current version is HTTP/1.1, which expands HTTP/1.0's capabilities with several features that improve the efficiency of transfers and address many of the needs of the rapidly growing modern Web.

HTTP Operational Model and Client/Server Communication

While the Web itself has many different facets, HTTP is concerned with only one basic function: the transfer of hypertext documents and other files from web servers to web clients. In terms of actual communication, clients are chiefly concerned with making requests to servers, which respond to those requests.

Thus, even though HTTP includes a lot of functionality to meet the needs of clients and servers, when you boil it down, you get a very simple, client/server, request/response protocol. In this respect, HTTP more closely resembles a rudimentary protocol like the Boot Protocol (BOOTP) or the Address Resolution Protocol (ARP) than it does other application layer protocols like the File Transfer Protocol (FTP) and the Simple Mail Transfer Protocol (SMTP), which involve multiple communication steps and command/reply sequences.

Basic HTTP Client/Server Communication

In its simplest form, the operation of HTTP involves only an HTTP client, usually a *web browser* on a client machine, and an HTTP server, more commonly known as a *web server*. After a TCP connection is created, the two steps in communication are as follows (see Figure 80-1):

Client Request The HTTP client sends a request message formatted according to the rules of the HTTP standard—an *HTTP Request*. This message specifies the resource that the client wishes to retrieve or includes information to be provided to the server.

Server Response The server reads and interprets the request. It takes action relevant to the request and creates an *HTTP Response* message, which it sends back to the client. The response message indicates whether the request was successful, and it may also contain the content of the resource that the client requested, if appropriate.

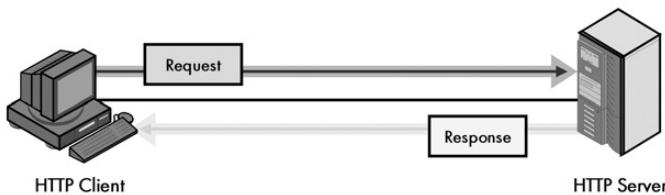


Figure 80-1: HTTP client/server communication In its simplest form, HTTP communication consists of an HTTP Request message sent by a client to a server, which replies with an HTTP Response message.

In HTTP/1.0, each TCP connection involves only one such exchange, as shown in Figure 80-1. In HTTP/1.1, multiple exchanges are possible, as you'll see soon. Note also that, in some cases, the server may respond with one or preliminary responses prior to sending the full response. This may occur if the server sends a preliminary response using the 100 Continue status code prior to the actual reply. See the description of HTTP status codes in Chapter 81 for more information.

KEY CONCEPT HTTP is a client/server-oriented, request/reply protocol. Basic communication consists of an *HTTP Request* message sent by an HTTP client to an HTTP server, which returns an *HTTP Response* message back to the client.

Intermediaries and the HTTP Request/Response Chain

The simple request/response pair between a client and server becomes more complex when *intermediaries* are placed in the virtual communication path between the client and server. These are devices such as *proxies*, *gateways*, or *tunnels* that are used to improve performance, provide security, or perform other necessary functions for particular clients or servers. Proxies are particularly commonly used on the Web, because they can greatly improve response time for groups of related client computers.

When an intermediary is involved in HTTP communication, it acts as a middle-man. Rather than the client speaking directly to the server and vice versa, each talks to the intermediary. This allows the intermediary to perform functions such as caching, translation, aggregation, and encapsulation. For example, consider an exchange through a single intermediary device. The two-step communication process described in the preceding section would become four steps:

1. **Client Request** The HTTP client sends a request message to the intermediary device.
2. **Intermediary Request** The intermediary processes the request, making changes to it if necessary. It then forwards the request to the server.
3. **Server Response** The server reads and interprets the request, takes appropriate action, and then sends a response. Since it received its request from the intermediary, its reply goes back to the intermediary.
4. **Intermediary Response** The intermediary processes the request, again possibly making changes, and then forwards it back to the client.

As you can see, the intermediary acts as if it were a server from the client's perspective and as a client from the server's viewpoint. Many intermediaries are designed to be able to intercept a variety of TCP/IP protocols, by posing as the server to a client and the client to a server. Most protocols are unaware of the existence of intermediaries. HTTP, however, includes special support for certain intermediaries such as proxy servers, providing headers that control how intermediaries handle HTTP requests and replies. (Proxy servers are discussed in Chapter 84.)

It is possible for two or more intermediaries to be linked together between the client and server. For example, the client might send a request to intermediary 1, which then forwards to intermediary 2, which then talks to the server, as illustrated in Figure 80-2. The process is reversed for the reply. The HTTP standard uses the phrase *request/response chain* to refer collectively to the entire set of devices involved in an HTTP message exchange.

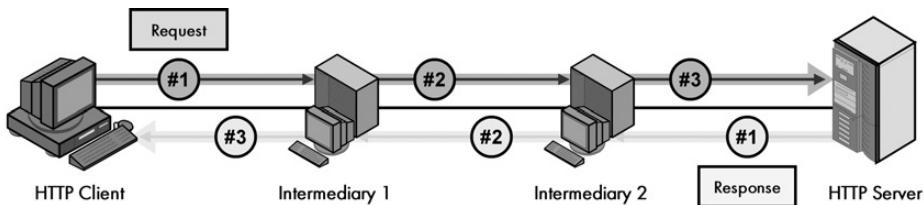


Figure 80-2: HTTP request/response chain using intermediaries Instead of being connected directly, an HTTP client and server may be linked using one or more intermediary devices such as proxies. In this example, two intermediaries are present. The HTTP Request message sent by the client will actually be transferred three times: from the client to the first intermediary, then to the second, and finally to the server. The HTTP Response message will be created once but transmitted three distinct times. The full set of devices participating in the message exchange is called the *request/response chain*.

KEY CONCEPT The simple client/server operational model of HTTP becomes more complicated when *intermediary devices* such as proxies, tunnels, or gateways are inserted in the communication path between the HTTP client and server. HTTP/1.1 is specifically designed with features to support the efficient conveyance of requests and responses through a series of steps from the client through the intermediaries to the server, and back again. The entire set of devices involved in such a communication is called the *request/response chain*.

The Impact of Caching on HTTP Communication

The normal HTTP communication model is changed through the application of *caching* to client requests. Various devices on the Web employ caching to store recently retrieved resources so they can be quickly supplied in reply to a request. The client itself will cache recently accessed web documents, so that if the user asks for them again, they can be displayed without even making a request to a server. If a request is required, any intermediary device can satisfy a request for a file if the file is in its cache.

When a cache is used, the device that has the cached resource requested returns it directly, circumventing the normal HTTP communication process. In the example shown in Figure 80-2, if intermediary 1 has the file the client needs, it will supply

it to the client directly, and intermediary 2 and the web server that the client was trying to reach originally will not even be aware that a request was ever made. Chapter 84 provides details on HTTP caching.

NOTE *Most requests for web resources are made using HTTP URLs based on a Domain Name System (DNS) host name. The first step in satisfying such requests is to resolve the DNS domain name into an IP address, but this process is separate from the HTTP communication itself.*

HTTP Transitory and Persistent Connections and Pipelining

You just learned that the basic HTTP communication process is a simple two-step procedure: A client sends a request to a server, and the server replies back to the client. Since this was all that HTTP was intended to do, the first version of the protocol was designed so that after a TCP connection was established between the client and server, a single request/response exchange was performed. After the request was satisfied, the TCP connection was terminated. These *transitory* connections were the only type supported by the original HTTP/0.9, and the same model was maintained in the more widely deployed HTTP/1.0.

The advantage of this connection model is its conceptual simplicity. The problem with it is that it is inefficient when the client needs to make many requests to the same server. This is often the case with modern hypertext documents, which usually carry inline references to images and other media. A typical client request for the home page of a website begins with a single request for a Hypertext Markup Language (HTML) file, but then leads to subsequent requests for each of the other related files that go with that document.

With transitory connections, each of these requests made by the client requires a new, distinct TCP connection to be set up between the client and server. Every connection takes server resources and network bandwidth, so needing to establish a new one for each file is woefully inefficient. Suppose that you were having a conversation with someone whom you needed to ask a series of questions. Now imagine that after answering each question, the other person hung up the phone, and you had to call her again! You get the picture.

There are some people who consider the temporary nature of HTTP/0.9 and HTTP/1.0 connections to be a design flaw of these early versions of HTTP, but I don't think that this is fair. In the early days, this model of operation was really not a big issue; it became problematic only when the use of the Web and hypertext evolved. For the first few years of its existence, hypertext was primarily that: *text*. Having an HTTP session last just long enough for one request/response was generally sufficient, since the whole resource was in one file. It was only in the 1990s that hypertext became *hypermedia*, with a heavy emphasis on embedded graphics and other files. When web pages changed from simple text to multimedia marvels sporting dozens or even hundreds of embedded images, the limitations of HTTP/1.0 became obvious.

The solution to the problem came in HTTP/1.1, which allows an HTTP client and server to set up a *persistent connection*.

Persistent Connections

With persistent connections, the basic operation of HTTP is not changed. The main difference is that, by default, the TCP connection is kept open after each request/response set, so that the next request and response can be exchanged immediately. The session is closed only when the client is finished requesting all the documents it needs.

Keeping the TCP connection between an HTTP client and server alive between requests is probably the single most important way that HTTP/1.1 improves performance over HTTP/1.0. Clients are able to get their files more quickly because they don't need to wait for a TCP connection before each resource is retrieved. Server load is reduced, and memory use in busy servers is conserved. Network congestion is reduced through the elimination of unnecessary TCP handshaking segments.

Pipelining

Persistent connections offer another important performance-enhancing option to HTTP clients: the ability to *pipeline* requests. Suppose the client needs to send a request for Files A, B, and C to a server. Since the requests for all of these files will be sent in the same TCP session, there is no need for the client to wait for a response to its request for File A before sending the request for File B. The client can send requests in a rapid-fire fashion, one after the other. This also improves the efficiency of the server, which will be able to fill the requests in the order in which they are received, as soon as it is able, without needing to pause to wait for each new request to be sent.

KEY CONCEPT HTTP/0.9 and HTTP/1.0 supported only *transitory connections* between an HTTP client and server, where just a single request and response could be exchanged on a TCP connection. This is very inefficient for the modern Web, where clients frequently need to make dozens of requests to a server. By default, HTTP/1.1 operates using *persistent connections*. This means that once a TCP connection is established, the client can send many requests to the server and receive replies to each in turn. This allows files to be retrieved more quickly, and conserves server resources and Internet bandwidth. The client can even *pipeline* its requests, sending the second request immediately, without needing to wait for a reply to the first request. HTTP/1.1 still supports transitory connections for backward-compatibility, when needed.

The obvious advantages of persistent connections make them the default for modern HTTP communication, but they do have one drawback: They complicate the process of sending data from the server to the client. With transitory connections, the client knows that all of the data it receives back from the server is in reply to the one request it sent. Once it has all the bytes the server sent and the TCP session ends, the client knows the file is complete.

With persistent connections, and especially when pipelining is used, the server will typically be sending one file after the other to the client, which must differentiate them. Remember that TCP sends data as just a series of unstructured bytes; the application must take care of specifying where the dividing points are between files. This means that persistent connections and pipelining lead to data length issues that must be specially addressed in HTTP.

To provide compatibility with older versions of the software, HTTP/1.1 servers still support transitory connections, and they will automatically close the TCP connection after one response if they receive an HTTP/0.9 or HTTP/1.0 request. HTTP/1.1 clients may also specify in their initial request that they do not want to use persistent connections.

HTTP Persistent Connection Establishment and Management

As with most TCP/IP client/server protocols, in establishing a persistent connection, the HTTP server plays the passive role by listening for requests on a particular port number. The default port number for HTTP is well-known TCP port number 80, and is used by web browsers for most HTTP requests, unless a different port number is specified in the Uniform Resource Locator (URL). The client initiates an HTTP connection by opening a TCP connection from itself to the server it wishes to contact.

NOTE *A DNS name resolution step may precede the entire HTTP connection, since most URLs contain a host name, while HTTP requires that the client know the server's IP address. This can lead to confusion, because DNS uses the User Datagram Protocol (UDP), but HTTP uses TCP. This causes some people to think that HTTP uses UDP.*

Once the TCP connection is active, the client sends its first request message. The request specifies which version of HTTP the client is using. If this is HTTP/0.9 or HTTP/1.0, the server will automatically work in the transitory connection model, and it will send only one reply and then close the link. If it is HTTP/1.1, the assumption is that a persistent connection is desired. An HTTP/1.1 client can override this by including the special Connection: Close header in its initial request, which tells the server it does not want to keep the session active after the request it is sending has been fulfilled.

Assuming that a persistent connection is being used, the client may begin pipelining subsequent requests after sending its first request, while waiting for a response from the server to the initial query. As the server starts to respond to requests, the client processes them and takes action, such as displaying the data retrieved to the user. The data received from the server may also prompt the client to request more files on the same connection, as in the case of an HTML document that contains references to images.

The server will generally buffer a certain number of pipelined requests from the client. In the case where the client sends too many requests too quickly, the server may throttle back the client using the flow-control mechanism built into TCP. In theory, the server could also just decide to terminate the connection with

the client, but it is better for it to use TCP's existing features. Closing the connection will cause the client to initiate a new connection, potentially exacerbating any overloading problem.

The flow of requests and responses continues for as long as the client has requests. The connection can be gracefully terminated by the client by including the Connection: Close header in the last request it needs to send to the server. All requests are filled in order, so the server will satisfy all outstanding requests, and then close the session.

Since HTTP/1.1 supports pipelining of requests, there is usually no need for a client to establish more than one simultaneous connection to the same server. Clients occasionally do this anyway to allow them to get information from a server more quickly. This is considered by many to be "antisocial," because it can lead to a busy server's resources being monopolized by one client to the exclusion of others that want to access it.

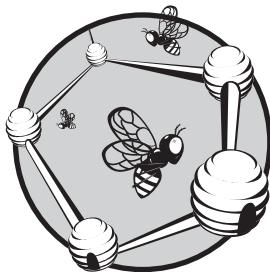
Under special circumstances, either the client or the server may unexpectedly close an active persistent connection. For example, if the client detects that too much time has elapsed since the server last replied, it may conclude that the server has crashed and terminate the connection. Similarly, the server might receive a shutdown command from its administrator or for other reasons end a session with a client abruptly. Servers normally avoid closing down a link during the middle of sending a response.

Both clients and servers must be able to handle abrupt session termination. For servers, there is not much to do; if the client terminates the connection, the server simply cleans up any resources associated with the connection, and then goes on to service the next client.

Clients have more to do when a server prematurely terminates a session, and this is especially the case when requests are pipelined. The client must keep track of all requests sent to the server to ensure that each is filled. If the server closes the session unexpectedly, the client will usually attempt to establish a new connection to retransmit the unfilled requests. Since an abrupt session termination is often a sign of a busy server, the HTTP standard specifies that clients use a binary exponential back-off algorithm to wait a variable but increasing amount of time before resubmitting requests for files (similar in concept to the method used to deal with collisions in Ethernet). This helps prevent clients from piling on requests to a device that is already overwhelmed.

81

HTTP MESSAGES, METHODS, AND STATUS CODES



As you saw in the previous chapter, the Hypertext Transfer Protocol (HTTP) is entirely oriented around the sending of client requests and server responses. These take the form of *HTTP messages* sent between clients and servers. As with all protocols, HTTP uses a special format that dictates the structure of both client Request messages and server Response messages. Understanding how these messages work is a big part of comprehending HTTP as a whole.

In this chapter, I describe the messages used by HTTP and the specific commands and responses issued by clients and servers. I begin with a look at the generic HTTP message format and the major components of every HTTP message. I then discuss the specific formats used for both Request and Response messages. I explain the different types of HTTP methods (commands) used in client requests and the HTTP status codes used in server replies.

NOTE Much of the functionality of HTTP is implemented using header fields that appear at the start of each HTTP Request and Response message. Headers are covered in detail in the next chapter.

HTTP Generic Message Format

As you learned in the previous chapter, all of the communication between devices using HTTP takes place via HTTP messages, of which there are only two types: *Request* and *Response* messages. Clients usually send requests and receive responses, while servers receive requests and send responses. Intermediate devices such as gateways or proxies may send and receive both types of messages.

All HTTP messages are created to fit a message structure that the standard calls the *generic message format*. Like most of the other TCP/IP messaging protocols, HTTP does not use a binary message format; rather, the messages are text-based. HTTP messages are based loosely on the electronic mail (email) RFC 822 and 2822 message standards, as well as the Multipurpose Internet Mail Extensions (MIME) standard (described in Chapter 76). I say “loosely” because HTTP messages are similar in construction to email messages but do not strictly follow all of the email or MIME format requirements. One difference is that not all of the RFC 822 and MIME headers are used; there are other differences as well, which we will soon examine.

The HTTP generic message format is as follows:

```
<start-line>
<message-headers>
<empty-line>
[<message-body>]
[<message-trailers>]
```

You can see that this is pretty much the same as the format used for email messages: headers, an empty line, and then a message body. All text lines are terminated with the standard carriage return-line feed (CRLF) control character sequence. The empty line contains just those two characters and nothing else. The headers are always sent as regular text. The body, however, may be either text or 8-bit binary information, depending on the nature of the data to be sent. (This is another way that HTTP does not adhere strictly to the RFC 822 standard; see the discussion of entities and media types in Chapter 83 for a full discussion.)

The generic message format has the following components:

Start Line The *start line* is a special text line that conveys the nature of the message. In a request, this line indicates the nature of the request, in the form of a *method*, as well as specifying a Uniform Resource Identifier (URI) to indicate the resource that is the object of the request. Responses use the start line to indicate status information in reply to a request. You can find more details on the use of the start line in the following sections in this chapter that detail HTTP Request messages and Response messages.

Message Headers Many dozens of message headers are defined in HTTP. These headers are organized into groups by function, as described in the following sections in this chapter. Almost all of these headers are optional; the one exception is the Host header, which must be present in each request in HTTP/1.1. Headers may be sent in any order, and they all follow the same header format used in email messages: <*header-name*>: <*header-value*>.

Message Body The message body is optional, because it is needed only for certain types of messages. The body may carry a set of information to be communicated between the client and server, such as a detailed error message in a response. More commonly, it carries a file or other resource, which is formally called an *entity* in the HTTP standard. Entities are most often found in the body of a Response message, since most client requests ask for a server to send a file or other resource. However, they can also be found in certain Request messages. HTTP supports many kinds of entities, as described in detail in Chapter 83.

Message Trailers As described in the previous chapter, HTTP/1.1 uses persistent connections by default, so messages are sent in a steady stream from client to server and server to client. This requires some means to mark where one message ends and the next begins, which is usually accomplished in one of two ways. The first is using a special header that indicates the length of the message, so the receiving device knows when the entire message has been received. The second is a method called *chunking*, where a message is broken into pieces for transmission, and the length of each piece is indicated in the message body. When chunking is done, a set of message *trailers* may follow the body of the message. Trailers are actually the same as headers, except for their position in the file, but they may only be used for entity headers. See Chapter 83 for more details on trailers and chunked data.

KEY CONCEPT All HTTP messages conform to a structure called the *generic message format*. This format is based on the RFC 822 and MIME electronic mail message standards, although HTTP does not follow those formats precisely. Each HTTP message begins with a start line, then contains a number of message headers, followed by an empty line and optionally a message body. The body of the message may contain a resource such as a file to be communicated between client and server, called an *entity*.

HTTP Request Message Format

The client initiates an HTTP session by opening a TCP connection to the HTTP server with which it wishes to communicate. It then sends *HTTP Request messages* to the server, each of which specifies a particular type of action that the user of the HTTP client would like the server to take. Requests can be generated either by specific user action (such as clicking a hyperlink in a web browser) or indirectly as a result of a prior action (such as a reference to an inline image in an HTML document leading to a request for that image).

HTTP Request messages use a format that is based on the generic message format described in the previous section, but specific to the needs of requests. The structure of this format is as follows (see Figure 81-1):

```
<request-line>
<general-headers>
<request-headers>
<entity-headers>
<empty-line>
[<message-body>]
[<message-trailers>]
```



Figure 81-1: HTTP Request message format This diagram shows the structural elements of an HTTP Request message and an example of the sorts of headers a Request message might contain. Like most HTTP requests, this one carries no entity, so there are no entity headers and the message body is empty. See Figure 81-2 for the HTTP Response message format.

Request Line

The generic *start line* that begins all HTTP messages is called a *request line* in Request messages. Its has three main purposes:

- To indicate the command or action that the client wants performed
- To specify a resource on which the action should be taken
- To indicate to the server what version of HTTP the client is using

The formal syntax for the request line is as follows:

```
<METHOD> <request-uri> <HTTP-VERSION>
```

Each of the request line components is discussed in the following sections.

Method

The *method* is simply the type of action that the client wants the server to take; it is always specified in uppercase letters. There are eight standard methods defined in HTTP/1.1, of which three are widely used: *GET*, *HEAD*, and *POST*. They are called *methods*, rather than *commands*, because the HTTP standard uses terminology from object-oriented programming. I explain this and also describe the methods themselves in the “HTTP Methods” section later in this chapter.

Request URI

The *request URI* is the URI of the resource to which the request applies. While URIs can theoretically refer to either Uniform Resource Locators (URLs) or Uniform Resource Names (URNs), currently, a URI is almost always an HTTP URL that follows the standard syntax rules of Web URLs, as described in Chapter 70.

Interestingly, the exact form of the URL used in the HTTP request line usually differs from that used in HTML documents or entered by users. This is because some of the information in a full URL is used to control HTTP itself. It is needed as part of the communication between the user and the HTTP client, but not in the request from the client to the server. The standard method of specifying a resource in a request is to include the path and filename in the request line (as well as any optional query information), while specifying the host in the special Host header that must be used in HTTP/1.1 Request messages.

For example, suppose the user enters a URL such as `http://www.myfavoritewebsite.com:8080/chatware/chatroom.php`. We obviously don’t need to send the `http:` to the server. The client would take the remaining information and split it so the URI was specified as `/chatware/chatroom.php` and the Host line would contain `www.myfavoritewebsite.com:8080`. Thus, the start of the request would look like this:

```
GET /chatware/chatroom.php HTTP/1.1
Host: www.myfavoritewebsite.com:8080
```

The exception to this rule is when a request is being made to a proxy server. In that case, the request is made using the full URL in its original form, so that it can be processed by the proxy just as the original client processed it. The request would look like this:

```
GET http://www.myfavoritewebsite.com:8080/chatware/chatroom.php HTTP/1.1
```

Finally, there is one special case where a single asterisk can be used instead of a real URL. This is for the *OPTIONS* method, which does not require the specification of a resource. (Nominally, the asterisk means the method refers to the server itself.)

HTTP Version

The *HTTP version* element tells the server which version the client is using, so the server knows how to interpret the request, and what to send and not to send the client in its response. For example, a server receiving a request from a client using HTTP/0.9 or HTTP/1.0 will assume that a transitory connection is being used

rather than a persistent one (as explained in the previous section), and the server will avoid using HTTP/1.1 headers in its reply. The version token is sent in uppercase letters, as HTTP/0.9, HTTP/1.0, or HTTP/1.1—just the way I've been doing throughout my discussion of the protocol.

Headers

After the request line come any of the headers that the client wants to include in the message. In these headers, details are provided to the server about the request. The headers all use the same structure, but are organized into the following categories based on the functions they serve and whether they are specific to one kind of message:

General Headers General headers refer mainly to the message itself, as opposed to its contents, and they are used to control its processing or provide the recipient with extra information. They are not particular to either Request or Response messages, so they can appear in either. Also, they are not specifically relevant to any entity the message may be carrying.

Request Headers These headers convey to the server more details about the nature of the client's request, and they give the client more control over how the request is handled. For example, special request headers can be used by the client to specify a conditional request—one that is filled only if certain criteria are met. Others can tell the server which formats or encodings the client is able to process in a Response message.

Entity Headers These are headers that describe the entity contained in the body of the request, if any.

KEY CONCEPT *HTTP Request messages* are the means by which HTTP clients ask servers to take a particular type of action, such as sending a file or processing user input. Each Request message begins with a *request line*, which contains three critical pieces of information: the *method* (type of action) the client is requesting, the *URI* of the resource on which the client wishes the action to be performed, and the version of HTTP that the client is using. After the request line comes a set of message headers related to the request, followed by a blank line, and then optionally, the message body of the request.

Request headers are obviously used only in Request messages, but both general headers and entity headers can appear in either a Request or a Response message. Since there are so many headers and most are not particular to one message type, I describe them in detail in the next chapter.

HTTP Response Message Format

Each Request message sent by an HTTP client to a server prompts the server to send back a *Response message*. Actually, in certain cases, the server may send two responses: a preliminary response, followed by the real one. Usually though, one

request yields one response, which indicates the results of the server's processing of the request, and a response often also carries an entity (file or resource) in the message body.

Like Request messages, Response messages use their own specific format that is based on the HTTP generic message format described earlier in this chapter. The formal syntax for the Response message header is as follows (see Figure 81-2):

```
<status-line>
<general-headers>
<response-headers>
<entity-headers>
<empty-line>
[<message-body>]
[<message-trailers>]
```

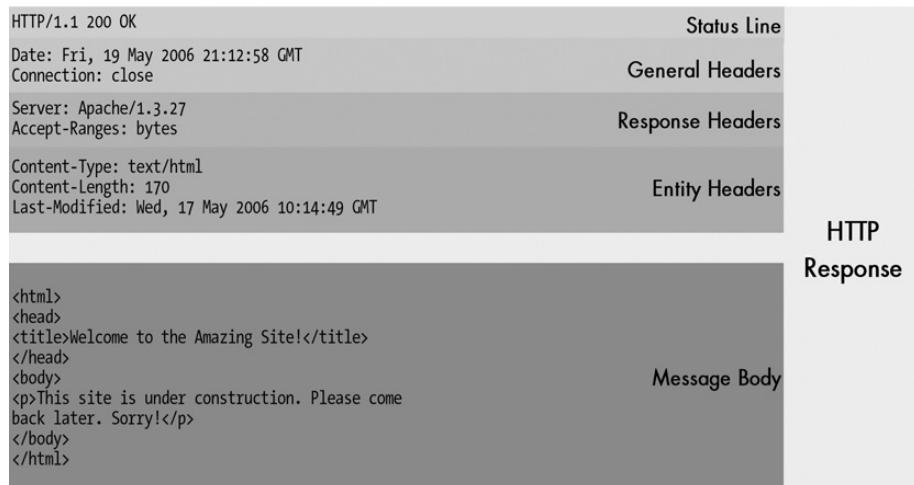


Figure 81-2: HTTP Response message format This figure illustrates the construction of an HTTP Response message and includes an example of both message headers and body. The status code 200 indicates that this is a successful response to a request; it contains a brief text HTML entity in the message body. See Figure 81-1 for the HTTP Request message format.

Status Line

The *status line* (note that this is not called the *response line*) is the start line used for Response messages. It has two functions: to tell the client what version of the protocol the server is using and to communicate a summary of the results of processing the client's request. The formal syntax for the status line is as follows:

```
<HTTP-VERSION> <status-code> <reason-phrase>
```

Each of the status line components is discussed in the following sections.

HTTP Version

The *HTTP-VERSION* label in the status line serves the same purpose as it does in the request line of a Request message (described in the previous section). Here, it tells the client the version number that the server is using for its response. It uses the same format as in the request line, with the version in uppercase as HTTP/0.9, HTTP/1.0, or HTTP/1.1. The server is required to return an HTTP version number that is no greater than the number the client sent in its request.

Status Code and Reason Phrase

The *status code* and *reason phrase* provide information about the results of processing the client's request in two different forms. The status code is a three-digit number that indicates the formal result that the server is communicating to the client. It is intended for the client HTTP implementation to process so the software can take appropriate action. The reason phrase is an additional, descriptive text string, which can be displayed to the human users of the HTTP client so they can see how the server responded. I describe status codes and reason phrases later in this chapter, and also list all of the standard codes.

Headers

The Response message will always include a number of headers that provide extra information about it. Response message headers fall into three categories:

General Headers General headers that refer to the message itself and are not specific to Response messages or the entity in the message body. These are the same as the generic headers that can appear in Request messages (though certain headers appear more often in responses, and others are more common in requests).

Response Headers These headers provide additional data that expands on the summary result information in the status line. The server may also return extra result information in the body of the message, especially when an error occurs.

Entity Headers These are headers that describe the entity contained in the body of the response, if any. These are the same entity headers that can appear in a Request message, but they are seen more often in response messages.

Most Response messages contain an entity in the message body. In the case of a successful request to retrieve a resource, this is the resource itself. Responses indicating unsuccessful requests usually contain detailed error information, often in the form of an HTML-formatted error message.

NOTE *Entity headers may appear in a Response message to describe the resource that is the subject of the request, even if the entity itself is not sent in the message. This occurs when the HEAD method is used to request only the headers associated with an entity.*

Response headers are used only in Response messages, while the others are general with respect to message type. See Chapter 82 for more details about HTTP headers.

KEY CONCEPT Each HTTP Request message sent by a client leads to a server returning one or more *HTTP Response messages*. Each Response message starts with a *status line* that contains the server's HTTP version number, and a numeric *status code* and text *reason phrase* that indicate the result of processing the client's request. The message then contains headers related to the response, followed by a blank line, and then the optional message body. Since most HTTP Request messages ask for a server to return a file or other resource, many HTTP Response messages carry an entity in the message body.

HTTP Methods

An HTTP Request message sent by a client to a server obviously requests that the server do something. All client/server protocols provide a way for the client to prompt the server to take action, generally by having the client give the server a series of commands. HTTP, in contrast, has *methods*, rather than commands. Each client Request message begins with the specification of the method that is the subject of the request.

What is the difference between a method and a command? In practical terms, nothing; they are the same. So why does HTTP use the term *method* instead of *command*? That's a good question. The answer can be found in the abstract of the standard defining HTTP/1.0, RFC 1945. It states, in part, that HTTP is "a generic, stateless, object-oriented protocol which can be used for many tasks. . ." In highly simplified terms, object-oriented programming is a technique in which software modules are described not as sets of procedures, but as *objects* that possess attributes. These modules send messages to each other to communicate and to cause actions to be performed, where the action taken depends on the nature of the object. In object-oriented programming, the procedures each object can perform are called *methods*.

HTTP is considered to be object-oriented because, in many cases, the action taken by a server depends on the object that is the subject of the request. For example, if you ask a server to retrieve a text document, it will send that document; but if you ask for a directory, the server may instead return a default document for that directory. In contrast, a request that specifies the name of a program will result in the program being executed and its output returned (as opposed to the program's source code being returned).

Common Methods

Each method allows the client to specify a particular type of action to be taken by the server. Method names are always in uppercase letters. There are three methods that are commonly used in HTTP: GET, HEAD, and POST.

GET

The GET method requests that server retrieve the resource specified by the URL on the HTTP request line and send it in a response back to the client. This is the most basic type of request and the one that accounts for the majority of HTTP traffic. When you enter a conventional URL or click a link to a document or other file, you are usually prompting your web browser to send a GET request.

The handling of a GET request depends on a number of factors. If the URL is correct and the server can find the resource, it will send back the appropriate response to the client. The exact resource returned depends on the nature of the object requested. If the request cannot be processed properly, an error message may result. Caching (discussed in Chapter 84) also comes into play, as a proxy server or even the client itself might satisfy the request before it gets to the server.

It's important to remember that the meaning of a GET request may change if certain headers, such as If-Modified-Since or If-Match, are used. These tell the server to send the resource only if certain conditions are met. A request of this sort is sometimes called a *conditional GET*. Similarly, the client may use the Range header to request that the server send it only part of a resource; this is usually used for large files. When this header is included, the request may be called a *partial GET*.

HEAD

The HEAD method is identical to the GET method, but it tells the server not to send the actual body of the message. Thus, the response will contain all of the headers that would have accompanied a reply to the equivalent GET message, including entity headers describing the entity that the server would have sent had the method been GET. The client often uses this method to check the existence, status, or size of a file before deciding whether it wants the server to send the whole file.

HEAD requests are processed in the same way as GET requests, except that only the headers are returned, not the actual resource.

POST

The POST method allows the client to send an entity containing arbitrary data to the server for processing. It is commonly used to enable a client to submit information such as an interactive HTML form to a program on the server, which then takes action based on that input and sends a response. This capability is now used for all sorts of online programs. The URL in the request specifies the name of the program on the server that is to accept the data. Contrast this with the PUT method described in the next section.

Other Methods

The other methods defined by the HTTP standard are not used as often, but I will describe them briefly, as you may still encounter them. Other HTTP methods include the following:

OPTIONS This method allows the client to request that the server send it information about available communication options. A URI of a resource may be specified to request information relevant to accessing that resource, or an asterisk (*) may be used to indicate that the query is about the server itself. The response includes headers that give the client more details about how the server may be accessed.

PUT This method requests that the server store the entity enclosed in the body of the request at the URL specified in the request line. In a PUT, the URI identifies the entity in the request; thus a PUT allows a file to be copied to a server, in the exact

complement to how a GET requests that a file be copied to the client. In contrast, with a POST, the URI identifies a program intended to *process* the entity in the request, so it's used for interactive programs. Now, would you like people to be able to store files on your server in the same way that they request them? Neither would I. This is one primary reason why PUT is not often used. It has valid uses, such as uploading content to a website, and it must be used with authentication in this case. However, storing files on a site is more often accomplished using other means, like the File Transfer Protocol (FTP).

DELETE This method requests that the specified resource be deleted. This has the same issues as PUT and is not often used for similar reasons.

TRACE This method allows a client to receive back a copy of the request that it sent to the server, for diagnostic purposes.

In addition to these, the standard reserves the method name CONNECT for future use. An earlier version of HTTP/1.1, RFC 2068, defined the methods PATCH, LINK, and UNLINK. These were removed in the final version, but you may still see references to them.

KEY CONCEPT Each HTTP client request specifies a particular type of action that the server should perform; in HTTP, these are called *methods*, rather than commands. The three most common HTTP methods are *GET*, which prompts a server to return a resource; *HEAD*, which returns just the headers associated with a resource; and *POST*, which allows a client to submit data to a server for processing.

Safe and Idempotent Methods

As you've seen, methods vary greatly in the type of behavior they cause the server to take. The HTTP standard defines two characteristics that can be used to differentiate methods based on the impact they have on a server:

Safe Methods These are methods that an administrator of a server can feel reasonably comfortable permitting a client to send because they are very unlikely to have any negative side effects. The methods usually put into this category are GET, HEAD, OPTIONS, and TRACE. The methods that cause data to be accepted by the server for processing, or lead to changes on the server, are deemed unsafe: POST, PUT, and DELETE. (The fact that they are considered unsafe doesn't mean a server never allows them—just that they require more care and detail in handling than the others.)

Idempotent Methods A method is said to be *idempotent* if repeating the same method request numerous times causes the exact same results, as if the method were issued only once. For example, if you load a web page in your browser, and then type the same URL in again, you get the same result, at least most of the time. In general, all of the methods in HTTP have this property inherently except one: POST.

The POST method is not idempotent because each instance of a POST request causes the receiving server to process the data in the Request message's body. Submitting a POST request two or more times can often lead to undesirable results. The classic example is clicking the Submit button on a form more than once, which can lead to annoyances such as a duplicate message on an Internet forum or a double order at an online store.

There are also situations where a method that is normally idempotent may not be. A GET request for a simple document is idempotent, but a GET for a script can change files on the server and therefore is not idempotent. Similarly, a sequence of idempotent methods may not be idempotent. For example, consider a situation where a PUT request is followed by a GET for the same resource. This sequence is not idempotent because the second request depends on the results of the first.

The significance of nonidempotence is that clients must handle such requests or sequences specially. The client must keep track of them, making sure that they are filled in order and only once. The HTTP standard also specifies that nonidempotent methods should not be pipelined, to avoid problems if an HTTP session is unexpectedly terminated. For example, if two POST requests were pipelined and the server got hung up handling them, the client would need to reissue them but might not know how many of the original requests had been successfully processed.

HTTP Status Codes and Reason Phrases

Every request sent by an HTTP client causes one or more responses to be returned by the server that receives it. As you saw earlier in the discussion of the Response message format, the first line of the response is a status line that contains a summary of the results of processing the request. The purpose of this line is to communicate quickly whether or not the request was successful and why.

HTTP status lines contain both a numeric status code and a text reason phrase. The reason for having both a number and a text string is that computers can more easily understand the results of a request by looking at a number and then can quickly respond accordingly. Humans, on the other hand, find text descriptions easier to comprehend. The idea of using both forms was taken directly from earlier application layer protocols such as FTP, the Simple Mail Transfer Protocol (SMTP), and the Network News Transfer Protocol (NNTP). The explanation of FTP reply codes in Chapter 72 discusses more completely the reasons why numeric reply codes are used in addition to descriptive text.

Status Code Format

HTTP status codes are three digits in length and follow a particular format, where the first digit has particular significance. Unlike the reply codes used by FTP and other protocols, the second digit does not stand for a functional grouping; the second and third digits together just make 100 different options for each of the categories indicated by the first digit. Thus, the general form of an HTTP status code is xyy , where the first digit, x , is specified as shown in Table 81-1.

Table 81-1: HTTP Status Code Format: First-Digit Interpretation

Status Code Format	Meaning	Description
1yy	Informational message	Provides general information; does not indicate success or failure of a request.
2yy	Success	The method was received, understood, and accepted by the server.
3yy	Redirection	The request did not fail outright, but additional action is needed before it can be successfully completed.
4yy	Client error	The request was invalid, contained bad syntax, or could not be completed for some other reason that the server believes was the client's fault.
5yy	Server error	The request was valid, but the server was unable to complete it due to a problem of its own.

In each of these five groups, the code where yy is 00 is defined as a generic status code for that group, while other two-digit combinations are more specific responses. For example, 404 is the well-known specific error message that means the requested resource was not found by the server, and 400 is the less specific Bad Request error. This system was set up to allow the definition of new status codes that certain clients might not comprehend. If a client receives a strange code, it just treats it as the equivalent of the generic response in the appropriate category. So, if a server response starts with the code 491, and the client has no idea what this is, it treats it as a 400 Bad Request reply.

Reason Phrases

The reason phrase is a text string that provides a more meaningful description of the error for people who are bad at remembering what cryptic codes stand for (which would be most of us!). The HTTP standard includes sample reason phrases for each status code, but server administrators can customize these phrases if desired. When a server returns a more detailed HTML error message in the body of its Response message, the reason phrase is often used for the title tag in that message body.

KEY CONCEPT Each HTTP Response message includes both a numeric *status code* and a text *reason phrase*, both of which indicate the disposition of the corresponding client request. The numeric code allows software programs to easily interpret the results of a request, while the text phrase provides more useful information to human users. HTTP status codes are three digits in length, with the first digit indicating the general class of the reply.

Table 81-2 lists in numerical order the status codes defined by the HTTP/1.1 standard, along with the standard reason phrase and a brief description of each.

Table 81-2: HTTP Status Codes and Reason Phrases

Status Code	Reason Phrase	Description
100	Continue	The client should continue sending its request. This is a special status code; see the next section in this chapter for details.
101	Switching Protocols	The client has used the Upgrade header to request the use of an alternative protocol and the server has agreed.
200	OK	This is the generic successful Request message response, which is the code sent most often when a request is filled normally.
201	Created	The request was successful and resulted in a resource being created. This is a typical response to a PUT method.
202	Accepted	The request was accepted by the server, but it has not yet been processed. This is an intentionally noncommittal response that does not tell the client whether or not the request will be carried out. The client determines the eventual disposition of the request in some unspecified way. It is used only in special circumstances.
203	Non-Authoritative Information	The request was successful, but some of the information returned by the server came from a third party, rather than from the original server associated with the resource.
204	No Content	The request was successful, but the server has determined that it does not need to return to the client an entity body.
205	Reset Content	The request was successful; the server is telling the client that it should reset the document from which the request was generated so that a duplicate request is not sent. This code is intended for use with forms.
206	Partial Content	The server has successfully fulfilled a partial GET request. See the section on methods earlier in this chapter for more details on this, as well as the description of the Range header in the next chapter.
300	Multiple Choices	The resource is represented in more than one way on the server. The server is returning information describing these representations, so the client can pick the most appropriate one, a process called agent-driven negotiation (discussed in Chapter 83).
301	Moved Permanently	The resource requested has been moved to a new URL permanently. Any future requests for this resource should use the new URL. This is the proper method of handling situations where a file on a server is renamed or moved to a new directory. Most people don't bother setting this up, which is why URLs break so often, resulting in 404 errors.
302	Found	The resource requested is temporarily using a different URL. The client should continue to use the original URL. See code 307.
303	See Other	The response for the request can be found at a different URL, which the server specifies. The client must do a fresh GET on that URL to see the results of the prior request.
304	Not Modified	The client sent a conditional GET request, but the resource has not been modified since the specified date/time, so the server has not sent it.
305	Use Proxy	To access the requested resource, the client must use a proxy, whose URL is given by the server in its response.
306	(unused)	Defined in an earlier version of HTTP and no longer used.
307	Temporary Redirect	The resource is temporarily located at a different URL than the one the client specified. Note that 302 and 307 are basically the same status code. Code 307 was created to clear up some confusion related to 302 that occurred in earlier versions of HTTP.
400	Bad Request	This is a generic response when the request cannot be understood or carried out due to a problem on the client's end.

(continued)

Table 81-2: HTTP Status Codes and Reason Phrases (continued)

Status Code	Reason Phrase	Description
401	Unauthorized	The client is not authorized to access the resource. This is often returned if an attempt is made to access a resource protected by a password or some other means without the appropriate credentials.
402	Payment Required	This is reserved for future use. Its mere presence in the HTTP standard has caused a lot of people to scratch their chins and go "hmm. . . ."
403	Forbidden	The request has been disallowed by the server. This is a generic "no way" response that is not related to authorization. For example, if the maintainer of website blocks access to it from a particular client, any requests from that client will result in a 403 reply.
404	Not Found	The most common HTTP error message, this is returned when the server cannot locate the requested resource. It usually occurs due to the server having moved (or removed) the resource or the client giving an invalid URL (usually due to misspellings).
405	Method Not Allowed	The requested method is not allowed for the specified resource. The response includes an Allow header that indicates which methods the server will permit.
406	Not Acceptable	The client sent a request that specifies limitations that the server cannot meet for the specified resource. This error may occur if an overly restrictive list of conditions is placed into a request such that the server cannot return any part of the resource.
407	Proxy Authentication Required	This is similar to 401, but the client must first authenticate itself with the proxy.
408	Request Timeout	The server was expecting the client to send a request within a particular time frame and the client didn't send it.
409	Conflict	The request could not be filled because of a conflict of some sort related to the resource. This most often occurs in response to a PUT method, such as if one user tries to PUT a resource that another user has open for editing.
410	Gone	The resource is no longer available at the server, which does not know its new URL. This is a more specific version of the 404 code that is used only if the server knows that the resource was intentionally removed. It is seen rarely (if ever).
411	Length Required	The request requires a Content-Length header field and one was not included.
412	Precondition Failed	This indicates that the client specified a precondition in its request, such as the use of an If-Match header, which evaluated to a false value. This indicates that the condition was not satisfied, so the request is not being filled. This is used by clients in special cases to ensure that they do not accidentally receive the wrong resource.
413	Request Entity Too Large	The server has refused to fulfill the request because the entity that the client is requesting is too large.
414	Request-URI Too Long	The server has refused to fulfill the request because the URL specified is longer than the server can process. This rarely occurs with properly formed URLs, but may be seen if clients try to send gibberish to the server.
415	Unsupported Media Type	The request cannot be processed because it contains an entity using a media type the server does not support.
416	Requested Range Not Satisfiable	The client included a Range header specifying a range of values that is not valid for the resource. An example might be requesting bytes 3000 through 4000 of a 2400-byte file.
417	Expectation Failed	The request included an Expect header that could not be satisfied by the server.
500	Internal Server Error	This is a generic error message indicating that the request could not be fulfilled due to a server problem.

(continued)

Table 81-2: HTTP Status Codes and Reason Phrases (continued)

Status Code	Reason Phrase	Description
501	Not Implemented	The server does not know how to carry out the request, so it cannot satisfy it.
502	Bad Gateway	The server, while acting as a gateway or proxy, received an invalid response from another server it tried to access on the client's behalf.
503	Service Unavailable	The server is temporarily unable to fulfill the request for internal reasons. This is often returned when a server is overloaded or down for maintenance.
504	Gateway Timeout	The server, while acting as a gateway or proxy, timed out while waiting for a response from another server it tried to access on the client's behalf.
505	HTTP Version Not Supported	The request used a version of HTTP that the server does not understand.

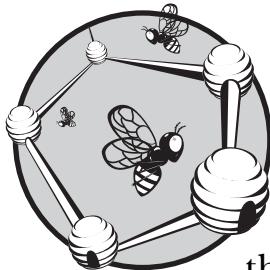
The 100 (Continue) Preliminary Reply

Now, let's go back to the top of the list in Table 81-2 and look at the special status code 100. Normally, a client sends a complete request to the server and waits for a response to it (while optionally pipelining additional requests, as described in the previous chapter). In certain circumstances, however, the client might wish to check in advance if the server is willing to accept the request before it bothers sending the whole message. This is not a common occurrence, because most requests are quite small, so checking first isn't worth the bother. However, in cases where a user wants to submit a very large amount of data to an online program or use PUT to store a large file, for example, checking with the server first can be a useful optimization.

In this situation, the client sends a request containing the special header Expect: 100-Continue. Assuming that the server supports the feature, it will process the request's headers and immediately send back the 100 Continue preliminary reply. This tells the client to continue sending the rest of the request. The server then processes it and responds normally. If the server doesn't send the 100 response after a certain amount of time, the client will typically just send the rest of the request anyway. Note that in some cases, servers send these preliminary replies even when they are not supposed to, so clients must be prepared to deal with them (they are simply discarded, since they contain no information).

82

HTTP MESSAGE HEADERS



As you have seen in the preceding two chapters, Hypertext Transfer Protocol (HTTP) communication takes place through the relatively simple exchange of request and response messages. There are only a small number of methods (commands) supported by the protocol, which might give you the impression that the protocol is quite limited. Looks can be deceiving, however. Much of the functionality in HTTP is actually implemented in the form of *message headers*, which convey important details between clients and servers.

Some headers can appear in only HTTP requests, some in only HTTP responses, and some in either type of message. Understanding these headers is important to learning how HTTP works. There are literally dozens of them, and many apply to both Request and Response messages.

In this chapter, I provide a description of each of the many headers used in HTTP Request and Response messages. The chapter is organized by the four basic types of HTTP headers: general headers, request headers, response headers, and entity headers.

BACKGROUND INFORMATION I assume here that you have already read the preceding chapter describing HTTP message formats.

NOTE For the purpose of determining how web caches treat HTTP messages, HTTP headers are categorized as either end-to-end or hop-by-hop headers. The former are meaningful only to the ultimate recipient of a message, while the latter are relevant to each device in the chain of devices (such as proxies) connecting a client and server. To avoid unnecessary complication, I have not categorized the headers using these categories; see the full discussion of caching in Chapter 84 for more information. In the descriptions of the individual headers, I indicate which headers are hop-by-hop; all others are end-to-end.

HTTP General Headers

HTTP *general headers* are so named because, unlike headers in the other three categories, they are not specific to any particular kind of message or message component (request, response, or message entity). General headers are used primarily to communicate information about the message itself, as opposed to what content it carries. They provide general information and control how a message is processed and handled.

Despite not being specific to either requests or replies, some general headers are used either mostly or entirely in one or the other type of message. There are also some general headers that can appear in either a Request or a Reply message, but have a somewhat different meaning in each.

Here, I describe the Cache-Control and Warning headers and then the other more straightforward headers.

Cache-Control Headers

A Cache-Control header specifies directives that manage how caching is performed either for an HTTP request or response. These directives affect the handling of a request or response by all devices in the request/response chain from the HTTP client, through any present intermediaries, to the HTTP server (or the other way, from the server, through intermediaries, to the client). They override any default caching behavior performed by a device. See the discussion of caching in Chapter 84 for a full exposition of the subject.

There are a dozen individual directives that can appear in this header, the full details of which can be found in RFC 2616. Even though this is a general header, some directives can appear only in a request or a response. Some also include an additional parameter, such as a number of seconds, that control their interpretation. Table 82-1 provides a brief summary of the different Cache-Control options and how they are used.

Note that only one directive may appear in a Cache-Control header, but more than one such header can appear in a message.

Table 82-1: HTTP Cache-Control Directives

Cache-Control Directive	HTTP Message Type	Description
no-cache	Request or Response	When present, forces a caching device to forward any subsequent requests for the same content to the server for revalidation; that is, the cache must check with the server to ensure that the cached data is still valid. Also see the Pragma header description, for an alternative way of accomplishing the same thing.
public	Response	Indicates that the response may be cached by any cache, including a shared one (a cache used by many clients). See Chapter 84 for more details on shared caches.
private	Response	Specifies that the response is intended for only a particular user and should not be placed into a shared cache.
no-store	Request or Response	Specifies that the entire request or response should not be stored in a cache. This is used sometimes to prevent the storing of sensitive documents in caches where unauthorized people might be able to access them. However, as the HTTP standard points out, this is really a very rudimentary security measure and should not be trusted a great deal (since a malicious cache operator could simply ignore the directive).
max-age	Request or Response	In a request, indicates that the client is willing to accept a response whose age is no greater than the value specified. In a response, indicates the maximum age of the response before it is considered stale. This is an alternative to the use of the Expires header and takes precedence over it.
s-maxage	Response	If present, specifies the maximum age for shared caches receiving the response. Private caches (ones that serve only a single client) use the max-age value (see the preceding description).
min-fresh	Request	Specifies that the client wants a response that is not only not stale at the time the request is received, but that will remain fresh for the specified number of seconds.
max-stale	Request	If sent without a parameter, indicates that the client is willing to accept a stale reply (one that has expired). If a numeric parameter is included, it indicates how stale, in seconds, the response may be.
only-if-cached	Request	Used only in special circumstances, forces the reply to come from a cache only; the content may not come from the actual specified HTTP server.
must-revalidate	Response	Instructs a cache to revalidate its cache entry for the given response with the original server after it becomes stale. This is used to prevent problems with certain types of transactions that can occur if stale cache entries are sent to a client (perhaps as a result of the client using the max-stale directive).
proxy-revalidate	Response	Similar to must-revalidate, but applies only to proxies that service many users. Private caches, such as those on individual client computers, are not affected.
no-transform	Request or Response	Some caches will, by default, change the form in which certain cached entries are stored, to save space or improve performance. In cases where this might cause problems, the client or server can use this directive to request that this transformation not be performed.

Warning

A Warning header is used when it's needed to provide additional information about the status of a message. Many of the defined warning header types are related to caching. More than one Warning header may appear in a message, and each

typically includes a three-digit numeric code as well as a plain text message, following the same basic format used in HTTP response status codes (described in Chapter 81). Table 82-2 briefly lists the warnings defined in RFC 2616.

Table 82-2: HTTP Warning Header Codes

Warning Code	Warning Text	Description
110	Response is stale	Must be included when a response provided by a cache is stale (that is, has passed the expiration time set for it).
111	Revalidation failed	A cache attempted to revalidate a cached entry but was unsuccessful, so it returned its (stale) cached entry.
112	Disconnected operation	The cache is disconnected from the rest of the network.
113	Heuristic expiration	Included if the cache chose a freshness lifetime of more than 24 hours, and the age of the response is also greater than 24 hours.
199	Miscellaneous warning	Catchall code for other, nonspecific warnings.
214	Transformation applied	Warns the recipient that an intermediate cache or proxy applied a transformation of some type to change the content coding or media type of the message or message body.
299	Miscellaneous persistent warning	Similar to code 199, but indicates a persistent warning.

Other HTTP General Headers

The following are the other types of HTTP general headers:

Connection Contains instructions that pertain only to this particular connection, and must not be retained by proxies and used for further connections. The most common use of this header is with the `close` parameter, as follows: `Connection: close`. This overrides the default persistent connection behavior of HTTP/1.1 (described in Chapter 80), forcing the connection to terminate after the server's response. `Connection` is a hop-by-hop header.

Date Indicates the date and time when the message originated. This is the same as the `Date` header in the RFC 822 email format (described in Chapter 76). A typical example is `Date: Wed, 17 May 2006 16:43:50 GMT`.

Pragma Used to enable implementation-specific directives to be applied to all devices in the request/response chain. One common use of this header is to suppress caching by including `Pragma: no-cache` in a message. This has the same meaning as a `Cache-Control: no-cache` header, and is included in HTTP/1.1 for backward-compatibility with HTTP/1.0 (which supports `Pragma` but not `Cache-Control`).

Trailer When chunked transfers are used (as described in Chapter 83), certain headers may be placed as trailers, after the data being sent. In this case, the `Trailer` header is included before the data, and it lists the names of the headers that are actually trailers in that message. This warns the recipient to look for them after the data. `Trailer` is a hop-by-hop header.

Transfer-Encoding Indicates what encoding has been used for the body of the message, to ensure that it is able to be transferred properly between devices. This header is most often used with the chunked transfer method. Note that this header describes encoding applied to an entire message, and is thus not the same as the Content-Encoding entity header, which specifically describes the entity carried in a message. See Chapter 83 for a full discussion. This header applies only to a single transfer, so it is a hop-by-hop header.

Upgrade Allows a client device to specify which additional protocols it supports. If the server also supports one of the protocols the client listed, the server may agree to upgrade the connection to the alternative protocol. It indicates the protocol to which it is upgrading by including an Upgrade header in a 101 (Switching Protocols) response to the client. This is a hop-by-hop header.

Via Included by intermediary devices to indicate to the recipient which gateways, proxies, and/or tunnels were used in conveying a request or response. This header allows easy tracing of the path a message took over a potentially complex chain of devices between a client and server.

KEY CONCEPT HTTP general headers can appear in either an HTTP Request or HTTP Response message. They are used to communicate information about the message itself, as opposed to its contents. General headers are used for functions such as specifying the date and time of a message, controlling how the message is cached, and indicating its transfer encoding method.

HTTP Request Headers

HTTP *request headers*, as you might imagine, are used only in HTTP Request messages, where they serve a number of functions. First, they allow the client to provide information about itself to the server. Second, they give additional details about the nature of the request that the client is making. Third, they allow the client to have greater control over how its request is processed and how (or even if) a response is returned by the server or intermediary.

This is the largest of the four categories of HTTP headers, containing more than a dozen different types, as follows:

Accept Allows the client to tell the server which Internet media types it is willing to accept in a response. The header may list several different Multipurpose Internet Mail Extensions (MIME) media types and subtypes that the client knows how to deal with. Each may be prepended with a quality value (*q* parameter) to indicate the client's preference. If this header is not specified, the default is for the server to assume any media type may be sent to the client. See the discussion of entity media types and content negotiation in Chapter 83 for more information about how this header is used.

Accept-Charset Similar to Accept, but specifies which character sets (Charsets) the client is willing to accept in a response, rather than which media types. Again, the listed charsets may use a q value, and again, the default if the header is omitted is for the client to accept any charset.

Accept-Encoding Similar to Accept and Accept-Charset, but specifies which content encodings the client is willing to accept. This is often used to control whether the server may send content in compressed form. (As you'll learn in Chapter 83, content codings are not the same as transfer encodings.)

Accept-Language Similar to the preceding Accept-type headers, but provides a list of *language tags* that indicates which languages the client supports or expects the server to use in its response.

Authorization Used by the client to present authentication information (called *credentials*) to the server to allow the client to be authenticated. This is required only when the server requests authentication, often by sending a 401 (Unauthorized) response to the client's initial request. This response will contain a WWW-Authenticate header providing the client with details on how to authenticate with the server. See the discussion of security and privacy in Chapter 84 for more information.

Expect Indicates certain types of actions that the client is expecting the server to perform. Usually, the server will accept the indicated parameters; if not, it will send back a 417 (Expectation Failed) response. The most common use of this field is to control when the server sends a 100 (Continue) response. The client indicates that it wants the server to send this preliminary reply by including the Expect: 100-Continue header in its request. (See the discussion of status codes at the end of Chapter 81 for details.)

From Contains the email address of the human user making the request. This is optional, and since it is easily spoofed, should be used only for informational purposes, and not for any type of access rights determination or authentication.

Host Specifies the Internet host as a Domain Name System (DNS) domain name and may also contain a port number specification as well (typically, only if a port other than the HTTP default of 80 is to be used). This header is used to allow multiple domains to be served by the same web server on a particular Internet Protocol (IP) host. It has the distinction of being the only mandatory header—it must be present in all HTTP/1.1 requests.

If-Match Makes a method conditional by specifying the *entity tag* (or tags) corresponding to the specific entity that the client wishes to access. This is usually used in a GET method, and the server responds with the entity only if it matches the one specified in this header. Otherwise, the server sends a 412 (Precondition Failed) reply.

If-Modified-Since Makes a method conditional by telling the server to return the requested entity only if it has been modified since the time specified in this header. Otherwise, the server sends a 304 (Not Modified) response. This is used to check if a resource has changed since it was last accessed, to avoid unnecessary transfers.

If-None-Match The opposite of If-Match; it creates a conditional request that is only filled if the specified tag(s) do not match the requested entity.

If-Range Used in combination with the Range header to effectively allow a client to both check for whether an entity has changed and request that a portion of it be sent in a single request. (The alternative is to first issue a conditional request, and if it fails, issue a second request.) When present, If-Range tells the server to send to the client the part of the entity indicated in the Range header if the entity has not changed. If the entity has changed, the server sends the entire entity in response.

If-Unmodified-Since The logical opposite of the If-Modified-Since header; the request is filled only if the resource has *not* been modified since the specified time. Otherwise, the server sends a 412 reply.

Max-Forwards Specifies a limit on the number of times a request can be forwarded to the next device in the request chain. This header is used with the TRACE or OPTIONS methods only, to permit diagnosis of forwarding failures or looping. When present in one of these methods, each time a device forwards the request, the number in this header is decremented. If a device receives a request with a Max-Forwards value of 0, it must not forward it, but rather it should respond back to the client. (In a way, this is somewhat analogous to how the Time to Live field is used in the IP datagram format, as described in Chapter 21.)

Proxy-Authorization Like the Authorization header, but used to present credentials to a proxy server for authentication, rather than to the end server. It is created using information sent by a proxy in a response containing a Proxy-Authenticate header. This is a hop-by-hop header, sent only to the first proxy that receives the request. If authentication is required with more than one proxy, multiple Proxy-Authorization headers may be put in a message, with each proxy consuming one of the headers.

Range Allows the client to request that the server send it only a portion of an entity, by specifying a range of bytes in the entity to be retrieved. If the requested range is valid, the server sends only the indicated part of the file, using a 206 (Partial Content) status code; if the range requested cannot be filled, the reply is 416 (Requested Range Not Satisfiable).

Referer Tells the server the Uniform Resource Locator (URL) of the resource from which the URL of the current request was obtained. Typically, when a user clicks a link on one web page to load another, the address of the original web page is put into the Referer line when the request for the clicked link is sent. This allows tracking and logging of how the server is accessed. If a human user manually enters a Uniform Resource Identifier (URI) into a web browser, this header is not included in the request. Since this header provides information related to how web pages are used, it has certain privacy implications.

NOTE *The proper spelling of this word is referrer. It was misspelled years ago in an earlier version of the HTTP standard, and before this was noticed and corrected, this spelling became incorporated into so much software that the Internet Engineering Task Force (IETF) chose not to correct the spelling in HTTP/1.1.*

TE Provides information to the server about how the client wishes to deal with transfer encodings for entities sent by the server. If extensions to the standard HTTP transfer encodings are defined, the client can indicate its willingness to accept them in this header. The client can also use the header **TE: trailers** to indicate its ability to handle having headers sent as trailers following data when chunking of data is done. This is a hop-by-hop header and applies only to the immediate connection.

User-Agent Provides information about the client software. This is normally the name and version number of the web browser or other program sending the request. It is used for server access statistic logging and also may be used to tailor how the server responds to the needs of different clients. Note that proxies do not modify this field when forwarding a request; rather, they use the **Via** header.

KEY CONCEPT *HTTP request headers* are used only in HTTP Request messages. They allow a client to provide information about itself to a server, provide more details about a request, and allow control over how the request is carried out.

HTTP Response Headers

The counterpart to request headers, *response headers*, appear only in HTTP responses sent by servers or intermediaries. They provide additional data that expands on the summary information that is present in the status line at the beginning of each server reply. Many of the response headers are sent only in response to the receipt of specific types of requests or even to particular headers within certain requests.

There are nine response headers defined for HTTP/1.1:

Accept-Ranges Tells the client whether the server accepts partial content requests using the Range request header, and if so, what type. For example, include **Accept-Range: bytes** indicates the server accepts byte ranges, and **Accept-Range: none** indicates range requests are not supported. Note that this is header is different from the other Accept- headers, which are used in HTTP requests to perform content negotiation.

Age Tells the client the approximate age of the resource, as calculated by the device sending the response.

ETag Specifies the entity tag for the entity included in the response. This value can be used by the client in future requests to uniquely identify an entity, using the **If-Match** (or similar) request header.

Location Indicates a new URL that the server is instructing the client to use in place of the one the client initially requested. This header is normally used when the server redirects a client request to a new location, using a 301, 302, or 307 reply. It is also used to indicate the location of a created resource in a 201 (Created) response to a PUT request. Note that this is not the same as the Content-Location entity header, which is used to indicate the location of the originally requested resource.

Proxy-Authenticate The proxy version of the WWW-Authenticate header (described next). It is included in a 407 (Proxy Authentication Required) response, to indicate how the proxy is requiring the client to perform authentication. The header specifies an authentication method, as well as any other parameters needed for authentication. The client will use this to generate a new request containing a Proxy-Authorization header. This is a hop-by-hop header.

Retry-After Sometimes included in unsuccessful requests—such as those resulting in a 503 (Service Unavailable) response—to tell the client when it should try its request again. It may also be used with a redirection response such as 301, 302, or 307 to indicate how long the client should wait before sending a request for the redirected URL. The Retry-After header may specify either a time interval to wait (in seconds) or a full date/time when the server suggests the client try again.

Server The server’s version of the User-Agent request header. It identifies the type and version of the server software generating the response. Note that proxies do not modify this field when forwarding a response; they put their identification information into a Via header instead.

Vary Specifies which request header fields fully determine whether a cache is allowed to use this response to reply to subsequent requests for the same resource without revalidation. A caching device inspects the Vary header to ascertain which other headers it needs to examine when the client makes its next request for the resource in this reply, to determine whether it can respond with a cached entry. (See Chapter 84 for more information about caching, which should make the use of this header easier to understand.)

WWW-Authenticate Included in a 401 (Unauthorized) response to indicate how the server wants the client to authenticate. The header specifies an authentication method as well as any other parameters needed for authentication. The client will use this to generate a new request containing an Authorization header.

KEY CONCEPT HTTP response headers appear in HTTP Response messages, where they provide additional information about HTTP server capabilities and requirements, and the results of processing a client request.

HTTP Entity Headers

Last, but not least, we come to the fourth group of HTTP headers: *entity headers*. These headers provide information about the resource carried in the body of an HTTP message, called an *entity* in the HTTP standards. They serve the overall purpose of conveying to the recipient of a message the information it needs to properly process and display the entity, such as its type and encoding method.

The most common type of entity is a file or another set of information that has been requested by a client, and for this reason, entity headers most often appear in HTTP Response messages. However, they can also appear in HTTP Request messages, especially those using the PUT and POST methods, which are the ones that transfer data from a client to a server.

At least one entity header should appear in any HTTP message that carries an entity. However, they may also be present in certain responses that do not have an actual entity in them. Most notably, a response to a HEAD request will contain all the entity headers associated with the resource specified in the request; these are the same headers that would have been included with the entity had the GET method been used instead of the HEAD method on the same resource. Entity headers may also be present in certain error responses to provide information to help the client make a successful follow-up request.

NOTE *Many of the entity headers have the same names as certain MIME headers, but they are often used in different ways. See the topic on HTTP Internet media types in Chapter 83 for a full discussion of the relationship between HTTP and MIME.*

There are ten entity headers defined for HTTP/1.1:

Allow Lists all the methods that are supported for a particular resource. This header may be provided in a server response as a guide to the client regarding what methods it may use on the resource in the future. The header must be included when a server returns a 405 (Method Not Allowed) response to a request containing an unsupported method.

Content-Encoding Describes any optional method that may have been used to encode the entity. This header is most often used when transferring entities that have been compressed. It tells the recipient which algorithm has been used so the entity can be uncompressed. Note that this header describes only transformations performed on the entity in a message; the Transfer-Encoding header describes encodings done on the message as a whole. See the discussion of content codings and transfer codings in Chapter 83 for more details.

Content-Language Specifies the natural (human) language intended for using the entity. This is an optional header, and it may not be appropriate for all resource types. Multiple languages may be specified, if needed. This header is intended to provide guidance so the entity can be presented to the correct audience; thus, the language should be selected based on who would best use the material, which may not necessarily include all of the languages used in the entity. For example, a German analysis of Italian operas would probably be best tagged only with the language *de*.

Content-Length Indicates the size of the entity in octets. This header is important, as it is used by the recipient to determine the end of a message. However, it may be included only in cases where the length of a message can be fully determined prior to transmitting the entity. This is not always possible in the case of dynamically generated content, which complicates message-length calculation; the discussion of data length and chunked transfer encoding in Chapter 83 contains a full exploration of this issue.

Content-Location Specifies the resource location of the entity, in the form of an absolute or relative URL. This is an optional header, and it is normally included only in cases where the entity has been supplied from a location different from the one specified in the request. This may occur if a particular resource is stored in multiple places.

Content-MD5 Contains a Message Digest 5 (MD5) digest for the entity, used for checking message integrity.

Content-Range Sent when a message contains an entity that is only part of a complete resource—for example, a fragment of a file sent in response to an HTTP GET request containing the Range header. The Content-Range header indicates which portion of the overall file this message contains, as well as the total size of the resource. This information is given as a byte range, with the first byte numbered 0. For example, if the entity contains the first 1200 bytes of a 2000-byte file, this header would have a value of 0-1199/2000.

Content-Type Specifies the media type and subtype of the entity, in a manner very similar to how this header is used in MIME. See Chapter 83 for a full discussion.

Expires Specifies a date and time after which the entity in the message should be considered stale. This may be used to identify certain entities that should be held in HTTP caches for longer or shorter periods of time than usual. This header is ignored if a Cache-Control header containing the max-age directive is present in the message.

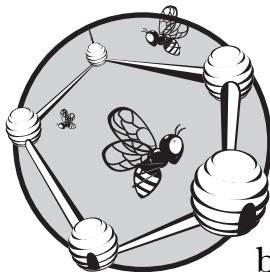
Last-Modified Indicates the date and time when the server believes the entity was last changed. This header is often used to determine if a resource has been modified since it was last retrieved. For example, suppose a client machine already contains a copy of a very large file that was obtained two months ago, and its user wants to check if an update to the file is available. The client can send a HEAD request for the file, and compare the value of the returned Last-Modified header to the date of the copy of the file it already has. Then it needs to request the entire file only if it has changed.

Note the use of the word “believes” in the preceding description of the Last-Modified header. The reason for this wording is that the server cannot always be certain of the time that a resource was modified. With files this is fairly simple—it is usually the last-modified time stored for the file by the operating system. For other more complex resources such as database records or virtual objects, however, it may be more difficult to ascertain when the last change occurred to a particular piece of information. In the case of dynamically generated content, the Last-Modified date/time may be the same as that of the message as a whole, as specified in the Date field.

KEY CONCEPT HTTP *entity headers* appear in either Request or Response messages that carry an entity in the message body. They describe the nature of the entity, including its type, language, and encoding, to facilitate the proper processing and presentation of the entity by the device receiving it.

83

HTTP ENTITIES, TRANSFERS, CODING METHODS, AND CONTENT MANAGEMENT



Hypertext Transfer Protocol (HTTP) message headers are very important, because they are the mechanism that HTTP uses to allow devices to specify the details of client requests and server responses. These headers, however, are only the means to an end, which is the transfer of resources (such as files, form input, and program output) from one device to another. When a resource is carried in the body of an HTTP message, it is called an *entity*. HTTP defines special rules for how these entities are identified, encoded, and transferred.

In this chapter, I explain how HTTP handles entities. I begin with a discussion of entities in general terms and a look at how their contents are identified. This includes an examination of the relationship between HTTP and Multipurpose Internet Mail Extensions (MIME). I discuss the issues behind the transfer of entities between clients and servers, and the difference between content encodings and transfer encodings. I describe the special issues associated with identifying the length of entities in HTTP messages, and detail the special chunked transfer coding and message

trailers. Finally, I describe the methods by which devices can perform content negotiation and how quality values allow clients to intelligently select different variations of a resource.

HTTP Entities and Internet Media Types

The presence of the word *text* in the name Hypertext Transfer Protocol is a reminder of the legacy of HTTP. As I explained in Chapter 80, HTTP was originally created to allow text documents to be linked together. This made sense, because at the time that the Web was being created, most computing was being done with text. Accordingly, the first version of HTTP (HTTP/0.9) supported only one type of message body: a plain ASCII text document.

In the early 1990s, the rapid increase in computing power and networking performance transformed the world of information technology from text to multimedia. These were also the Web's formative years, and it did not take long before many users wanted to exploit the power of the Web to share not only text files, but also pictures, drawings, sound clips, movies, and much more. Thus, HTTP had to evolve as well. Starting with HTTP/1.0, the protocol's developers made significant changes to allow HTTP to transport and process much more than just text. Today, HTTP really would be better described as dealing with *hypermedia* than *hypertext*.

One drawback of supporting many types of files in HTTP is added complexity. Previously, every message recipient knew the body contained ASCII text; now any message can contain any of many kinds of data. When HTTP was expanded to support flexible media, it needed a system that would address two specific issues: encoding entities of various types into an HTTP message body and clearly identifying the entity's characteristics for the recipient of the message.

At the same time that HTTP was being changed to support nontext entities, another important TCP/IP application was also moving away from its decades-long role as a text-messaging medium to one that could transport multimedia: electronic mail (email). This was accomplished using a technology called MIME (introduced in Chapter 76), which define a mechanism for encoding and identifying nontext data—exactly what HTTP needed to do. Since TCP/IP developers wisely reuse technologies that work, the creators of HTTP borrowed many concepts from MIME, including many of the MIME email headers that are used to identify the contents of a MIME message.

Media Types and Subtypes

The most important concept that HTTP adopted from MIME was the use of standardized Internet media types, which describe the contents of an HTTP entity. The formal syntax of an HTTP media type is the same as that used in MIME:

```
<type>/<subtype> [; parameter1 ; parameter2 ... ; parameterN ]
```

Each media type consists of a top-level media type that defines its general nature and a more specific subtype that indicates its form or structure. For example, text documents use the top-level media type `text`, with subtypes such as `plain` for regular unformatted text and `html` for HTML documents. So, an HTML

document of the type commonly transported using HTTP will be identified with a media type of `text/html`. Similarly, `image` is a top-level media type, with subtypes such as `jpeg`, `gif`, and `tiff`. Photographs usually are identified as `image/jpeg`, while line drawings are often seen as `image/gif`. Additional parameters may also be supplied to provide more information to help a recipient interpret the entity.

HTTP's Use of Media Types

In HTTP, media types are most often seen in a special `Content-Type` entity header, which is present in any HTTP message that carries an entity. This header uses the same format as the header of the same name in MIME:

`Content-Type: <type>/<subtype> [; parameter1 ; parameter2 ... ; parameterN]`

RELATED INFORMATION I provide a more complete description of both the `Content-Type` header and Internet media types, including a description of many types and subtypes, in Chapter 76. HTTP can also support composite media types, such as the `multipart` media type.

The other place where media types are used in HTTP is in the `Accept` request header, which may appear in an HTTP request sent by a client. If present, the purpose of this header is to tell the server what sorts of media types the client can handle, so the server will not send a response that cannot be processed. For example, if a client can process only text documents, it might send a request specifying this in an `Accept` header. This is part of the overall content negotiation process supported by HTTP, which I describe in the “HTTP Content Negotiation and Quality Values” section later in this chapter.

When a media type is specified in an `Accept` header, either the subtype or both the type and subtype can be replaced by the asterisk (*) wildcard to represent any acceptable type. For example, in an `Accept` header, the specification `text/html` refers to an HTML document, while `text/*` means any text type. The string `*/*` means any type of media; this is usually used in combination with a `q` value, as explained in the discussion of the HTTP content negotiation process later in this chapter.

KEY CONCEPT While HTTP is most often associated with hypertext, its messages can transport a large variety of different types of files, including images, audio, video, and much more. To indicate the type of entity contained in an HTTP message, its sender must identify its media type and subtype. This is done using the `HTTP Content-Type` header, which was borrowed from the Multipurpose Internet Mail Extensions (MIME) specification.

Differences in HTTP and MIME Constructs

In addition to media types, HTTP also borrows from MIME in several other ways, such as MIME’s notion of content codings and the use of a header to indicate the length of an entity. It’s important to recognize, however, that even though HTTP’s handling of Internet media is very similar to that of MIME, it is not identical. In fact, there was an early proposal that HTTP use MIME exactly as defined, but HTTP’s developers specifically decided not to do this. We will explore a possible reason why HTTP is not strictly MIME-compliant in the next section.

The bottom line is that HTTP's developers chose to adopt concepts from MIME that made sense and to leave other parts out. As a result, HTTP messages are not MIME-compliant, even though you may see several headers in HTTP messages starting with MIME's Content- prefix. For example, even though HTTP has a Content-Encoding header, its use is quite different from that of MIME's. The fact that HTTP does not use the MIME-Version header that is required in MIME messages confirms the difference between HTTP and MIME.

KEY CONCEPT Even though HTTP borrows several concepts and header types from MIME, the protocol is not MIME-compliant.

HTTP Content and Transfer Encodings

Two specific issues that HTTP must address in order to carry a wide variety of media types in its messages are encoding the data and identifying its type and characteristics. HTTP borrows from MIME the notion of media types and the Content-Type header to handle type identification, as explained in the previous section. It similarly borrows concepts and headers from MIME to deal with the encoding issue. Here, however, we run into some of the important differences between HTTP and MIME.

Encoding was a significant issue for MIME, because it was created for the specific purpose of sending nontext data using the old RFC 822 email message standard (discussed in Chapter 76). RFC 822 imposes several significant restrictions on the messages it carries, the most important of which is that data must be encoded using 7-bit ASCII. RFC 822 messages are also limited to lines of no more than 1,000 characters that end in a carriage return/line feed (CRLF) sequence.

These limitations mean that arbitrary binary files, which have no concept of lines and consist of bytes that can each contain a value from 0 to 255, cannot be sent using RFC 822 in their native format. In order for MIME to transfer binary files, they must be encoded using a method such as base64 (described in Chapter 76), which converts three 8-bit characters to a set of four 6-bit characters that can be represented in ASCII. When this sort of transformation is done, the MIME Content-Transfer-Encoding header is included in the message, so the recipient can reverse the encoding to return the data to its normal form. Although this technique works, it is less efficient than sending the data directly in binary, because base64 encoding increases the size of the message by 33 percent (three bytes are encoded using four ASCII characters, each of which takes one byte to transmit).

HTTP messages are transmitted directly between the client and server over a Transmission Control Protocol (TCP) connection, and they do not use the RFC 822 standard. Thus, binary data can be sent between HTTP clients and servers without the need for base64 encoding or other transformation techniques. Since it is more efficient to send the data unencoded, this may be one reason why HTTP's developers decided not to make the protocol strictly MIME-compliant.

HTTP's Two-Level Encoding Scheme

So, encoding would seem to be an area where HTTP is simpler than MIME. There is no need to encode the entity, and thus no need for the Content-Transfer-Encoding header, so we have one less thing to worry about. It is true that HTTP could have been designed so that all entities were just sent one byte at a time with no need to specify encodings, but the developers of the protocol recognized that this would have made the protocol inflexible. There are situations where it might be useful to transform or encode an entity or message for transmission, and then reverse the operation on receipt.

This effort to make HTTP flexible resulted in a system of representing encodings that is actually more complicated than MIME's! The key to understanding it is to recognize that HTTP/1.1 actually splits MIME's notion of content transfer encoding into two different encoding levels:

Content Encoding This encoding is applied specifically to the entity carried in an HTTP message, to prepare or package it prior to transmission. Content encodings are said to be *end-to-end*, because the encoding of the entity is done once before it sent by the client or server, and decoded only on receipt by the ultimate recipient: server or client. When this type of encoding is done, the method is identified in the special Content-Encoding entity header. A client may also specify which content encodings it can handle, using the Accept-Encoding header, as you will see in the section on content negotiation later in this chapter.

Transfer Encoding This encoding is done specifically for the purpose of ensuring that data can be safely transferred between devices. It is applied across an entire HTTP message and not specifically to the entity. This type of encoding is *hop-by-hop*, because a different transfer encoding may be used for each hop of a message that is transmitted through many intermediaries in the request/response chain. The transfer encoding method, if any, is indicated in the Transfer-Encoding general header.

Use of Content and Transfer Encodings

Since the content and transfer encodings are applied at different levels, it is possible for both to be used at the same time. A content encoding may be applied to an entity and then placed into a message. On some or all of the hops that are used to move the message containing that entity, a transfer encoding may be applied to the entire message (including the entity). The transfer encoding is removed first, and then the content encoding is removed.

So, what are these types of encodings used for in practice? The answer is not a great deal. The HTTP standard defines a small number of content and transfer encodings, and specifies that additional methods may be registered with the Internet Assigned Numbers Authority (IANA). Currently, only the ones defined in the HTTP/1.1 standard are in use.

Content encodings are used only to implement compression. This is a good example of an encoding that, while not strictly necessary, can be useful since it improves performance dramatically for some types of data. RFC 2616 defines three different encoding algorithms:

- gzip, which is the compression used by the UNIX gzip program, described in RFC 1952
- compress, which also represents the compression method used by the UNIX program of that name
- deflate, which is a method defined in RFCs 1950 and 1951

NOTE *It is also possible to apply compression to an entire HTTP message as a transfer encoding. Obviously, if the entity is already compressed using content encoding, this will result in some duplication of effort. Since the size of HTTP headers is not that large compared to some entities that HTTP messages carry, it is usually simpler just to compress the entity using content encoding.*

Since transfer encodings are intended to be used to make data safe for transfer, and we've already discussed the fact that HTTP can handle arbitrary binary data, this suggests that transfer encodings are not really necessary. However, there is one situation where safe transport does become an issue: the matter of identifying the end of a message. This issue is the subject of the next section.

KEY CONCEPT HTTP supports two levels of codings for data transfer. The first is *content encoding*, which is used in certain circumstances to transform the entity carried in an HTTP message. The second is *transfer encoding*, which is used to encode an entire HTTP message to ensure its safe transport. Content encodings are often employed when entities are compressed to improve communication efficiency. Transfer encoding is used primarily to deal with the problem of identifying the end of a message.

HTTP Data Length Issues, Chunked Transfers, and Message Trailers

As you've learned, two different levels of encodings are used in HTTP: *content encodings*, which are applied to HTTP entities, and *transfer encodings*, which are used over entire HTTP messages. Content encodings are used for convenience to package entities for transmission. Transfer encodings are hop-specific, and they are intended for use in situations where data needs to be made safe for transfer.

However, we've already seen that HTTP can transport arbitrary binary data, so unlike the situation where MIME needed to make binary data safe (as defined in RFC 822), this is not an issue with HTTP. Therefore, why are transport encodings needed at all? In theory, they are not, and HTTP/1.0 did not even have a Transfer-Encoding header (though it did use content encodings). The concept of transfer encoding became important in HTTP/1.1 due to another key feature of that version of HTTP: persistent connections (described in Chapter 80).

Dynamic Data Length

Recall that HTTP uses TCP for connections. One of the key characteristics of TCP is that it transmits all data as a stream of unstructured bytes (see Chapter 46). TCP itself does not provide any way of differentiating between the end of one piece of data and the start of the next; this is left up to each application. In HTTP/1.0 (and HTTP/0.9), this was not a problem, because those versions used only transitory connections. Each HTTP session consisted of only one request and one response. Since the client and server each sent only one piece of data, there was no need to worry about differentiating HTTP messages on a connection.

HTTP/1.1's persistent connections improve performance by letting devices send requests and responses one after the other over a single TCP connection. However, the fact that messages are sent in sequence makes differentiating them a concern. There are two usual approaches to dealing with this sort of data length issue: using an explicit delimiter to mark the end of the message, or including a length header or field to tell the recipient how long each message is. The first approach could not really have been done easily while maintaining compatibility with older versions of the protocol. This left the second approach. Since HTTP already had a Content-Length entity header, the solution was to use this to indicate the length of each message at transmission time.

Using the Content-Length header works fine in cases where the size of the entity to be transferred is known in advance, such as when transmitting a text document, an image, or an executable program needs. However, there are many types of resources that are generated dynamically. In those cases, the total size of such a resource is not known until it has been completely processed.

While not typical in HTTP's early days, dynamic resources account for a large percentage of Web traffic today. Many web pages are often not static Hypertext Markup Language (HTML) files, but instead are created as output from scripts or programs based on user input; discussion forums are a good example. Even modern HTML files are often not static. They usually contain program elements such as *server-side includes (SSIs)* that cause code to be generated on the fly, so their exact size cannot be determined in advance.

The problem of unknown message length could be resolved by buffering the entire resource before transmission. However, this would be wasteful of server memory and would delay the transmission of the entity unnecessarily, since no part could be sent until the entire entity was ready. Instead, a special transfer encoding method was developed to handle the particular problem of not knowing the length of a file. The method is called *chunking*.

Chunked Transfers and Message Trailers

When the chunking technique is used, instead of sending an entity as a raw sequence of bytes, it is broken into, well, chunks. This allows HTTP to send a dynamically generated resource, such as output from a script, a piece at a time as the data becomes available from the software processing it. To indicate that this

method has been used, the special header `Transfer-Encoding: chunked` is placed in the message. A special format is also used for the body of the HTTP message to delineate the chunks:

```
<chunk-1-length>
<chunk-1-data>
<chunk-2-length>
<chunk-2-data>
...
0
<message-trailers>
```

Basically, instead of putting the whole entity in the body and indicating its length in a `Content-Length` header, each chunk is placed in the body sequentially, each preceded by the length of the chunk. The length is specified in hexadecimal and represented using ASCII characters. All chunk lengths and chunk data are terminated with a CRLF sequence. The recipient knows it has received the last chunk when it sees a chunk length of zero.

NOTE An HTTP/1.1 client can specify that it does not want to use persistent connections by including the `Connection: close` header in its request. In this case, the server does not have to use chunking in its response. Since the server will close the connection after the first response message, the client knows that everything it receives from the server is part of that response. However, some servers may use chunked transfers anyway, even in this situation.

When chunked transfer encoding is used, the sender of the message may also choose to specify one or more *message trailers*. These are the same as entity headers, describing the contents of the message body, but appear *after* the entity, rather than *before* it. Message trailers provide flexibility in the same way that chunking itself does: They allow a device to include an HTTP header that may contain information that was not available when the HTTP message transmission began. A good example would be an integrity check field calculated based on the byte values of the entire entity.

Trailers are optional, and they will not always be needed. When they are used, they are processed just like regular entity headers. To give the recipient of a message a “heads up” that trailers have been used, the special `Trailer` header is included at the start of the message, which lists the names of each header that appears as a trailer.

Yes, I really did say that headers can actually be trailers, in which case, a header called `Trailer` lists each header that is actually a trailer. An example will help clarify matters somewhat. Suppose we have a server that contains a program that, when supplied with a filename, returns a simple HTML response that contains the size and last modification date of the file. This is obviously dynamic content, so the length of the response cannot be determined in advance. If the server were to

buffer the entire output of this program (since it is small), it could construct a conventional HTTP response using the Content-Length header, as shown in the sample output of Listing 83-1.

```
HTTP/1.1 200 OK
Date: Tue, 22 Mar 2005 11:15:03 GMT
Content-Type: text/html
Content-Length: 129
Expires: Sun, 27 Mar 2005 21:12:00 GMT

<html><body><p>The file you requested is 3,400 bytes long and was last modified:
Sun, 20 Mar 2005 21:12:00 GMT.</p></body></html>
```

Listing 83-1: Example of an HTTP Response using a Content-Length header

Using chunking instead allows the server to send out parts of the response as soon as they become available from the program. The equivalent output of the example shown in Listing 83-1 using chunked transfers is shown in Listing 83-2.

```
HTTP/1.1 200 OK
Date: Tue, 22 Mar 2005 11:15:03 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Trailer: Expires

29
<html><body><p>The file you requested is
5
3,400
23
bytes long and was last modified:
1d
Sun, 20 Mar 2005 21:12:00 GMT
13
.</p></body></html>
0
Expires: Sun, 27 Mar 2005 21:12:00 GMT
```

Listing 83-2: Example of an HTTP Response using chunked transfer encoding

In Listing 83-2, notice that the Expires header is now a trailer, so it can be calculated based on the output of the program, and this is indicated by the Trailer: Expires header. Remember that the Content-Length header specifies the length as a decimal number while chunking specifies chunk lengths in hexadecimal; the chunks in this example are 41, 5, 35, 29, and 19 decimal bytes, respectively.

KEY CONCEPT Since HTTP/1.1 uses persistent connections that allow multiple requests and responses to be sent over a TCP connection, clients and servers need some way to identify where one message ends and the next begins. The easier solution is to use the Content-Length header to indicate the size of a message, but this works only when the length of a message can be determined in advance. For dynamic content or other cases where message length cannot be easily computed before sending the data, the special *chunked* transfer encoding can be used, where the message body is sent as a sequence of chunks, each preceded by the length of the chunk. When chunked transfer encoding is used, the sender of the message may move certain headers from the start of the message to the end, where they are known as *trailers*. Trailers are interpreted in the same way as normal headers by the recipient. The special Trailer header is used in such messages to tell the recipient to look for trailers after the body of the message.

HTTP Content Negotiation and Quality Values

Many Internet resources have only one representation, meaning a single way in which they are stored or made available. In this situation, a client request to a server is an all-or-nothing proposition. The client may specify conditions under which it would like the server to send the resource, using the If- series of request headers described in the previous chapter. If the condition is met, the resource will be sent in the server's response in the one form in which it exists; if the condition is not met, no entity will be returned.

Other resources, however, may have multiple representations. The most common example is a document that is available in multiple languages, or one that is stored using more than one character set. Similarly, a graphical image might exist in two different formats: a Tagged Image File Format (TIFF) file, for those who want maximum image quality despite the large size of TIFF images, and a more compact JPEG file, for those who need to see the image quickly and don't care as much about its quality level.

To provide flexibility in allowing clients to obtain the best version of resources that exist in multiple forms, HTTP/1.1 defines a set of features that are collectively called *content negotiation*.

Content Negotiation Techniques

The HTTP/1.1 standard defines two basic methods by which this negotiation may be performed.

Server-Driven Negotiation In this technique, the client includes headers in its request that provide guidance to the server about its desired representation for the resource. The server uses an algorithm that processes this information and provides the version of the resource that it feels best matches the client's preferences.

Agent-Driven Negotiation This method puts the client in charge of the negotiation process. It first sends a preliminary request for the resource to the server. If the resource is available in multiple forms, the server typically sends back a 300 (Multiple Choices) response, which contains a list of the various representations in which the resource is available. The client then sends a second request for the one that it prefers.

To draw an analogy, suppose a co-worker offers to go out at lunchtime to pick up lunch for the two of you. He is going to a new restaurant, where neither of you have eaten before. You could provide him with some parameters regarding what you like to eat—“I like roast beef sandwiches, fish and chips, and pizza, but not chicken”—and then trust him to pick something you will like. Alternatively, he could go to the restaurant, call you on his cell phone, read the menu to you, and let you make a selection. This former approach is like server-driven negotiation; the latter is like agent-driven negotiation.

This analogy not only points out the differences between the two methods, but it also highlights the key advantages and disadvantages of each. Trusting your co-worker with your lunch selection is simple and efficient, but not foolproof. It’s possible that the restaurant may not have any of the items you specified, or that your friend may get you something containing another ingredient that you don’t like but forgot to mention. Similarly, server-based negotiation is a best-guess process that does not guarantee that the client will receive the resource in the format it wants. This is exacerbated by the fact that there are only so many ways for the client to specify its preferences using a handful of request headers.

Agent-based negotiation, on the other hand, allows the client to select exactly what it wants from the available choices, just as you can choose your favorite dish from the menu of the restaurant. The problem here is that it is inefficient, because two requests and responses are required for each resource access. (Would you really want to read a restaurant’s menu over the phone to someone so he could choose his ideal dish?)

In practice, server-based negotiation is the type that is most commonly used today. The client specifies its preferences using a set of four request headers that indicate what it would prefer in the representation of the resource. The headers each represents one characteristic of a resource: Accept (media type), Accept-Charset (character set), Accept-Encoding (content encoding), and Accept-Language (resource language). Any or all of these may be included in the request. Each Accept- header contains a list of acceptable values that is appropriate to the characteristic that it specifies, separated by a comma. For example, the Accept header lists media types the client considers acceptable, and Accept-Language contains language tags.

For example, suppose you have a friend who is trilingual in English, French, and Spanish. She can read a particular document in any of these languages, so she might instruct her browser to include the following header in her requests:

Accept-Language: en, fr, sp

KEY CONCEPT HTTP includes a feature called *content negotiation* that allows the selection of a particular variation of a resource that has more than one representation. There are two negotiation techniques: *server-driven*, where the client includes headers in its request that indicate what it wants and the server does its best to select the most appropriate variant, and *agent-driven*, where the server sends the client a list of the available resource alternatives and the client chooses one.

Quality Values for Preference Weights

To improve the results of server-driven negotiation, HTTP allows the client to *weight* each of the items in such a list, to indicate which is preferred of the alternatives. The client specifies weights by adding a decimal *quality value* after each parameter using the syntax `q=<value>`, which represents the relative priority of that parameter relative to others. The highest priority is 1, and the lowest priority is 0. The default if no value is indicated is 1. A value of 0 means that the client will not accept documents with that characteristic.

For example, suppose your trilingual friend knows English, French, and Spanish, but her French is a bit rusty. Furthermore, she may need to share the document she is requesting with a friend of hers who knows only a little Spanish, so it would be best if she got the document in English. Finally, she knows there is a German version of the resource that she definitely does not want. This could be represented as follows:

Accept-Language: en, fr;q=0.3, sp;q=0.7, de;q=0

Translated to English, this means, “I would prefer if you sent me the document in English. If not, Spanish is okay, or French if that is all you have, but definitely don’t send it to me in German.”

Incidentally, the name *quality value* is the one used in the HTTP standard, but it is really a poor choice of terminology (a point which, to be fair, is also mentioned in the standard). These values do not have anything to do with quality; for all we know, the German version of this document may be the original and the others could be lousy translations. The q values specify only the relative preference of the client making the request.

Finally, the asterisk (*) wildcard can be used in the Accept family of headers to represent any value or everything else. This is often used to tell the server, “If you can’t find what I specifically asked for, then here are my preferences for the alternatives.” Let’s take an example using the Accept header:

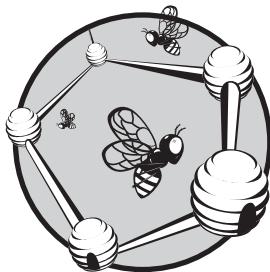
Accept: text/html, text/*;q=0.6, */*;q=0.1

This header represents the client saying, “My preference (q=1, the default since no q value is indicated) is an HTML text document. If not available, I would prefer some other type of text document. Failing that, you may send me any other type of document relevant to the requested resource.”

KEY CONCEPT Server-driven content negotiation is the type most often used in HTTP. A client sending a request can include up to four different headers that provide information about how the server should fill its request. These may include optional *quality values* that specify the client’s relative preference among a set of alternative resource characteristics such as media type, language, character set, and encoding.

84

HTTP FEATURES, CAPABILITIES, AND ISSUES



The previous chapters covered the fundamental concepts and basic operation of the Hypertext Transfer Protocol (HTTP).

Modern HTTP, however, goes beyond the simple mechanics by which HTTP requests and responses are exchanged. It includes a number of features and capabilities that extend the basic protocol to improve performance and meet the various needs of organizations using modern TCP/IP internetworks.

In this chapter, I complete my description of HTTP by discussing several important matters that are essential to the operation of the modern World Wide Web. I begin with an overview of HTTP caching, which is the single most important feature that promotes efficiency in web transactions. I discuss the different uses of proxies in HTTP and some of the issues associated with them. I briefly examine the issues related to security and privacy in HTTP and conclude with a discussion of the matter of state management and how it is implemented despite HTTP being an inherently stateless protocol.

HTTP Caching Features and Issues

The explosive growth of the Web was a marvel for its users but a nightmare for networking engineers. The biggest problem that the burgeoning Web created was an overloading of the internetworks over which it ran. Many of the features that were added to HTTP/1.1 were designed specifically to improve the efficiency of the protocol and reduce unnecessary bandwidth consumed by HTTP requests and responses. Arguably, the most important of these is a set of features designed to support *caching*.

The subject of caching comes up again and again in discussions of computers and networking, because of a phenomenon that is widely observed in these technologies: Whenever a user, hardware device, or software process requests a particular piece of data, there is a good chance it will ask for that same data again in the near future. Thus, by storing recently retrieved items in a cache, we can eliminate duplicated effort. This is the reason that caching plays an important role in the efficiency of protocols such as the Address Resolution Protocol (ARP) and the Domain Name System (DNS).

Benefits of HTTP Caching

Caching is important to HTTP because Web users tend to request the same documents over and over again. For example, in writing this section on HTTP, I made reference to RFC 2616 many, many times. Each time, I loaded it from a particular web server. Since the document never changes, it would have been more efficient to just load it from a local cache rather than needing to retrieve it from the distant web server each time.

However, caching is even more essential to HTTP than to most other protocols or technologies where it used. The reason is that web documents tend to be structured so that a request for one resource leads to a request for many others. Even if you load a number of *different* documents, they may each refer to common elements that do not change between your requests. Thus, caching can be of benefit in HTTP even if a user never asks for the same document twice, or if a single document changes over time so that caching the document itself would be of little value.

For example, suppose that each morning, you load CNN's website to see what is going on in the world. Obviously, the headlines will be different every day, so caching of the main CCN.com home page won't be of much value. However, many of the graphical elements on the page (CNN's logo, dividing bars, perhaps a "breaking news" graphic, and so on) will be the same every day, and these can be cached. Another example would be a set of discussion forums on a website. As you load different topics to read, each one is different, but they have common elements (such as icons and other images) that would be wasteful to need to retrieve over and over again.

Caching in HTTP yields two main benefits:

- Reduced bandwidth use, by eliminating unneeded transfers of requests and responses
- Faster response time for the user loading a resource

Consider that on many web pages today, the image files are much larger than the HTML page that references them. Caching these graphics will allow the entire page to load far more quickly. Figure 84-1 illustrates how caching reduces bandwidth and speeds up resource retrieval by short-circuiting the request/response chain.

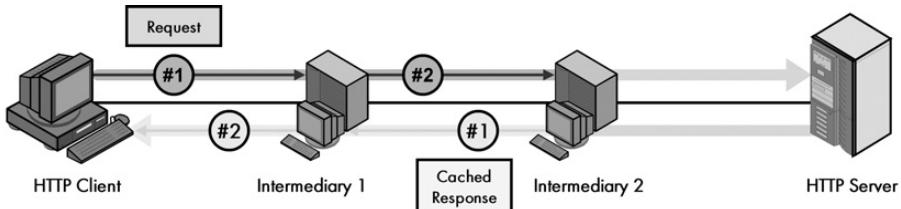


Figure 84-1: Impact of caching on the HTTP request/response chain This diagram illustrates the impact of caching on the request/response chain of (see Figure 80-2 in Chapter 80). In this example, intermediary 2 is able to satisfy the client's request from its cache. This short-circuits the communication chain after two transfers, which means the client gets its resource more quickly, and the HTTP server is spared the need to process the client's request.

The obvious advantages of caching have made it a part of the Web since pretty much the beginning. However, it was not until HTTP/1.1 that the importance of caching was really recognized in the protocol itself, and many features were added to support it. Where the HTTP/1.0 standard makes passing mention of caching and some of the issues related to it, HTTP/1.1 devotes 26 full pages to caching (more than 20 percent of the main body of the document!).

Cache Locations

HTTP caching can be implemented in a variety of places in the request/response chain. The choice of location involves the fundamental trade-off that always occurs in caching: proximity versus universality. Simply put, the closer the cache is to the requester of the information, the more savings that result when data is pulled from the cache, rather than being fetched from the source. However, the further the cache is from the requester (and thus closer to the source), the greater the number of devices that can benefit from the cache. Let's see how this manifests itself in the three classes of devices where caches may be found: the web client, intermediary, and web server.

Caching on the Web Client

The cache with which most Internet users are familiar is that found on the local client. It is usually built into the web browser software, and for this reason, it's called a *web browser cache*. This cache stores recent documents and files accessed by a particular user, so that they can be made quickly available if that user requests them again.

Since the cache is in the user's own machine, a request for an item that the cache contains is filled instantly, resulting in no network transaction and instant gratification for the user. However, that user is the only one who can benefit from the cache, so it's sometimes called a *private cache*.

Caching on the Intermediary

Devices such as proxy servers that reside between web clients and servers are also often equipped with a cache. If users want documents that are not in their local client cache, the intermediary may be able to provide it, as shown in Figure 84-1. This is not as efficient as retrieving from the local cache, but far better than going back to the web server.

An advantage is that all devices using the intermediary can benefit from its cache, which may be termed a *public* or *shared cache*. This can be useful, because members of an organization often access similar documents. For example, in an organization developing a hardware product to be used on Apple computers, many different people might be accessing documents on Apple's website. With a shared cache, a request from User A would often result in items being cached that could be used by User B as well.

Caching on the Web Server

Web servers themselves may also implement a cache. While it may seem a bit strange to have a server maintain a cache of its own documents, this can be of benefit in some circumstances. A resource might require a significant amount of server resources to create. For example, consider a web page that is generated using a complex database query. If this page is retrieved frequently by many clients, there can be a large benefit to creating it periodically and caching it, rather than generating it on the fly for each request.

Since the web server cache is the farthest from the users, this results in the least savings for a cache hit, as the client request and server response must still travel the full path over the network between the client and server. However, this distance from the client also means that all users of the server can benefit from the cache.

KEY CONCEPT The most important feature that improves the efficiency of operation of HTTP is *caching*—the storing of recently requested resources in a temporary area. If the same resource is then needed again a short time later, it can be retrieved from the cache rather than requiring a fresh request to the server, resulting in a savings of both time and bandwidth. Caching can be performed by web clients, web servers, and intermediaries. The closer the cache is to the user, the greater the efficiency benefits; the farther from the user, the greater the number of users who can benefit from the cache.

Cache Control

Caching in clients and servers is controlled in the same manner as most other types of control are implemented in HTTP: through the use of special headers. The most important of these is the Cache-Control general header, which has a number of directives that allow the operation of caches to be managed. There are other important caching-related headers, including Expires and Vary. For a great deal of more specific information related to HTTP caching, see RFC 2616, section 13.

Important Caching Issues

While the performance advantages of caching are obvious, caching has one significant drawback: it complicates the operation of HTTP in a number of ways. The following are some of the more important issues that HTTP/1.1 clients, servers, and intermediaries need to address. This list is not exhaustive, but it gives you an idea of what is involved with caching in HTTP.

Cache Aging and Staleness When users retrieve a document directly from its original source on the server, they are assured of getting the current version of that resource. When caching is used, that is no longer the case. While many resources change infrequently, almost all will change at some point. For example, at CNN’s website, it is probable that the CNN logo won’t change very often, but it’s possible that the site may be redesigned periodically and the logo modified in some way, such as its size or color. For this reason, a device cannot keep items in an HTTP cache indefinitely. The longer an item is held in a cache—a process called *aging*—the more likely it is that the resource on the server has changed and the cache has become stale. To make matters even more complex, some resources become stale more quickly than others. As a result, much of the caching-related functionality of HTTP involves dealing with this matter of cache aging.

Cache Expiration and Validation One of the ways that HTTP deals with the cache aging issue is through headers and logic that allow caches, clients, and servers to specify how long items should be cached before they expire and must be refreshed. A validation process allows a cache to check with a server at appropriate times to see if an item it has stored has been modified.

Communication of Cache Status to the User In most cases, the fact that an item has been retrieved from a cache rather than its source is transparent to users (though they may notice that the resource loads faster than expected). In certain cases, however, the user may need to be informed that a resource came from a cache and not its original source. This is especially true when a cached item may be stale; in which case, the client should warn the user that the information might be out-of-date.

Header Caching Caching in HTTP is complicated by the fact that it can occur in multiple places, and some HTTP headers are treated differently than others. HTTP headers are divided into two general categories: *end-to-end headers* that are intended to accompany a resource all the way to its ultimate recipient, and *hop-by-hop headers* that are used only for a particular communication between two devices (by the client, server, or intermediary device). End-to-end headers must be stored with a cached resource. Hop-by-hop headers have meaning only for a particular transfer and are not cached.

Impact of Resource Updates Some HTTP methods (discussed in Chapter 81) will automatically cause cache entries to become invalidated, because they inherently cause a change to the underlying resource. For example, if a user performs a PUT

on a resource that was previously retrieved using GET, any cached copies of that resource should be automatically invalidated to prevent the old version from being supplied from the cache.

Privacy Concerns In the case of shared caches (such as might exist in a proxy), there are potential privacy issues. In most cases, having User A's cached resource be made available to User B is advantageous, but we must be careful not to cache any items that might be specific to User A, which User B should not see.

HTTP Proxy Servers and Proxying

In my overview of the HTTP operational model in Chapter 80, I described how HTTP was designed to support not just communication between a client and server, but also the inclusion of intermediaries that may sit in the communication path between them. One of the most important types of intermediary is a device called a *proxy server*, or more simply, just a *proxy*.

A proxy is a middleman that acts as both a client and a server. It accepts requests from a client as if it were a server, then forwards those requests (possibly modifying them) to the real server, which sees the proxy as a client. The server responds back to the proxy, which forwards the reply back to the client. Proxies can be either *transparent*, meaning that they do not modify requests and responses, or *nontransparent*, if they do modify messages in order to provide a particular service.

NOTE *The term transparent proxy can also be used to refer to a proxy that is interposed automatically between a client and server—such as an organization-wide firewall—as opposed to one that a user manually configures.*

Benefits of Proxies

Since proxies have the ability to fully process all client requests and server responses, they can be extremely useful in a number of circumstances. They can be used to implement or enhance many important capabilities, such as the following:

Security Proxies can be set up to examine both outgoing requests and incoming responses, to address various security concerns. For example, filtering can be set up to prevent users from requesting objectionable content or to screen out harmful replies, such as files containing hidden viruses.

Caching As you saw earlier, it can be advantageous to set up a shared cache that is implemented on an intermediary, so resources requested by one client can be made available to another. This can be done within a proxy server.

Performance In some circumstances, using a proxy server can significantly improve performance, particularly by reducing latency.

An excellent example of how a proxy server can improve performance is how proxying is used by my own satellite Internet connection. Due to the distance from the Earth to the satellite, it takes more than 500 milliseconds for a round-trip request/response cycle between my PC and my Internet server provider (ISP). If I loaded a web page containing images, I would need to wait 500+ milliseconds to get

the HTML page, and then my browser would need to generate new requests for each graphical element, meaning another 500+ millisecond delay for each. Instead, my ISP has a proxy server to which I send my requests for web pages. The proxy server looks through the HTML of these pages and automatically requests any elements such as graphics for me. It then sends them straight back to my machine, thus drastically reducing the time required to display a full web page.

KEY CONCEPT One of the most important types of intermediary devices in HTTP is a *proxy server*, which acts as a middleman between the client and server, handling both requests and responses. A proxy server may transport messages unchanged or may modify them to implement certain features and capabilities. Proxies are often used to increase the security and/or performance of Web access.

Comparing Proxies and Caches

Proxying and caching are concepts that have a number of similarities, especially in terms of the impact that they have on basic HTTP operation. Like caching, proxying has become more important in recent years, and it also complicates HTTP in a number of ways. The HTTP/1.1 standard includes a number of specific features to support proxies, and it also addresses a number of concerns related to proxying.

The fact that both proxying and caching represent ways in which basic HTTP client/server communication is changed, combined with the ability of proxies to perform caching, sometimes leads people to think caches and proxies are the same, which is not true. A proxy is a separate element that resides in the HTTP request/response chain. Caches can be implemented within any device in that chain, including a proxy.

Another key way that caches and proxies differ is that caches are used automatically when they are enabled, but proxies are not. To use a proxy, client software must be told to use the proxy and supplied with its IP address or domain name. The client then sends all requests to the proxy, rather than to the actual server that the user specifies.

NOTE Most of my explanations here have focused on hardware proxy servers, but proxies are also commonly implemented as software in a client device. A software proxy performs the same tasks of processing requests and responses. A software proxy is much cheaper to implement than a hardware proxy, but it cannot be shared by many devices.

Important Proxying Issues

As with caching, issues arise when proxies are used in HTTP. The following are some of the more important ones. (For much more information about proxying, refer to RFC 2616).

Capability Inconsistencies Issues arise when a client and server don't use the same version of HTTP or don't support the same features. For example, some servers may not support all of the methods that a client may try to use. This becomes more complex when a proxy enters the picture. Of particular concern is the situation

where a client and server may agree on a particular feature that the proxy does not. The proxy must make sure that it passes along headers or other elements that it may not comprehend.

Authentication Requirements The use of proxy servers often introduces new authentication or security requirements. In addition to authenticating with an end server, the proxy may specify that the client needs to present separate authentication credentials to it as well. This is done using the HTTP Proxy-Authorization and Proxy-Authenticate headers, as discussed in the next section in this chapter.

Caching Interaction Not only do both caching and proxying both complicate HTTP, they can complicate each other. Many of the issues in handling caching—such as header caching, expiration, and validation—become more complex when proxies are involved. Some of the Cache-Control general header directives are specific to proxying. Another issue is that the use of proxying and caching together can lead to distortions in the apparent number of times that a web resource is accessed. This is important in situations where web pages are supported by advertising, based on the number of times the page is accessed. In some cases, special codes called *cache busters* are placed in URLs to force pages not to be stored in shared caches.

Encodings Content encodings (discussed in Chapter 83) are applied end-to-end and so should not be affected by proxies. Transfer encoding is done hop-by-hop, so a proxy may use different encodings in handling different transfers of a single request or response.

Tracing Proxy Handling It is useful in some circumstances, especially when multiple proxies may be in the request/response chain, to be able to trace which proxies have processed a particular message. To this end, HTTP/1.1 requires that each proxy that handles a message identify itself in the Via header.

HTTP Security and Privacy

Many TCP/IP protocols lack security measures, largely because they were developed when security wasn't a big concern. As the Internet has developed, security has become extremely important, however. In the case of the Web, the issue is even more important due to the significance of the changes that have occurred in the content of HTTP messages since the protocol was first developed.

HTTP has become the vehicle for transporting any and every kind of information, including a large amount of personal data. HTTP was initially designed to carry academic documents such as memos about research projects. Today, an HTTP message is more likely to carry someone's mortgage application, credit card number, or medical details. Thus, not only does HTTP have the usual security issues such as preventing unauthorized access, but it also needs to deal with privacy concerns.

HTTP Authentication Methods

The main HTTP/1.1 standard, RFC 2616, does not deal extensively with security matters. These are addressed in detail instead in the companion document, RFC 2617, which explains the two methods of HTTP authentication:

Basic Authentication This is a conventional user name/password type of authentication. When a client sends a request to a server that requires authentication to access a resource, the server sends a response to the client's initial request that contains a WWW-Authenticate header. The client then sends a new request containing the Authorization header, which carries a base64-encoded user name and password combination. Basic authentication is not considered strong security because it sends credentials unencrypted, which means that they can be intercepted.

Digest Authentication Digest authentication uses the same headers as basic authentication, but employs more sophisticated techniques, including encryption, that protect against a malicious person snooping credentials information. Digest authentication is not considered as strong as public key encryption, but it is a lot better than basic authentication. It's also a lot more complicated. The full details of how it works are in RFC 2617.

Security and Privacy Concerns and Issues

Both RFC 2616 and 2617 address some of the specific security concerns and threats that can potentially affect HTTP clients and servers. These include actions such as spoofing, counterfeit servers, replay attacks, and much more. One concern addressed is the potential for man-in-the-middle attacks, where an attacker interposes between the client and server. Since proxies are inherently middlemen, they represent a security concern in this area. The same authentication methods used for servers can also be applied to authentication with proxies. In this case, the Proxy-Authenticate and Proxy-Authorization headers are used instead of WWW-Authenticate and Authorization headers.

The HTTP standards also discuss a number of privacy issues. The following are particularly worthy of examining.

Sensitive Information Handling The HTTP protocol can carry any type of information, and it does not inherently protect the privacy of data in HTTP message entities. To ensure the privacy of sensitive information, other techniques must be used (as described in the next section).

Information in URLs One issue that sometimes arises in HTTP is that poorly designed websites may inadvertently encode private information into URLs. These URLs may be recorded in web logs, where they could fall into the hands of people who could abuse them. An example of this is a website that submits a user name and password to a server by encoding them as parameters of a GET request such as this: `GET http://www.somesite.com/login?name=xxx&password=yyy`. The POST method should be used instead for this sort of functionality, because it transmits its data in the body of the message instead of putting it into the URL.

Information in Accept Headers While this may seem strange at first, it is possible that private information about the user could be transmitted through the use of certain Accept headers used for content negotiation. For example, some users might not want others to know what languages they speak, so they may be concerned about who looks at the Accept-Language header.

Information in Referer Headers The Referer (yes, that's how it's spelled; see my note in Chapter 82) request header is a double-edged sword. It can be very useful to those who operate websites because it lets them see the sources of links to their resources. At the same time, it can be abused by those who might employ it to study users' Web-access patterns. There are also potential privacy issues that the HTTP standard raises. For example, a user might not want the name of a private document that references a public web page to be transmitted in a Referer header.

Methods for Ensuring Privacy in HTTP

As mentioned earlier, HTTP does not include any mechanism to protect the privacy of transmitted documents or messages. There are two different methods by which this is normally accomplished:

Encryption The simplest way is to encrypt the resource on the server and supply valid decryption keys only to authorized users. Even if the entire message is intercepted, the entity itself will still be secured. The level of protection here depends on the quality of the encryption.

Secure Sockets Layer (SSL) Another more common method is to use a protocol designed specifically to ensure the privacy of HTTP transactions. The one often used today is called *Secure Sockets Layer (SSL)*. Servers employ SSL to protect sensitive resources, such as those associated with financial transactions. They are accessed by using the URL scheme *https* rather than *http* in a web browser that supports the protocol. SSL was originally developed by Netscape and is now widely used across the Web.

HTTP State Management Using Cookies

Even though modern HTTP has a lot of capabilities and features, it is still, at its heart, a simple request/reply protocol. One of the unfortunate problems that results from this is that HTTP is entirely *stateless*. This means that each time a server receives a request from a client, it processes the request, sends a response, and then forgets about the request. The next request from the client is treated as independent of any previous ones.

NOTE *The persistent connection feature of HTTP/1.1 (described in Chapter 80) does not change the stateless nature of the protocol. Even though multiple requests and responses can be sent on a single Transmission Control Protocol (TCP) connection, they are still not treated as being related in any way.*

So why is HTTP being stateless a problem? Isn't this what we would expect of a protocol designed to allow a client to quickly and efficiently retrieve resources from a server? Well, this is, yet again, another place where HTTP's behavior was well

suited to its original intended uses but not to how the Web is used today. Sure, if all we want to do is to say, “Hey server, please give me that file over there,” then the server doesn’t need to care about whether or not it may have previously provided that client with any other files in the past. This is how HTTP was originally intended to be used.

Today, the Web is much more than a simple resource-retrieval protocol. If you go to an online store, you want to be able to select a number of items to put into a “shopping cart” and have the store’s server remember them. You might also want to participate in a discussion forum, which requires you to provide a user name and password in order to post a message. Ideally, the server should let you log in once, and then remember who you are so you can post many messages, without needing to enter your login information each and every time. (I have used forums where the latter is required—it gets old very quickly, believe me.)

For these and other interactive applications, the stateless nature of HTTP is a serious problem. The solution was the addition of a new technology called *state management*, which allows the state of a client session with a server to be maintained across a series of HTTP transactions. Initially developed by Netscape, this technique was later made a formal Internet standard in RFC 2109, later revised in RFC 2965, “HTTP State Management Mechanism.” This feature is actually not part of HTTP; it is an optional element, but one that has been implemented in most web browsers due to its usefulness.

The idea behind state management is very simple. When a server implements a function that requires state to be maintained across a set of transactions, it sends a small amount of data called a *cookie* to the web client. The cookie contains important information relevant to the particular web application, such as a customer name, items in a shopping cart, or a user name and password. The client stores the information in the cookie, and then uses it in subsequent requests to the server that set the cookie. The server can then update the cookie based on the information in the new request and send it back to the client. In this manner, state information can be maintained indefinitely, allowing the client and server to have a memory that persists over a period of time.

NOTE *Cookie may seem like an odd term, but it is used in a few contexts to refer to a small piece of significant data. Another example is found in the Boot Protocol (BOOTP) and Dynamic Host Configuration Protocol (DHCP) message format. Today, most knowledgeable web users would blink at you if you mentioned the “HTTP state management mechanism,” but they usually know what cookies are.*

Issues with Cookies

Cookies sound like a great idea, right? Cookies are absolutely essential for many of the applications that make the Web the powerhouse it is today. Online shopping and discussion forums are just two of the many interactive applications that benefit from cookies. Most of the time, cookies are used for these sorts of useful and benign purposes. Unfortunately, some people have turned cookies to the “dark side” by finding ways to abuse them. There can even be potential problems with cookies when there is no nefarious intent. For this reason, cookies are rather controversial.

Here are some of the issues with cookies:

Transmission of Sensitive Information Suppose you use an online banking system. You log in to the server, which then stores your user name and password (which controls access to your account) in a cookie. If the application is not implemented carefully, the message containing that cookie could be intercepted, giving someone access to your account. Even if it is not intercepted, someone knowledgeable who gained access to your computer could retrieve the information from the file where cookies are stored.

Undesirable Use of Cookies In theory, cookies should be a help to the user, not a hindrance. However, any server can set a cookie for any reason. In some cases, a server could set a cookie for the purpose of tracking the websites that a user visits, which some people consider a violation of their privacy. Since some web browsers do not inform the user when a cookie is being set, the user may not even be aware that this is happening.

Third-Party or Unintentional Cookies While most people think of cookies as being set in the context of a resource they specifically request, a cookie may be set by any server to which a request is sent, whether the user realizes it or not. Suppose you send a request to <http://www.myfavoritesite.com/index.htm>, and that page contains a reference to a tiny image that is on the server <http://www.bigbrotherishere.com>. The second site can set a cookie on your machine even though you never intended to visit it. This is called a *third-party cookie*.

KEY CONCEPT HTTP is an inherently *stateless* protocol, because a server treats each request from a client independently, forgetting about all prior requests. This characteristic of HTTP is not an issue for most routine uses of the Web, but is a problem for interactive applications such as online shopping where the server needs to keep track of a user's information over time. To support these applications, most HTTP implementations include an optional feature called *state management*. When enabled, a server sends to a client a small amount of information called a *cookie*, which is stored on the client machine. The data in the cookie is returned to the server with each subsequent request, allowing the server to update it and send it back to the client again. Cookies thus enable servers to remember user data between requests. However, they are controversial, because of certain potential privacy and security concerns related to their use.

Managing Cookie Use

The RFCs describing the cookie state management technique deal extensively with these and other issues, but there is no clear-cut resolution to these concerns. Like most security and privacy matters, the most important determinant of how significant potential cookie abuse may be is your own personal comfort level. Millions of people browse the Web every day letting any and all sites set whatever cookies they want and never have a problem. Others consider cookies an offensive idea and disable all cookies, which eliminates the privacy concerns but can cause problems with useful applications like interactive websites. As usual, the best approach is usually something in the middle, where you choose when and how you will allow cookies to be set.

The degree to which cookie control is possible depends greatly on the quality and feature set of your web client software. Many browsers do not provide a great deal of control in how and when cookies are set; others are much better in this regard. Some browsers allow cookies to be disabled, but come with them turned on by default. Since many people are not even aware of the issues associated with cookies, they do not realize when cookies are being sent. Most notable in this regard is the popular Microsoft Internet Explorer, which normally comes set by default to accept all cookies without complaint or even comment.

Internet Explorer does allow you to disable cookies, but you must do it yourself. It also allows you to differentiate between first-party and third-party cookies, but again, you must turn on this feature. Other browsers have more sophisticated settings, which will let you dictate conditions under which cookies may be set and others when they may not. Some browser will even let you allow certain websites to send cookies while prohibiting them from others. Better browsers will also let you visually inspect cookies, and selectively clear the ones you do not want on your machine.

Third-party cookies can be used by online advertising companies and others to track the sites that a Web user visits. For this reason, they are considered by many people to fall into the general category of undesirable software called *spyware*. There are numerous tools that will allow you to identify and remove tracking cookies from your computer; many are available free on the Web.

PART III-9

OTHER FILE AND MESSAGE TRANSFER APPLICATIONS

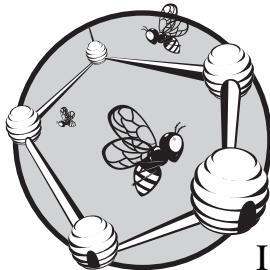
The previous three parts of this book have examined several of the most widely used TCP/IP file and message transfer protocols: the File Transfer Protocol (FTP), the Trivial File Transfer Protocol (TFTP), electronic mail (email), and the World Wide Web.

Of course, hundreds of other applications are in use on the Internet today, and we couldn't possibly examine them all here. However, there are a couple of other protocols that are considered part of the group of classic applications of TCP/IP like FTP, email, and the Web that I feel are worth discussing.

This part contains two chapters that cover these other file and message transfer applications. The first chapter describes Usenet (network news), which is one of the original methods of group communication on the Internet. The second chapter describes the Gopher protocol, which while no longer widely used today is worth a brief discussion, especially due to its role as a historical precursor of the Web.

85

USENET (NETWORK NEWS) AND THE TCP/IP NETWORK NEWS TRANSFER PROTOCOL (NNTP)



Electronic mail (email) is one of the stalwarts of message transfer on the modern Internet, but it is really designed only for communication within a relatively small group of specific users. There are many situations in which email is not ideally suited, such as when information needs to be shared among a large number of participants, not all of whom may necessarily even know each other. One classic example of this is sharing *news*. In this case, the person providing news often wants to make it generally available to anyone who is interested, rather than specifying a particular set of recipients.

For distributing news and other types of general information over internetworks, a messaging system called both *Usenet* and *network news* was created. Like email, this application allows messages to be written and read by large numbers of users. However, it is designed using a very different model than email—one that is focused on public sharing and feedback. In Usenet, anyone can write a message that can be read by any number of recipients, and anyone can respond to messages written by others. Usenet

was one of the first widely deployed internetwork-based group communication applications, and it has grown into one of the largest online communities in the world, used by millions of people for sharing information, asking questions, and discussing thousands of different topics.

In this chapter, I describe Usenet and network news in detail, discussing how they are used and how they work. I provide an overview and history of Usenet, a high-level look at its model of communication and how messages are created and manipulated, an explanation of Usenet newsgroups, and a description of the Usenet message format and headers. Then I provide a detailed description of the operation of the Network News Transfer Protocol (NNTP), the means used for transferring messages on modern Usenet. Starting as usual with an overview of the protocol, I then explain the two fundamental ways that NNTP is used: for the propagation of news articles between servers and for client article posting and access. From there, I move on to the technical details of NNTP commands, command extensions, responses, and response codes.

BACKGROUND INFORMATION Several aspects of how Usenet works are closely related to the standards and techniques used for email. If you have not read Part III-7, which covers email, I suggest that you at least review the overview of the email system in Chapter 74 and the discussion of the email message format in Chapter 76, since Usenet messages are based on the RFC 822 email message standard.

Usenet Overview, History, and Operation

Where email is the modern equivalent of the handwritten letter or the interoffice memo, *Usenet* is the updated version of the company newsletter, the cafeteria bulletin board, the coffee break chat, and the watercooler gossip session, all rolled into one. Spread worldwide over the Internet, Usenet newsgroup messages provide a means for people with common interests to form online communities to discuss happenings, solve problems, and provide support to each other, as well as to engage in plain old socializing and entertainment.

We are by nature both highly social and creative animals, and as a result, we are always finding new ways to communicate. It did not take long after computers were first connected together for it to be recognized that those interconnections provided the means to link together people as well. The desire to use computers to create an *online community* led to the creation of Usenet more than two decades ago.

History of Usenet

Like almost everything associated with networking, Usenet had very humble beginnings. In 1979, Tom Truscott was a student at Duke University in North Carolina, and he spent the summer as an intern at Bell Laboratories, the place where the UNIX operating system was born. He enjoyed the experience so much that when he returned to school that autumn, he missed the intensive UNIX environment at Bell Labs. He used the *Unix-to-Unix Copy Protocol (UUCP)* to send information from his local machine to other machines and vice versa, including establishing electronic connectivity back to Bell Labs.

Building on this idea, Truscott and a fellow Duke student, Jim Ellis, teamed up with other UNIX enthusiasts at Duke and the nearby University of North Carolina (UNC) at Chapel Hill, to develop the idea of an online community. The goal was to create a system where students could use UNIX to write and read messages, to allow them to obtain both technical help and maintain social contacts. They designed the system based on an analogy to an online newsletter that was open to all users of a connected system. To share information, messages were posted to *newsgroups*, where any user could access the messages to read them and respond to them.

The early work at Duke and UNC resulted in the development of both the initial message format and the software for the earliest versions of this system, which became known both as *network news (net news)* and *Usenet* (a contraction of *User's network*). At first, the system had just two computers, sharing messages posted in a pair of different newsgroups. The value of the system was immediately recognized, however, and soon many new sites were added to the system. These sites were arranged in a structure to allow messages to be efficiently passed using direct UUCP connections. The software used for passing news articles also continued to evolve and become more capable, as did the software for reading and writing articles.

The newsgroups themselves also changed over time. Many new newsgroups were created, and a hierarchical structure was defined to help keep the newsgroups organized in a meaningful way. As more sites and users joined Usenet, more areas of interest were identified. Today, there are a staggering number of Usenet newsgroups: more than 100,000. While many of these groups are not used, many thousands of active ones discuss nearly every topic imaginable—from space exploration, to cooking, to biochemistry, to PC troubleshooting, to raising horses. There are also regional newsgroups devoted to particular areas; for example, there is a set of newsgroups for discussing events in Canada and another for discussing happenings in the New York area, and so on.

Usenet Operation and Characteristics

Usenet begins with a user writing a message to be distributed. After the message is *posted* to say, the group on TCP/IP networking, it is stored on that user's local news server, and special software sends copies of it to other connected news servers. The message eventually propagates around the world, where anyone who chooses to read the TCP/IP networking newsgroup can see the message.

The real power of Usenet is that after reading a message, any user can respond to it on the same newsgroup. Like the original message, the reply will propagate to each connected system, including the one used by the author of the original message. This makes Usenet very useful for sharing information about recent happenings, for social discussions, and especially for receiving assistance with problems, such as resolving technical glitches or getting help with a diet program.

What is particularly interesting about Usenet is that it is not a formalized system in any way, and it is not based on any formally defined standards. It is a classic example of the development of a system in an entirely ad hoc manner: The software was created, people started using it, the software was refined, and things just

took off from there. Certain standards have been written to codify how Usenet works—such as RFC 1036, which describes the Usenet message format—but these serve more as historical documents than as prescriptive standards.

There is likewise no central authority that is responsible for Usenet's operation, even though new users often think there is one. Unlike a dial-up bulletin board system or Web-based forum, Usenet works simply by virtue of cooperation between sites; there is no manager in charge. For this reason, Usenet is sometimes called an anarchy, but this is not accurate. It isn't the case that there are no rules. It is up to the managers of participating systems to make policy decisions such as which newsgroups to support. There are also certain dictatorial aspects of the system, in that only certain people (usually system administrators) can decide whether to create some kinds of new newsgroups. The system also has socialistic elements in that machine owners are expected to share messages with each other. So, the simplified political labels really don't apply to Usenet.

Every community has a *culture*, and the same is true of online communities, including Usenet. There is an overall culture that prescribes acceptable behavior on Usenet, and also thousands of newsgroup-specific cultures in Usenet, each of which has evolved through the writings of thousands of participants over the years. There are even newsgroups devoted to explaining how Usenet itself operates, where you can learn about *newbies* (new users), *netiquette* (rules of etiquette for posting messages), and related subjects.

Usenet Transport Methods

As I said earlier, Usenet messages were originally transported using UUCP, which was created to let UNIX systems communicate directly, usually using telephone lines. For many years, all Usenet messages were simply sent from machine to machine using computerized telephone calls (just as email once was). Each computer joining the network would connect to one already on Usenet and receive a *feed* of messages from it periodically. The owner of that computer had to agree to provide messages to other computers.

Once TCP/IP was developed in the 1980s and the Internet grew to a substantial size and scope, it made sense to start using it to carry Usenet messages rather than UUCP. The *Network News Transfer Protocol (NNTP)* was developed specifically to describe the mechanism for communicating Usenet messages over the Transmission Control Protocol (TCP). It was formally defined in RFC 977, published in 1986, with NNTP extensions described in RFC 2980, published in October 2000.

For many years, Usenet was carried using both NNTP and UUCP, but NNTP is now the mechanism used for the vast majority of Usenet traffic, and for this reason is the primary focus of my Usenet discussion. NNTP is employed not only to distribute Usenet articles to various servers, but also for other client actions, such as posting and reading messages. It is thus used for most of the steps in Usenet message communication.

NOTE Many people often equate the Usenet system as a whole with the NNTP protocol that is used to carry Usenet messages on the Internet. They are not the same however; Usenet predates NNTP, which is simply a protocol for conveying Usenet messages.

It is because of the critical role of NNTP and the Internet in carrying messages in today's Usenet that the concepts are often confused. It's essential to remember, however, that Usenet does not refer to any type of physical network or internetworking technology; rather, it is a logical network of users. That logical network has evolved from UUCP data transfers to NNTP and TCP/IP, but Usenet itself is the same.

Today, Usenet faces competition from many other group messaging applications and protocols, including Web-based bulletin board systems and chat rooms. After a quarter of a century, however, Usenet has established itself and is used by millions of people every day. While to some, the primarily text-based medium seems archaic, it is a mainstay of global group communication and likely to continue to be so for many years to come.

KEY CONCEPT One of the very first online electronic communities was set up in 1979 by university students who wanted to keep in touch and share news and other information. Today, *Usenet* (for *User's network*), also called *network news*, has grown into a logical network that spans the globe. By posting messages to a Usenet newsgroup, people can share information on a variety of subjects of interest. Usenet was originally implemented in the form of direct connections established between participating hosts. Today, the Internet is the vehicle for message transport.

Usenet Communication Model

When the students at Duke University decided to create their online community, email was already in wide use, and there were many mailing lists in operation as well. Email was usually transported using UUCP—the same method that Usenet was designed to employ—during these pre-Internet days. Then why not simply use email to communicate between sites?

The main reason is that email is not designed to facilitate the creation of an online community where information can be easily shared in a group. The main issue with email in this respect is that only the individuals who are specified as recipients of a message can read it. There is no facility whereby someone can write a message and put it in an open place where anyone who wants to can read it, analogous to posting a newsletter in a public place.

Another problem with email in large groups is related to efficiency. Consider that if you put 1,000 people on a mailing list, each message sent to that list must be duplicated and delivered 1,000 times. Early networks were limited in bandwidth and resources, so using email for wide-scale group communication was possible, but far from ideal.

KEY CONCEPT While email can be used for group communications, it has two important limitations. First, a message must be specifically addressed to each recipient, making public messaging impossible. Second, each recipient requires delivery of a separate copy of the message, so sending a message to many recipients requires the use of a large number of resources.

Usenet's Public Distribution Orientation

To avoid the problems of using email for group messaging, Usenet was designed using a rather different communication and message-handling model than email. The defining difference between the Usenet communication model and that used for email is that Usenet message handling is oriented around the concept of *public distribution*, rather than private delivery to an individual user. This affects every aspect of how Usenet communication works, as follows:

Addressing Messages are not addressed from a sender to any particular recipient or set of recipients, but rather to a *group*, which is identified with a newsgroup name.

Storage Messages are not stored in individual mailboxes, but rather in a central location on a server, where any user of the server can access them.

Delivery Messages are not conveyed from the sender's system to the recipient's system, but rather are spread over the Internet to all connected systems so anyone can read them.

Usenet Communication Process

To help illustrate in more detail how Usenet communication works, let's take a look at the steps involved in writing, transmitting, and reading a typical Usenet message (also called an *article*—the terms are used interchangeably). Let's suppose the process begins with a user, Ellen, posting a request for help with a sick horse to the newsgroup misc.rural. Since she is posting the message, she would be known as the message *poster*. Simplified, the steps in the process (illustrated in Figure 85-1) are as follows:

1. **Article Composition** Ellen begins by creating a Usenet article, which is structured according to the special message format required by Usenet. This message is similar to an email message in that it has a *header* and a *body*. The body contains the actual message to be sent, while the header contains header lines that describe the message and control how it is delivered. For example, one important header line specifies for which newsgroup(s) the article is intended.
2. **Article Posting and Local Storage** After completing her article, Ellen submits the article to Usenet, a process called *posting*. A client software program on Ellen's computer transmits Ellen's message to her local Usenet server. The message is stored in an appropriate file storage area on that server. It is now immediately available to all other users of that server who decide to read misc.rural.
3. **Article Propagation** At this point, Ellen's local server is the only one that has a copy of her message. The article must be sent to other sites, a process called *distribution*, or more commonly, *propagation*. Ellen's message travels from her local Usenet server to other servers to which her server directly connects. It then propagates from those servers to others they connect to, and so on, until all Usenet servers that want it have a copy of the message.

4. **Article Access and Retrieval** Since Usenet articles are stored on central servers, in order to read them, they must be accessed on the server. This is done using a Usenet *newsreader* program. For example, some other reader of *misc.rural* named Jane might access that group and find Ellen's message. If Jane were able to help Ellen, she could reply to Ellen by posting an article of her own. This would then propagate back to Ellen's server, where she could read it and reply. All other readers of *misc.rural* could jump into the conversation at any time as well, which is what makes Usenet so useful for group communication.

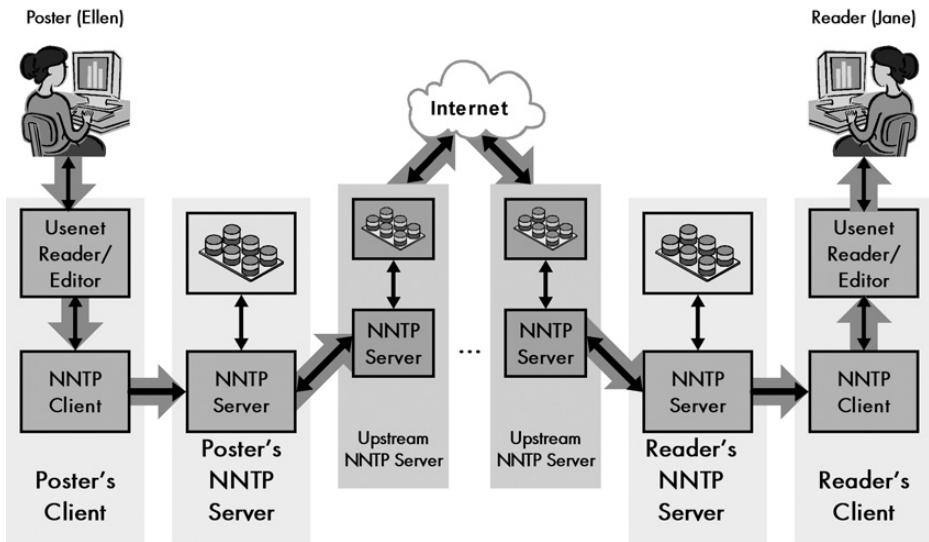


Figure 85-1: Usenet (network news) communication model This figure illustrates the method by which messages are created, propagated, and read using NNTP on modern Usenet; it is similar in some respects to the email model diagram (Figure 74-1 in Chapter 74). In this example, a message is created by the poster, Ellen, and read by a reader, Jane. The process begins with Ellen creating a message in an editor and posting it. Her NNTP client sends it to her local NNTP server. It is then propagated from that local server to adjacent servers, usually including its upstream server, which is used to send the message around the Internet. Other NNTP servers receive the message, including the one upstream from Jane's local server. It passes the message to Jane's local server, and Jane accesses and reads the message using an NNTP client. Jane could respond to the message; in which case, the same process would repeat, but going in the opposite direction, back to Ellen (and also back to thousands of other readers, not shown here).

KEY CONCEPT Usenet communication consists of four basic steps. A message is first composed and then posted to the originator's local server. The third step is propagation, where the message is transmitted from its original server to others on the Usenet system. The last step in the process is article retrieval, where other members of the newsgroup access and read the article. The Network News Transfer Protocol (NNTP) is the technology used for moving Usenet articles from one host to the next.

Message Propagation and Server Organization

Propagation is definitely the most complex part of the Usenet communication process. In the past, UUCP was used for propagation. Each Usenet server would be programmed to regularly dial up another server and give it all new articles it had received since the last connection. Articles would *flood* across Usenet from one server to another. This was time-consuming and inefficient, and it worked only because the volume of articles was relatively small.

As I noted in the previous section, in modern Usenet, NNTP is used for all stages of transporting messages between devices. Articles are posted using an NNTP connection between a client machine and a local server, which then uses the same protocol to propagate the articles to other adjacent NNTP servers. The client news-reader software also uses NNTP to retrieve messages from a server.

NNTP servers are usually arranged in a hierarchy of sorts, with the largest and fastest servers providing service to smaller servers downstream from them. Depending on how the connections are arranged, an NNTP server may establish a connection to immediately send a newly posted article to an upstream server for distribution to the rest of Usenet, or the server may passively wait for a connection from the upstream server to ask if there are any new articles to be sent. With the speed of the modern Internet, it typically takes only a few minutes (or seconds) for articles to propagate from one server to another, even across continents.

It is also possible to restrict the propagation of a Usenet message, a technique often used for discussions that are of relevance only in certain regions or on certain systems. Discussing rural issues such as horses is of general interest, and Ellen might find help anywhere around the world, so global propagation of her message makes sense. However, if Ellen lived in the Boston area and was interested in knowing the location of a good local restaurant, posting a query to *ne.food* (New England food discussions) with only local distribution would make more sense. There are also companies that use Usenet to provide “in-house” newsgroups that are not propagated off the local server at all. However, because so many news providers are now national or international, limiting the distribution of messages has largely fallen out of practice.

Usenet Addressing: Newsgroups

A key concept in Usenet communication is the *newsgroup*. Newsgroups are the addressing mechanism for Usenet, and sending a Usenet article to a newsgroup is equivalent to sending email to an email address. Newsgroups are analogous to other group communication venues such as mailing lists, chat rooms, Internet Relay Chat (IRC) channels, or bulletin board system (BBS) forums (though calling a newsgroup a *list*, *room*, *channel*, or *BBS* is likely to elicit a negative reaction from Usenet old-timers!).

Like any addressing mechanism, newsgroups must be uniquely identifiable. Each newsgroup has a *newsgroup name* that describes the topic of the newsgroup and differentiates it from other newsgroups. Since there are many thousands of different newsgroups, they are arranged into sets called *hierarchies*. Each hierarchy contains a tree structure of related newsgroups.

The Usenet Newsgroup Hierarchies

The total collection of newsgroup hierarchies is in many ways similar to the domain name tree structure used in the Domain Name System (DNS). Each Usenet hierarchy is like a collection of all the domain names within a DNS top-level domain. Just as a domain name like `www.pcguide.com` is formed by appending the label of the top-level domain `.com` to the second-level domain name `pcguide` and the subdomain `www`, newsgroup names are created in the same way. They are created from a top-level newsgroup hierarchy name, to which are attached a set of descriptive labels that describes the newsgroup's place in the hierarchy.

One difference between DNS and Usenet hierarchies is that while DNS names are created from right to left as you go down the tree, Usenet newsgroup names are formed in the more natural (for English speakers) left-to-right order. For example, one of the main Usenet hierarchies is the `comp` hierarchy, devoted to computer topics. Within `comp` is a subhierarchy on data communications called `dcom`, and within that is a group that discusses data cabling. This group is called `comp.dcom.cabling`. Almost all newsgroups are structured in this manner.

The “Big Eight” Newsgroup Hierarchies

One problem with the decentralized nature of Usenet is ensuring coordination in certain areas where we want everyone to be on the same page, and one of these is newsgroup naming. If we let just anyone create a newsgroup, we might end up with many groups that all discuss the same topic. Imagine that someone had a question on data cabling and didn't realize that `comp.dcom.cabling` existed, so he created a new group called `comp.datacomm.cabling`. The two groups could coexist, but this would lead to both confusion and fragmenting of the pool of people interested in this topic.

To avoid problems with newsgroup creation, administrators of large Usenet systems collaborated on a system for organizing many of the more commonly used Usenet groups into eight hierarchies, and devised a specific procedure for creating new newsgroups within them. Today, these are called the *Big Eight* Usenet hierarchies, which are summarized in Table 85-1.

Table 85-1: Usenet Big Eight Newsgroup Hierarchies

Hierarchy	Description
comp.*	Newsgroups discussing computer-related topics, including hardware, software, operating systems, and techniques
humanities.*	Newsgroups discussing the humanities, such as literature and art
misc.*	Newsgroups discussing miscellaneous topics that don't fit into other Big Eight hierarchies
news.*	Newsgroups discussing Usenet itself and its administration
rec.*	Newsgroups discussing recreation topics, such as games, sports, and activities
sci.*	Science newsgroups, covering specific areas such as physics and chemistry, research topics, and so forth
soc.*	Society and social discussions, including groups on specific cultures
talk.*	Newsgroups primarily oriented around discussion and debate of current events and happenings

These eight hierarchies contain many of the most widely used groups on Usenet today. For example, professional baseball is discussed in *rec.sport.baseball*, Intel computers in *comp.sys.intel*, and Middle East politics in *talk.politics.mideast*.

The Big Eight hierarchies are rather tightly controlled in terms of their structure and the newsgroups they contain. The process to create a new Big Eight newsgroup is democratic and open. Anyone can propose a new group, and if there is enough support, it will be created by the cooperating system administrators who agree to follow the Big Eight system. However, this creation process is rather complex and time-consuming. Some people find this unacceptable and even object to the entire concept of this restricted process. Others consider the system advantageous, as it keeps the Big Eight hierarchies relatively orderly by slowing the rate of change to existing newsgroups and the number of new groups added.

Alt and Other Newsgroup Hierarchies

For those who prefer a more freewheeling environment and do not want to submit to the Big Eight procedures, there is an alternative Usenet hierarchy, which begins with the hierarchy name *alt*. This hierarchy includes many thousands of groups. Some are quite popular, but many are not used at all; this is a side effect of the relative ease with which an alt group can be created.

In addition to these nine hierarchies, there are dozens of additional, smaller hierarchies. Many of these are regional or even company-specific. For example, the *ne*. hierarchy contains a set of newsgroups discussing issues of relevance to New England; *fr.** covers France, and *de.** pertains to Germany. Microsoft has its own set of public newsgroups in the *microsoft.** hierarchy. Figure 85-2 shows the Big Eight hierarchies and some of the other hierarchies that exist.

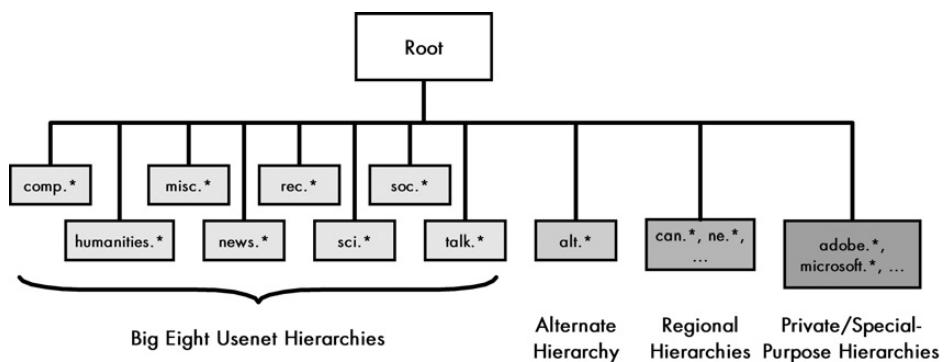


Figure 85-2: Usenet newsgroup hierarchies Usenet newsgroups are arranged into tree-like structures called hierarchies. Eight of these are centralized, widely used, general-purpose hierarchies, which are today called the Big Eight. The alternate (*alt*) hierarchy is a very loosely structured set of thousands of groups covering every topic imaginable. In addition to these, there are many hundreds of regional, private, and special-purpose hierarchies.

KEY CONCEPT Usenet messages are not addressed to individual users; rather, they are posted to newsgroups. Each newsgroup represents a topic. Those with an interest in the subject of a group can read messages in it and reply to them as well. Usenet newsgroups are arranged into tree-like hierarchies that are similar in structure to DNS domains. Many of the most widely used newsgroups are found in a collection of general-interest hierarchies called the Big Eight. An alternate (alt) hierarchy offers an alternative to the Big Eight. There are also many regional and special-purpose hierarchies.

Unmoderated and Moderated Newsgroups

Most newsgroups are open to all to use and are called *unmoderated* because a message sent to them goes directly out to the whole Usenet server internetwork. In contrast, a small percentage of newsgroups is *moderated*, which means that all messages sent to the group are screened and only the ones that are approved by a moderator (or moderator team) are really posted.

The purpose of moderated groups is to ensure that discussions in a particular group remain on-topic. They are often created to handle topics that are controversial, to ensure that debates remain constructive and disruption is avoided. For example, rec.guns is moderated to ensure that discussions focus on the use of guns and not on endless political arguments related to gun control and the like (which has a place, in talk.politics.guns). Moderated groups are also sometimes used for specialty groups intended only for announcements, or for groups where the content is restricted. For example, rec.food.recipes is moderated so that it contains only recipes and recipe requests, which helps people find recipes easily without needing to wade through a lot of discussion. Finally, moderated versions of unmoderated groups are sometimes created when a few disruptive elements choose to post large volumes in the unmoderated groups, making normal discussion difficult.

Cross-Posting to Multiple Newsgroups

It is possible for a single article to be posted to multiple newsgroups. This process, called *cross-posting*, is used when a message pertains to two topics, or to allow a sender to reach a wider audience. For example, if you live in the Seattle area and have a problem with your house, you might legitimately cross-post to seattle.general and misc.consumers.house.

Cross-posting is more efficient than posting the same message to each group independently for two reasons:

- Only one copy of the message will be stored on each Usenet server rather than two.
- Usenet participants who happen to read both groups won't see the message twice.

However, cross-posting to very large numbers of newsgroups is usually considered disruptive and a breach of Usenet etiquette.

Usenet Message Format and Special Headers

Usenet is designed to permit users to exchange information in the form of messages that are sent from one computer to another. As is necessary with any message-based networking application, all Usenet client software and server software agree to use a common *message format*. This ensures that all devices and programs are able to interpret all Usenet articles in a consistent manner.

While Usenet was created as an alternative to email, and there are obviously differences in how each treats messages, there are also many similarities. Both are text-oriented messaging systems with similar needs for communicating content and control information. The creators of Usenet realized that there would be many advantages to basing the Usenet message format on the one used for email, rather than creating a new format from scratch. The email message format was already widely used, and adopting it for Usenet would save implementation time and effort. It would also enhance compatibility between email and Usenet messages, allowing software designed to process or display email to also work with Usenet articles. For this reason, the Usenet message format was defined based on the RFC 822 standard for email messages (introduced in Chapter 76).

RFC 822 messages begin with a set of *headers* that contain control and descriptive information about the message, followed by a blank line and then the message *body*, which contains the actual content.

One important attribute of the RFC 822 standard is the ability to define custom headers that add to the regular set of headers defined in the standard itself. Usenet articles require some types of information not needed by email, and these can be included in specially defined headers while still adhering to the basic RFC 822 format. At the same time, headers specific to email that are not needed for Usenet can be omitted. Thus, there is no structural difference at all between a Usenet article and an email message. They differ only in the kinds of headers they contain and the values for those headers. For example, a Usenet message will always contain a header specifying the newsgroup(s) to which the article is being posted, but will not carry a “To:” line as an email message would.

KEY CONCEPT Usenet articles use the same RFC 822 message format as email messages. The only difference between a Usenet article and an email message is in the header types and values used in each.

Usenet Header Categories and Common Headers

All Usenet headers are defined according to the standard header format specified in RFC 822: <header name>: <header value>. As with email messages, headers may extend onto multiple lines, following the indenting procedure described in the RFC 822 standard.

The current standard for Usenet messages, RFC 1036, describes the header types for Usenet messages. The headers are divided into two categories: *mandatory* headers (see Table 85-2) and *optional* headers (see Table 85-3). Some are the same as headers of the equivalent name used for email, some are similar to email headers but used in a slightly different way, and others are unique to Usenet.

Table 85-2: Usenet Mandatory Headers

Header Name	Description
From:	The email address of the user sending the message, as for email.
Date:	The date and time that the message was originally posted to Usenet. This is usually the date/time that the user submitted the article to his or her local NNTP server.
Newsgroups:	Indicates the newsgroup or set of newsgroups to which the message is being posted. Multiple newsgroups are specified by separating them with a comma; for example: Newsgroups: news.onegroup,rec.secondgroup.
Subject:	Describes the subject or topic of the message. Note that this header is mandatory on Usenet despite being optional for email; it is important because it is used by readers to decide what messages to open.
Message-ID:	Provides a unique code for identifying a message; normally generated when a message is sent. The message ID is very important in Usenet, arguably more so than in email. The reason is that delivery of email is performed based on recipient email addresses, while the propagation of Usenet messages is controlled using the message ID header.
Path:	An informational field that shows the path of servers that a particular copy of a message followed to get to the server where it is being read. Each time a server forwards a Usenet article, it adds its own name to the list in the Path header. The entries are usually separated by exclamation points. For example, if a user on Usenet Server A posts a message, and it is transported from Server A to Server G, then Server X, then Server F, and finally to Server Q, where a second user reads it, the person on Server Q would see something like this in the Path header: "Q!FIX!GIA."

Table 85-3: Usenet Optional Headers

Header Name	Description
Reply-To:	It is possible to reply back to a Usenet article author using email, which by default, goes to the address in the From: line. If this header is present, the address it contains is used instead of the default From: address.
Sender:	Indicates the email address of the user who is sending the message, if different from the message originator. This is functionally the same as the Sender: header in email messages, but is used in a slightly different way. Normally, when a Usenet message is posted, the sender's email address is automatically filled in to the From: line. If the user manually specifies a different From: line, the address from which the message was actually sent is usually included in the Sender: line. This is used to track the true originating point of articles.
Followup-To:	A reply to a Usenet message is usually made back to Usenet itself and is called a <i>follow-up</i> . By default, a follow-up goes to the newsgroup(s) specified in the original message's Newsgroups: header. However, if the Followup-To: header is included, follow-ups to that message go to the newsgroups specified in the Followup-To: header instead. This header is sometimes used to route replies to a message to a particular group. Note, however, that when a user replies to a message, this field controls only what appears in the new message's Newsgroups: line by default. The user can override the Newsgroups: header manually.
Expires:	All Usenet messages are maintained on each server for only a certain period of time, due to storage limitations. The expiration interval for each newsgroup is controlled by the administrator of each site. If present, this line requests a different expiration for a particular message; it is usually used only for special articles. For example, if a weekly announcement is posted every Monday morning, each article might be set to expire the following Monday morning, to make sure that people see the most current version.
References:	Lists the message IDs of prior messages in a conversation. For example, if someone posts a question to a newsgroup with message ID AA207, and a reply to that message is made, the software will automatically insert the line "References: AA207" into the reply. This is used by software to group together articles into conversations (called <i>threads</i>) to make it easier to follow discussions on busy newsgroups.

(continued)

Table 85-3: Usenet Optional Headers (continued)

Header Name	Description
Control:	Indicates that the article is a control message and specifies a control action to be performed, such as creating a new newsgroup.
Distribution:	By default, most messages are propagated on Usenet worldwide. If specified, this line restricts the distribution of a message to a smaller area, either geographical or organizational.
Organization:	Describes the organization to which the article sender belongs. Often filled in automatically with the name of the user's Internet service provider (ISP).
Keywords:	Contains a list of comma-separated keywords that may be of use to the readers of the message. Keywords can be useful when searching for messages on a particular subject matter. This header is not often used.
Summary:	A short summary of the message. This is rarely used in practice.
Approved:	Added by the moderator of a moderated newsgroup to tell the Usenet software that the message has been approved for posting.
Lines:	A count of the number of lines in the message.
Xref:	While Usenet articles are identified by message ID, they are also given a number by each Usenet server as they are received. These article numbers, which differ from one system to the next, are usually listed in this cross-reference header. This information is used when a message is cross-posted to multiple groups. In that case, as soon as a user reads the message in one group, it is marked as having been read in all the others where it was posted. This way, if the user later reads one of those other groups, that user will not see the message again.

Additional Usenet Headers

Usenet messages may also contain additional headers, just as is the case with email messages. Some of these are custom headers included by individual users to provide extra information about an article. Others are used in many current Usenet articles and have become almost de facto standard headers through common use. Many of these custom headers are preceded by X, indicating that they are experimental or extra headers.

Some of the more frequently encountered additional Usenet headers are shown in Table 85-4.

Table 85-4: Common Additional Usenet Headers

Header Name	Description
NNTP-Posting-Host:	Specifies the IP address or the DNS domain name of the host used to originally post the message. This is usually either the address of the client that the author used for posting the message or the sender's local NNTP server.
User-Agent: (or) X-Newsreader:	The name and version number of the software used to post the message.
X-Trace:	Provides additional information that can be used to trace the message.
X-Complaints-To:	An email address to use to report abusive messages. This header is now included automatically by many ISPs.

Usenet MIME Messages

Since Usenet follows the RFC 822 standard, Multipurpose Internet Mail Extensions (MIME) can be used to format Usenet messages. When this is done, you will see the usual MIME headers (such as MIME-Version, Content-Type, and so forth) in the message.

Note that the use of MIME in Usenet messages is somewhat controversial. Some newsreaders are not MIME-compliant and make a mess when trying to display some of these messages, and many Usenet veterans object to the use of anything but plain text in Usenet messages. Despite this, MIME messages are becoming more common, for better or worse.

NNTP Overview and General Operation

As I explained earlier in this chapter, Usenet started out as an informal network of UNIX computers using dial-up UUCP connections to transmit messages between servers. This arrangement arose out of necessity, and it worked fairly well, though it had a number of problems. Once the Internet became widely used in the 1980s, it provided the ideal opportunity for a more efficient means of distributing Usenet articles. NNTP was developed as a special TCP/IP protocol for sending these messages. Now NNTP carries billions of copies of Usenet messages from computer to computer every day.

BACKGROUND INFORMATION *NNTP is similar to the Simple Mail Transfer Protocol (SMTP) in many ways, including its basic operation and command set and reply format. You may find the information about NNTP easier to understand if you are familiar with SMTP, covered in Chapter 77.*

Usenet began as a *logical* internetwork of cooperating hosts that contacted each other directly. In the early Usenet, a user would post a message to her local server, where it would stay until that server either contacted or was contacted by another server. The message would then be transferred to the new server, where it would stay until the second server contacted a third one, and so on. This transport mechanism was functional, but seriously flawed in a number of ways.

Servers were not continually connected to each other; they could communicate only by making a telephone call using an analog modem. Thus, messages would often sit for hours before they could be propagated. Modems in those days were also very slow by today's standards—2400 bits per second or even less—so it took a long time to copy a message from one server to another. Worst of all, unless two sites were in the same city, these phone calls were long distance, making them quite expensive.

Why was this system used despite all of these problems? The answer is simply because there was no alternative. In the late 1970s and early 1980s, there was no Internet as we know it, and no other physical infrastructure existed to link Usenet sites together. It was either use UUCP over telephone lines or nothing.

That all changed as the fledgling ARPAnet grew into the modern Internet. As the Internet expanded, more and more sites connected to it, including many sites that were participating in Usenet. Once both sites in an exchange were on the Internet, it was an easy decision to use the Internet to send Usenet articles, rather

than relying on slow, expensive phone calls. Over time, more and more Usenet sites joined the Internet, and it became clear that just as email had moved from UUCP to the TCP/IP Internet, the future of Usenet was on the Internet as well.

The shifting of Usenet from UUCP connections to TCP/IP internetworking meant that some rethinking was required as to how Usenet articles were moved from server to server. On the Internet, Usenet was just one of many applications, and the transfer of messages had to be structured using TCP or the User Datagram Protocol (UDP). Thus, like other applications, Usenet required an application-level protocol to describe how to carry Usenet traffic over TCP/IP. Just as Usenet had borrowed its message format from email's RFC 822, it made sense to model its message delivery protocol on the one used by email: SMTP. The result was the creation of NNTP, published as RFC 977 in February 1986.

The general operation of NNTP is indeed very similar to that of SMTP. NNTP uses TCP, with servers listening on well-known TCP port 119 for incoming connections, either from client hosts or other NNTP servers. As in SMTP, when two servers communicate using NNTP, the one that initiates the connection plays the role of client for that exchange.

After a connection is established, communication takes the form of commands sent by the client to the server and replies returned from the server to the client device. NNTP commands are sent as plain ASCII text, just like those used by SMTP, the File Transfer Protocol (FTP), the Hypertext Transfer Protocol (HTTP), and other protocols. NNTP responses take the form of three-digit reply codes as well as descriptive text, again just like SMTP (which, in turn, borrowed this concept from FTP).

NNTP was designed to be a comprehensive vehicle for transporting Usenet messages. It is most often considered as a delivery protocol for moving Usenet articles from one server to another, but it is also used for connections from client hosts to Usenet servers for posting and reading messages. Thus, the NNTP command set is quite extensive and includes commands to handle communications between servers and between clients and servers. For message propagation, a set of commands allows a server to request new articles from another server or to send new articles to another server. For message posting and access, commands allow a client to request lists of new newsgroups and messages, and to retrieve messages for display to a user.

The commands defined in RFC 977 were the only official ones for over a decade. However, even as early as the late 1980s, implementers of NNTP server and client software were adding new commands and features to make NNTP both more efficient and useful to users. These *NNTP extensions* were eventually documented in RFC 2980, published in 2000. I describe them in more detail later in this chapter, in the “NNTP Commands and Command Extensions” section.

KEY CONCEPT The Network News Transfer Protocol (NNTP) is the protocol used to implement message communication in modern Usenet. It is used for two primary purposes: to propagate messages between NNTP servers and to permit NNTP clients to post and read articles. It is a stand-alone protocol, but shares many characteristics with email's Simple Mail Transfer Protocol (SMTP).

NNTP is used for all of the transfer steps in the modern Usenet communication process. However, NNTP is most often associated with the process of Usenet article *propagation*. This is arguably the most important function of NNTP: providing an efficient means of moving large volumes of Usenet articles from one server to another. It is thus a sensible place to start looking at the protocol.

NNTP Interserver Communication Process: News Article Propagation

To understand how NNTP propagation works, we must begin with a look at the way that the modern Usenet network itself is organized. Usenet sites are now all on the Internet, and theoretically, any NNTP server can contact any other to send and receive Usenet articles. However, it would be ridiculous to have a new article submitted to a particular server need to be sent via separate NNTP connections to each and every other NNTP server. For this reason, the Usenet logical network continues to be very important, even in the Internet era.

The Usenet Server Structure

In theory, all that is required of the Usenet structure is that each site be connected to at least one other site in some form. The logical network could be amorphous and without any formal structure at all, as long as every site could form a path through some sequence of intermediate servers to each other one. However, the modern Usenet is very large, with thousands of servers and gigabytes of articles being posted every day. This calls for a more organized structure for distributing news.

For this reason, the modern Usenet logical network is structured loosely in a hierarchy. A few large Internet service providers (ISPs) and big companies with high-speed Internet connections and large servers are considered to be at the top of the hierarchy, in what is sometimes called the Usenet *backbone*. Smaller organizations connect to the servers run by these large organizations; these organizations are considered to be *downstream* from the backbone groups. In turn, still smaller organizations may connect further downstream from the ones connected to the large organizations.

This hierarchical structure means that most Usenet servers maintain a direct connection only to their upstream neighbor and to any downstream sites to which they provide service. A server is said to receive a *news feed* from its upstream connection, since that is the place from which it will receive most of its news articles. It then provides a news feed to all the servers downstream from it. I illustrated this structure earlier in Figure 85-1.

As an example, suppose Company A runs a large Usenet server called Largenews that is connected to the backbone. Downstream from this server is the NNTP server Mediumnews. That server provides service to the server named Smallnews. If a user posts an article to Mediumnews, it will be placed on that server immediately. That server will send the article downstream, to Smallnews, so that it can be read by that server's users. Mediumnews will also, at some point, send the article to Largenews. From Largenews, the message will be distributed to other backbone sites, which will

pass the message down to their own downstream sites. In this way, all sites eventually get a copy of the message, even though Mediumnews needs to connect directly to only two other servers.

The term used to describe how news is propagated with NNTP is *flooding*. This is because of the way that a message begins in one server and floods outward from it, eventually reaching the backbone sites, and then going down all the downstream “rivers” to reach every site on Usenet.

Even though I described the logical Usenet network as a hierarchy, it is not a strict hierarchy. For redundancy, many NNTP servers maintain connections to multiple other servers to ensure that news propagates quickly. The transmission of articles can be controlled by looking at message IDs to avoid duplication of messages that may be received simultaneously by one server from more than one neighbor.

Basic NNTP Propagation Methods

Now let’s look at how messages are actually propagated between servers using NNTP. There are two techniques by which this can be done:

- In the *push model*, as soon as a server receives a new message, it immediately tells its upstream and downstream neighbors about the message and asks them if they want a copy of it.
- In the *pull model*, servers do not offer new articles to their neighbors. The neighboring servers must ask for a list of new messages if they want to see what has arrived since the last connection was established, and then request that the new messages be sent to them.

Both techniques have advantages and disadvantages, but pushing is the model most commonly used today.

KEY CONCEPT One important role that NNTP plays is its propagation of articles between Usenet servers, which is what makes the entire system possible. Two models are used for article propagation: the push model, in which a server that receives a new message offers it to connected servers immediately, and the pull model, where servers that receive new messages hold them until they are requested by other servers. The push model is usually preferred since it allows for quicker communication of messages around the system.

Article Propagation Using the Push Model

Using the push model, when the administrators of an NNTP server establish a service relationship with an upstream Usenet service provider, they furnish the provider with a list of newsgroups that the downstream server wants to carry. Whenever a new article arrives at the upstream server within that list of groups, it is automatically sent to the downstream site. This saves the downstream server from constantly having to ask whether anything has arrived.

In the classic NNTP protocol as defined in RFC 977, the exchange of articles is based on the push model and performed using the IHAVE command. Returning to the example in the previous section, suppose three new messages arrive at the Largenews server. It would establish an NNTP connection to the Mediumnews

server and use IHAVE to provide the message IDs of each of the three new messages, one at a time. (NNTP commands are described later in this chapter.) The Mediumnews server would respond to each one, indicating whether or not it already had that message. If not, Lagenews would send it the message. An example of an article transaction using the push model of propagation is illustrated in Figure 85-3.

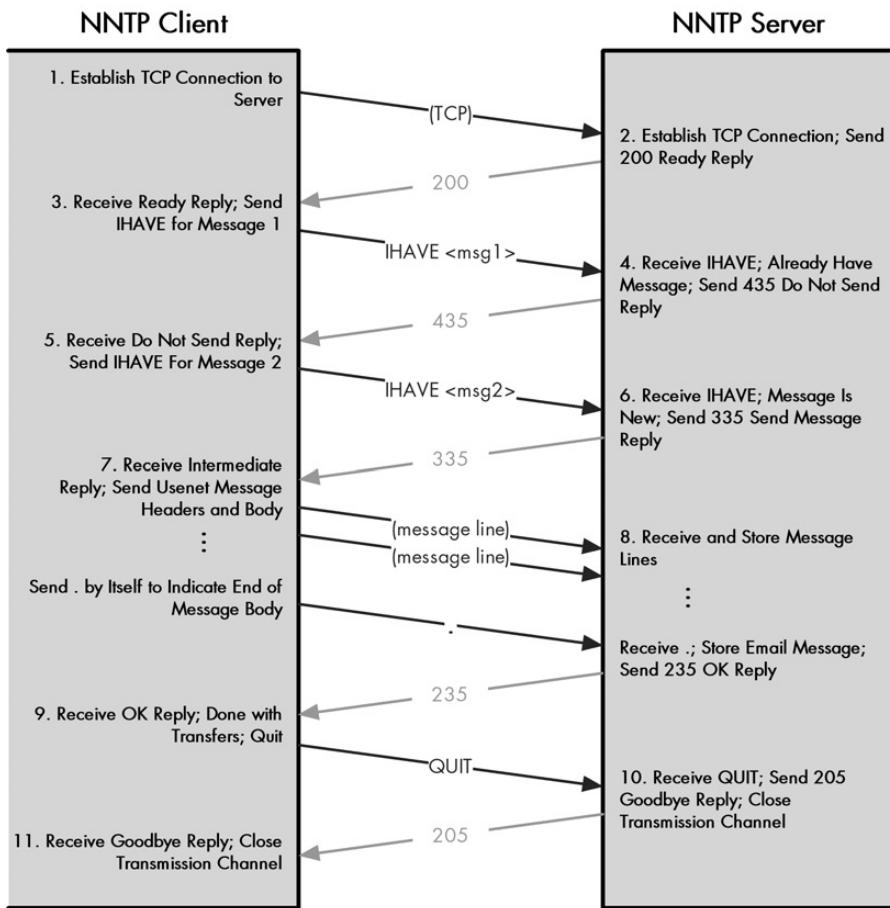


Figure 85-3: NNTP article propagation using the push model This example shows how Usenet articles are moved between servers using the conventional push model of propagation. Here, the device acting as an NNTP client (which may, in fact, be an NNTP server) has two messages available to offer to the server. It sends the IHAVE command specifying the message ID of the first message, but the server already has that message, so it sends a 435 (Do Not Send) reply. The client then issues an IHAVE with the second message ID. The server wants this one, so it sends a 335 reply. The client sends the Usenet message, ending with a single period on a line by itself. The server indicates that it received the message, and the client, finished with its transactions, quits the session.

The main advantage of this technique is that it ensures that a server is not sent a duplicate copy of a message that it already has. The problem with it in modern Usenet is that it is slow, because the server must respond to the IHAVE command before the message or the next command can be sent by the client.

Improving Propagation Efficiency with Streaming Mode

One of the more important NNTP extensions is *streaming mode*, which changes how news pushing is done. (NNTP command extensions are described later in this chapter.) When this mode is enabled, the client machine uses the CHECK command instead of IHAVE to ask the server if it wants a particular message. The server responds to indicate if it wants the message; if it does, the client sends the message with the TAKETHIS command.

The benefit of CHECK/TAKETHIS is that the client does not need to wait for a reply to CHECK before sending the next command. While the client is waiting for a reply to the first CHECK command, it can do something else, like sending the next CHECK command, allowing commands to be streamed for greater efficiency. So, the client could send a CHECK command for the first new message, then a CHECK for the second, while waiting for a reply from the server to the first one. Many CHECK commands could be sent in a stream, and then TAKETHIS commands sent for each reply received to CHECK commands sent earlier indicating that the message was wanted by the server.

Article Propagation Using the Pull Model

The pull model is implemented using the NEWNEWS and ARTICLE commands. The client connects to the server and sends the NEWNEWS command with a date specifying the date and time that it last checked for new messages. The server responds with a set of message IDs for new articles that have arrived since that date. The client then requests each new message using the ARTICLE command.

Note that the push and pull models can be combined in a single session. A client can connect to a server, use NEWNEWS to check for new messages on that server, and then IHAVE or CHECK inform the server about new messages the client wants to send. In practice, it is more common for only one or the other of the models to be used between a pair of servers for any given exchange.

In addition to propagating new messages, NNTP is also used to allow servers to communicate information about new newsgroups that have been created. This is done using the NEWGROUPS command, which is specified with a date and time like NEWNEWS. In response, the server sends to the client a list of new newsgroups that have been created since the specified date and time.

NNTP Client-Server Communication Process: News Posting and Access

One critical area where NNTP differs from its progenitor, SMTP, is that NNTP is not just used for interserver communication. It is also used for the initial posting of Usenet messages, as well as reading the messages. In fact, the majority of NNTP commands deals with the interaction between user client machines and NNTP servers, not communication between servers.

An NNTP client is any software program that knows the NNTP protocol and is designed to provide user access to Usenet. NNTP clients are usually called *newsreaders*, and they provide two main capabilities to a user: *posting* and *reading*.

Usenet messages. Usenet newsreaders exist for virtually all hardware and software platforms, and they range greatly in terms of capabilities, user interface, and other characteristics. Most people today use a Usenet newsreader on a client computer that must make NNTP connections to a separate NNTP server to read and post news. These programs are analogous to email clients, and, in fact, many email clients also function as NNTP clients.

News Posting, Access, and Reading

Posting a Usenet message is the first step in the overall Usenet communication process (although many Usenet articles are actually replies to other articles, so it's a bit of a chicken-and-egg situation). Article posting is quite straightforward with NNTP. The client establishes a connection to the server and issues the POST command. If the server is willing to accept new articles, it replies with a prompt for the client to send it the article. The article is then transmitted by the client to the server. Some newsreaders may batch new articles, so they can be sent in a single NNTP session, rather than submitting them one at a time.

Newsreaders also establish an NNTP connection to a server to read Usenet articles. NNTP provides a large number of commands to support a variety of different article access and retrieval actions that may be taken by a user. The first step in reading is sometimes to examine the list of available newsgroups. Using the LIST command, the client requests from the server a list of the newsgroups available for reading and posting. RFC 977 defines the basic LIST command, which returns a list of all groups to the client. RFC 2980 defines numerous extensions to the command to allow a client to retrieve only certain types of information about groups on the server. Since the number of Usenet newsgroups is so large today, this listing of newsgroups is usually skipped unless the user specifically requests it.

The next step in Usenet message access is typically to select a newsgroup to read from the list of groups available. Again, since there are so many groups today, most newsreaders allow a user to search for a group name using a pattern or partial name string. The GROUP command is then sent to the server with the name of the selected group. The server returns the first and last current article numbers for the group to the client.

Messages are identified in two ways: one absolute and the other site-specific. The article's message ID is a fixed identifier that can be used to uniquely represent it across Usenet; this is what is used in interserver communication to determine whether each site has a copy of a given message. In contrast, *article numbers* are server-specific; they represent the numbers assigned to those articles as they arrived at that server and are used as a shorthand to more easily refer to articles in a newsgroup. Thus, the same message will have a different article number on each NNTP server. Article numbers are used for convenience, since they are much shorter than message IDs. During a session, the NNTP server also maintains a current article pointer, which can be used for stepping sequentially through a newsgroup.

News Access Methods

There are several different ways that the newsreader can access messages in a group, depending on how it is programmed and what the user of the software wants. The news access methods include the following:

Full Newsgroup Retrieval The brute-force technique is for the client to simply request that the server send it all the messages in the group. The client issues the ARTICLE command to select the first current message in the group, using the first article number returned by the GROUP command. This sets the server's internal pointer for the session to point to the first article, so it can be retrieved. The NEXT command is then used to advance the pointer to the next message, and the ARTICLE command is used to retrieve it. This continues until the entire group has been read. Figure 85-4 illustrates the process. The retrieved messages are stored by the newsreader and available for instant access by the user. This method is most suitable for relatively small newsgroups and/or users with fast Internet connections.

Newsgroup Header Retrieval Since downloading an entire newsgroup is time-consuming, many newsreaders compromise by downloading the headers of all messages instead of the full message. The process is the same as for full newsgroup retrieval, but the HEAD command is used to retrieve just an article's headers. This takes less time than retrieving each message in its entirety using the ARTICLE command. The XHDR command extension can also be used, if the server supports it, to more efficiently retrieve only a subset of the headers for the messages, such as the subject line and author.

Individual Article Retrieval It is also possible to retrieve a single message from a group, using the ARTICLE command and specifying the article's message identifier.

KEY CONCEPT While NNTP is best known for its role in interserver propagation, it is also used by Usenet clients to write and read articles. Different commands provide flexibility in how articles can be read by a client device. A client can retrieve an entire newsgroup, only a set of newsgroup headers, or individual articles. Other commands also support various administrative functions.

Other Client/Server Functions

In addition to reading and posting, NNTP includes commands to support other miscellaneous tasks that a Usenet user may wish to perform. The client can ask the server for help information by using the HELP command or get a list of new newsgroups by using the NEWGROUPS command.

Most modern newsreaders include capabilities that go far beyond the basic posting and reading functions previously described. Most maintain their own sets of configuration files that allow a user to maintain a set of favorite subscribed newsgroups, rather than needing to choose a group to read from the master list each time Usenet is accessed. Newsreaders also keep track of which articles have been read by a user in each subscribed newsgroup, so users do not need to wade through a whole newsgroup to see new messages that have been posted.

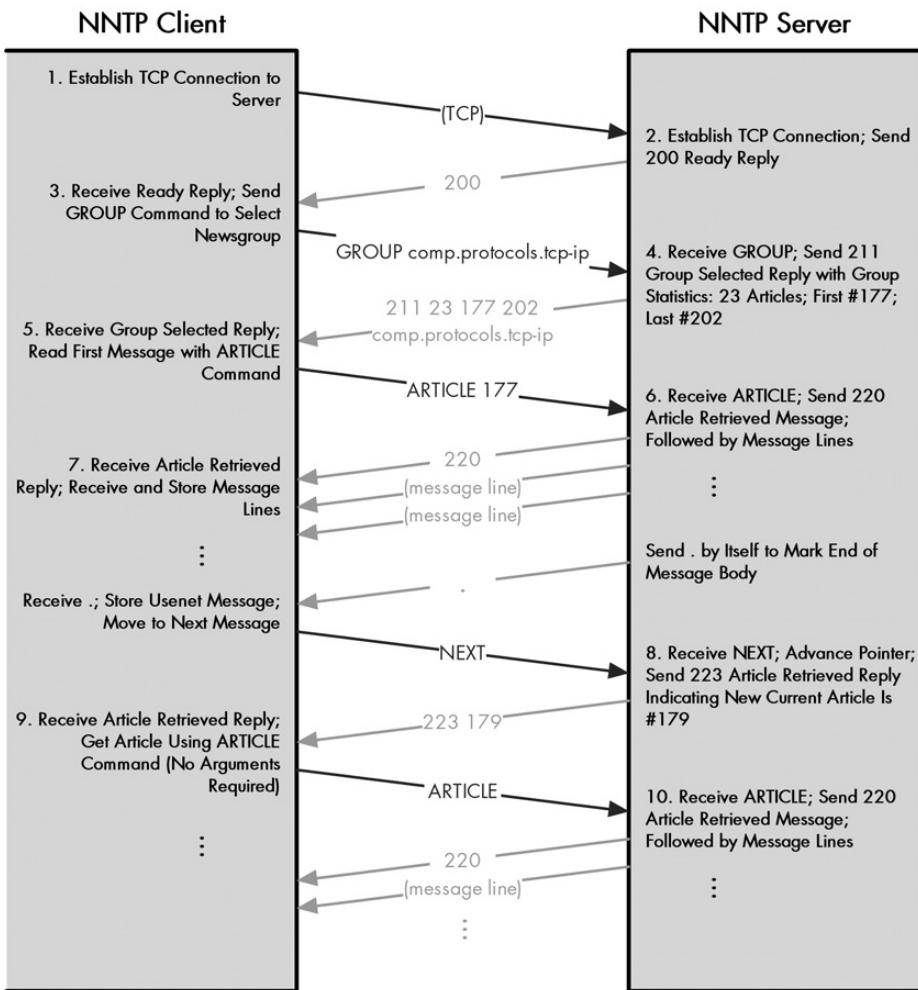


Figure 85-4: NNTP full newsgroup retrieval process There are many ways that an NNTP client can access and read Usenet messages on a server. One common method is to retrieve the entire contents of a newsgroup. In this example, the client uses the GROUP command to select the newsgroup comp.protocols.tcp-ip for reading; the server responds with a 211 (Group Selected) reply, which includes important statistics about the group. The client uses the ARTICLE command with the number of the first article in the group, 177, to read it from the server. The server then sends the message line by line, ending it with a single period on a line. The client uses the NEXT command to tell the server to advance its internal article pointer to the next message, which often will not be the next consecutive number after the one just read; here it is 179. The client can then read that message by sending the ARTICLE command by itself. Since no parameters are given, the server returns the current message (179).

Article Threading

One particularly useful enhancement to basic Usenet article reading is *threading*. This feature allows a newsreader to display articles not strictly in either alphabetical or chronological order, but rather grouped into conversations using the information

in the articles' References headers. Threading is especially useful in busy newsgroups, as it allows users to see all the articles in a particular discussion at once, rather than trying to juggle messages from many conversations simultaneously.

A problem with threading is that it takes a long time for a newsreader to sift through all those References lines and construct the article threads. To speed up this process, many servers now cache extra threading or overview information for newsgroups, which can be retrieved by the client to save time when a newsgroup is opened. This is done using the XTHREAD or XOVER NNTP command extensions.

NNTP Commands and Command Extensions

One of the great strengths of the open, cooperative process used to develop Internet standards is that new protocols are usually designed by building on older ones. This saves development time and effort, and promotes compatibility between technologies. As I explained earlier in the chapter, NNTP was based in many ways on principles from SMTP; SMTP, in turn, borrowed ideas from earlier protocols: Telnet and FTP. This legacy can be seen in the similarities between NNTP commands and those of these earlier protocols.

Command Syntax

As in SMTP, all NNTP commands are ASCII text that are sent over the NNTP TCP connection to an NNTP server, from the device acting as the client (which may be a newsreader client or an NNTP server itself). These are standard text strings adhering to the Telnet Network Virtual Terminal (NVT) format, terminated by the two-character carriage return/line feed (CRLF) sequence. As is the case with SMTP and FTP, you can conduct an interactive session with an NNTP server by using Telnet to connect to it on port 119.

The basic syntax of an NNTP command is *<command-code> <parameters>*. Unlike SMTP, NNTP commands are not restricted to a length of four characters. The parameters that follow the command are separated by one or more space characters, and are used to provide necessary information to allow the server to execute the command. NNTP commands are not case-sensitive.

Base Command Set

The main NNTP specification, RFC 977, describes the base set of commands supported by NNTP clients and servers. They are not broken into categories, but rather listed alphabetically, as I have done in Table 85-5. (The details on how many of these commands are used for news article propagation and news posting/access were provided earlier in this chapter.)

Table 85-5: NNTP Base Commands

Command Code	Command	Parameters	Description
ARTICLE	Retrieve Article	Message ID or server article number	Tells the server to send the client a particular Usenet article. The article to be retrieved may be specified using either its absolute, universal message ID or its locally assigned article number. When the command is issued with an article number, this causes the server's internal message pointer to be set to the specified article. If the message pointer is already set to a particular article, the ARTICLE command can be issued without an article number, and the current message will be retrieved.
HEAD	Retrieve Article Headers	Message ID or server article number	Same as the ARTICLE command, but retrieves only the article's headers.
BODY	Retrieve Article Body	Message ID or server article number	Same as the ARTICLE command, but returns only the body of the article.
STAT	Retrieve Article Statistics	Server article number	Conceptually the same as the ARTICLE command, but does not return any message text, only the message ID of the article. This command is usually used for setting the server's internal message pointer, so STAT is normally invoked only with an article number (and not a message ID).
GROUP	Select Newsgroup	Newsgroup name	Tells the server the name of the newsgroup that the client wants to access. Assuming the group specified exists, the server returns to the client the numbers of the first and last articles currently in the group, along with an estimate of the number of messages in the group. The server's internal article pointer is also set to the first message in the group.
HELP	Get Help Information	None	Prompts the server to send the client help information, which usually takes the form of a list of valid commands that the server supports.
IHAVE	Offer Article to Server	Message ID	Used by the client in an NNTP session to tell the server that it has a new article that the server may want. The server will check the message ID provided and respond to the client, indicating whether or not it wants the client to send the article.
LAST	Go to Last Message	None	Tells the server to set its current article pointer to the last message in the newsgroup.
LIST	List Newsgroups	None	Asks the server to send a list of the newsgroups that it supports, along with the first and last article number in each group. The command as described in RFC 977 is simple, supporting no parameters and causing the full list of newsgroups to be sent to the client. NNTP command extensions significantly expand the syntax of this command, as described in the following section of this chapter.
NEWGROU PS	List New Newsgroups	Date and time, and optional distribution specification	Prompts the server to send a list of new newsgroups created since the date and time specified. The client may also restrict the command to return only new newsgroups within a particular regional distribution.
NEWNEWS	List New News Articles	Date and time, and optional distribution specification	Requests a list from the server of all new articles that have arrived since a particular date and time. Like the NEWGROUPS command, this may be restricted in distribution. The server responds with a list of message IDs of new articles.

(continued)

Table 85-5: NNTP Base Commands (continued)

Command Code	Command	Parameters	Description
NEXT	Go to Next Message	None	Advances the server's current article pointer to the next message in the newsgroup.
POST	Post Article	None	Tells the server that the client would like to post a new article. The server responds with either a positive or negative acknowledgment. Assuming that posting is allowed, the client then sends the full text of the message to the server, which stores it and begins the process of propagating it to other servers.
QUIT	End Session	None	Terminates the NNTP session. To be "polite," the client should issue this command prior to closing the TCP connection.
SLAVE	Set Slave Status	None	Intended for use in special configurations where one NNTP server acts as a subsidiary to others. It is not often used in practice.

KEY CONCEPT The main NNTP standard defines a number of base NNTP commands that are used by the device initiating an NNTP connection to accomplish article propagation, posting, and reading functions. NNTP commands consist of a command code and, optionally, parameters that specify how the command is to be carried out.

NNTP Command Extensions

The base command set described in RFC 977 was sufficient to enable client-server and interserver functionality, but in many ways, it was quite basic and limited in efficiency and usefulness. As Usenet grew larger and more popular in the late 1980s, NNTP needed changes to improve its usability. In 1991, work began on a formal revision to the NNTP standard, but was never completed. Despite this, many of the concepts from that effort were adopted informally in NNTP implementations in subsequent years. In addition, some Usenet software authors created their own nonstandard features to improve the protocol, and some of these features also became de facto standards through widespread adoption.

As a result, by the late 1990s, most Usenet software actually implemented variations of NNTP with capabilities far exceeding what was documented in the standard. Naturally, not all NNTP software supported the same extra features, leading to potential compatibility difficulties between servers and clients. RFC 2980, "Common NNTP Extensions," was published in October 2000 to formalize many of these extensions to the base NNTP standard as defined in RFC 977.

The NNTP extensions primarily consist of new NNTP commands that are added to the basic NNTP command set, as well as some minor changes to how other commands and functions of NNTP work. The extensions generally fall into three categories:

- Extensions that improve the efficiency of NNTP message transport between servers
- Extensions that make NNTP more effective for client message access
- Miscellaneous extensions, which don't fall into either of the preceding groups

NNTP Transport Extensions

The first group is called the NNTP *transport extensions* and consists of a small group of related commands that are designed to improve interserver message propagation. Most of these implement NNTP's *stream mode*, which provides a more effective way of moving large numbers of articles from one server to another, as described in the discussion of interserver communication earlier in this chapter. Table 85-6 describes the new transport commands.

Table 85-6: NNTP Transport Extensions

Command Code	Command	Parameters	Description
MODE STREAM	Set Stream Mode	None	Used to tell the server that the client wants to operate in stream mode, using the CHECK and TAKETHIS commands.
CHECK	Check If Article Exists	Message ID	Used in stream mode by a server acting as a client to ask another server if it has a copy of a particular article. The server responds back indicating whether or not it wishes to be sent a copy of the article. This command is similar to IHAVE, except that the client does not need to wait for a reply before sending the next command.
TAKETHIS	Send Article to Server	Message ID	When a server responds to a CHECK command indicating that it wants a copy of a particular message, the client sends it using this command.
XREPLIC	Replicate Articles	List of newsgroups and article numbers	Created for the special purpose of copying large numbers of articles from one server to another. It is not widely used.

NNTP Newsreader Extensions

The second group of extensions defined by RFC 2980 consists of *newsreader extensions*, which focus primarily on commands used by newsreader clients in interactions with NNTP servers. These extensions consist of several new commands, as well as significant enhancements to one important command that was very limited in its functionality in RFC 977: LIST.

The original LIST command has no parameters and only allows a client to retrieve the entire list of newsgroups a server carries. This may have been sufficient when there were only a few hundred Usenet newsgroups, but there are now tens of thousands. RFC 2980 defines a number of new variations of the LIST command to allow the client much more flexibility in the types of information the server returns. Table 85-7 shows the new LIST command variations.

Table 85-7: NNTP LIST Command Extensions

Command Code	Command	Parameters	Description
LIST ACTIVE	List Active Newsgroups	Newsgroup name or pattern	Provides a list of active newsgroups on the server. This is semantically the same as the original LIST command, but the client may provide a newsgroup name or a pattern to restrict the number of newsgroups returned. For example, the client can ask for a list of only the newsgroups that contain "football" in them.

(continued)

Table 85-7: NNTP LIST Command Extensions (continued)

Command Code	Command	Parameters	Description
LIST ACTIVE.TIMES	List Active Newsgroup Creation Times	None	Prompts the server to send the client its <i>active.times</i> file, which contains information about when the newsgroups carried by the server were created.
LIST DISTRIBUTIONS	List Distributions	None	Causes the server to sent the client the contents of the <i>distributions</i> file, which shows what regional distribution strings the server recognizes (for use in the Distribution header of a message).
LIST DISTRIB.PATS	List Distribution Patterns	None	Asks the server for its <i>distribution.pats</i> file, which is like the distributions file but uses patterns to summarize distribution information for different newsgroups.
LIST NEWSGROUPS	List Newsgroups	Newsgroup name or pattern	Provides a list of newsgroup names and descriptions. This differs from LIST ACTIVE in that only the newsgroup name and description are returned, not the article numbers for each newsgroup. It is functionally the same as XGTITLE (see Table 85-8) and is usually employed by a user to locate a newsgroup to be added to his or her subscribed list.
LIST OVERVIEW.FMT	Display Overview Format	None	Prompts the server to display information about the format of its overview file. See the XOVER command description in Table 85-8 for more information.
LIST SUBSCRIPTIONS	Retrieve Default Subscription List	None	Asks the server to send the client a default list of subscribed newsgroups. This is used to set up a new user with a suggested list of newsgroups. For example, if an organization has an internal support newsgroup, it could put this group on the default subscription list so all new users learn about it immediately when they first start up their newsreader.

In addition to these changes to the LIST command, many new newsreader-related command extensions are defined, which are described in Table 85-8.

Table 85-8: NNTP Newsreader Extensions

Command Code	Command	Parameters	Description
LISTGROUP	List Article Numbers In Newsgroup	Newsgroup name	Causes the server to return a list of local article numbers for the current messages in the newsgroup. The server's current article pointer is also set to the first message in the group.
MODE READER	Set Newsreader Mode	None	Tells the server that the device acting as a client is a client newsreader and not another NNTP server. While technically not required—all commands can be sent by any device acting as client—some servers may be optimized to respond to newsreader-oriented commands if given this command.

(continued)

Table 85-8: NNTP Newsreader Extensions (continued)

Command Code	Command	Parameters	Description
XTITLE	Retrieve Newsgroup Descriptions	Newsgroup name or pattern	Used to list the descriptions for a newsgroup or a set of newsgroups matching a particular text pattern. This command is functionally the same as the LIST NEWSGROUP command extension (see Table 85-7). It is therefore recommended that XTITLE no longer be used.
XHDR	Retrieve Article Headers	Header name and optionally, either a message ID or a range of article numbers	Allows a client to ask for only a particular header from a set of messages. If only the header name is provided, the header is returned for all messages in the current group. Otherwise, the header is provided for the selected messages. This extension provides a newsreader client with a more efficient way of retrieving and displaying important headers in a newsgroup to a user.
XINDEX	Retrieve Index Information	Newsgroup name	Retrieves an <i>index</i> file, used by the newsreader called <i>TIN</i> to improve the efficiency of newsgroup perusal. TIN now supports the more common overview format, so the XOVER command is preferred to this one.
XOVER	Retrieve Overview Information	Article number or range of article numbers in a newsgroup	Retrieves the overview for an article or set of articles. Servers supporting this feature maintain a special database for their newsgroups that contains information about current articles in a format that can be used by a variety of newsreaders. Retrieving the overview information allows features like message threading to be performed more quickly than if the client had to retrieve the headers of each message and analyze them manually.
XPAT	Retrieve Article Headers Matching a Pattern	Header name, pattern, and either a message ID or a range of article numbers	Similar to XHDR in that it allows a particular header to be retrieved for a set of messages. The difference is that the client can specify a pattern that must be matched for the header to be retrieved. This allows the client to have the server search for and return certain messages, such as those with a subject line indicating a particular type of discussion, rather than requiring the client to download all the headers and search through them.
XPATH	Retrieve File Name Information	Message ID	Allows a client to ask for the name of the actual file in which a particular message is stored on the server.
XROVER	Retrieve Overview Reference Information	Article number or range of article numbers in a newsgroup	Like the XOVER command, but specifically retrieves information in the References header for the indicated articles. This is the header containing the data needed to create threaded conversations.
XTHREAD	Retrieve Threading Information	Optional DBINIT parameter	Similar to XINDEX, but retrieves a special threading information file in the format used by the newsreader named <i>TRN</i> . Like TIN, TRN now supports the common overview format, so XOVER is preferred to this command. The DBINIT parameter can be used to check for the existence of a thread database.

Other NNTP Extensions

The last extension group contains the miscellaneous extensions not strictly related to either interserver or client-server NNTP interaction. There are two commands in this group: AUTHINFO and DATE. The latter is a simple command that causes the server to tell the client its current date and time. AUTHINFO is more interesting. It is used by a client to provide authentication data to a server.

You may have noticed that there are no commands related to security described in the RFC 977 protocol. That's because the original NNTP had no security features whatsoever. Like many protocols written before the modern Internet era, security was not considered a big issue back in the early 1980s. Most news servers were used only by people within the organization owning the server, and simple security measures were used, such as restricting access to servers by IP address or through the use of access lists.

One of the more important changes made by many NNTP software implementations as soon as Usenet grew in size was to require authentication. Modern clients will usually issue AUTHINFO as one of their first commands on establishing a connection to a server, because the server will refuse to accept most other commands before this is done. A special reply code is also added to NNTP for a server to use if it rejects a command due to improper authentication.

The AUTHINFO command can be invoked in several different ways. The original version of the command required the client to issue an AUTHINFO USER command with a user name, followed by AUTHINFO PASS with a password. This is simple user/password login authentication. A variation of this is the AUTHINFO SIMPLE command, where the client needs to send just a password.

A client and server can also agree to use more sophisticated authentication methods by employing the AUTHINFO GENERIC command. The client provides to the server the name of the authentication method it wants to use, along with any arguments required for authentication. The client and server then exchange messages and authentication information as required by the particular authenticator they are using.

KEY CONCEPT A number of limitations in its base command set led to a proliferation of nonstandard enhancements to NNTP during the 1980s and 1990s. These were eventually documented in a set of NNTP command extensions that formally supplement the original RFC 977 commands. The extensions are conceptually divided into three groups: transport extensions that refine how NNTP propagates messages, newsreader extensions that improve client article access, and miscellaneous extensions. The most important miscellaneous extension is AUTHINFO, which adds security to NNTP.

NNTP Status Responses and Response Codes

Each time the device acting as a client in an NNTP connection sends a command, the server sends back a *response*. The response serves to acknowledge receipt of the command, to inform the client of the results of processing the command, and possibly to prompt for additional information. Since NNTP commands are structured and formatted in a way very similar to that of SMTP commands, I'm sure it

will come as no great surprise that NNTP responses are very similar to those of SMTP (described in Chapter 77). In turn, SMTP responses are based on the system designed for replies in FTP.

The first line of an NNTP response consists of a three-digit numerical *response code*, as well as a line of descriptive text that summarizes the response. These response codes are structured so that each digit has a particular significance, which allows the client to quickly determine the status of the command to which the reply was sent. After the initial response line, depending on the reply, a number of additional response lines may follow. For example, a successful LIST command results in a 215 response code, followed by a list of newsgroups.

BACKGROUND INFORMATION *The discussion of FTP reply codes in Chapter 72 explains the reasons why numeric reply codes are used in addition to descriptive text.*

As with SMTP and FTP, NNTP reply codes can be considered to be of the form xyz, where x is the first digit, y the second, and z the third. The first reply code digit (x) indicates the success, failure, or progress of the command in general terms; whether a successful command is complete or incomplete; and the general reason why an unsuccessful command did not work. The values of this digit are defined slightly differently than they are in SMTP and FTP. In some cases, the terminology is just simplified; for example, the second category is Command OK, instead of the more cryptic Positive Completion Reply. Table 85-9 shows the specific meaning of the possible values of this digit.

Table 85-9: NNTP Reply Code Format: First Digit Interpretation

Reply Code Format	Meaning	Description
1yz	Informative Message	General information; used for help information and debugging.
2yz	Command OK	The command was completed successfully.
3yz	Command OK So Far; Send the Rest	An intermediate reply, sent to prompt the client to send more information. Typically used for replies to commands such as IHAVE or POST, where the server acknowledges the command, and then requests that an article be transmitted by the client.
4yz	Command Was Correct, but Couldn't Be Performed	The command was valid but could not be performed. This type of error usually occurs due to bad parameters, a transient problem with the server, a bad command sequence, or similar situations.
5yz	Command Unimplemented or Incorrect, or Serious Program Error	The command was invalid or a significant program error prevented it from being performed.

The second reply code digit (y) is used to categorize messages into functional groups. This digit is used in the same general way as in SMTP and FTP, but the functional groups are different, as described in Table 85-10.

Table 85-10: NNTP Reply Code Format: Second Digit Interpretation

Reply Code Format	Meaning	Description
x0z	Connection, Setup, and Miscellaneous	Generic and miscellaneous replies.
x1z	Newsgroup Selection	Messages related to commands used to select a newsgroup.
x2z	Article Selection	Messages related to commands used to select an article.
x3z	Distribution Functions	Messages related to the transfer of messages.
x4z	Posting	Messages related to posting messages.
x5z	Authentication	Messages related to authentication and the AUTHINFO command extension. (This category is not officially listed in the standard, but these responses have a middle digit of 5.)
x8z	Nonstandard Extensions	Reserved for private, nonstandard implementation use.
x9z	Debugging	Debugging output messages.

The third reply code digit (*z*) indicates a specific type of message within each of the functional groups described by the second digit. The third digit allows each functional group to have ten different reply codes for each reply type given by the first code digit.

As in FTP and SMTP, these *x*, *y*, and *z* digit meanings are combined to make specific reply codes. For example, the reply code 435 is sent by the server if a client issues the IHAVE command but the server doesn't want the article being offered. The command was correct but the reply is negative, thus it starts with 4, and the message is related to message distribution, so the middle digit is 3.

Table 85-11 contains a list of some of the more common NNTP reply codes in numerical order, along with typical reply text from the standard and additional descriptive information.

Table 85-11: NNTP Reply Codes

Reply Code	Reply Text	Description
100	help text follows	Precedes response to HELP command.
111	(date and time)	Response to DATE command extension.
199	(debugging output)	Debugging information.
200	server ready - posting allowed	Sent by the server on initiation of the session, if the client is allowed to post messages.
201	server ready - no posting allowed	Sent by the server on initiation of the session, if the client is not allowed to post messages.
202	slave status noted	Response to the SLAVE command.
203	streaming is ok	Successful response to MODE STREAM command.
205	closing connection - goodbye!	Goodbye message sent in response to a QUIT message.
211	n f l s group selected	Successful response to the GROUP command, indicating the estimated number of messages in the group (<i>n</i>), first and last article numbers (<i>f</i> and <i>l</i>) and group name (<i>s</i>).

(continued)

Table 85-11: NNTP Reply Codes (continued)

Reply Code	Reply Text	Description
215	list of newsgroups follows (OR) information follows	Successful response to LIST command. The second form is for variations of LIST defined as NNTP command extensions.
218	tin-style index follows	Successful response to XINDEX command extension.
220	n <a> article retrieved - head and body follow	Successful response to the ARTICLE command, indicating the article number and message ID of the article.
221	n <a> article retrieved - head follows	Successful response to the HEAD command, indicating the article number and message ID of the article.
222	n <a> article retrieved - body follows	Successful response to the BODY command, indicating the article number and message ID of the article.
223	n <a> article retrieved - request text separately	Successful response to the STAT command, indicating the article number and message ID of the article.
224	overview information follows	Successful response to the XOVER command extension.
230	list of new articles by message-id follows	Successful response to the NEWNEWS command.
235	article transferred ok	Successful response to the IHAVE command, after the article has been sent.
239	article transferred ok	Successful response to the TAKETHIS command.
240	article posted ok	Successful response to the POST command, after the article has been posted.
250 or 281	authentication accepted	Successful authentication using the AUTHINFO command extension.
282	list of groups and descriptions follows	Positive response to the XGTITLE command extension.
288	binary data to follow	Successful response to the XTHREAD command extension.
335	send article to be transferred	Preliminary response to the IHAVE command.
340	send article to be posted	Preliminary response to the POST command.
381	more authentication information required	Preliminary response to the AUTHINFO command extension.
400	service discontinued	Session is being terminated, perhaps due to user request.
411	no such newsgroup	Invalid newsgroup name specified.
412	no newsgroup has been selected	Attempt to issue a command that refers to the current newsgroup before one has been selected using GROUP.
420	no current article has been selected	Attempt to issue a command that refers to the current article using the server's current article pointer, before the pointer has been set through article selection.
421	no next article in this group	Response to NEXT command when at the last article of a newsgroup.
422	no previous article in this group	Possible response to LAST (I have no idea why the word "previous" is in there).
423	no such article number in this group	Command with invalid article number.
430	no such article found	Article not found; it may have been deleted.

(continued)

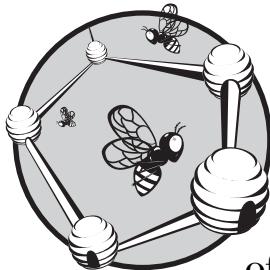
Table 85-11: NNTP Reply Codes (continued)

Reply Code	Reply Text	Description
435	article not wanted - do not send it	Negative response to IHAVE if server doesn't need the article.
436	transfer failed - try again later	Temporary failure of article transfer; retry.
437	article rejected - do not try again	Article refused for whatever reason.
438	already have it, please don't send it to me	Same as reply code 435, but for the CHECK command extension.
440	posting not allowed	POST command issued when posting is not allowed.
441	posting failed	POST command failed.
450	authorization required for this command	Response sent when server requires authentication but client has not yet authenticated.
452	authorization rejected	Failed authentication.
480	transfer permission denied	Response to CHECK if transfer is not allowed.
500	command not recognized	Bad command.
501	command syntax error	Bad syntax in command.
502	access restriction or permission denied	Permission denied; sent if the client has not properly authentication but the server requires it.
503	program fault - command not performed	General fatal error message.

KEY CONCEPT Each command sent by the device acting as the client in an NNTP connection results in the server returning a reply. NNTP replies consist of a three-digit reply code and a string of descriptive text. These codes are modeled after those of SMTP, and in turn, FTP.

86

GOPHER PROTOCOL (GOPHER)



Let's suppose that I told you I was going to describe a TCP/IP application layer protocol designed for the specific purpose of distributed document search and retrieval.

This protocol uses a client/server model of operation, where servers provide links to related resources such as files or programs that users access with client software that displays options for the user to select. You might think that I was talking about the World Wide Web, and for good reason. However, in this case, I am actually talking about one of the Web's predecessors: the *Gopher Protocol*.

In this chapter, I briefly describe Gopher's history, operation, differences from the World Wide Web, and role in the modern Internet.

Gopher Overview and General Operation

A good place to start our discussion of this protocol is with its name, which is well chosen for a number of reasons. The Gopher Protocol was developed at the University of Minnesota, whose sports teams are called the Golden Gophers (Minnesota is known as the Gopher State). This is the direct origin

of the name, but it is also appropriate because the rodent that shares it is known for burrowing, just as the protocol is designed to “burrow” through the Internet. And of course, the term *gopher* also applies to a person who performs errands, such as retrieving documents (they “go fer” this and “go fer” that).

The Gopher Protocol was developed in the late 1980s to provide a mechanism for organizing documents for easy access by students and faculty at the university. The core principle that guided the development of the system was *simplicity*. Gopher is designed on the basis of a small number of core principles, and it uses a very straightforward mechanism for passing information between client and server devices. It is described in RFC 1436, published in March 1993.

Information Storage on Gopher Servers

Information accessible by Gopher is stored as files on *Gopher servers*. It is organized in a hierarchical manner similar to the file system tree of a computer such as a Windows PC or UNIX workstation.

Just as a file system consists of a top-level directory (or folder) that contains files and subdirectories (subfolders), Gopher servers present information as a top-level directory that contains resources such as files, and/or subdirectories containing additional resources. Resources on different servers can be linked together by having them mentioned in each others’ resource hierarchies. It is also possible for virtual resources to be created that act as if they were files, such as programs that allow Gopher servers to be searched.

Gopher Client/Server Operation

Typical use of Gopher begins with a user on a client machine creating a TCP connection to a Gopher server using well-known TCP port number 70. After the connection is established, the server waits for the client to request a particular resource by sending the server a piece of text called a *selector string*. Often, when a user first accesses a server, he does not know what resource to request, so a null (empty) selector string is sent. This causes the server to send back to the client a list of the resources available at the top (root) directory of the server’s file system tree.

A directory list sent by the server consists of a set of lines, each of which describes one available resource in that directory. Each line contains the following elements, each separated by a tab character:

Type Character and Resource Name The first character of the line tells the client software what sort of resource the line represents. The most common type characters are 0 (zero) for a file, 1 for a subdirectory, and 7 for a search service. The rest of the characters up to the first tab character contain the name of the resource to be presented to the user.

Selector String The string of text to be sent to the server to retrieve this resource.

Server Name The name of the server where the resource is located.

Server Port Number The port number to be used for accessing this resource’s server; normally 70.

Each line ends with a carriage return/line feed (CRLF) character sequence consistent with the Telnet Network Virtual Terminal (NVT) specification. Upon sending the directory listing (or any other response) the connection between the client and server is closed.

After receiving this sort of directory list, the Gopher client software will display a menu to the user containing all the resource names the server provided. The user then selects his desired item from the menu, and the client retrieves it by making a connection to the appropriate server and port number, and sending the selector string of that resource. If this itself represents a subdirectory, the server will send a new directory listing for that subdirectory; if it represents some other type of resource, it will be accessed according to the requirements of the resource type.

For example, suppose this line were sent from the server to the client:

```
0Gopher Introduction<Tab>intro<Tab>gopher.someserver.org<Tab>70
```

This would be presented to the user as the file called Gopher Introduction in a menu containing other options. If the user chose it, the client would initiate a connection to the Gopher server gopher.someserver.org at port 70, and then send the selector string intro to that server to retrieve the document.

Important Differences Between Gopher and the Web

As I hinted at the start of this discussion, both Gopher and the Web are intended for the same basic purpose: providing access to repositories of information, with links between related documents and resources. However, they take a very different approach to how that information is accessed, especially in two key areas: user interface and resource linking.

Gopher's presentation to the user is entirely oriented around its hierarchical file system. As a result, Gopher is inherently menu-based, and the user interface is usually based on a simple text presentation of those menus. In contrast, information on web servers can be organized in any manner and presented to the user in whatever form or fashion the owner of the server desires. The Web is much more free-form, and there is no need to use a directory structure unless that is advantageous in some way.

Linking in the Web is done directly between documents, most often using Hyper-text Markup Language (HTML) tags. When someone writing Document A mentions something relevant to Document B, she puts a link to Document B directly in Document A. Gopher, on the other hand, is not designed to use links in this way. Instead, linking is intended to be done using the directory tree I described earlier.

Gopher's Role in the Modern Internet

There are some people who believe that Gopher is technically superior to the Web in a number of respects. They consider it cleaner to have servers do the linking, rather than having links embedded in documents. An argument can also be made that the text orientation of Gopher is efficient, better able to ensure compatibility

between platforms, and also more suited to special needs situations such as low-bandwidth links and access by those with visual impairments. Some Gopher enthusiasts thus consider it to be a purer hypertext system than the Web.

However, history shows us that despite Gopher predating the Web, the Web overtook it in popularity in only a few short years. Today, the Web is the 900-pound gorilla of the Internet, while most people have never even heard of Gopher. What happened?

I believe the main reason why Gopher lost out to the Web is that the Web is far more flexible. Gopher's use of text hyperlinks and server directory structures may be efficient, but it is limiting. In contrast, the Web allows information to be presented in a wide variety of ways. The open, unstructured nature of the Web makes it an ideal vehicle for the creativity of information providers and application developers. In the mid-1990s, the Web was also perfectly poised to support the transition of computing from text to graphics, and Gopher was not.

Simply put, you can do more with the Web than you can with Gopher, and most people care more about functionality and breadth of options than straight efficiency. Once the Web started to gain momentum, it very quickly snowballed. It took only a couple of years before Web use was well entrenched, and Gopher was unable to compete.

For its part, the University of Minnesota likely hastened Gopher's demise with its controversial decision to charge licensing fees to companies that wanted to use Gopher for commercial purposes. I do not believe there was anything nefarious about this. The university was on a limited budget and wanted companies that could afford it to pay a small fee to support development of Gopher software. However, computing history has shown time and time again that there is no faster way to kill a protocol or standard than to try to charge licensing or royalty fees for it, no matter what the reason.

By the late 1990s, Gopher was well on its way to obsolescence. As use of the protocol dwindled, many organizations could no longer justify the cost of continuing to run Gopher servers. Even the University of Minnesota itself eventually shut down its own Gopher servers due to low utilization. The final nail in the coffin for Gopher occurred in 2002, when a security vulnerability related to Gopher was discovered in Internet Explorer, and Microsoft chose to simply remove Gopher support from the product rather than fix the problem. Today, Gopher is still around, but it is a niche protocol used only by a relatively small group of enthusiasts and a handful of organizations that have a past history of using it.

KEY CONCEPT The Gopher Protocol is a distributed document search and retrieval protocol that was developed at the University of Minnesota in the late 1980s. Resources are stored on Gopher servers, which organize information using a hierarchical directory structure. Gopher clients access servers to retrieve directory listings of available resources, which are presented to the user as a menu from which an item may be selected for retrieval. Gopher's chief advantage is simplicity and ease of use, but it lacks flexibility in presentation and the ability to effectively present graphics and multimedia. For this reason, despite Gopher predating the World Wide Web, the Web has almost entirely replaced it, and Gopher is now a niche protocol.

PART III-10

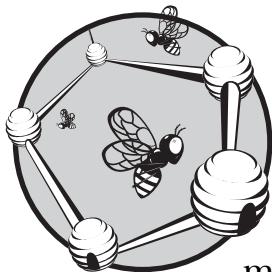
INTERACTIVE AND ADMINISTRATIVE UTILITIES AND PROTOCOLS

File and message transfer applications include the File Transfer Protocol (FTP), electronic mail (email), and the World Wide Web, which makes file and message transfer the most important category of classic TCP/IP applications. However, those applications do not represent the only ways that TCP/IP internetworks are used. While not as glamorous as some of the application protocols we have examined so far in this section, interactive and administrative protocols are also important and worth understanding.

This final part of the book covers a couple other categories of TCP/IP applications. The first chapter describes interactive and remote application protocols, which are used traditionally to allow a user of one computer to access another, or to permit the real-time exchange of information. The second chapter discusses TCP/IP administration and troubleshooting utilities, which can be employed by both administrators and end users to manage TCP/IP networks and diagnose problems with them.

87

TCP/IP INTERACTIVE AND REMOTE APPLICATION PROTOCOLS



When it comes to TCP/IP applications, file and message transfer applications get the most attention, because they are the ones used most often on modern internetworks. Another category of TCP/IP application protocols that is less well known is the group that allows users to interactively access and use other computers directly over an internetwork, such as the public Internet. These applications are not often employed by end users today, but they are still important—both from a historical perspective and because of their usefulness in certain circumstances, especially to network administrators.

In this chapter, I provide a brief description of the classic interactive and remote application protocols used in TCP/IP. I first describe the Telnet Protocol, one of the earliest and most conceptually important application protocols in TCP/IP. This discussion includes a description of Telnet client/server communication, the Telnet Network Virtual Terminal (NVT), and Telnet's protocol commands and options. I then describe the Berkeley remote access family, often called the *r commands* or protocols because their

command names begin with that letter. Finally, I provide a brief overview of the Internet Relay Chat (IRC) protocol, the original interactive chat application of the Internet and one still used widely today.

Telnet Protocol

In the very earliest days of internetworking, one of the most important problems that computer scientists needed to solve was how to allow someone operating one computer to access and use another as if that remote user were connected to it locally. The protocol created to meet this need was called *Telnet*, and the effort to develop it was tied closely to that of the Internet and TCP/IP as a whole. Even though most Internet users today never invoke the Telnet Protocol directly, they use some of its underlying principles indirectly all the time. Every time you send a piece of email, use the File Transfer Protocol (FTP) to transfer a file, or load a web page, you are using technology based on Telnet. For this reason, the Telnet Protocol can make a valid claim to the title of the most historically important application protocol in TCP/IP.

BACKGROUND INFORMATION A basic comprehension of the Transmission Control Protocol (TCP), especially its sliding window mechanism and flow control features, will be helpful in understanding Telnet. Those topics are covered in Chapters 48 and 49.

Telnet Overview, History, and Standards

The history of Telnet actually goes back over a decade before the modern TCP/IP protocol suite that we know today. As I mentioned in my overview of FTP, the early developers of TCP/IP internetworking technologies identified two overall application needs for networks to fill: enabling *direct access* to resources and also allowing *indirect access* to resources. FTP was created for indirect access, by allowing users to retrieve a resource from a remote host, use it locally, and if desired, copy it back to its source. Telnet was designed for direct access, by allowing users to access a remote machine and use it as if they were connected to it locally.

Telnet History

Telnet was initially developed in the late 1960s. This was well before the era of the small personal computers that so many of us use exclusively today. All computers of that period were large and usually shared by many users. To work on a computer, you had to access a physical terminal connected to that machine, which was usually specially tailored to the needs and requirements of the host. Two specific issues resulted from this situation:

- If an organization had several different computers, each user needed a separate terminal to access each computer that he or she used. This was expensive and inefficient. I can recall reading a quote from a book that compared this situation to having a room containing a number of television sets, each of which could only display a single channel.

- Perhaps a more significant issue was the difficulty in allowing a user at one site to access and use a machine at another site. The only method at the time for accomplishing this was to install a dedicated data circuit from the site of the computer to the site of the user, to connect the user's terminal to the remote machine. Again, each circuit would enable access to only one machine. Every combination of user and computer required a separate, expensive circuit to be installed and maintained.

The solution to both of these issues was to create a more general way of allowing any terminal to access any computer. The underlying internetwork provided the mechanism for communicating information between computers. This became the physical network connecting sites and the TCP/IP protocol suite connecting networks. On top of this ran an application protocol that allowed a user to establish a session to any networked computer and use it. That application protocol is Telnet.

Telnet was the first application protocol demonstrated on the fledgling ARPAnet, in 1969. The first RFC specifically defining Telnet was RFC 97, "First Cut at a Proposed Telnet Protocol," published in February 1971. Development of Telnet continued throughout the 1970s, with quite a number of different RFCs devoted to revisions of the protocol and discussions of issues related to it. It took many years to refine Telnet and resolve all the difficulties that were associated with its development. The final version of the protocol, "Telnet Protocol Specification," was published as RFC 854 in May 1983. Over the years, other RFCs have been published to clarify the use of the protocol and address various issues such as authentication. There are also a number of other RFCs that define Telnet options, as discussed in the "Telnet Options and Option Negotiation" section later in this chapter.

Fundamental Telnet Concepts

At first glance, it may be surprising that Telnet took so long to develop, because in theory, it should be a very simple protocol to define. All it needs to do is send keystrokes and program output over the network like any other protocol. Its definition would be simple if every terminal and computer used the same communication method, but they do not. Telnet becomes complicated because it needs to allow a terminal from one manufacturer to be able to talk to a computer that may use a very different data representation.

Telnet solves this problem by defining a method that ensures compatibility between terminal types and computers, while allowing special features to be used by computers and terminals that agree to support them. The protocol is built on a foundation of three main concepts.

Network Virtual Terminal (NVT) Telnet defines a standardized, fictional terminal called the *Network Virtual Terminal (NVT)* that is used for universal communication by all devices. A Telnet client takes input from a user and translates it from its native form to the NVT format to send to a Telnet server running on a remote computer. The server translates from NVT to whatever representation the computer being accessed requires. The process is reversed when data is sent from the remote computer back to the user. This system allows clients and servers to communicate even if

they use entirely different hardware and internal data representations. Special Telnet commands are interspersed with the data to allow the client and server devices to perform various functions needed to manage the operation of the protocol.

Options and Option Negotiation Having Telnet clients and servers act as NVTs avoids incompatibilities between devices, but does so by stripping all terminal-specific functionality to provide a common base representation that is understood by everyone. Since there are many cases where more intelligent terminals and computers may wish to use more advanced communication features and services, Telnet defines a rich set of options and a mechanism by which a Telnet client and server can negotiate their use. If the client and server agree on the use of an option, it can be enabled; if not, they can always fall back on the NVT to ensure basic communication.

Symmetric Operation While Telnet is a client/server protocol, it is specifically designed to not make assumptions about the nature of the client and server software. Once a Telnet session is established, the computers can each send and receive data as equals. They can also each initiate the negotiation of options. This makes the protocol extremely flexible and has led to its use in a variety of places, as discussed in the next section.

Telnet Applications

Telnet is most often associated with remote login, which is its common traditional use. A user typically uses a Telnet client program to open a Telnet connection to a remote server, which then treats the Telnet client like a local terminal, allowing the user to log in and access the server's resources as if he were using a directly attached terminal. Telnet is still used this way quite extensively by UNIX users, who often need to log in to remote hosts from their local machines (I use Telnet in this manner every day to access a machine hundreds of miles away). However, this use of Telnet is not nearly as common among the majority of Internet users who work on Windows or Apple computers, where network resources are accessed not through direct login, but by other means.

Although remote login is a big part of what Telnet is about, the protocol was not inherently designed for that specific function. When Telnet is used to access a remote device, the protocol itself is used only to set up the connection between the client and server machines, encode data to be transmitted according to the rules of the Telnet NVT, and facilitate the negotiation and use of options. The client and server devices decide whether Telnet is used for remote access or for some other purpose.

This flexibility, combined with Telnet's age in the TCP/IP suite, has led to its being adopted for a variety of other protocols. Since Telnet doesn't make assumptions about what a client is and what a server is, any program or application can use it. Many of the file and message transfer applications—such as FTP, Simple Mail Transfer Protocol (SMTP), Network News Transfer Protocol (NNTP), and Hypertext Transfer Protocol (HTTP)—communicate by sending text commands and messages, and use Telnet's NVT specification to ensure the compatibility of communication between devices. They don't actually establish Telnet sessions or use features like option negotiation; they just send data in a manner consistent with how Telnet works. Thus, even though modern Internet users may never

intentionally invoke Telnet specifically, they use it indirectly every time they send or receive email or browse the Web. Administrators can even use Telnet client software to access devices such as FTP and HTTP servers, and send those devices commands manually.

KEY CONCEPT *Telnet* is one of the oldest protocols in the TCP/IP suite, first developed in the 1960s to allow a user on one computer system to directly access and use another. It is most often used for remote login, with Telnet client software on a user's machine establishing a session with a Telnet server on a remote host to let the user work with the host as if connected directly. To ensure compatibility between terminals and hosts that use different hardware and software, communication between Telnet client and server software is based on a simplified, fictional data representation, called the *Network Virtual Terminal (NVT)*, which can be enhanced through the negotiation of options.

Telnet Connections and Client/Server Operation

Telnet's overall function is to define a means by which a user or process on one machine can access and use another machine as if it were locally connected. This makes Telnet inherently client/server in operation, like so many other application protocols in TCP/IP. Usually, the Telnet client is a piece of software that acts as an interface to the user, processing keystrokes and user commands and presenting output from the remote machine. The Telnet server is a program running on a remote computer that has been set up to allow remote sessions.

TCP Sessions and Client/Server Communication

Telnet is used for the interactive communication of data and commands between a client and server over a prolonged period of time, and is thus strongly based on the concept of a *session*. For this reason, Telnet runs over the connection-oriented Transmission Control Protocol (TCP). Telnet servers listen for connections on well-known TCP port number 23. When a client wants to access a particular server, it initiates a TCP connection to the appropriate server, which responds to set up a TCP connection using the standard TCP three-way handshake (described in Chapter 47).

The TCP connection is maintained for the duration of the Telnet session, which can remain alive for hours, days, or even weeks at a time. The quality of service features of TCP guarantee that data is received reliably and in order, and ensure that data is not sent at too high a rate for either client or server. A machine offering Telnet service can support multiple simultaneous sessions with different users, keeping each distinct by identifying it using the IP address and port number of the client.

Since TCP is a full-duplex protocol, both the client and server can send information at will over the Telnet session. By default, both devices begin by using the standard NVT method for encoding data and control commands (which we will explore fully a little later in this chapter). They can also negotiate the use of Telnet options to provide greater functionality for the session. While option negotiation

can occur at any time, it is normal for there to be a burst of such option exchanges when a Telnet session is first established and only occasional option command exchanges thereafter.

With the TCP connection in place and the Telnet session active, the client and server software begin their normal jobs of interfacing the user to the remote host. To the user, the Telnet session appears fundamentally the same as sitting down at a terminal directly connected to the remote host. In most cases, the server will begin the user's session by sending a login prompt to ask for a user name and password. The Telnet client will accept this information from the user and send it to the server. Assuming the information is valid, the user will be logged in and can use the host in whatever manner her account authorizes.

As mentioned in the Telnet overview, even though the protocol is commonly used for remote login, it does not need to be used in this manner. The administrator of the computer that is running the Telnet server determines how it is to be used on that machine. As just one example, a Telnet server can be interfaced directly to a process or program providing a service. I can recall years ago using an Internet server that provided weather information to the public using Telnet. After using the protocol to connect to that machine, users were presented not with a login prompt, but with a menu of weather display options. Today, the Web has replaced most of such facilities, as it is far better suited to this type of information retrieval.

KEY CONCEPT Telnet is a client/server protocol that uses TCP to establish a session between a user terminal and a remote host. The Telnet client software takes input from the user and sends it to the server, which feeds it to the host machine's operating system. The Telnet server takes output from the host and sends it to the client to display to the user. While Telnet is most often used to implement remote login capability, it is not specifically designed for logins. The protocol is general enough to allow it to be used for a variety of functions.

Use of Telnet to Access Other Servers

The Telnet NVT representation is used by a variety of other protocols such as SMTP and HTTP. This means that the same Telnet client that allows you to access a Telnet server can be used to directly access other application servers. All you need to do is specify the port number corresponding to the service. For example, the following command will allow you to directly interface to a web server:

```
telnet www.someserversomewhere.org 80
```

You will not receive a login prompt, but instead the server will wait for you to send an HTTP Request message, as if you were a web browser. If you enter a valid request, the server will send you an HTTP Response message. Used in this way, Telnet can be very valuable as a diagnostic tool.

KEY CONCEPT The Telnet Network Virtual Terminal (NVT) data representation has been adopted by a host of other TCP/IP protocols as the basis for their messaging systems. Telnet client software can thus be used not only to connect to Telnet servers, but also to connect to servers of protocols such as SMTP and HTTP, which is useful for diagnostic purposes.

Telnet Communications Model and the Network Virtual Terminal (NVT)

At its heart, Telnet is a rather simple protocol. Once a TCP connection is made and the Telnet session begins, the only real task for the client and server software is to capture input and output, and redirect it over the network. So, when the user presses a key on his local terminal, the Telnet client software captures it and sends it over the network to the remote machine. There, the Telnet server software sends the keypress to the operating system, which treats it as if it had been typed locally. When the operating system produces output, the process is reversed: Telnet server software captures the output and sends it over the network to the user's client program, which displays it on the printer or monitor.

To invoke two well-known clichés, I could say that this looks good on paper, but that the devil is in the details. This simplified implementation would work only if every computer and terminal used the exact same hardware, software, and data representation. Of course, this is far from the case today, and was even worse when Telnet was being developed. Computers back in the “good old days” were highly proprietary and not designed to interoperate. They differed in numerous ways—from the type of keyboard a terminal used and the keystrokes it could send, to the number of characters per line and lines per screen on a terminal, to the character set used to encode data and control functions. In short, Computer A was designed to accept input in a particular form from its own terminals, and not those of Computer B.

This is actually a fairly common issue in the world of networking, and one to which I can draw a real-world analogy to help explain the problem and how it may be solved. Suppose that an important international conference was attended by 30 ambassadors from different nations, each of which had one assistant. Every ambassador and assistant pair spoke only their own language and thus could only speak to each other—just like a computer and terminal designed to interface only to each other. To allow the assistant from one country to speak to the ambassador from the others, one solution would be to train the assistants to speak the languages of all the other attending nations. Back in the computing world, this would be like defining the Telnet Protocol so that every Telnet client software implementation understood how to speak to every computer in existence. This would work, but it would be quite impractical and difficult to do.

An alternative approach is to define a single common language and have all the ambassadors and assistants learn it. While this would require some work, it would be a lot less than requiring people to learn dozens of languages. Each ambassador and assistant would speak both a native language and this chosen common language. Each could communicate with all of the others using this common language, without needing to know all of the languages that might be used by anyone at the

conference. Even more important, if an ambassador and assistant showed up at the conference speaking a new, 31st language, all the other delegates wouldn't need to learn it.

Telnet uses a very similar approach for dealing with its problem of hardware and software compatibility. Rather than having terminals and hosts communicate using their various native languages, all Telnet clients and servers agree to send data and commands that adhere to a fictional, virtual terminal type call the NVT.

The NVT

The NVT defines a set of rules for how information is formatted and sent, such as character set, line termination, and how information about the Telnet session itself is sent.

Each Telnet client running on a terminal understands both its native language and the NVT language. When users enter information on their local terminal, it is converted to NVT form for transmission over the network. When the Telnet server receives this information, it translates it from NVT form to the format that the remote host expects to receive it. The identical process is performed for transmissions from the server to the client, in reverse. This is illustrated in Figure 87-1.

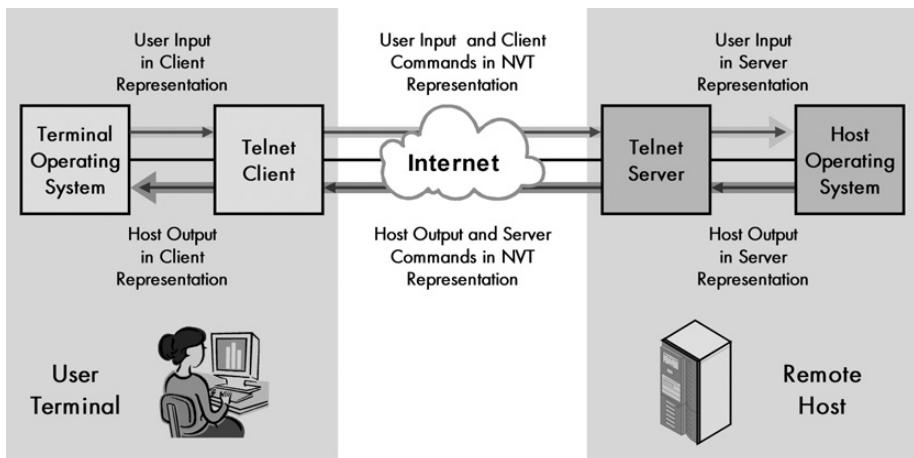


Figure 87-1: Telnet communication and the Network Virtual Terminal (NVT) Telnet uses the Network Virtual Terminal (NVT) representation to allow a user terminal and remote host that use different internal formats to communicate.

KEY CONCEPT The Telnet Network Virtual Terminal (NVT) is a uniform data representation that ensures the compatibility of communication between terminals and hosts that may use very different hardware, software, and data formats. The Telnet client translates user input from the terminal's native form to NVT form for transport to the Telnet server, where it is converted to the host's internal format. The process is reversed for output from the host to the user.

The NVT is defined to consist of a logical keyboard for input and a logical printer for output (the age of the protocol is reflected in these terms; decades ago there were no monitors, all output was on paper). NVT uses the 7-bit *United States ASCII* (US-ASCII) character set. Each character is encoded using one 8-bit byte.

However, a client and server can use Telnet options to negotiate other data representations, including the transmission of either extended ASCII or even full 8-bit binary data.

NVT ASCII Control Codes

Regular ASCII consists of 95 regular, printable characters (codes 32 through 126) and 33 control codes (0 through 31 and 127). The Telnet standard specifies that the output device must be able to handle all the printable characters, and it mandates how several of the other common ASCII control codes should be interpreted. Of these codes, three (0, 10, and 13) are required to be accepted by all Telnet software; five others are optional, but if supported, must be interpreted in a manner consistent with the Telnet specification. Table 87-1 describes the standard Telnet NVT ASCII control codes.

Table 87-1: Interpretation of Standard Telnet NVT ASCII Control Codes

ASCII Value (Decimal)	ASCII Character Code	ASCII Character	Description	Support Optional/Mandatory
0	NUL	Null	No operation (no effect on output).	Mandatory
7	BEL	Bell	Produces an audible or visible signal on the output without moving the print head. This notification may be used to get the user's attention, as in the case of an error.	Optional
8	BS	Backspace	Moves the print position one character to the left.	Optional
9	HT	Horizontal Tab	Moves the printer to the next horizontal tab stop. The standard does not specify how devices agree on tab stop positions; this can be negotiated using Telnet options.	Optional
10	LF	Line Feed	Moves the printer to the next line, keeping the print position the same.	Mandatory
11	VT	Vertical Tab	Moves the print line to the next vertical tab stop. As with the HT character, devices must use an option to come to an agreement on vertical tab stop positions.	Optional
12	FF	Form Feed	Moves the printer to the top of the next page (or on a display, clears the screen and positions the cursor at the top).	Optional
13	CR	Carriage Return	Moves the printer to the left margin of the current print line.	Mandatory

The Telnet NVT scheme defines the combination of the carriage return (CR) and line feed (LF) characters to represent the end of a line of ASCII text. The literal meaning of these two characters is return to the left margin (the CR) and go to the next line (the LF). However, NVT treats the CRLF sequence as more than just two independent characters; they are taken collectively to define a *logical end-of-line character*. This is necessary because not all terminal types define an end of line using both CR and LF. Translation of end-of-line characters between the native and NVT formats is one of the functions that Telnet client and server software must perform to ensure compatibility between terminals and hosts.

KEY CONCEPT The Telnet NVT format is based on 7-bit US-ASCII, with each byte carrying one character. The standard specifies that devices must handle all standard printable ASCII characters, as well as three mandatory control characters. Two of these are the carriage return (CR) and line feed (LF) characters; when combined, these define the logical end of a line of text. The Telnet standard also describes the interpretation of five other optional ASCII control characters.

Half-Duplex and Full-Duplex Modes

Another artifact of the age of Telnet is that for maximum compatibility, the NVT specification is designed under the assumption of half-duplex operation: only one device can transmit at a time. A device that is sending data is supposed to end its transmission with the special Telnet Go Ahead command, telling the other device that it may now transmit (the next section describes Telnet protocol commands). This is similar to how people using walkie-talkies end each transmission with “Over,” to tell their partners that they may now respond.

Of course, modern networks operate in a full-duplex mode, and using half-duplex communication would be needlessly inefficient. In most cases, the Telnet client and server agree to use an option (Suppress Go Ahead) that eliminates the need to send this command. However, having this as the default is a good example of how NVT acts as a least common denominator in Telnet, in case the simpler operating mode is needed by either device.

Telnet Protocol Commands

Most of the input that users enter at a terminal takes the form of data and commands that are sent to the application program they are using. However, computer systems also provide a means by which users can instruct the terminal to send certain commands that control how the terminal itself operates, and how it interacts with the computer to which it is connected. The best example of this is the command to interrupt a process, which is usually sent by pressing a special key or key combination on the user terminal.

Telnet needs to have a way to allow such commands to be entered by the user. However, here we run into the same problem that arises in the communication of data between terminals and computers: a lack of uniformity in representation. While all terminals and computers support the ability to interrupt a running program, for example, they may each use a different keystroke to invoke it. For example, on most UNIX systems, the key combination CTRL-C interrupts a program, but typing this on a Windows system will not (it usually represents the copy data function!).

Since the problem is the same as the one we ran into in representing data flow, it's not too surprising that the solution is the same: the use of a universal representation for a set of standard commands to be passed between the terminal and host computer. All keystrokes that represent these commands are translated to the standard Telnet codes for transmission, and then translated to the specific needs of the host computer. So, if a user presses CTRL-C on a UNIX terminal where this is

defined as the interrupt function, instead of sending that exact keystroke, the Telnet client sends the special Telnet Interrupt Process command, which is translated by the Telnet server to the command code appropriate for the connected host.

The Telnet standard includes a number of these special codes to allow a user to control the operation of the remote computer. It also defines a set of commands that are specific to the Telnet Protocol itself; these let the Telnet client and Telnet server software communicate. Collectively, these are called Telnet *protocol commands*.

All Telnet commands are sent in the same communication stream as regular data. They are represented using special byte values in the range from 240 to 254. To differentiate between data bytes of these values and Telnet commands, every command is preceded by a special *escape character*, given the name *Interpret As Command (IAC)*. IAC has a value of 255; when the recipient sees this character, it knows the next byte is a command, not data. So, since the Telnet Interrupt Process command has the value 244, to send this command, the Telnet client would transmit the byte 255 and then 244. If the actual data byte value 255 needs to be sent, it is transmitted as two 255 bytes. Some Telnet commands also include additional bytes of data, which are sent after the command code itself. A good example is the use of parameters in Telnet option negotiation, as you will see in the “Telnet Options and Option Negotiation” section later in this chapter.

KEY CONCEPT The Telnet Protocol defines a set of *protocol commands* that are used for two purposes: first, to represent standard control functions that need to be sent between a terminal and host, such as the command to interrupt a process, and second, to enable protocol communication between the Telnet client and server software. Protocol commands are sent in the normal data communication stream over the Telnet session’s TCP connection. Each is represented by a byte value from 240 to 254, and is preceded by the Interpret As Command (IAC) command, byte value 255, which tells the recipient that the next byte in the stream is a command.

You may be wondering at this point why the IAC character is needed at all. After all, Telnet uses US-ASCII, which is 7-bit data in the byte range of 0 to 127, and the Telnet commands have values higher than 127. One general rationale for using the IAC escape character is to be explicit that a command is being sent. A more specific reason is to accommodate the optional sending of 8-bit binary data over Telnet, which the client and server can negotiate. If this mode were enabled and commands were not preceded by the IAC character, this would require all data bytes with values from 240 to 255 to be marked somehow so they would be interpreted as data and not commands. It is more efficient to include an extra byte for commands than data, since commands are sent less frequently. By escaping commands, only data byte value 255 requires two bytes to be sent.

Table 87-2 lists the Telnet protocol commands in numerical byte value order, showing for each its command code and name, and describing its meaning and use.

Table 87-2: Telnet Protocol Commands

Command Byte Value (Decimal)	Command Code	Command	Description
240	SE	Subnegotiation End	Marks the end of a Telnet option subnegotiation, used with the SB code to specify more specific option parameters. See the “Telnet Options and Option Negotiation” section later in this chapter for details.
241	NOP	No Operation	Null command; does nothing.
242	DM	Data Mark	Used to mark the end of a sequence of data that the recipient should scan for urgent Telnet commands. See the discussion of Telnet interrupt handling in the following section for details.
243	BRK	Break	Represents the pressing of the “break” or “attention” key on the terminal.
244	IP	Interrupt Process	Tells the recipient to interrupt, abort, suspend, or terminate the process currently in use.
245	AO	Abort Output	Instructs the remote host to continue running the current process but discard all remaining output from it. This may be needed if a program starts to send unexpectedly large amounts of data to the user.
246	AYT	Are You There	May be used to check that the remote host is still “alive.” When this character is sent, the remote host returns some type of output to indicate that it is still functioning.
247	EC	Erase Character	Instructs the recipient to delete the last undeleted character from the data stream. Used to undo the sending of a character.
248	EL	Erase Line	Tells the recipient to delete all characters from the data stream back to (but not including) the last end-of-line (CRLF) sequence.
249	GA	Go Ahead	Used in Telnet half-duplex mode to signal the other device that it may transmit.
250	SB	Subnegotiation	Marks the beginning of a Telnet option subnegotiation, used when an option requires the client and server to exchange parameters. See the “Telnet Options and Option Negotiation” section later in this chapter for a full description.
251	WILL	Will Perform	In Telnet option negotiation, indicates that the device sending this code is willing to perform or continue performing a particular option.
252	WONT	Won’t Perform	In Telnet option negotiation, indicates that the device sending this code is either not willing to perform a particular option or is now refusing to continue to perform it.
253	DO	Do Perform	In Telnet option negotiation, requests that the other device perform a particular option or confirms the expectation that the other device will perform that option.
254	DONT	Don’t Perform	In Telnet option negotiation, specifies that the other party not perform an option or confirms a device’s expectation that the other party not perform an option.
255	IAC	Interpret As Command	Precedes command values 240 through 254 as described in the preceding descriptions. A pair of IAC bytes in a row represents the data value 255.

Perhaps ironically, the Telnet commands are not used as much today as they were when Telnet was in its early days, because many of the compatibility issues that we discussed earlier no longer exist. ASCII has become the standard character set of the computing world, so many of the functions such as aborting output or interrupting a process no longer require the use of Telnet commands. They are still widely used, however, for internal Telnet operations such as option negotiation.

Telnet Interrupt Handling

All the bytes of data sent from a Telnet client to a server are received in the order that they were sent, and vice versa. This is the way that we expect an application to operate. In fact, ensuring that data is not received out of order is one of the jobs that we assume of the reliable transport protocol TCP, over which Telnet runs. However, this can cause a problem for Telnet because of the way Telnet sends both data and commands over the same connection.

The most important case where this issue arises is when a user needs to interrupt a process. Suppose that you are using Telnet to run an interactive program that takes user input, processes it, and then produces output. You are merrily typing away when you notice that you haven't seen any output from the program for a while. It has apparently hung up due to a programming error or other glitch.

If you were using the program on a directly connected terminal, you would simply use the key or keystroke command appropriate to that terminal to interrupt or abort the process and restart it. Instead, you are using Telnet, so you enter the appropriate keystroke, which gets converted to the special Telnet Interrupt Process command code (byte value 244, preceded by the Telnet Interpret As Command code, 255).

Since Telnet uses only a single stream for commands and data, that code is placed into the TCP data stream to be sent over to the Telnet server. Since you were entering data for a while, that Telnet Interrupt Process code will be sitting behind a bunch of regular data bytes. Now the remote process has stopped reading this data, which means the TCP receive buffer on the server will start to fill up. The Interrupt Process command will thus remain stuck in the buffer, waiting to be read. In fact, if the number of data bytes in front of the command is high enough, the TCP buffer on the server may fill entirely, causing the server to close the client's TCP send window. This means the Interrupt Process command will wait in the client's outgoing TCP queue and *never* be sent to the remote host!

What we need here is some way to be able to flag the Interrupt Process command, so that it can be sent to the remote host regardless of the number of data bytes outstanding in front of it. If you've already perused the chapters devoted to TCP, you may be thinking that you have already read about a feature of that protocol that seems ideally suited for this exact problem, and you would be correct! The TCP urgent function (described in Chapter 48) allows an important piece of data to be marked so that it is given priority over regular data, a process sometimes called *out-of-band signaling* (because the signal is outside the normal data stream). Telnet uses this feature of TCP to define what it calls the *synch function*.

When needed, the *synch* function is invoked by the client sending the special Telnet Data Mark (DM) protocol command, while instructing its TCP layer to mark that data as urgent. The URG bit in the TCP segment carrying this command causes

it to bypass TCP's normal flow control mechanism so it is sent over to the remote host. The Telnet server software, seeing the synch in the data stream, searches through all of the data in its buffer looking only for Telnet control commands such as Interrupt Process, Abort Output, and Are You There. These commands are then executed immediately. The server continues to search for important commands up to the point where the Data Mark command is seen. All intervening data is discarded; it will need to be retransmitted. After the Data Mark is processed, the server returns to normal operation.

It is also possible for the server to use the synch function in communication with the user on the client device. For example, if the user sends the Abort Output command to the server, she is telling the server to discard all remaining output from the current process. The server will stop sending that output, and can also use the synch function to clear all outstanding data that is waiting in buffers to be sent to the client machine (since it causes data to be discarded).

KEY CONCEPT Telnet protocol commands are sent in the same stream with user data, which means a problem with the remote host that stops the flow of data might cause user commands to become backed up and never received by the host. Since this may include commands issued by the user to try to fix the problem on the host, this can be a serious problem. To alleviate this situation, Telnet includes the *synch function*, which uses TCP's urgent data transmission feature to force the receipt of essential commands, even when regular data is not being processed.

Telnet Options and Option Negotiation

The basic Telnet NVT specification solves the problem of compatibility between different terminal and computer types by defining a common representation for data and commands that every Telnet client and server uses. The price for this universal representation, however, is very high: All of the advanced or special capabilities of terminals and hosts are stripped off. The result is a language that everyone can speak but that is not capable of much more than basic conversation.

The creators of Telnet recognized that, while it was important to define NVT as a common base to ensure cross-device compatibility, it was also essential that some means be provided by which clients and servers could agree to use more advanced means of communication. They defined a set of *Telnet options* and a mechanism by which a Telnet client and server can *negotiate* which options they want to use.

Most Telnet options are used for improving the efficiency of how data is transferred between devices. For example, by default, the NVT assumes half-duplex operation with each device, requiring it to use the Go Ahead command after each transmission. However, virtually all hardware now supports full-duplex communication, so devices will usually agree to use the Suppress Go Ahead option to eliminate the need to send this character. Similarly, it is possible for devices to negotiate the sending of 8-bit binary data instead of the standard 7-bit ASCII of the Telnet NVT.

The process of Telnet option negotiation is described in the main Telnet standard document, RFC 854, as well as a companion document, RFC 855, “Telnet Option Specifications.” The options themselves are described in a separate set of Internet standards. Several of these were published at the same time as RFCs 854 and 855; others were defined earlier as part of previous versions of Telnet; and still

others have been added over the years. There are now several dozen different Telnet options in existence. A master list is maintained by Internet Assigned Numbers Authority (IANA), just as it maintains other TCP/IP parameters. An up-to-date listing of all Telnet options can be found on the IANA website at <http://www.iana.org/assignments/telnet-options>.

Common Telnet Options

Each Telnet option is identified using a decimal byte code with a possible value of 0 to 254. The value 255 is reserved to extend the option list should more than 255 options ever be needed. Each option also has a text code string associated with it, which is often used as a symbol in place of the code number in both protocol discussions and diagnostic output. Table 87-3 lists some of the more interesting Telnet options and provides a brief description of each.

Table 87-3: Common Telnet Options

Option Number	Option Code	Option Name	Description	Defining RFC
0	TRANSMIT-BINARY	Binary Transmission	Allows devices to send data in 8-bit binary form instead of 7-bit ASCII.	856
1	ECHO	Echo	Allows devices to negotiate any of a variety of different echo modes. (When you press a key on a terminal, you also expect to see the character you entered appear on the terminal screen as output; this is called <i>echoing</i> the input.)	857
3	SUPPRESS-GO-AHEAD	Suppress Go Ahead	Allows devices not operating in half-duplex mode to no longer need to end transmissions using the Telnet Go Ahead command.	858
5	STATUS	Status	Lets a device request the status of a Telnet option.	859
6	TIMING-MARK	Timing Mark	Allows devices to negotiate the insertion of a special timing mark into the data stream, which is used for synchronization.	860
10	NAOCRD	Output Carriage Return Disposition	Lets the devices negotiate how carriage returns will be handled.	652
11	NAOHTS	Output Horizontal Tab Stops	Allows the devices to determine what horizontal tab stop positions will be used for output display.	653
12	NAOHTD	Output Horizontal Tab Stop Disposition	Allows the devices to negotiate how horizontal tabs will be handled and by which end of the connection.	654
13	NAOFFD	Output Form Feed Disposition	Allows the devices to negotiate how form feed characters will be handled.	655
14	NAOVTS	Output Vertical Tab Stops	Used to determine what vertical tab stop positions will be used for output display.	656
15	NAOVTD	Output Vertical Tab Disposition	Lets devices negotiate the disposition of vertical tab stops.	657
16	NAOLFD	Output Line Feed Disposition	Allows devices to decide how line feed characters should be handled.	658

(continued)

Table 87-3: Common Telnet Options (continued)

Option Number	Option Code	Option Name	Description	Defining RFC
17	EXTEND-ASCII	Extended ASCII	Lets devices agree to use extended ASCII for transmissions and negotiate how it will be used.	698
24	TERMINAL-TYPE	Terminal Type	Allows the client and server to negotiate the use of a specific terminal type. If they agree, this allows the output from the server to be ideally customized to the needs of the particular terminal the user is using.	1091
31	NAWS	Negotiate About Window Size	Permits communication of the size of the terminal window.	1073
32	TERMINAL-SPEED	Terminal Speed	Allows devices to report on the current terminal speed.	1079
33	TOGGLE-FLOW-CONTROL	Remote Flow Control	Allows flow control between the client and the server to be enabled and disabled.	1372
34	LINEMODE	Line Mode	Allows the client to send data one line at a time instead of one character at a time. This improves performance by replacing a large number of tiny TCP transmissions with a smaller number of larger ones.	1184
37	AUTHENTICATION	Authentication	Lets the client and server negotiate a method of authentication to secure connections.	1416

KEY CONCEPT The Telnet NVT specification ensures that all devices using Telnet can talk to each other, but accomplishes this communication at the lowest level. To allow the use of more sophisticated formats and services, Telnet defines a number of *options*. If a client and server both implement a particular option, they can enable its use through a process of *negotiation*.

Telnet Option Negotiation

The first stage in Telnet option negotiation is for the client and server to decide whether they want to enable a particular option. One of the aspects of Telnet's symmetry of operation is that either device may choose to initiate the use of an option. The initiating device may either specify that it wants to start using an option or that it wants the other device to start using it. The responding device may agree or disagree. An option can be enabled only if both devices agree to its use.

This negotiation is performed using four Telnet protocol commands: WILL, WONT, DO, and DONT.

To specify that it wants to start using an option, the initiator sends the WILL command to the other device. There are two possible replies by the responding device:

DO Sent to indicate agreement that the initiator should use the option; it is then considered enabled.

DONT Sent to specify that the initiator must not use the option.

If the initiator wants the other device to start using an option, it sends the DO command. That device may respond in two ways:

WILL Sent to specify that the responding device will agree to use the option; the option is enabled.

WONT Sent to tell the initiator that the responder will not use the option requested.

The symmetry of Telnet and the fact that both DO and WILL can be used either to initiate a negotiation or respond to one make Telnet's option negotiation potentially complicated. Since either device can initiate negotiation of an option at any time, this could result in acknowledgment loops if both devices were to try to enable an option simultaneously or each kept responding to the other's replies. For this reason, the Telnet standard specifies restrictions on when the WILL and DO commands are used. One is that a device may send a negotiation command only to request a change in the status of an option; it cannot send DO or WILL just to confirm or reinforce the current state of the option. Another is that a device receiving a request to start using an option it is already using should not acknowledge it using DO or WILL.

Since an option may be activated only if both devices agree to use it, either may disable the use of an option at any time by sending one of these commands:

WONT Sent by a device to indicate that it is going to stop using an option. The other device must respond with DONT as a confirmation.

DONT Sent by a device to indicate that it wants the other device to stop using an option. The other device must respond with WONT.

KEY CONCEPT Either device may choose to negotiate the use of a Telnet option. The initiator uses the WILL command to specify that it wants to start using a particular option; if the other device agrees, it responds with DO; otherwise, it sends DONT. Alternatively, the initiator can use the DO command to indicate that it wants the other device to start using an option; that device responds with WILL if it agrees to do so or WONT if it does not. Either device may disable the use of an option at any time by sending the other a WONT or DONT command.

Option Subnegotiation

All of the DO/DONT/WILL/WONT negotiation just described serves only to enable or disable an option. Some options, such as the binary transmission option (TRANSMIT-BINARY), are either only off or on; in which case, this option negotiation is sufficient. Other options require that after they are enabled, the client and server exchange parameters to control how the option works. For example, the TERMINAL-TYPE option requires some way for the client to send the server the name of the terminal. Telnet allows the client and server to send an arbitrary amount of data related to the option using a process called *option subnegotiation*.

A device begins the subnegotiation process by sending a special sequence of Telnet protocol commands and data. First, the device sends the SB (subnegotiation) command, followed by the option number and parameters as defined by the particular option, and then ending the subnegotiation data by sending the SE (subnegotiation end) command. Both SB and SE must be preceded by the Interpret As Command (IAC) command byte.

Let's take the terminal type negotiation as an example. Suppose the server supports this option and would like the client to use it. The server starts option negotiation by sending the DO command:

IAC DO TERMINAL-TYPE

Assuming the client agrees, it will respond with the WILL command:

IAC WILL TERMINAL-TYPE

Now the terminal type option is in effect, but the server still doesn't know which terminal the client is using. It can prompt the client to provide that information by sending this command:

IAC SB TERMINAL-TYPE SEND IAC SE

The client receiving this option subnegotiation command will respond with the following:

IAC SB TERMINAL-TYPE IS <some_terminal_type> IAC SE

KEY CONCEPT The WILL and DO commands only turn on a Telnet option that a client and server agree to use. In some cases, an option requires additional information to be sent between the client and server device for it to function properly. This is accomplished through a process of *option subnegotiation*. Either device sends the other a set of data relevant to the option, bracketed by the SB (subnegotiation) and SE (subnegotiation end) Telnet protocol commands.

Berkeley Remote (r) Commands

TCP/IP has achieved success in large part due to its universality—it has been implemented on virtually every major computing platform. While the suite is thus not specific to any operating system, there is no denying that its history is closely tied to a particular one—UNIX. Most of the computers on the early Internet used UNIX, and the development of TCP/IP has paralleled that of UNIX in a number of respects.

One of the most important organizations involved in the development of UNIX, and thus TCP/IP indirectly, was the University of California at Berkeley (UCB). The well-known UCB-developed *Berkeley Software Distribution (BSD)* UNIX has been in widespread use for over 20 years. They also developed a set of commands for BSD UNIX to facilitate various remote operation functions over a TCP/IP internetwork. Each of these programs begins with the letter *r* (for remote), so they have come to be known as both the *Berkeley remote commands* (or utilities) and also

simply the *r commands*. Since their initial creation, they have been adopted for most variations of UNIX and some other operating systems as well.

BACKGROUND INFORMATION *This section will probably make much more sense to those who have some understanding of the UNIX operating system than those who do not.*

Berkeley Remote Login (*rlogin*)

The head of the Berkeley remote protocol family is the remote login command, *rlogin*. As the name clearly implies, the purpose of this program is to allow a user on a UNIX host to log in to another host over a TCP/IP internetwork. Since Telnet is also often used for remote login, *rlogin* and Telnet are sometimes considered alternatives to each other for TCP/IP remote login. While they can be used in a similar way, they are quite different in a few respects.

From a conceptual standpoint, Telnet is designed as a protocol to enable terminal/host communication. As I mentioned in the Telnet overview earlier in this chapter, that protocol was not designed specifically for the purpose of remote login. In contrast, *rlogin* was intended for that specific purpose, and this is reflected in its operation.

The protocol requires *rlogin* server software to be running on the host that is going to allow remote access; it is usually called *rlogind* (for *rlogin daemon*, the latter word being the standard UNIX term for a background server process). The server listens for incoming connection requests on TCP port 513. Users who want to remotely log in to the server run the *rlogin* command on their local host and specify the name of the server. The client makes a TCP connection to the server and then sends to the server a string containing the following information:

- The login name of the user on the client machine
- The login name that the user wants to use on the server (which is often the same as the user's login name on the client, but not always)
- Control information such as the type and speed of the terminal

The server processes this information and begins the login process. It will normally prompt the user for a password to log in to the remote host. Assuming the password is correct, the user will be logged in to the remote host and can use it as if the user were locally connected.

From a practical standpoint, the *rlogin* command is much simpler than Telnet; it does not support Telnet's full command structure, nor capabilities such as option negotiation. It does include a small set of commands, however. The client is able to send to the server one key piece of information: the current size of the terminal window in use. The server is able to tell the client to turn on or off flow control, request that the client send it the current window size, or ask the client to flush pending output that the server has sent, up to a certain point in the data stream.

Some organizations have many different UNIX hosts that are used every day, and needing to constantly type passwords when using *rlogin* can be somewhat of a chore. On these systems, it is possible for administrators to set up control files that specify combinations of host names, user names, and passwords. If set up correctly, this enables an authorized user to use *rlogin* to remotely access a host automatically, without needing to enter either a login name or password.

As originally designed, rlogin is a classic example of a protocol from the early days of TCP/IP, since it emphasizes simplicity and usability over security. This is especially true of the automated login process just described. The original schemes used by rlogin for authentication are considered inadequate for modern TCP/IP internetworks, especially those connected to the Internet. Later versions of rlogin have been enhanced with more secure authorization methods. There is also a newer program called slogin (for *secure login*) that uses stronger authentication and encryption, which is intended to replace rlogin on newer systems.

KEY CONCEPT The Berkeley remote, or *r*, commands facilitate remote operations between UNIX hosts on a TCP/IP internetwork. The base command of the family is the *remote login* command, rlogin, which allows a device on one host to access and use another as if it were locally connected to it. rlogin is often used as an alternative to Telnet. It is simpler than Telnet, both conceptually and practically.

Berkeley Remote Shell (rsh)

A user would normally use rlogin when he needs to log in to a server to perform a number of tasks. There are some situations, however, where a user needs to only enter one command on a remote host. With rlogin, the user would need to log in to the host, execute the command, and then log back out again. This isn't exactly an earth-shattering amount of inconvenience, especially when the correct configuration files are set up to allow automatic login. Over the course of time, however, all the extra logging in and out can become tedious. As a convenience, a variation of rlogin, called rsh (for *remote shell*), allows a user to access a remote host and execute a single command on it without requiring the login and logout steps.

NOTE *Shell* is the standard term used in UNIX to refer to the user interface that accepts commands from the user and displays output on the screen.

The rsh command is based on rlogin and works in much the same way, except that it is oriented around executing a command rather than establishing a persistent login session. The server process on the remote host is usually called rshd (for *remote shell daemon*) and listens for incoming rsh requests. When one is received, the user is logged in through the same mechanism as rlogin. The command runs on the remote host, and then the user is automatically logged out.

rsh is most useful when automatic login is employed, so that the program can be run without the need for the user to enter a login name or password. In that case, it is possible to have programs use rsh to automatically run commands on remote hosts without the need for human intervention, which opens up a number of possibilities for UNIX users. The normal UNIX user interface concepts of *standard input (stdin)*, *standard output (stdout)*, and *standard error (stderr)* also apply to rsh, so you can use it to execute a remote command and redirect the output to a local file. For example, the following command would let a user get a listing of his home directory on the host server and store it in the local file named remotelist:

```
rsh <somehost> ls -l >remotelist
```

KEY CONCEPT The `rsh` (remote shell) command is similar to the Berkeley `rlogin` command, but instead of opening a login session on a remote host, it executes a single, user-provided command. `rsh` can be helpful for users who need to perform a quick operation on a remote host, and it can also be employed by other programs to automate network tasks.

Since `rsh` is based on `rlogin`, all of the concerns that apply to `rlogin` are also relevant here, especially with regard to security. (We really don't want unauthorized users running commands on our servers!) As with `rlogin`, newer versions of `rsh` support more advanced authentication options than the original software. Also, just as `slogin` is a newer, more secure version of `rlogin`, there is a program called `ssh` (for *secure shell*) that replaces `rsh` on many systems.

NOTE *On some systems, if rsh is entered without a command specified to execute, an interactive remote session is established, exactly as if the rlogin command had been entered instead of rsh.*

Other Berkeley Remote Commands

The `rlogin` and `rsh` commands are the generic members of the Berkeley *r* family of programs that allow remote access to a host. To complement these, the developers also defined a small number of specific remote commands. These are essentially remote versions of some of the more common UNIX functions. Instead of the command being applied to only one system, however, it is used between two systems or across all systems on a TCP/IP network.

All of these commands are based on `rlogin` in the same way as `rsh` is. They work in the same way, but instead of opening a session or passing a user-specified command to the remote host, they execute a particular function. The following are the most common of these remote commands:

Remote Copy (rcp) This is the remote version of the UNIX copy (`cp`) command. It allows a file to be copied between the local host and the remote host or between two remote hosts. The usual syntax is basically the same as the regular `cp` command, but the source and/or destination is specified as being on a remote host. The `rcp` command can be used in a manner similar to FTP, but is much simpler and less capable. Or, to put it another way, `rcp` is to FTP what `rlogin` is to Telnet. (That's not a perfect analogy, but it's pretty close.)

Remote Uptime (ruptime) The UNIX command `uptime` displays how long a computer has been running since it was last booted, along with information related to its current load. `ruptime` is the remote version of this command; it displays the current status of each machine on the network (up or down), how long each up machine has been up since its last boot, and its load statistics.

Remote Who (rwho) This is the remote version of the `who` command. Where `who` shows all the users logged on to the host where it is run, `rwho` shows all users logged on to all machines on the network.

The `ruptime` and `rwho` commands both rely on the presence of the `rwhod` (for *remote who daemon*) running in the background on networked machines. These processes routinely share information with each other about host uptime and who is logged on to each system, so it can be quickly displayed when either `ruptime` or `rwho` is run.

On some operating systems, other remote commands may also be implemented. As with `rlogin` and `rsh`, security issues may apply to these commands, and there may be efficiency concerns with others (such as `rwho`). For these reasons, on many networks, these commands are no longer used.

Internet Relay Chat Protocol (IRC)

The primary advantage that electronic mail (email) offers over conventional mail is *speed*. Instead of needing to wait for days or weeks for a message to be delivered, it usually arrives in minutes or even seconds. This makes email far more useful than the regular postal service for most types of information transfer. There are some cases, however, where speed of delivery is not sufficient to make email an ideal mechanism for communication. One such case is where a *dialogue* is required between two parties.

Consider that even though email may be delivered very quickly, it uses a decoupled model of communication. Say that Ellen sends an email to Jane. The message may show up in Jane's inbox in a matter of seconds, but Jane may not be around to read it at the time it arrives. Jane might not see the message until hours later. Then Jane would send a response to Ellen, who might not see it for a while. If the subject they are discussing requires several dozen iterations of this sort, it could take a very long time before the exchange is completed.

In the real world, of course, most of us would never use email for such a conversation, preferring instead that high-tech communication device that we call the telephone. Many people using computers realized that it would be useful to have a way for two or more people to interactively discuss issues in a manner similar to a telephone conversation. In the online world, this is commonly called *chatting*, and one of the first and most important application protocols designed to implement it in TCP/IP was the *Internet Relay Chat (IRC) Protocol*.

Prior to the widespread use of the Internet, people with computers would often communicate by dialing in to a *bulletin board system (BBS)* or other proprietary service. IRC was originally created by a gentleman from Finland named Jarkko Oikarinen, based on his experience with chat applications on BBSes. He wrote the first client and server software in 1988. The protocol was later formally defined in RFC 1459, “Internet Relay Chat Protocol,” published May 1993. In April 2000, the IRC standard was revised and enhanced with several new extended capabilities, and published as a set of four smaller documents: RFCs 2810 through 2813. Each of these focuses on one particular area of IRC functionality.

NOTE RFC 1459 has the experimental RFC status, and the RFC 2810 to 2813 group is designated Informational. This makes IRC optional; it does not need to be implemented on TCP/IP devices.

IRC Communication Model and Client/Server Operation

IRC is an interesting protocol in that it is not based strictly on the standard client/server model of TCP/IP protocol operation. *IRC servers* are TCP/IP machines that run IRC server software. They are configured with information that allows them to establish TCP connections to each other. IRC uses TCP because the connections are maintained over a long period of time, and reliable transport of data is required. Server connections are used to exchange control information and user data, forming a logical *IRC network* at the application level, which allows any server to send to any other server, using intermediate servers as conduits. Servers are managed by *IRC operators (IRCsops)* who have special privileges that allow them to ensure that everything runs smoothly on the network.

The IRC network forms the backbone of the IRC communication service. A user can access the network by running *IRC client* software on any TCP/IP-enabled device. The user enters the name of one of the servers on the network and establishes a TCP connection to that server. This causes the user to be connected directly to one server, and thus, indirectly to all of the others on the network. This allows that user to send and receive messages to and from all other users connected either to the user's server or other servers.

Messaging and IRC Channels

The most common type of communication in IRC is *group messaging*, which is accomplished using *IRC channels*. A channel is a virtual meeting place of sorts and is also sometimes called a *chat room* (though IRC purists scoff at the use of that term). Every IRC network has hundreds or even thousands of different channels, each of which is dedicated to a particular type of discussion, ranging from the serious to the silly. For example, a group of people interested in talking about meteorology could establish a channel called #weather, where they would meet regularly to discuss various aspects of climatology and interesting weather events.

IRC is an inherently text-based protocol (though it is also possible to use IRC clients to transfer arbitrary files between users, including images and executable programs). To communicate in a channel, all a user needs to do is enter text in the appropriate spot in the IRC client program, and then the program automatically sends this text to every other member of the channel. The IRC network handles the relaying of these messages in real time from the sender's connected server to other servers in the network, and then to all user machines on those servers. When other users see the first user's message, they can reply with messages of their own, which will, in turn, be propagated across the network. Each IRC user chooses a nickname (often abbreviated *nick*) that is like a *handle* used for communication while connected to the network.

IRC also supports one-to-one communication, which can be used for private conversation. To use this method, a user just needs the nickname of another user to whom she wants to talk. She uses a special command to send messages directly to that user, who can respond in kind. This is not a secure form of communication, since the messages are not encrypted, and they pass through servers where they could be monitored. However, there is so much traffic on a typical IRC network that any given message is unlikely to be monitored.

The IRC Protocol defines a rich command set that allows users to perform essential functions, such as joining or leaving a channel, changing nicknames, changing servers, setting operating modes for channels, and so forth. The exact command set and features available depend both on the specific software used for the user's IRC client and the features available on the IRC network itself. Not all IRC networks run the same version of the protocol.

IRC and the Modern Internet

IRC became very popular in the early 1990s because of the powerful way that it allows users from anywhere on the Internet to meet and share information dynamically. It acts like a text-based telephone, but users across the globe don't have the expense of long-distance calls.

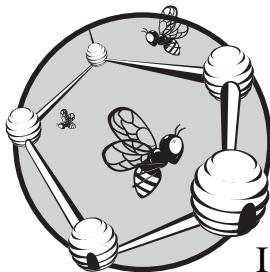
One of the most important characteristics of IRC is its open-ended nature; it gives every person the freedom to communicate in whatever way he or she considers best. For example, every IRC channel has an owner, who has certain rights related to how the channel is used, including the ability to decide who should be allowed in the channel. This may seem autocratic, but IRC lets anyone start a new channel instantly and become that channel's owner, without the need for prior registration or authorization. This means that if you don't like how a particular channel is run, you can start your own with a minimum of fuss. You are not forced to adhere to anyone's rules, other than the rules set forth for the server (which are usually just intended to prevent abuse).

This same principle extends to the IRC networks themselves. There isn't just one single IRC network; there are dozens of different ones. Some are large, well-established networks that may have more than 100 servers and thousands of users; others are smaller and devoted to specific areas of interest or geographical regions. Anyone can set up their own IRC network if they have the hardware and software, and some organizations have set up private, dedicated IRC servers for their own use.

IRC is considered by many to be the most important ancestor of the related interactive applications collectively known as *instant messaging*. These services are offered by several organizations, including America Online (AOL), Yahoo, and Microsoft's MSN. The idea behind them is very similar to that of IRC. Each allows a message sent by one user to be displayed immediately to another, though most are focused primarily on user-to-user messages rather than group messaging. Instant messaging has surpassed IRC in overall use, perhaps due to the large subscriber base of services like AOL. However, IRC is still widely used by thousands of enthusiasts on a daily basis for both entertainment and business purposes.

88

TCP/IP ADMINISTRATION AND TROUBLESHOOTING UTILITIES AND PROTOCOLS



This final chapter on application protocols is a bit different from the previous ones. It doesn't describe applications designed for end users. Rather, it discusses a set of TCP/IP troubleshooting utilities and protocols, which are normally the province of internetwork administrators. Even though millions of people use TCP/IP every day without even knowing that these applications exist—much less how they work—they are critically important to those who maintain TCP/IP internetworks. Since many of you are studying TCP/IP so that you can implement and administer this technology, understanding how these applications work is well worth your time.

In this chapter, I provide an overview of a number of software utilities that are commonly employed to help set up, configure, and maintain TCP/IP internetworks. These programs allow a network administrator to perform functions such as checking the identity of a host, verifying connectivity between two hosts, checking the path of routers between devices, examining the configuration of a computer, and looking up a Domain Name System (DNS) domain name.

The goal of this chapter is to provide explanations of the general purpose and function of troubleshooting utilities, so you will know how they can help you manage TCP/IP networks. As part of these descriptions, I demonstrate the typical syntax used to invoke each utility in both UNIX and Windows. Due to variations in software implementations, you will need to consult your operating system documentation for the details on exactly how each program should be used on your network. On Windows systems, try `<program> /?` to see the syntax of the program; on UNIX/Linux, try `man <program>`.

BACKGROUND INFORMATION Many of the software tools described in this section are designed to manage the operation of other TCP/IP protocols, such as the Internet Protocol (IP), the Domain Name System (DNS), and the Dynamic Host Configuration Protocol (DHCP). To fully appreciate how these utilities work, you need to understand the basics of these and other key TCP/IP protocols. In particular, a number of the utilities discussed here communicate use Internet Control Message Protocol (ICMP) messages, so I would recommend familiarity with ICMP (discussed in Part II-6) before proceeding.

TCP/IP Host Name Utility (`hostname`)

One of the most fundamental of tasks in diagnosing problems with a networked computer is identifying it. Just as the first thing we usually do when we meet someone is exchange names, one of the first actions an administrator takes when accessing a device is to determine its name, if it is not known. This is accomplished using the `hostname` utility.

You may recall from our discussion of TCP/IP name systems in Part III-1 that there are two different ways that hosts can be named. The first way is to manually assign flat names to devices using host tables or equivalent means; this is most often used for devices that not going to be accessed on the public Internet. The second is to give a device a domain name within DNS. The `hostname` utility can be used for both types of named hosts, but it functions in a slightly different way for each.

On most systems, including Windows and many UNIX implementations, the `hostname` utility is very simple. When you enter the command by itself on a line with no arguments, it displays the full name of the host. If it is entered with the `-s` (short) parameter and the host name is a fully qualified DNS domain name, only the local label of the node is shown and not the full domain name; if the host has a flat (non-DNS) name, the `-s` parameter has no effect. Here is a simple example:

```
% hostname
fearn.pair.com
% hostname -s
fearn
```

The `hostname` utility is also intended to allow an administrator to set the name of a host. The syntax for this is also simple; you just supply the name of the host as a parameter, as follows:

```
hostname <new-hostname>
```

However, in most implementations, the use of the `hostname` command for setting a device's name is either disabled or restricted. In Windows systems, a special applet in the Control Panel is used to set the device's name; attempting to set it using `hostname` will result in an error message. In UNIX, the superuser of the system can use `hostname` to set the device's name, but it is more common for this to be done by other means, such as editing the configuration file `/etc/hosts`. If a simple flat name is being assigned to this host, the administrator has full control over it. However, if DNS is used, then the proper procedures for registering the name must be followed.

NOTE *The `hostname` utility is not, strictly speaking, tied into the operation of DNS or other formal mechanisms for identifying a host. It simply displays what the administrator has set it to show. It makes sense for this to be set to the host's DNS name, but there may be exceptions, such as in small networks that might not use DNS.*

In most operating systems, the `-s` parameter is the only one that this command supports. The parameter is not supported on all implementations of the `hostname` command, however. On some implementations, if you use `hostname -s`, the system may report its host name as being `-s`. On certain Linux systems, the `hostname` utility includes a few additional parameters that allow different ways for the host name to be displayed, as well as some miscellaneous functions such as showing the version number of the program.

KEY CONCEPT The simplest and most basic of TCP/IP administrative utilities is `hostname`, which returns the name of the host on which it is run.

TCP/IP Communication Verification Utility (`ping`)

One of the most common problems that network administrators are asked to solve is that two hosts are not able to communicate. For example, a user on a corporate network might not be able to retrieve one of his files from a local server, or another user might be having difficulty loading her favorite website. In these and similar situations, one important step in diagnosing the problem is to verify that basic communication is possible between the TCP/IP software stacks on the two machines. This is most often done using the `ping` utility, or `ping6` in Internet Protocol version 6 (IPv6) implementations. The IPv6 version of `ping` works in much the same way as IPv4 `ping`, but `ping6`'s options and parameters reflect the changes made in addressing and routing in IPv6.

NOTE *Some people say that `ping` is an acronym for Packet Internet Groper, while others insist that it is actually based on the use of the term to refer to a sonar pulse sent by a submarine to check for nearby objects. I really don't know which of these is true, but I prefer the second explanation. Consider that the utility works in a way similar to a sonar ping, and that it was originally written by a gentleman named Mike Muuss, who worked at the United States Army Ballistics Research Laboratory.*

`ping` is one of the most commonly used diagnostic utilities, and it is present in just about every TCP/IP implementation. It is usually implemented and accessed as a command-line utility, though there are also now graphical and menu-based versions of the program on some operating systems.

Operation of the ping Utility

The `ping` utility is implemented using Internet Control Message Protocol (ICMP) Echo (Request) and Echo Reply messages, which are designed specifically for this type of diagnostic use. When Device A sends an ICMP Echo message to Device B, Device B responds by sending an ICMP Echo Reply message back to Device A. The same functionality exists in ICMPv6, the IPv6 version of ICMP; the ICMPv6 Echo and Echo Reply messages differ from the IPv4 ones only slightly in their field structure.

This would seem to indicate that `ping` would be an extremely simple utility that would send one Echo message and wait to see if an Echo Reply was received back. If so, this would mean that the two devices were able to communicate; if not, this would indicate a problem somewhere on the internetwork between the two. However, almost all `ping` implementations are much more complex than this. They use multiple sets of Echo and Echo Reply messages, along with considerable internal logic, to allow an administrator to determine all of the following, and more:

- Whether or not the two devices can communicate
- Whether congestion or other problems exist that might allow communication to succeed sometimes but cause it to fail in others, seen as packet loss; if so, how bad the loss is
- How much time it takes to send a simple ICMP message between devices, which gives an indication of the overall latency between the hosts and also indicates if there are certain types of problems

Basic Use of ping

The most basic use of the `ping` command is to enter it by itself with the IP address of a host. Virtually all implementations also allow you to use a host name, which will be resolved to an IP address automatically. When you invoke the utility with no additional options, it uses default values for parameters such as what size message to send, how many messages to be sent, how long to wait for a reply, and so on. The utility will transmit a series of Echo messages to the host and report back whether or not a reply was received for each. If a reply is seen, it will also indicate how long it took for the response to be received. When the program is finished, it will provide a statistical summary showing what percentage of the Echo messages received a reply and the average amount of time it took for them to be received.

NOTE *While the inability to get a response from a device to a ping has traditionally been interpreted as a problem in communication, this is not always necessarily the case. In the current era of increased security consciousness, some networks are set up to not respond to Echo messages, to protect against attacks that use floods of such messages. In this case, a ping will fail, even though the host may be quite reachable.*

Listing 88-1 shows an example of using the `ping` command on a Windows XP computer (mine!), which, by default, sends four 32-byte Echo messages and allows four seconds before considering an Echo message lost. I use a satellite Internet connection that has fairly high latency and also occasionally drops packets. This isn't great for me, but it is useful for illustrating how `ping` works.

```
D:\aa>ping www.pcguide.com
Pinging pcguide.com [209.68.14.80] with 32 bytes of data:

Reply from 209.68.14.80: bytes=32 time=582ms TTL=56
Reply from 209.68.14.80: bytes=32 time=601ms TTL=56
Request timed out.
Reply from 209.68.14.80: bytes=32 time=583ms TTL=56

Ping statistics for 209.68.14.80:
    Packets: Sent = 4, Received = 3, Lost = 1 (25% loss),
Approximate round trip times in milli-seconds:
    Minimum = 582ms, Maximum = 601ms, Average = 588ms
```

***Listing 88-1:** Verifying communication using the ping utility*

Methods of Diagnosing Connectivity Problems Using ping

Most people find that using ping with default settings is enough for their needs. In fact, the utility can be used in this simplest form to perform a surprising number of diagnostic checks. In many cases, you can use the ping command to diagnose connectivity problems by issuing it multiple times in sequence, often starting with checks at or close to the transmitting device and then proceeding outward toward the other device with which the communication problem has been observed. Here are some examples of how ping can be used in this way:

Internal Device TCP/IP Stack Operation By performing a ping on the device's own address, you can verify that its internal TCP/IP stack is working. This can also be done using the standard IP loopback address, 127.0.0.1.

Local Network Connectivity If the internal test succeeds, it's a good idea to do a ping on another device on the local network, to verify that local communication is possible.

Local Router Operation If there is no problem on the local network, it makes sense to ping whatever local router the device is using to make sure it is operating and reachable.

Domain Name Resolution Functionality If a ping performed on a DNS domain name fails, you should try it with the device's IP address instead. If that works, this implies either a problem with domain name configuration or resolution.

Remote Host Operation If all the preceding checks succeed, you can try performing a ping to a remote host to see if it responds. If it does not, you can try a different remote host. If that one works, it is possible that the problem is actually with the first remote device itself and not with your local device.

KEY CONCEPT The TCP/IP ping utility is used to verify the ability of two devices on a TCP/IP internetwork to communicate. It operates by having one device send ICMP Echo (Request) messages to another, which responds with Echo Reply messages. The program can be helpful in diagnosing a number of connectivity issues, especially if it is used to test the ability to communicate with other devices in different locations. It also allows the average round-trip delay to exchange messages with another device to be estimated.

ping Options and Parameters

In addition to the basic uses described in the previous sections, all ping implementations include a number of options and parameters that allow an administrator to fine-tune how it works. They allow ping to be used for more extensive or specific types of testing. For example, ping can be set in a mode where it sends Echo messages continually, to check for an intermittent problem over a long period of time. You can also increase the size of the messages sent or the frequency with which they are transmitted, to test the ability of the local network to handle large amounts of traffic.

As with the other utilities described in this chapter, the exact features of the ping program are implementation-dependent. Even though UNIX and Windows systems often include many of the same options, they usually use completely different option codes. Table 88-1 shows some of the more important options that are often defined for the utility on many UNIX systems, and where appropriate, the parameters supplied with the option. Table 88-2 shows ping options for a typical Windows system.

Table 88-1: Common UNIX ping Utility Options and Parameters

Option/ Parameters	Description
<code>-c <count></code>	Specifies the number of Echo messages that should be sent.
<code>-f</code>	Flood mode; sends Echo packets at high speed to stress test a network. This can cause serious problems if not used carefully!
<code>-i <wait-interval></code>	Tells the utility how long to wait between transmissions.
<code>-m <ttl-value></code>	Overrides the default Time to Live (TTL) value for outgoing Echo messages.
<code>-n</code>	Numeric output only; suppresses lookups of DNS host names to save time.
<code>-p <pattern></code>	Allows a byte pattern to be specified for inclusion in the transmitted Echo messages. This can be useful for diagnosing certain odd problems that may occur only with certain types of transmissions.
<code>-q</code>	Quiet output; only summary lines are displayed at the start and end of the program's execution, while the lines for each individual message are suppressed.
<code>-R</code>	Tells the utility to include the Record Route IP option, so the route taken by the ICMP Echo message can be displayed. This option is not supported by all implementations. Using the traceroute utility (described in the next section) is usually a better idea.
<code>-s <packet-size></code>	Specifies the size of outgoing message to use.
<code>-S <src-addr></code>	On devices that have multiple IP interfaces (addresses), allows a ping sent from one interface to use an address from one of the others.
<code>-t <timeout></code>	Specifies a timeout period, in seconds, after which the ping utility will terminate, regardless of how many requests or replies have been sent or received.

Table 88-2: Common Windows ping Utility Options and Parameters

Option/ Parameters	Description
-a	If the target device is specified as an IP address, forces the address to be resolved to a DNS host name and displayed.
-f	Sets the Don't Fragment bit in the outgoing datagram.
-i <ttl-value>	Specifies the Time to Live (TTL) value to be used for outgoing Echo messages.
-j <host-list>	Sends the outgoing messages using the specified loose source route.
-k <host-list>	Sends the outgoing messages using the indicated strict source route.
-l <buffer-size>	Specifies the size of the data field in the transmitted Echo messages.
-n <count>	Tells the utility how many Echo messages to send.
-r <count>	Specifies the use of the Record Route IP option and the number of hops to be recorded. It's usually preferable to use the traceroute utility (described in the next section).
-s <count>	Specifies the use of the IP Timestamp option to record the arrival time of the Echo and Echo Reply messages.
-t	Sends Echo messages continuously until the program is interrupted.
-w <timeout>	Specifies how long the program should wait for each Echo Reply before giving up, in milliseconds (default is 4,000, for 4 seconds).

TCP/IP Route Tracing Utility (traceroute)

The ping utility is extremely helpful for checking whether two devices are able to communicate with each other. However, it provides very little information regarding what is going on between those two devices. In the event that ping shows either a total inability to communicate or intermittent connectivity with high loss of transmitted data, administrators need to know more about what is happening to IP datagrams as they are carried across the internetwork. This is especially important when the two devices are far from each other, especially if you are trying to reach a server on the public Internet.

I described in my overview of IP datagram delivery that when two devices are not on the same network, data sent between them must be delivered from one network to the next until it reaches its destination. This means that any time data is sent from Device A on one network to Device B on another, it follows a route, which may not be the same for each transmission. When communication problems arise, it is very useful to be able to check the specific route taken by data between two devices. A special route tracing utility is provided for this function, called traceroute (abbreviated tracert in Windows systems, a legacy of the old eight-character limit for DOS program names).

The IPv6 equivalent of this program is called traceroute6, which functions in a very similar manner to its IPv4 predecessor. It obviously uses IPv6 datagrams instead of IPv4 ones, and responses from traced devices are in the form of ICMPv6 Time Exceeded and Destination Unreachable messages rather than their ICMPv4 counterparts.

Operation of the traceroute Utility

Like the ping utility, traceroute is implemented using ICMP messages. However, unlike ping, traceroute was not originally designed to use a special ICMP message type intended exclusively for route tracing. Instead, it makes clever use of the IP and ICMP features that are designed to prevent routing problems.

Recall that the IP datagram format includes a Time to Live (TTL) field. This field is set to the maximum number of times that a datagram may be forwarded before it must be discarded; it exists to prevent datagrams from circling an internetwork endlessly. If a datagram must be discarded due to expiration of the TTL field, the device that discards it is supposed to send an ICMP Time Exceeded message back to the device that sent the discarded datagram. (This is explained in detail in Chapter 32.) Under normal circumstances, this occurs only when there is a problem, such as a router loop or another misconfiguration issue. What traceroute does is to force each router in a route to report back to it by intentionally setting the TTL value in test datagrams to a value too low to allow them to reach their destination.

Suppose you have Device A and Device B, which are separated by Routers R1 and R2—three hops total (A to R1, R1 to R2 and R2 to B). If you do a traceroute from Device A to Device B, here's what happens (see Figure 88-1):

1. The traceroute utility sends a dummy User Datagram Protocol (UDP) message (sometimes called a *probe*) to a port number that is intentionally selected to be invalid. The TTL field of the IP datagram is set to 1. When Router R1 receives the message, it decrements the field, which will make its value 0. That router discards the probe and sends an ICMP Time Exceeded message back to Device A.
2. Device A sends a second UDP message with the TTL field set to 2. This time, Router R1 reduces the TTL value to 1 and sends it to Router R2, which reduces the TTL field to 0 and sends a Time Exceeded message back to Device A.
3. Device A sends a third UDP message, with the TTL field set to 3. This time, the message will pass through both routers and be received by Device B. However, since the port number was invalid, the message is rejected by Device B, which sends back a Destination Unreachable message to Device A.

So Device A sends out three messages to Device B, and it gets back three error messages and is happy about it! The route to Device B is thus indicated by the identities of the devices sending back the error messages, in sequence. By keeping track of the time between when it sent each UDP message and received back the corresponding error message, the traceroute utility can also display how long it took to communicate with each device. In practice, usually three dummy messages are sent with each TTL value, so their transit times can be averaged by the user if desired.

NOTE *Not all traceroute utility implementations use the technique described here. Microsoft's tracert works by sending ICMP Echo messages with increasing TTL values, rather than UDP packets. It knows it has reached the final host when it gets back an Echo Reply message. A special ICMP Traceroute message was also developed in 1993, which was intended to improve the efficiency of traceroute by eliminating the need to send many UDP messages for each route tracing. Despite its technical advantages, since this message was introduced long after TCP/IP was widely deployed, it never became a formal Internet standard and its use is not seen as often as the traditional method.*

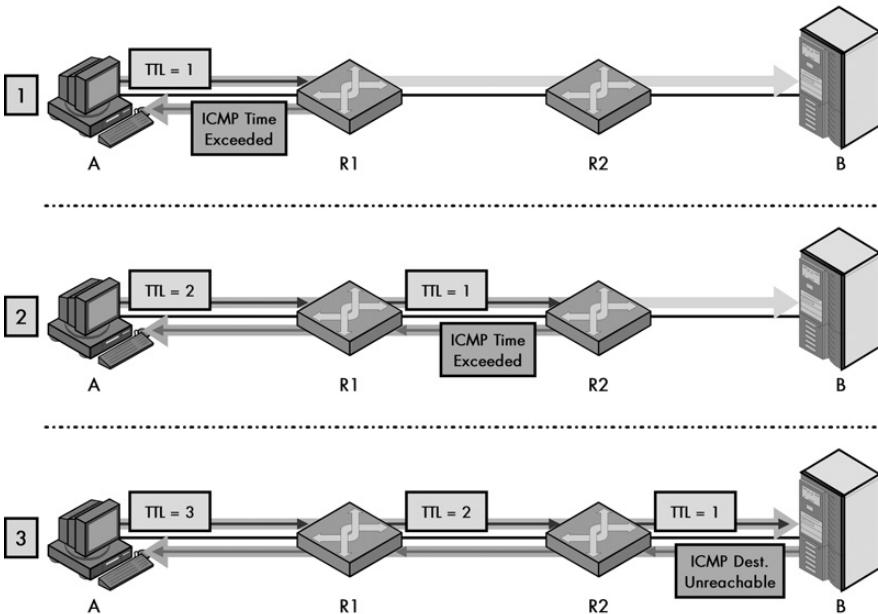


Figure 88-1: Operation of the traceroute/tracert utility The traceroute utility identifies the devices in a route by forcing them to report back failures to route datagrams with parameters intentionally set to invalid values. The first message sent by Device A here has a Time to Live (TTL) value of 1, which will cause Router R1 to drop it and send an ICMP Time Exceeded message back to Device A. The second message has a TTL value of 2, so it will be dropped and reported by Router R2. The third message will pass both routers and get to the destination host, Device B, but since the message is deliberately chosen with a bogus port number, this will cause an ICMP Destination Unreachable message to be returned. These error messages identify the sequence of devices in the route between Devices A and B.

KEY CONCEPT The traceroute utility takes the idea behind ping one step further, allowing administrators to not only check communication between two devices, but also letting them see a list of all the intermediate devices between the pair. It works by having the initiating host send a series of test datagrams with TTL values that cause each to expire sequentially at each device on the route. The traceroute program also shows how much time it takes to communicate with each device between the sending host and a destination device.

Basic Use of the traceroute Utility

Listing 88-2 shows an example of a traceroute sent between two of the UNIX computers I use on a regular basis. I added the -q2 parameter to change the default of three dummy messages per hop to two, so the output would fit better on the page.

```
traceroute -q2 www.pcguide.com
traceroute to www.pcguide.com (209.68.14.80), 40 hops max, 40 byte packets
 1 cisco0fe0-0-1.bf.sover.net (209.198.87.10) 1.223 ms 1.143 ms
 2 cisco1fe0.bf.sover.net (209.198.87.12) 1.265 ms 1.117 ms
 3 cisco0a5-0-102.wnskvtao.sover.net (216.114.153.170) 8.004 ms 7.270 ms
 4 207.136.212.234 (207.136.212.234) 7.163 ms 7.601 ms
 5 sl-gw18-nyc-2-0.sprintlink.net (144.232.228.145) 15.948 ms 20.931 ms
```

```

6 sl-bb21-nyc-12-1.sprintlink.net (144.232.13.162) 21.578 ms 16.324 ms
7 sl-bb27-pen-12-0.sprintlink.net (144.232.20.97) 18.296 ms *
8 sl-bb24-pen-15-0.sprintlink.net (144.232.16.81) 18.041 ms 18.338 ms
9 sl-bb26-rly-0-0.sprintlink.net (144.232.20.111) 20.259 ms 21.648 ms
10 sl-bb20-rly-12-0.sprintlink.net (144.232.7.249) 132.302 ms 37.825 ms
11 sl-gw9-rly-8-0.sprintlink.net (144.232.14.22) 23.085 ms 20.082 ms
12 sl-exped4-1-0.sprintlink.net (144.232.248.126) 43.374 ms 42.274 ms
13 * *
14 pcguide.com (209.68.14.80) 41.310 ms 49.455 ms

```

Listing 88-2: Route tracing using the traceroute utility

In this case, the servers are separated by 14 hops. Notice how the elapsed time generally increases as the distance from the transmitting device increases, but it is not consistent because of random elements in the delay between any two devices (see the incongruously large value in hop 10, for example). Also notice the asterisk (*) in the seventh hop, which means that no response was received before the timeout period for the second transmission with a TTL value of 7. Finally, there is no report at all for hop 13. This machine may have been configured not to send Time Exceeded messages.

Additional unusual results may be displayed under certain circumstances. For example, the traceroute program may display a code such as !H, !N, or !P to indicate receipt of an unexpected Destination Unreachable message for a host, network, or protocol, respectively. Other error messages may also exist, depending on the implementation.

traceroute Options and Parameters

As is the case with ping, traceroute can be used with an IP address or host name. If no parameters are supplied, default values will be used for key parameters. On the system I use, the defaults are three probes for each TTL value, a maximum of 64 hops tested, and packets 40 bytes in size. However, my implementation also supports a number of options and parameters to give me more control over how the utility functions (such as the -q parameter I used in Listing 88-2). Some of the typical options available in UNIX systems are described in Table 88-3. A smaller set of options exists in Windows, as shown in Table 88-4.

Table 88-3: Common UNIX traceroute Utility Options and Parameters

Option/Parameters	Description
-g <host-list>	Specifies a source route to be used for the trace.
-M <initial-ttl-value>	Overrides the default value of 1 for the initial TTL value of the first outgoing probe message.
-m <max-ttl-value>	Sets the maximum TTL value to be used. This limits how long a route the utility will attempt to trace.
-n	Displays the route using numeric addresses only, rather than showing both IP addresses and host names. This speeds up the display by saving the utility from needing to perform reverse DNS lookups on all the devices in the route [ICMP messages use IP addresses, not domain names].

(continued)

Table 88-3: Common UNIX traceroute Utility Options and Parameters (continued)

Option/Parameters	Description
-p <port-number>	Specifies the port number to be used as the destination of the probe messages.
-q <queries>	Tells the utility how many probes to send to each device in the route (the default is 3).
-r	Tells the program to bypass the normal routing tables and send directly to a host on an attached network.
-s <src-addr>	On devices that have multiple IP interfaces (addresses), allows the device to use an address from one interface on a traceroute using another interface.
-S	Instructs the program to display a summary of how many probes did not receive a reply.
-v	Sets verbose output mode, which informs the user of all ICMP messages received during the trace.
-w <wait-time>	Specifies how long the utility should wait for a reply to each probe, in seconds (the typical default is 3 to 5).

Table 88-4: Common Windows tracert Utility Options and Parameters

Option/Parameters	Description
-d	Displays the route using numeric addresses only, rather than showing both IP addresses and host names, for faster display. This is the same as the -n option on UNIX systems.
-h <maximum-hops>	Specifies the maximum number of hops to use for tracing (the default is 30).
-j <host-list>	Sends the outgoing probes using the specified loose source route.
-w <wait-time>	Specifies how long to wait for a reply to each probe, in milliseconds (the default is 4,000, or 4 seconds).

TCP/IP Address Resolution Protocol Utility (arp)

All devices on an internetwork are considered to be virtually connected at layer 3, since the process of routing lets any device communicate with any other device. However, there is no way for devices on distant networks to communicate directly. The internetwork communication at layer 3 actually consists of a number of steps, called *hops*, that carry the data from its source to destination. Each hop in a route requires that data be sent between a pair of hardware devices, and each transmission must use layer 2 hardware addresses. Since TCP/IP uses layer 3 addresses, this means each hop requires that we translate the IP address of the target of the hop to a hardware address. This is called *address resolution*; the reasons why it is needed and the methods used for it are explained in detail in Chapter 13.

In TCP/IP, address resolution functions are performed by the aptly named Address Resolution Protocol (ARP). When a device needs to transmit to a device with a particular IP address, it can use ARP's request/reply messaging protocol to find out which hardware device corresponds to that IP address. However, each such message exchange takes time and network bandwidth, so for efficiency, every device maintains an ARP cache, which is a table containing mappings between IP and hardware addresses. The ARP cache table can contain a combination of static cache entries that are manually inserted for frequently accessed devices, and

dynamic entries, which are entered automatically when a request/reply resolution is done. The next time it is necessary to send a device mapped in the ARP cache table, the lookup process can be avoided.

To allow administrators to manage this ARP cache table, TCP/IP devices include an arp utility. It has the following three basic functions, which are invoked using three different versions of the command (which, for once, are the same in UNIX and Windows):

ARP Cache Table Display When the -a option is used with the utility, it displays the current contents of the ARP cache table. The syntax is `arp -d <host-name>`. Each entry in the table shows the IP address and hardware address pair for one device (interface, actually). Usually, it also indicates whether each entry is static or dynamic. The exact format of the display varies from one implementation to the next; some programs show IP addresses, others show host names, and still others may show both. Some systems default to displaying host names but allow the -n option to also be used to force only IP addresses (not names) to be displayed.

ARP Cache Table Entry Addition This version allows an administrator to make a new manual ARP cache table entry that maps the given host name to the specified hardware address. The syntax is `arp -s <host-name> <hw-addr>`.

ARP Cache Table Entry Deletion Using arp with the -d option removes the specified cache entry from the table. Some implementations allow the addition of another parameter to specify that all entries should be removed from the cache. The basic syntax is `arp -d <host-name>`.

KEY CONCEPT The TCP/IP arp utility is used by an administrator to inspect or modify a host's ARP cache table, which contains mappings between TCP/IP host names and IP addresses.

Certain versions of the software may also supplement these basic commands with additional features. One common additional option on UNIX systems is the ability to specify a file from which cache table entries may be read, using the syntax `arp -f <file-name>`. This saves a considerable amount of time and effort compared to typing each entry manually using `arp -s`.

Note also that the operating system may allow only authorized users to access options that cause the ARP cache table to be changed. This is especially true of the delete function.

TCP/IP DNS Name Resolution and Lookup Utilities (`nslookup`, `host`, and `dig`)

DNS is a critically important part of TCP/IP internetworks, especially the modern Internet, because it allows hosts to be accessed using easily remembered names rather than confusing numerical addresses. Two different primary types of devices are involved in the operation of DNS: DNS name servers that store information about domains and DNS resolvers that query DNS servers to transform names into addresses, as well as perform other necessary functions.

DNS resolvers are employed by Internet users on a continual basis to translate DNS names into address, but under normal circumstances, they are always invoked indirectly. Each time a user types a DNS name into a program such as a web browser or File Transfer Protocol (FTP) client—or even uses it in one of the other utilities described in this chapter, such as ping or traceroute—the resolver automatically performs the name resolution without the user having to ask. For this reason, there is no need for users to manually resolve DNS names into addresses.

However, administrators often do need to perform a DNS resolution manually. For example, when troubleshooting a problem, the administrator may know a host's name but not its address. In the case of a security problem, the address may show up in a log file but the host name may not be known. In addition, even though users do not need to know the specifics of the resource records that define a DNS domain, administrators often need to be able to check these details, to make sure a domain is set up properly. Administrators also need some way to be able to diagnose problems with DNS servers themselves. To support all of these needs, modern TCP/IP implementations come equipped with one or more DNS name resolution and information lookup utilities. Here, we will look at three such utilities: nslookup, host, and dig.

The nslookup Utility

One of the most common DNS diagnostic utilities is nslookup (for name server lookup), which has been around for many years. The details of how the program is implemented depend on the operating system, though most of them offer versions that are quite similar in operation and settings. The utility can normally be used in two modes: interactive or noninteractive.

The noninteractive version of nslookup is the simplest, and it is most often used when an administrator wants to just quickly translate a name into an address or vice versa. To run this version, issue the nslookup command using the following simple syntax:

```
nslookup <host> [<server>]
```

Here, *<host>* can be a DNS domain name, for performing a normal resolution, or it may be an IP address, for a reverse resolution to return the associated DNS domain name. The *<server>* parameter is optional; if it's omitted, the program uses the default name server of the host where the command was issued. Listing 88-3 shows a simple example of noninteractive use of nslookup.

```
D:\aa>nslookup www.pcguide.com
Server: ns1-mar.starband.com
Address: 148.78.249.200

Non-authoritative answer:
Name: pcguide.com
Address: 209.68.14.80
Aliases: www.pcguide.com
```

Listing 88-3: DNS name resolution using the nslookup utility

This example was done on my home PC that uses the Starband satellite Internet service, which is configured to use Starband's name server (`ns1-mar.starband.com`). The answer provided here is labeled *non-authoritative*, because it came from the Starband name server's DNS cache, rather than one of the DNS name servers that is a DNS authority for `www.pcguide.com`.

NOTE *It is also possible to specify one or more options to modify the behavior of the lookup in noninteractive mode. These options are the same as the parameters controlled by the nslookup set command described in Table 88-5. They are specified by preceding them with a dash. For example, nslookup -timeout=10 www.pcguide.com would perform the same lookup as in Listing 88-3, but with the timeout interval set to 10 seconds.*

The interactive mode of nslookup is selected by simply issuing the name of the command with no parameters. This will cause the program to display the current default name server's DNS name and address, and then provide a prompt at which the administrator may enter commands. Interactive mode allows someone to perform multiple lookups easily without having to type nslookup each time. More important, it provides more convenient control over the types of information that can be requested and how the lookups are performed.

You can usually determine the exact command set available in an nslookup implementation by issuing the command help or ? at the nslookup prompt. Table 88-5 shows some of the commands that are usually found in most nslookup implementations.

Table 88-5: Typical nslookup Utility Commands

Command and Parameters	Description
<code><host> [<server>]</code>	Look up the specified host, optionally using the specified DNS name server. Note that there is no actual command here; you just enter the name directly at the command prompt.
<code>server <server></code>	Change the default server to <code><server></code> , using information obtained from the current default server.
<code>lserver <server></code>	Change the default server to <code><server></code> , using information obtained from the initial name server; that is, the system's default server that was in place when the nslookup command was started (prior to any preceding changes of the current name server in this session).
<code>root</code>	Changes the default name server to one of the DNS root name servers.
<code>ls [-t <type>] <name></code>	Requests a list of information available for the specified domain name, by conducting a zone transfer. By default, the host names and addresses associated with the domain are listed; the <code>-t</code> option may be used to restrict the output to a particular record type. Other options may also be defined. (Most servers restrict the use of zone transfers to designated slave servers, so this command may not work for ordinary clients.)
<code>help</code>	Displays help information (usually a list of valid commands and options).
<code>?</code>	Same as help (works on only some systems).
<code>set all</code>	Displays the current value of all nslookup options.
<code>set <option>[=<value>]</code>	Sets an option to control the behavior of the utility. Most implementations include quite a number of options, some of which are controlled by just specifying a keyword, while others require a value for the option. For example, <code>set recurse</code> tells the program to use recursive resolution, while <code>set norecurse</code> turns it off. <code>set retry=3</code> sets the number of retries to 3.
<code>exit</code>	Quits the program.

The nslookup utility is widely deployed on both UNIX and Windows systems, but the program is not without its critics. The complaints about it mainly center around its use of nonstandard methods of obtaining information, rather than standard resolution routines. I have also read reports that it can produce spurious results in some cases. One example of a significant problem with the command is that it will abort if it is unable to perform a reverse lookup of its own IP address. This can cause confusion, because users mistake that error for an error trying to find the name they were looking up. For this and other reasons, a number of people in UNIX circles consider nslookup to be a hack of sorts. In some newer UNIX systems, nslookup has been deprecated (still included in the operating system for compatibility, but not recommended and may be removed in the future). Instead, a pair of newer utilities is provided: host and dig.

The host Utility

The host utility is most often used for simple queries such as those normally performed using nslookup's noninteractive mode. It is invoked in the same way as noninteractive nslookup:

```
host <host> [<server>]
```

The output is also similar to that of noninteractive nslookup, but less verbose. Here is an example:

```
%host www.pcguide.com
www.pcguide.com is an alias for pcguide.com.
pcguide.com has address 209.68.14.80
```

Even though host does not operate interactively, it includes a number of options that can allow an administrator to get the same information that would have been obtained using nslookup's interactive mode. Some of the more common options are shown in Table 88-6.

Table 88-6: Typical host Utility Options and Parameters

Option/ Parameters	Description
-d	Turns on debug mode.
-l	Provides a complete list of information for a domain; this is similar to the ls command in interactive nslookup. This may be used with the -t option to select only a particular type of resource record for the domain.
-r	Disables recursion in the request. When this is specified, only the server directly queried will return any information; it will not query other servers.
-t <query-type>	Specifies a query for a particular resource record type, allowing any type of DNS information to be retrieved.
-v	Uses verbose mode for output (additional details are provided).
-w	Waits as long as necessary for a response (no timeout).

The dig Utility

The second alternative to nslookup is dig, which stands for Domain Information Groper (likely a play on the supposed origin of the name ping). It differs from the host command in that it provides considerably more information about a domain, even when invoked in the simplest of ways. It is also quite a bit more complicated, with a large number of options and features, such as a batch mode for obtaining information about many domains.

The basic syntax for the dig command is different from that of nslookup and host. If you specify a nondefault name server, it is prepended with an at sign (@) and comes before the host to be looked up. You can also specify a specific type of resource record, like this:

```
dig [<@server>] <host> [<type>]
```

Listing 88-4 shows the output from running dig on the same domain (www.pcguide.com) that I used as an example for nslookup (Listing 88-3) and host. You can see that it provides much more information about the domain.

```
%dig www.pcguide.com
; <>> DiG 9.2.1 <>> www.pcguide.com
;; global options: printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 15912
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 2, ADDITIONAL: 0
;; QUESTION SECTION:
;www.pcguide.com.           IN      A

;; ANSWER SECTION:
www.pcguide.com.        3600    IN      CNAME   pcguide.com.
pcguide.com.            3600    IN      A       209.68.14.80

;; AUTHORITY SECTION:
pcguide.com.          3600    IN      NS      ns0.ns0.com.
pcguide.com.          3600    IN      NS      ns23.pair.com.

;; Query time: 1840 msec
;; SERVER: 209.68.1.87#53(209.68.1.87)
;; WHEN: Tue Nov 18 16:05:08 2003
;; MSG SIZE  rcvd: 109Server: ns1-mar.starband.com
```

Listing 88-4: DNS name resolution using the nslookup utility

NOTE The dig utility is very useful, but has still not been implemented on some systems. Fortunately, there is an online dig utility you can access using your browser on the Internet. Find it at <http://www.gont.com.ar/tools/dig>.

The dig command includes dozens of options and settings. Since this chapter is already getting very long and dig is by far the most advanced of the three utilities, I will stop here. Consult your system's documentation for the full instructions on how dig works and a list of its parameters.

KEY CONCEPT Most TCP/IP implementations provide one or more utilities that can be employed by an administrator to manually resolve DNS domain names to IP addresses or perform related searches for DNS information. One of the most common is nslookup, which allows a host name to be translated to an address or vice versa; it has both interactive and noninteractive modes. On some operating systems, nslookup has been replaced by the host utility for simple DNS lookups and by the dig program for more detailed inspections of DNS resource information.

TCP/IP DNS Registry Database Lookup Utility (**whois/nickname**)

Utilities such as nslookup and host allow administrators to resolve a DNS domain name to an address and also view detailed information about a domain's resource records. There are cases, however, where administrators need to know its DNS registration information, rather technical information about a domain. This includes details such as which organization owns the domain, when its registration expires, and who are the designated contacts who manage it.

In the early days of DNS, all domain names were centrally registered by a single authority, called the Internet Network Information Center (InterNIC or just NIC). To allow Internet users to look up information about domains and contacts, InterNIC set up a special server. To allow users to retrieve information from this server, developers created a protocol called both **nickname** and **whois**. It was initially described in RFC 812 (in 1982) and then later in RFC 943 (in 1985). Over time, the name **whois** has become the preferred of the two, and it is the one used today for the utility program that allows an administrator to look up DNS registration data. (It can also be used to look up information about IP addresses, but is used for that purpose much less commonly.)

As the Internet grew and expanded, it moved away from having a single centralized authority. The modern Internet has a hierarchical structure of authorities that are responsible for registering domain names in different portions of the DNS name space. In recent years, this has been further complicated by the deregulation process that allows multiple registries for the generic top-level domains such as .COM, .NET, and .ORG. All of this means that more work is needed to look up domain registration information, since it is distributed across many databases on different servers.

To make it easier for administrators to find information about domains in this large distributed database, modern TCP/IP implementations generally come with an intelligent version of the **whois** utility. It is able to accept as input the name of a domain and automatically locate the appropriate registry in which that domain's information is located. The utility is usually used as follows:

```
whois [-h <whois-host>] <domain>
```

In this syntax, *<domain>* represents the name about which registration information is requested. The administrator can use the -h parameter to force the program to query a particular whois server, but again, this is usually not required. Some implementations also include other options that can be used to direct queries to particular registries.

Listing 88-5 shows the sample output of the *whois* command on a FreeBSD UNIX machine (I have stripped out some of the preliminary general information and legal disclaimers to shorten the listing).

```
%whois pcguide.com
Registrant:
    The PC Guide
    2080 Harwood Hill Road
    Bennington, VT 05201
    US

    ixl@fearn.pair.com
    +1.8025555555

Domain Name: PCGUIDE.COM

Administrative & Technical Contact:
    Charles Kozierok
    The PC Guide
    2080 Harwood Hill Road
    Bennington, VT 05201
    US

    ixl@fearn.pair.com
    +1.8025555555

Domain Name Servers:
    NS23.PAIR.COM
    NS0.NSO.COM

Created:     August 25, 1997
Modified:    July 7, 2003
Expires:    August 24, 2008

** Register Now at http://www.pairNIC.com/ **
```

Listing 88-5: DNS domain registry lookup using the *whois* program

In this case, the registrar of the domain `pcguide.com` is pairNIC, the DNS registry division of pair Networks, the company I have used for web hosting for many years (since 1997, as you can see). This output is public information and lets anyone who has an interest in `pcguide.com` determine that I own the domain and

learn how to contact me. (No, 555-5555 is not my real phone number.) It also tells them that pair Networks runs the name servers that contain domain information for my domain.

Many operating systems, including Windows, do not come with a `whois` command implementation, but there are third-party programs that will support the function. In recent years, many different organizations have also set up websites that implement the `whois` function, which is much more convenient and user-friendly to those more accustomed to graphical user interface operating systems like Windows. Many of these sites are provided as free services by DNS registrars, so customers can check if a name they are interested in is already taken, and if so, by whom.

One drawback of some of these systems is that they usually do not have the intelligence to check all the different registries where domain name records are stored. In most cases, a `whois` service provided by a registrar will search for names only in the particular top-level domains in which the registrar operates. So, if the registrar deals with .COM, .ORG, and .NET, it may support `whois` queries only for those top-level domains. To check the registration information for domains in more obscure domains, such as some of the less common geopolitical (country code) domains, a considerable amount of searching may be required.

KEY CONCEPT The TCP/IP `whois` utility allows registration information to be displayed for a DNS domain, such as its owner, contact information, and the date that its registration expires. The program is most commonly found on UNIX operating systems, where it is given intelligence that allows it to automatically query the correct servers to find the information for most domains. Newer Web-based `whois` utilities also exist, but they are usually limited to displaying information about domains in only a specific subset of top-level domains.

TCP/IP Network Status Utility (`netstat`)

Given how complex TCP/IP is, it's actually quite amazing that most of the time, all of the different protocols, services, and programs perform their jobs both efficiently and silently. Most of us don't even realize just how much is going on in the background, and that's as it should be. On the other hand, when a problem does occur on a TCP/IP network, the administrator charged with fixing it needs to obtain as much information as possible about what all those bits and pieces of the suite are doing behind the scenes. The network status utility, `netstat`, serves this purpose.

The `netstat` program is very simple in concept, being designed for only one purpose: to show information about the operation of TCP/IP on a device. The complexity of TCP/IP, however, leads to `netstat` being rather elaborate itself. The program can provide a large variety of information. As usual, the options and output of `netstat` depend on the particular operating system type and version. It is somewhat different on UNIX and Windows machines, so I will describe each platform's version separately.

The UNIX netstat Utility

On most UNIX systems, the netstat utility is very full-featured, with a typical implementation including dozens of options that can be used to control what information is displayed. These options may not all be used simultaneously; rather, they are arranged into option groups, each of which presents one class of information. Within each group, one option is mandatory, and that is the one that identifies the group, and hence the general kind of information that will be displayed. Other options are also possible in each group, which are optional and modify the command to provide better control of exactly what is output. In essence, netstat is like many related utilities rolled into one.

Table 88-7 provides a simplified summary of the option groups for a typical UNIX netstat implementation, in this case FreeBSD.

Table 88-7: Typical UNIX netstat Option Groups, Options, and Parameters

Option Group, Options, and Parameters	Description
netstat [-AaLSW] [-f <family>] [-p <protocol>] [-n]	Default invocation of netstat, with no mandatory options. It prompts the utility to display a list of active sockets on the host machine. The other options shown can be used to control what precisely is output; for example, -a also shows server processes.
netstat -i [-abdt] [-f <family>] [-n] netstat -I <interface> [-abdt] [-f <family>] [-n]	Tells netstat to provide information about all network interfaces (-i) or a particular network interface (-I <interface>). The -a option shows multicast addresses as well, -b displays bytes of data in and out on the interface, -d shows the number of dropped packets, and -t displays the value of watchdog timers.
netstat -w <interval> -d [-I <interface>]	Displays packet traffic information on all interfaces every <interval> seconds, or just on the specified interface if -I <interface> is included. If -d is included, it also indicates the number of dropped packets.
netstat -s [-s] [-z] [-f <family>] [-p <protocol>]	Shows systemwide statistics for each of the protocols on the system (which may be modified to show information for only a particular address family or protocol). If the -s option is repeated, counters that have a value of zero are suppressed. The -z option resets the statistics after they are displayed.
netstat -i -s [-f <family>] [-p <protocol>] netstat -I <interface> -s [-f <family>] [-p <protocol>]	Displays statistics as for netstat -s, but on a per-interface basis rather than aggregated for the whole system.
netstat -m	Outputs memory management routine statistics.
netstat -r [-Aa] [-f <family>] [-n] [-W]	Displays the contents of the host's routing tables. The options -A and -a provide additional information about the routes.
netstat -rs [-s]	Displays routing statistics. The -s option suppresses counters with a zero value.
netstat -g [-W] [-f <family>]	Shows multicast routing information.
netstat -gs [-s] [-f <family>]	Shows multicast routing statistics. The -s option suppresses counters with a zero value.

Most of the options shown in the option groups in Table 88-7 are particular to those groups; for example, you cannot use -s when issuing the command netstat -i. However, there are also a number of universal options that can be used with more than one of these groups to modify the behavior of netstat variations in a consistent way. These options are described in Table 88-8.

Table 88-8: Typical UNIX netstat Universal Options and Parameters

Option/ Parameters	Description
-f <family>	Limits the output of the command to information on a particular protocol address family, for hosts running multiple protocol suites. For example, the address family for regular TCP/IP is inet; for IPv6, it is inet6. Others may also be supported.
-p <protocol>	Restricts output to data related only to a particular protocol, such as IP, TCP, UDP, or ICMP.
-n	Shows network addresses in numeric form, instead of showing them as symbolic names. Also shows ports as numbers instead of converting well-known UDP and TCP port numbers to the protocol names that use them (for example, 23 rather than telnet).
-W	Suppresses the automatic truncation of addresses (which is sometimes done for display formatting).

The netstat command can produce a startling amount of output, especially if you do not restrict it with some of the options in Table 88-8. This is particularly true for netstat by itself and with the -s option. Listing 88-6 shows sample output from running “plain” netstat, but I have truncated the list of connections so it would not be too long (I also reformatted the listing so it would fit on the page better). Notice the last column, which shows the current state of the TCP connection (see the TCP finite state machine description in Chapter 47).

```
%netstat
Active Internet connections
  Prot  Rcv  Snd Local Address      Foreign Address      (state)
  tcp4  0    0   pcguide.com.http  c-24-118-141-124.3384  ESTABLISHED
  tcp4  0    827 pcguide.com.http  webcacheB03a.cac.46075  ESTABLISHED
  tcp4  0    0   qs36.smtp       MV1-24.171.17.64.1339  ESTABLISHED
  tcp4  0    0   pcguide.com.http  1Cust234.tnt1.le.1338  ESTABLISHED
  tcp4  0    0   pcguide.com.http  1Cust234.tnt1.le.1337  FIN_WAIT_1
  tcp4  0    84   pcguide.com.http  dial81-131-97-70.2902  FIN_WAIT_1
  tcp4  0    0   pcguide.com.http  216.76.14.221.9954  FIN_WAIT_2
  tcp4  0    0   pcguide.com.http  216.76.14.221.9945  FIN_WAIT_2
  tcp4  0    0   pcguide.com.http  1Cust234.tnt1.le.1326  TIME_WAIT
```

Listing 88-6: Sample connections list from the UNIX netstat utility

Listing 88-7 shows an example of the output of netstat -s. Here, I have limited the output by using -p ip to tell the program to show me only the statistics for IP.

```
%netstat -s -p ip
ip:
      57156204 total packets received
      0 bad header checksums
      4 with size smaller than minimum
      0 with data size < data length
      0 with ip length > max ip packet size
      0 with header length < data size
      0 with data length < header length
      0 with bad options
```

```

0 with incorrect version number
138 fragments received
6 fragments dropped (dup or out of space)
128 fragments dropped after timeout
2 packets reassembled ok
57085912 packets for this host
24736 packets for unknown/unsupported protocol
0 packets forwarded (0 packets fast forwarded)
44957 packets not forwardable
4 packets received for unknown multicast group
0 redirects sent
66183465 packets sent from this host
177 packets sent with fabricated ip header
0 output packets dropped due to no bufs, etc.
0 output packets discarded due to no route
0 output datagrams fragmented
0 fragments created
0 datagrams that can't be fragmented
0 tunneling packets that can't find gif
22 datagrams with bad address in header

```

Listing 88-7: Sample IP statistics from the UNIX netstat utility

The Windows netstat Utility

The Windows netstat utility is quite a bit simpler than the UNIX one, because it has a lot fewer options. This is good news for those learning about the program, but not so wonderful for those who want maximum power and flexibility in using it.

Like the UNIX netstat version, the Windows utility has a set of option groups that dictate the general type of information shown, and a few universal options that can be used with multiple groups. The option groups and generic options are shown in Tables 88-9 and 88-10, respectively.

Table 88-9: Typical Windows netstat Option Groups, Options, and Parameters

Option Group, Options, and Parameters	Description
netstat [-n] [-o] [<interval>] netstat -a [-n] [-o]	When called with no mandatory options, netstat displays information about active TCP connections.
[-p <protocol> [<interval>]	Displays all active TCP connections, as well as both TCP and UDP ports to which the host is listening.
netstat -e [<interval>]	Shows statistics for Ethernet interfaces.
netstat -r [<interval>]	Displays the current routing table for the device.
netstat -s [-p <protocol>] [<interval>]	Displays TCP/IP statistics for the system by protocol.

Table 88-10: Typical Windows netstat Universal Options and Parameters

Option/ Parameters	Description
-n	Displays network addresses in numeric form instead of symbolic name form. Also shows ports in numeric form instead of displaying standard process names associated with well-known UDP or TCP port numbers.
-o	Displays the process ID associated with each connection.
-p <protocol>	Limits the display to only the information associated with the specified protocol.
<interval>	Causes the netstat command to be repeated every <interval> seconds, rather than just displaying its information once. This can be used with any of the netstat option groups. For example, netstat -s 5 displays TCP/IP statistics every 5 seconds.

The output from the Windows netstat program is fairly similar to that of the UNIX utility when the same or similar options are given, but the UNIX version usually provides more details. Listing 88-8 shows an example illustrating TCP/IP statistics on my home Windows XP machine, using -p icmp to restrict the output to ICMP statistics only.

```
D:\aa>netstat -s -p icmp
ICMPv4 Statistics
```

	Received	Sent
Messages	243	248
Errors	0	0
Destination Unreachable	9	4
Time Exceeded	7	0
Parameter Problems	0	0
Source Quenches	0	0
Redirects	0	0
Echos	224	20
Echo Replies	3	224
Timestamps	0	0
Timestamp Replies	0	0
Address Masks	0	0
Address Mask Replies	0	0

Listing 88-8: Sample ICMP statistics from the Windows netstat utility

Listing 88-9 shows the routing table display from netstat (which I modified slightly to fit the page). You would get similar output using the UNIX netstat -s -p icmp or netstat -r command, but with additional information.

```
D:\aa>netstat -r
Route Table
=====
Interface List
0x1 ..... MS TCP Loopback interface
0x2 ...00 04 76 4e 75 3f ..... 3Com 10/100 Mini PCI Ethernet
```

```
=====
=====
Active Routes:
 Network Dest     Netmask        Gateway        Interface      Met
 0.0.0.0          0.0.0.0       148.64.128.1  148.64.133.73  30
 127.0.0.0        255.0.0.0    127.0.0.1     127.0.0.1     1
 148.64.128.0    255.255.192.0 148.64.133.73  148.64.133.73  30
 148.64.133.73   255.255.255.255 127.0.0.1     127.0.0.1     30
 148.64.255.255  255.255.255.255 148.64.133.73  148.64.133.73  30
 224.0.0.0        240.0.0.0    148.64.133.73  148.64.133.73  30
 255.255.255.255 255.255.255.255 148.64.133.73  148.64.133.73  1
Default Gateway:   148.64.128.1
=====
=====
Persistent Routes:
 None
```

Listing 88-9: Sample routing table display from the Windows netstat utility

KEY CONCEPT TCP/IP implementations include the netstat utility to allow information about network status to be displayed. On UNIX systems, netstat is a full-featured program with many options arranged into option groups, each of which shows a particular type of information about the operation of TCP/IP protocols. On Windows systems, netstat is somewhat more limited in function, but it still can display a considerable amount of information.

TCP/IP Configuration Utilities (**ifconfig**, **ipconfig**, and **winipcfg**)

A significant part of any network administrator's job is setting up and maintaining the devices that make a TCP/IP network function, a process generally called *configuration*. Networked hosts consist of both hardware and software that work together to implement all the layers and functions of the protocol stack. An administrator uses hardware tools to configure physical devices, performing tasks such as installing network interface cards, connecting cables, and manipulating switches and other hardware settings. Similarly, administrators need tools to configure the software that runs TCP/IP interfaces and controls the operation of higher-layer protocols on networked hosts. UNIX administrators use the ifconfig utility. On Windows NT, 2000, and XP, the configuration tool is ipconfig. Earlier versions of Windows have the winipcfg utility.

The ifconfig Utility for UNIX

On UNIX systems, administrators use the interface configuration utility, ifconfig, to view and modify the software settings that control how TCP/IP functions on a host. It is a very powerful program that allows an administrator to set up and manage a very wide array of network settings. The implementation of ifconfig varies greatly between flavors of UNIX; while most are similar in general terms, they may have different options and syntaxes.

You can use the ifconfig program for a variety of purposes: to create or remove a network interface, change its settings, or simply examine the existing configuration. Thus, like the netstat utility, ifconfig is like several related programs combined into one, and how it works depends on the syntax you used to invoke it. And also like netstat, ifconfig has a number of universal options that can be applied to many of its different uses.

Table 88-11 provides a simplified summary of the different functions that ifconfig can perform and the syntaxes that are used to specify each in a typical UNIX implementation (NetBSD in this case). You can use ifconfig to modify an interface's configuration by setting any of several dozen configuration parameters, using the syntax shown in the last row of that table. Table 88-12 describes the common options and parameters that can be used for many of these different modes. I have provided a brief description of some sample parameters in Table 88-13 (see your ifconfig documentation for a complete list).

Table 88-11: Typical UNIX ifconfig Syntaxes, Options, and Parameters

Syntax, Options and Parameters	Description
ifconfig [-L] [-m] <interface>	When ifconfig is called with just an interface specification and no other options (other than possibly -L and -m), it displays the configuration information for that network interface. Note that entering ifconfig by itself with no interface displays just help information for the parameter. To see all interfaces, use the -a parameter.
ifconfig -a [-L] [-m] [-b] [-d] [-u] [-s] [<family>]	Displays information about all the interfaces on the host. The output may be restricted using the universal parameters shown or by specifying an address family (see Table 88-12).
ifconfig -l [-b] [-d] [-u] [-s]	Lists all available interfaces on the system.
ifconfig <interface> create	Creates the specified logical network interface on the host, which is then configured using the syntax shown in the last row of this table. Note that some variations of UNIX allow certain parameters to be set at the time of creation.
ifconfig <interface> destroy	Destroys the specified logical interface.
ifconfig <interface> [<family>] [<address> [<dest-address>]] [<parameters>]	Configures parameters for a particular interface on the host. If the address is being set, it is the first parameter specified, after the optional address family, if present. The <dest-address> is used to specify a destination address for a point-to-point link. After this, any of several dozen parameters may be specified for the interface, some of which are shown in Table 88-13.

Table 88-12: Typical UNIX ifconfig Universal Options and Parameters

Option/Parameter	Description
-L	Displays the address lifetime for IPv6 addresses.
-m	Displays all supported media for the interface.
-b	Limits the display of interface information to broadcast interfaces.
-d	Shows only interfaces that are presently down (disabled).
-u	Shows only interfaces that are presently up (operational).

(continued)

Table 88-12: Typical UNIX ifconfig Universal Options and Parameters (continued)

Option/Parameter	Description
-s	Shows only interfaces that may be connected.
<family>	Specifies a particular address family, either to limit output or indicate what address type is being configured. The value <code>inet</code> is used for IPv4 and <code>inet6</code> for IPv6.

Table 88-13: Typical UNIX ifconfig Interface Configuration Parameters

Parameters	Description
<code>alias / -alias</code>	Establishes or removes a network address alias.
<code>arp / -arp</code>	Enables or disables the use of ARP on this interface.
<code>delete</code>	Removes the specified network address.
<code>down</code>	Marks an interface as being down, disabling it.
<code>media <type></code>	Sets the media type of the interface to a particular value.
<code>mtu <n></code>	Sets the maximum transmission unit (MTU) of the interface.
<code>netmask <mask></code>	Sets the network or subnet mask for the interface's address.
<code>prefixlen <n></code>	Same as netmask but allows the mask to be specified using a CIDR-style prefix length.
<code>up</code>	Sets an interface up, enabling it.

NOTE Since creating, destroying, or modifying interfaces can cause a host to stop working properly, administrative (superuser) rights are generally required on most systems in order to do anything with ifconfig other than examining the existing configuration.

Listing 88-10 shows sample output of the `ifconfig -a` command on one of the UNIX machines I use regularly, showing the settings for its interfaces.

```
%ifconfig -a
fxp0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    address: 00:a0:c9:8c:f4:a1
    media: Ethernet autoselect (100baseTX full-duplex)
    status: active
    inet 166.84.1.3 netmask 0xffffffff broadcast 166.84.1.31
        inet alias 166.84.1.13 netmask 0xffffffff broadcast 166.84.1.13
lo0: flags=8009<UP,LOOPBACK,MULTICAST> mtu 33228
    inet 127.0.0.1 netmask 0xff000000
```

Listing 88-10: Sample output of the UNIX ifconfig -a command

The ipconfig for Windows NT, 2000, and XP

Windows takes a somewhat different approach to network configuration than UNIX. As described in the previous section, you can use the UNIX `ifconfig` program both to view and modify a wide range of configuration parameters. In Windows, however, most setup and parameter modification is done using the Windows Control Panel. Windows does include a utility that is somewhat similar to UNIX's `ifconfig`,

but it has far less functionality and is used mainly to inspect the existing configuration, not change it. It also allows an administrator to easily perform a few simple functions on a host.

On Windows NT, 2000, and XP, the equivalent of ifconfig is a command-line utility called ipconfig. Like ifconfig, the Windows utility is controlled using options that are supplied to the program. However, because it is so much simpler than ifconfig, there are only a few options, as summarized in Table 88-14.

Table 88-14: Typical Windows ipconfig Options and Parameters

Option/Parameters	Description
(none)	When called with no options or parameters, ipconfig displays the IP address, subnet mask, and default gateway for each interface on the host.
/all	Similar to calling ipconfig with no options, but displays more detailed configuration information about the host's interfaces.
/release [<adapter>]	Releases (terminates) the DHCP lease on either the specified adapter (interface) or all interfaces, if none is provided.
/renew [<adapter>]	Manually renews the DHCP lease for either the specified adapter (interface) or all adapters, if none is mentioned.
/displaydns	Displays the contents of the host's DNS resolver cache.
/flushdns	Clears the host's DNS resolver cache.
/registerdns	Refreshes (renews) all DHCP leases and also reregisters any DNS names associated with the host.
/showclassid <adapter>	Displays DHCP class IDs associated with this adapter (these are used to arrange clients into groups that are given different treatment by DHCP servers). The adapter must be specified, even if there is only one.
/setclassid <adapter> [<classid>]	Modifies the DHCP class ID for the specified adapter.

As mentioned earlier, ipconfig is most often used to just examine the existing configuration. You can see from the list of options in Table 88-14 that most of the other uses of ipconfig are related to controlling the operation of protocols such as DNS and the Dynamic Host Configuration Protocol (DHCP), rather than configuring a host. One common use of ipconfig is to force a host to seek out a new DHCP lease, which can be done using ipconfig /release followed by ipconfig /renew.

Listing 88-11 shows an example of the output from using the ipconfig command without any options. For detailed information on interfaces, you can use the /all option, as shown in the example in Listing 88-12 (which I've modified slightly so it is easier to read).

```
D:\aa>ipconfig  
Windows IP Configuration
```

Ethernet adapter Local Area Connection 2:

```
Connection-specific DNS Suffix . :  
IP Address. . . . . : 148.64.133.73
```

Subnet Mask : 255.255.192.0
Default Gateway : 148.64.128.1

Listing 88-11: Simplified configuration information from the Windows ipconfig utility

```
D:\aa>ipconfig /all  
Windows IP Configuration
```

```
Host Name . . . . . : ixl
Primary Dns Suffix . . . . . :
Node Type . . . . . : Hybrid
IP Routing Enabled. . . . . : No
WINS Proxy Enabled. . . . . : No
```

Ethernet adapter Local Area Connection 2:

```
Connection-specific DNS Suffix . . . : 
Description . . . . . : 3Com PCI Ethernet Adapter
Physical Address. . . . . : 00-04-76-4E-75-3F
Dhcp Enabled. . . . . : Yes
Autoconfiguration Enabled . . . . : Yes
IP Address. . . . . : 148.64.133.73
Subnet Mask . . . . . : 255.255.192.0
Default Gateway . . . . . : 148.64.128.1
DHCP Server . . . . . : 148.64.128.1
DNS Servers . . . . . : 148.78.249.200
                                         148.78.249.201
Lease Obtained. . . . . : April 19, 2003 11:51:37 AM
Lease Expires . . . . . : April 19, 2003 12:21:37 PM
```

Listing 88-12: Detailed configuration information from the Windows ipconfig utility

The *winipcfg* Utility for Windows 95, 98, and Me

Windows 95, 98, and Me have a graphical tool called `winipcfg`, instead of the `ipconfig` command-line utility. This program allows you to examine the configuration parameters in much the same way as `ipconfig`, and also to release and renew DHCP leases, but it does not support the other options of `ipconfig` (such as displaying the host's DNS cache). An example of the main `winipcfg` screen is shown in Figure 88-2.

KEY CONCEPT On UNIX systems, the ifconfig utility can be used to display or modify a large number of TCP/IP configuration settings. Windows systems provide either the command-line utility ipconfig or the graphical tool winipcfg. Both let an administrator see basic TCP/IP configuration information for a host and allow tasks to be performed such as renewing a DHCP lease, but they are otherwise quite limited compared with the UNIX ifconfig program.

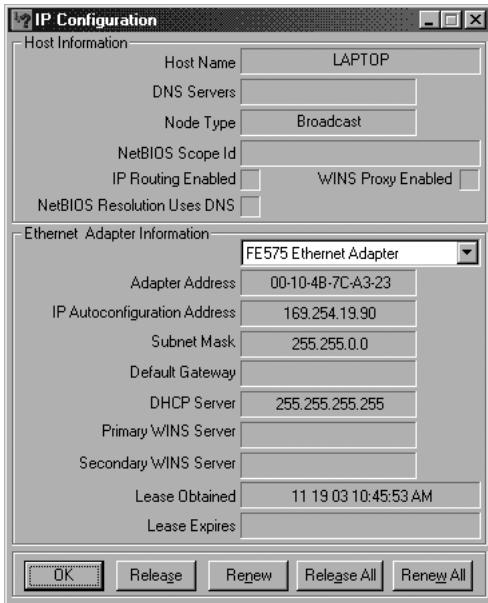


Figure 88-2: Windows 95/98/Me winipcfg utility The winipcfg utility can be used in older, consumer-oriented versions of Windows to check the configuration of a host and release/renew DHCP leases.

Miscellaneous TCP/IP Troubleshooting Protocols

As soon as you set up a network, it will very quickly develop problems that you will need to address. Recognizing that the complexity of TCP/IP internetworks would make diagnosing certain problems difficult, the suite's architects defined a number of miscellaneous utility protocols that can be helpful in testing and troubleshooting networks. Despite having been around for more than 20 years, these protocols are somewhat obscure and get little attention. However, even though they are no longer implemented on many systems, I feel they are worth a quick look.

These simple protocols are designed to be implemented as services that run on TCP/IP servers. Each listens for requests on a dedicated well-known port number and then responds with a particular type of information. These protocols can be used with both TCP and UDP, enabling each transport protocol to be tested. In the case of UDP, the server counts each UDP datagram sent to it as a request and sends a response to it. When used with TCP, a connection is first established by the client to the server. In some of the protocols, this connection is then used to send data continuously between the client and server; in others, the establishment of the connection is considered an implied request to the server, which will immediately send a response and then close the connection.

Table 88-15 provides a brief description of each of these troubleshooting protocols under both UDP and TCP. I have shown for each the port number that the service uses and also the RFC that defines it.

Table 88-15: Miscellaneous TCP/IP Troubleshooting Protocols

Protocol	Well-Known Port Number	Defining RFC	Description
Echo Protocol	7	862	Echoes received data back to its originator. When used on UDP, the payload of each message is simply packaged into a return UDP datagram and sent back. For TCP, each byte sent by the client is echoed back by the server until the connection is closed.
Discard Protocol	9	863	Throws away all data that is sent to it.
Character Generator Protocol	19	864	Generates random characters of data and sends them to a device. When used with UDP, each UDP message sent to the server causes it to send back a UDP message containing a random amount (0 to 512 bytes) of data. When used with TCP, the server just starts sending characters as soon as a client establishes a connection, and continues until the connection is terminated by the client.
Quote of the Day Protocol	17	865	Sends a short message (selected by the server's administrator) to a client device. For UDP, the message is sent for each incoming UDP message; for TCP, the message is sent by the server once when the connection is established, which is then closed.
Active Users	11	866	Sends a list of active users to a device. For UDP, the list is sent for each incoming UDP message; if it is longer than 512 bytes, it will be sent in multiple messages. For TCP, the list is sent automatically when the connection is made to the server, and then the connection is terminated.
Daytime Protocol	13	867	Returns the current time on the server in human-readable form, in response to receipt of a UDP message or an incoming TCP connection.
Time Protocol	37	868	Returns the current time in machine-readable form—specifically, the number of seconds since midnight, January 1, 1900 GMT. The time is sent for each UDP message received by the server or upon establishment of a TCP connection. Note that this protocol cannot be used for time synchronization of servers, because it does not compensate for variability in the time required for the messages to be carried over the internetwork.

INDEX

Numbers in *italics* refer to tables; numbers in **bold** refer to figures.

Numbers

- 100BASE-TX Fast Ethernet, 34
- 100VG-AnyLAN, 47
- 127.0.0.1, IPv4 loopback address, 266
- 30-Second Timer, RIP, 605
- 64-bit extended unique identifier. *See* EUI-64
- 6BONE experimental IPv6 network, 370
- 7-bit and 8-bit message encoding, MIME, 1258

A

- AAAA (IPv6 Address) DNS resource record, 949
- abbreviation, of URLs, 1156
- absolute and relative domain names, DNS, 865
- abstraction level, OSI Reference Model layers, 88
- Accept, 1361
- Accept-Charset, 1362
- Accept-Encoding, 1362
- Accept-Language, 1362
- Accept-Ranges, 1364
- access and retrieval
 - email, 1285
 - Usenet articles, 1403
- access control commands, FTP, 1185
- Access, MIB object characteristic, 1088
- accredited registrars for DNS name registration, 877

- Accredited Standards Committee X3, Information Technology, 50
- ACFC (Address and Control Field Compression), 158, 186
- ACK, 747, 753
- acknowledgment ambiguity, TCP, 804
- Acknowledgment messages, TFTP, 1213, 1213
- Acknowledgment Number, 784
- active and passive opens, 751
- active data connections, 1178
- adaptive retransmission, 803
- address allocation, 1000
- Address and Control Field Compression (ACFC)
 - PPP LCP option, 158
 - use in PPP, 186
- address books, 1230
- address classes, 256
- address designations, 214
- address embedding, 392
- address hexadecimal notation, 378
- Address Mask Request and Address Mask Reply messages, ICMP4, 543, 544
- address ranges (scopes), 1009
- address resolution, 203, 204
 - at layers 2 and 3, 204
- direct mapping, 206, 207
- ARCNet example, 207
 - problems with large hardware address size, 208
- dynamic, 209, 210
 - caching, 211
- email addressing and, 1226
- IP. *See* ARP
- IPv6, 224

address resolution, *continued*
 MAC addresses and, 205
 methods, 206
 multicast
 IEEE 802 addresses, 223
 organizationally unique identifier (OUI), 223
Address Resolution Protocol. *See* ARP
Address Resolution Protocol utility (arp), 1471
Address resource record, DNS, 937, 937, 946
address size
 IP, 245
 IPv6, 376
address space
 IP, 246
 IPv6, 376
 allocation of, 381
address terminology, IP NAT, 430
address types, IPv6, 375
Address Unreachable, 549
addressing, 565

- ANSI (American National Standards Institute), 50
- any state commands, IMAP, 1304, **1304**
- anycast address. *See* IPv6: address
- anycast routing, 421
- API (application program interface), 109
- APIPA (Automatic Private IP Addressing), DHCP. *See* DHCP
- application and service parameters, 1049
- application categories, 1164
- Application Layer
- OSI Reference Model. *See* OSI Reference Model
 - TCP/IP model, 130
 - TCP/IP protocol summary, 133
 - TCP/IP protocols at, 821
- application media type, MIME, 1251
- application program interface (API). *See* API
- applications
- direct and indirect methods, 1170
 - general file transfer, 1164
 - SNMP, 1073
 - TCP, 691, 742
 - TCP/IP, 707
 - Telnet, 1440
 - UDP, 692, 717
 - using both TCP and UDP, 718
- architecture
- DNS name, 858
 - DNS name server, 888
 - IPsec. *See* IPsec: architectures
 - name, 831
 - NFS, 957
 - PPP, 145, **145**
 - PPP Multilink Protocol (MP), 176
 - routing protocol, 591
 - SNMP Framework, 1076
 - TCP/IP, 119
- ARCNet, and direct mapping address resolution, 207
- area border routers, OSPF, 631
- area directors, IETF working groups, 53
- areas, OSPF, 630
- ARIN (American Registry for Internet Numbers), 56
- arithmetic, 72
- arp (Address Resolution Protocol utility), 1471
- ARP (Address Resolution Protocol), 212, 214
- address designations, 214
 - caching, 218
 - gratuitous, 499
 - messages
 - format, 216, 217, **218**
 - Request and Reply, 214
- Mobile IP interaction, 498
- proxy ARP, 221, **222**
- Mobile IP and, 499, **500**
 - role in IP datagram delivery, 354
 - transaction process, **215**
- ARPA (Advanced Research Projects Agency)
- ARPAnet, 122, 842, 1170
 - ARPAnet host name lists, 842
 - Network Control Protocol (NCP), 720, 1170
 - security issues, 450
 - TCP/IP history and, 122, 720
- .ARPA generic TLD, 872
- ARPAnet. *See* ARPA
- article propagation, NNTP, **1415**
- articles, Usenet. *See* Usenet
- AS (autonomous system)
- BGP. *See* BGP
 - routers, 593
 - routing architecture, 592
- ASCII control codes, 1445
- ASCII data type, 1183
- ASN.1, SNMP use of, 1116
- Asymmetric Digital Subscriber Line (ADSL), 41
- asymmetry, 41
- Asynchronous Transfer Mode. *See* ATM
- ATM (Asynchronous Transfer Mode)
- IP and, 17
 - PPP over ATM (PPPoA), 146
 - quality of service and, 44
- attachments, 1243
- audio media type, 1250
- AUTH extension, 1281
- Authenticate-Ack messages, 162
- Authenticated state, 1301
- commands, 1307, *1308*
- Authenticate-Nak, 162

- Authenticate-Request, 162
 - authentication
 - DHCP, 1064
 - ESP. *See* IPsec
 - FTP, 1175
 - HTTP, 1389
 - HTTP proxy servers, 1388
 - IMAP, 1306
 - OSPF message, 639
 - POP-before-SMTP, 1278
 - PPP, 162
 - RIP-2, 618
 - RIPng, 621
 - Authentication Header, 461
 - Authentication phase, 149
 - Authentication Protocol, 158
 - authorities
 - DNS, 868
 - Internet registration, 55
 - IP address, 252
 - authority, 882
 - Authorization state, 1290
 - Authorization, 1362
 - autoconfiguration
 - DHCP, 1058
 - IPv6, 399
 - ND, 583
 - automatic allocation, DHCP, 1002
 - Automatic Private IP Addressing. *See* DHCP
 - autonomous system (AS) routing
 - architecture, 592, **594**
- B**
- backbone routers and topology, OSPF, 631
 - backbone, Usenet, 1413
 - BACP (PPP Bandwidth Allocation Control Protocol), 178–179
 - Bad Length, 534
 - bandwidth, 35
 - Bandwidth Allocation Control Protocol, 178
 - Bandwidth Allocation Protocol, 178
 - bandwidth reservation, 44
 - BAP (PPP Bandwidth Allocation Protocol), 178–179
 - base 2 numbers, 65
 - base 8 numbers, 67
 - base 10 numbers, 65
 - base 16 numbers, 67
 - base64, MIME encoding method, 1258, 1259
 - base URLs, 1150
 - basic rate interface (BRI), 175
 - basic topology, OSPF, 628
 - Baudot, Jean-Maurice-Émile, 38
 - baud, 38
 - BBS (bulletin board system), 1458
 - Bell Laboratories, 1398
 - Bellman-Ford routing protocol
 - algorithm, 594
 - Berkeley Remote (r) Commands, 1454
 - Berkeley Standard Distribution (BSD), UNIX
 - ephemeral port number ranges, 704
 - popularized RIP, 614
 - Berners-Lee, Tim, 1318
 - Best Current Practice, RFC category, 58
 - best-effort, description of IP, 237
 - BGP (Border Gateway Protocol), 647, 648
 - algorithm, 657
 - AS (autonomous system)
 - internal peer, 652
 - multihomed, 654
 - routing policies, 654
 - stub, 654
 - transit, 655
 - types, 654
 - border routers, 652
 - connection establishment, 662
 - decision process, 659
 - error reporting, 663
 - external BGP (EBGP), 652
 - functions and features, 650
 - hold timer, 672
 - hybrid routing protocol algorithm, 595
 - Internal BGP (IBGP), 652
 - limitations on efficient route selection, 661
 - Marker field, 665
 - message generation and transport, 663
 - messages
 - general format, 664, *664*, **665**

- BGP, *continued*
- messages, *continued*
 - Keepalive, 672, 673, **673**
 - Notification, 673, 674, **674**, 675
 - Open, 666, 666, 667, **668**
 - Update, 671, 667, 668, 669, **670**
 - messaging, 662
 - neighbors and peers, 652
 - operation, 662, 663
 - path attributes, 659, 659
 - route determination, 659
 - route information exchange, 662
 - route storage, 656
 - router roles, 652
 - routing information bases (RIBs), 656
 - routing policy issues, 655
 - speakers, 652
 - standards, 649, 650
 - topology, 651, **653**
 - traffic flow and types, 653
 - versions, 649, 649
- BGP-4. *See* BGP
- bidirectional, IP NAT, 437–439
 - Big Eight newsgroup hierarchies, 1405
 - billing contact, 896
 - binary
 - addition, 72
 - arithmetic, 72
 - conversion from decimal to, 70
 - digit (bit), 63
 - information, 62
 - group representations and terms, 63, **63**, **64**
 - transistors and, 62
 - notation, IP addresses, 245
 - numbers, decimal equivalent, 65, 66
 - octal and hexadecimal digit conversion, 68
 - representation of IPv6 addresses, **378**
 - representation of DNS resource records, 890
 - units, 38
- bis, 1300
- bit (binary digit), 63
- bit mask, 76
- bit masking
 - IP subnet masks, 277
 - using Boolean logical functions, 75
- BITS (Bump in the Stack) IPsec architecture, 455
- bits and bytes, 37
- bits per second (bps), 38
- bits, 63, 76
- BITW (Bump in the Wire) IPsec architecture, 456
- .BIZ generic TLD, 872
- block mode, FTP, 1182
- bogus authentication information, 1158
- Boolean logic
 - expressions, 73, 75
 - functions, 73
 - AND and OR, 74
 - Exclusive-OR (XOR, EOR), 75
 - NOT, 73
 - true and false values, 73
- BOOTP (Bootstrap Protocol), 977
- addressing, 980
 - basis for DHCP, 999
 - bootstrapping and, 977, 983
 - broadcasts and port use, 981
 - Client IP Address (CIAAddr) field, 984
 - client/server messaging, 980
 - clients, 980
 - forwarding agents, 991–994, **994**, 1056
 - history, 978
 - interoperability with DHCP, 1062
 - magic cookie, 989
 - messages
 - format, 985, 986, **988**
 - HType values, 987
 - messaging, 981
 - operation, 983, **985**
 - relay agent, 991
 - broadcast message use, 994
 - DHCP message relaying, 1056
 - function, 992
 - normal operation, 993, **994**
 - retransmission of lost messages, 982
 - server, 980
 - standards, 978
 - transport, 981
 - Vendor Information
 - extensions, 989
 - field format, 990
 - fields, 990
 - Vendor-Specific Area, 988

- BOOTP/DHCP relay agents, 991–994, 1056
- Bootstrap Protocol. *See* BOOTP bootstrapping
- BOOTP and, 977
 - RARP and, 228
 - TFTP and, 1200
- Border Gateway Protocol. *See* BGP border routers
- AS in, 593
 - BGP, 652
- bottlenecks, 40
- bps (bits per second), 38
- branch, DNS name tree, 860
- BRI (basic rate interface), 175
- broadcast methods
- name registration, 836
 - name resolution, 837
- broadcast addresses
- IPv6, 375
 - subnet, 314
- browsers, 1321
- buffers, 525
- bulletin board system (BBS), 1458
- Bump in the Stack (BITS), 455
- Bump in the Wire (BITW), 456
- Bush, Vannevar, 1318
- business name conflicts, 879
- byte, 64
- versus octet, 64
- C**
- cable modem, 41
- cable segment, 29
- Cache-Control directives, 1359
- caching, 902
- ARP, 218
 - DNS, 903
 - dynamic address resolution, 210
 - HTTP. *See* HTTP: caching
 - name resolution, 839
 - negative, DNS, 904
 - NFS, 963
- caching-only, DNS name servers, 895
- Calculator program, Windows, 68
- Callback-Request messages, 179
- Callback-Response messages, 179
- Call-Request messages, 179
- Call-Response messages, 179
- Call-Status-Indication messages, 179
- Call-Status-Response messages, 179
- campus area network (CAN), 27
- Canonical Name resource record, 937, 938, 946
- care-of addresses. *See* Mobile IP: addresses
- Carriage Control/FORTRAN, 1185
- carriage return (CR), RFC 822 email message format, 1235
- carriage return/line feed sequence. *See* CRLF
- case
- in DNS labels, 863
 - in DNS master files, 945
- CBEMA (Computer and Business Equipment Manufacturers Association), 50
- .CC country code TLD, 875
- CCITT (Comité Consultatif International Téléphonique et Télégraphique), 50
- OSI Reference Model and, 82
- CCP (Compression Control Protocol). *See* PPP: CCP
- ccTLDs, DNS country code (geopolitical) TLDs, 874
- cell, message type, 18
- central authority, DNS root domain, 868
- centralized registration authorities, 55
- CERN (Conseil Européen pour la Recherche Nucléaire), 1318
- Challenge messages, PPP CHAP, 164
- channels, IRC, 1459
- CHAP (Challenge Handshake Authentication Protocol), PPP. *See* PPP: CHAP
- character, information storage term, 64
- checksum calculation, TCP, 774
- checksum recalculations, IP NAT, 446
- child, DNS name term, 862
- chunked transfers, HTTP, 1375
- CIAddr (Client IP Address) Field, BOOTP, 984
- CIDR (Classless Inter-Domain Routing) address blocks, 323
- addressing example, 324

CIDR, *continued*
difference between VLSM and, 296
hierarchical addressing, 319
hierarchical division of /15 CIDR address block, **325**
IANA and, 321
notation, 319, **320**
prefix length in, 319
circuit switching, **14**
circuit-switching and packet-switching networks, 13
Cisco Systems, 681
class determination algorithm, 258, **259**
classes
 BGP path attribute, 658
 DNS resource record, 893
 ICMP message, 512
 IP address, 256
 SNMP PDU, 1103
classful addressing, 258, **259**
classless addressing, 281. *See also* CIDR
Classless Inter-Domain Routing (CIDR)
 notation, 281
clearing bits, 63, 76
 using AND bit mask, 77, 77
client and server roles, 23
client hardware, 126
client identifier, DHCP, 1017
client implementations, DHCP, 1056
client processes, 697
client software, 23, 126
client, BOOTP, 980
client/server
 application port use, TCP/IP, 703
 model of operation, TCP/IP, **126**, 697
 name resolution method, 837
 networking, 23, **25**
 port number use, 705
 structural models, 23
closing TCP send window, 808
CMIP (Common Management Information Protocol), 1070
CName (Canonical Name) DNS
 resource record, 892, 937
 master file format, 946
coaxial cable segment, 29
Code-Reject messages
 PPP CCP, 170
PPP ECP, 173
PPP LCP, 159
PPP NCPs, 160
collision domain, 29
co-located care-of address, Mobile IP, 485
colon hexadecimal notation, 379
.COM generic TLD, 872
Comité Consultatif International Téléphonique et Télégraphique (CCITT), 50
command extensions, NNTP, 1422
command tagging, IMAP, 1303
command-line, FTP interface, 1193
comments, in DNS master files, 945
common Internet scheme syntax for URLs, 1143
Common Management Information Protocol (CMIP), 1070
Communication with Destination
 Administratively Prohibited, 549
Communication with Destination Host
 Is Administratively Prohibited, 524
Communication with Destination Network
 Is Administratively Prohibited, 524
communication, OSI Reference Model
 horizontal (corresponding layer), 93
 vertical (adjacent layer), 91
community strings, SNMPv1, 1111
community-based SNMPv2 (SNMPv2c), 1078
composite media types, MIME, 1253
compressed SLIP (CSLIP), 143
compression
 DNS message, 941
 OSI Reference Model Presentation Layer function, 111
 PPP field, 185
compression algorithms, 171, **171**
Computer and Business Equipment Manufacturers Association (CBEMA), 50
computing IP subnet host address
 shortcuts, 313
configuration parameter management
 DHCP, 1016

- Configure-Ack, Configure-Nak,
 Configure-Reject, Configure-
 Request messages
- PPP CCP, 170
 PPP ECP, 173
 PPP LCP, 157, 158
 PPP NCPs, 160
- conflicts in DNS public registration, 878–879
- congestion
 collapse, 817
 handling and avoidance, 817
- connection identification using TCP/IP
 socket pairs, 706
- Connection, HTTP general header, 1360
- connectionless and connection-oriented
 protocols, 15
- contacts, DNS domain, 896
- content and transfer encodings, 1372
- content negotiation, HTTP, 1378
- Content-Description, MIME header, 1247–1248
- Content-Encoding, HTTP entity header, 1366
- Content-ID, MIME header, 1247
- Content-Language, HTTP entity
 header, 1366
- Content-Length
 HTTP entity header, 1366
 MIME header, 1248
- Content-Location
 HTTP entity Header, 1366
 MIME header, 1248
- Content-MD5, HTTP entity header, 1367
- Content-Range, HTTP entity header, 1367
- Content-Transfer-Encoding, MIME
 header, 1247, 1257
- Content-Type
 HTTP entity header, 1367
 MIME header, 1246, 1248
- Continue, HTTP preliminary reply, 1356
- control connection, FTP, 1172
- control messages, PPP. *See* PPP
- conventional IP addressing, 258, 259
- conventions
- ICMP message processing, 517
 SMI textual, 1090
- convergence, routing protocol, 607
- cookie, magic, BOOTP, 989
- cookies, HTTP, 1391
- .COOP generic TLD, 873
- Copied flag, and IP datagram
 fragmentation, 347
- core protocols, IPsec, 453
- core protocols, TCP/IP lower-layer, 135
- core routers and core routing
 architecture, 592
- corporate name conflicts, 878
- counters, PPP LQR, 168
- counting to infinity, RIP, 607, 608
- country code TLDs and authorities,
 DNS, 874
- CRLF
 logical end-of-line character, Telnet
 NVT, 1445
 RFC 822 messages and, 1235
- cross-posting, Usenet, 1407
- cross-resolution, dynamic resolution, 211
- CSLIP (Compressed SLIP), 143
- custom subnet mask. *See* IP: subnetting:
 custom masks
- cybersquatting, DNS public registration, 879

D

- DARPA (Defense Advanced Research
 Projects Agency). *See* ARPA
- data block numbering, TFTP, 1205
- data connections, FTP, 1177
- data definition, NFS, 959
- data encapsulation, OSI Reference
 Model, 95, 96
- Data Encryption Standard (DES), 473
- data framing, SLIP, 141
- data handling, TCP, 728
- Data Link Layer (Layer 2), OSI
 Reference Model, 103
- Data messages, TFTP, 1212, 1212
- data representation, 61
- data, message, 19

Database Description messages, OSPF, 638
format of, 641
datagram
 IP. *See* IP: datagrams
 IPv6. *See* IPv6: datagrams
datagram placement and linking, 462
Date, HTTP general header, 1360
DDDS (Dynamic Delegation Discovery System), 1162
DDNS (Dynamic DNS), 907
de facto standards, 48
deceptive naming practices, 879
decimal numbers
 binary equivalent, 66
 conversion, 69, 70, 71
decimal units, 38
decision process, BGP, 659
decrypting data, PPP ECP, 174
dedicated circuits, 15
default gateway, IP, 251
default IP subnet masks, 282
default MSS, TCP, 778
default router, 530
default routes, RIP, 604
Defense Advanced Research Projects Agency (DARPA). *See* ARPA
delayed delivery, TCP/IP email, 1222
DELETE, HTTP method, 1351
delimiting, URLs, 1155
delivery
 IP datagram, 352
 IPv6 datagram, 420
demultiplexing, 698, 699
Department of Defense (DoD), United States, 371
deregistration of mobile nodes, Mobile IP, 491
deregulation of DNS registration, 877
DES (Data Encryption Standard), 473
descriptors, MIB object, 1091
Destination Host Unknown, 524
Destination Host Unreachable for Type of Service, 524
Destination Network Unknown, 524
Destination Network Unreachable for Type of Service, 524
Destination Options, IPv6 extension header, 413
Destination Unreachable messages
 ICMP4, 522, 522
 ICMP6, 548, 549
device
 renumbering, IPv6, 400
 roles, Mobile IP, 481
 types, SNMP, 1072
dhclient, UNIX DHCP client daemon, 1056
DHCP (Dynamic Host Configuration Protocol), 998, 1058, **1060**
 address assignment and allocation mechanisms, 1000
 address pool size selection, 1009
 APIPA, 1059, **1060**
 IP address block for, 267
 limitations, 1060
 authentication, 1064
 autoconfiguration, 1058
 automatic allocation, 1002
 BOOTP interoperability, 1062
 BOOTP relay agents for, 1057
 client and server responsibilities, 1014
 client finite state machine, 1017, **1018, 1020**
 client identifier, 1017
 client/server roles, 1015
 clients, 1056
 BOOTP servers and, 1063
 dhclient UNIX daemon, 1056
 dhcpcd UNIX daemon, 1056
 responsibilities, 1015
 configuration, 1013
 conflict detection, 1061
 development of, 980
 DHCPv6, 1065
 dynamic allocation, 1001
 extensions, 1050
 general operation, 1017
 history, 998
 IP Version 6. *See* DHCP; DHCPv6
 leases, 1003
 address ranges (scopes), 1009, **1010**
 allocation, 1021, **1025**
 early termination, 1031
 issues with Infinite leases, 1005
 length, 1003

DHCP, *continued*

- leases, *continued*
 - life cycle phases, 1006, **1007**
 - multiple-server non-overlapping scopes, **1011**
 - reallocation, 1026, **1028**
 - release, 1031
 - renewal and rebinding, 1028, **1032**
 - renewal and rebinding timers, 1008
- manual allocation, 1001
- message
 - format, 1038, **1039**, *1039*
 - generation, 1036
 - transport, 1036
 - types, **1051**
- options
 - See also* DHCP/BOOTP options
 - categories, **1043**
 - format, **1042**, *1043*
 - overloading, 1044
- parameter
 - configuration, 1033, **1034**
 - storage, 1017
- rebinding timer (T2), 1008
- relaying, 1057
- renewal timer (T1), 1008
- retransmission of lost messages, 1037
- security, 1063
- servers
 - BOOTP clients and, 1063
 - configuration parameters, 1014
 - conflict detection, 1061
 - implementations, 1054
 - Microsoft, 1061
 - number of, 1055
 - placement, setup, and maintenance, 1055
 - responsibilities, 1014
 - software features, 1054
- standards, 998
- transaction identifier (XID), 1036
- winipcfg and, 1056

DHCP/BOOTP options, 1045

- application and service parameters, **1049**
- DHCP extensions, **1050**
- IP layer parameters
 - per host, **1047**
 - per interface, **1047**

link layer parameters per interface, **1048**

RFC 1497 vendor extensions, **1045**

TCP parameters, 1048, **1048**

DHCP/BOOTP relay agents. *See* BOOTP: relay agents

dhcpd, DHCP UNIX client daemon, 1056

DHCPv6 (DHCP for IP Version 6), 1065

dial-up remote server, email access, 1311

Differentiated Services (DS), IP datagram TOS field, 335

Diffie-Hellman group, IKE, 473

dig utility, 1476

digital information, 62

direct and indirect

- (routed) delivery of IP datagrams, **352**
- network applications, 1170

direct mapping, address resolution. *See* address resolution: direct mapping

direct server email access, 1311

directives, DNS master file, 945

disconnected access model, 1287

discrete media, MIME, 1246, 1248

diskless workstations, 1200

dispute resolution, DNS, 878

disruption, TCP connection, 760

Distance Vector Multicast Routing Protocol (DVMRP), 361

Distance-Vector (Bellman-Ford) routing protocol algorithm, 594

distributed name database, DNS, 869

DNS (Domain Name System), 848, 941

- A resource record, 937, **937**, 946
- AAAA (IPv6 Address) resource record, 949
- absolute domain names, 865
- Address resource record, 937, **937**, 946
- authority structure, 868
- caching, 903, 915
- Canonical Name resource record, 892, **938**, 946
- changes to support IPv6, 948
- client/server messaging, 928
- CName resource record, 892, **938**, 946

DNS, *continued*
 components, 853
 country code designations, 874
 leasing/sale of, 875
 country code TLDs and authorities, 874
 DDNS, 907
 design goals, 851
 development, 848
 dispute resolution, name registration, 880
 distributed name database, 869
 DNS Update (Dynamic DNS), 907
 domain contacts, 896
 domain names, 863
 domain-related terminology, 860
 domains, 858
 email
 address notation, 941
 communication and, 1266
 name resolution, 924
 extension mechanisms for, 929
 fully qualified domain names (FQDNs), 865
 functions, 853
 generic TLDs, 870, 872, 872
 geopolitical (country code) TLDs, 874
 IN-ADDR.ARPA reverse name resolution hierarchy, 922
 inverse querying, 921
 IPv6 Address (AAAA) DNS resource record, 949
 labels and domain name construction, 863, 864
 lookup utility, whois/nickname, 1477
 Mail Exchange resource record, 892, 939, 947
 master files
 directives, 945
 format, 943
 resource record examples, 946
 sample, 948
 syntax rules, 945
 master name servers, 893
 messages
 compression, 941
 format, 930, 930, 931
 generation and transport, 928
 header format, 932, 933, 934
 question section format, 935, 935, 935
 resource record field formats, 935, 937
 transport using UDP and TCP, 929
multiple address records, 904
MX resource record, 892, 939, 947
name registration, 867
 authorities, 877
 coordination, 878
 dispute resolution, 880
 second-level domains, 876
name resolution, 912, 913, 917
 email, 924
 example, 917, 919
 handling aliases, 920
 impact of zones on, 882
IN-ADDR.ARPA hierarchy, 922
local resolution, 915
recursive, 913, 914
reverse, 920
techniques, 911
name resolvers
 caching, 916
 functions and operation, 910
 stub, 911
 user interface, 910
name servers, 887
 architecture, 833, 888
 caching, 901
 data storage, 890
 efficiency issues, 894
 enhancements, 905
 load balancing, 904
 multiple, 905
 redundancy, 894
 resource record, 892, 937, 946
 support functions, 889
 types and roles, 893
names
 absolute, 865
 architecture, 832, 858
 distributed database, 869
 fully qualified domain names (FQDNs), 865
 hierarchical tree structure, 858
 name space, 854, 857, 859, 862
 notation, 941

DNS, *continued*

- names, *continued*
 - partially qualified domain names (PQDNs), 866
 - terminology, 860–862
 - NAT and, 437, 444
 - notify, 906
 - NS resource record, 892, 937, 946
 - operation, 888
 - organizational TLDs, 870, **872**, 872
 - partially qualified domain names (PQDNs), 866
 - pointer resource record, 892, 938, 947
 - PQDNs (partially qualified domain names), 866
 - primary name servers, 893
 - private name registration, 884
 - PTR resource record, 892, 938, 947
 - relative domain names, 865
 - resolution. *See DNS: name resolution*
 - resolvers. *See DNS: name resolvers*
 - resource records, 892, 892
 - address, 937, 937, 946
 - canonical name, 892, 938, 946
 - classes, 893
 - format, common, 936, 936
 - IPv6 address, 949
 - Mail Exchange, 892, 939, 947
 - master file (text) representation, 891
 - master file format, common, 944
 - name server, 892, 937, 946
 - pointer, 892, 938, 947
 - RData field formats, 936
 - representations, 890
 - setup for reverse resolution, 922
 - Start Of Authority, 892, 938, 946
 - text, 892, 940, 947
 - reverse name resolution, 920
 - root
 - authority, 868
 - domain, 860
 - name servers, 899
 - RR. *See DNS: resource records*
 - secondary name servers, 893
 - slave name servers, 893
 - SOA resource record, 892, 938, 946
 - standards, 849, 850
 - Start Of Authority resource record, 892, 938, 946
 - subdomains, 860
 - text resource record, 892, 940, 947
 - top-level domains (TLDs), 860
 - tree-related terminology, 860, **861**
 - TXT resource record, 892, 940, 947
 - Uniform Domain Name Dispute Resolution Policy (UDRP), 880
 - whois/nickname utility, 1477
 - zones, **883**, 895
 - domain contacts, 896
 - transfers, 898, 907

DoD (United States Department of Defense), 371

DoD (United States Department of Defense) Model, 87

domains. *See DNS*

don't fragment flag, IP, 347

dotted decimal notation

IP address, 245

subnet mask, 278

double word, unit, 64

downstream Usenet servers, 1413

Duke University, 1398

DVMRP (Distance Vector Multicast Routing Protocol), 361

dynamic address

mappings, IP NAT, 433

resolution, 209, **210**

dynamic allocation, DHCP, 1001

dynamic data length, HTTP, 1375

Dynamic Delegation Discovery System (DDDS), 1162

Dynamic DNS (DDNS), 907

dynamic router selection, ND, 581

dynamic routing protocols, 595

E

EBCDIC, FTP data type, 1183

EBGP (External BGP), 652

Echo (Request) and Echo Reply messages

ICMP4, 536, **536**, **537**

ICMP6, 558, 559

ping utility, use of, 1464

ECP (Encryption Control Protocol), 173

editor, RFC, 57

EDNS0 (Extension Mechanisms for DNS), 929
.EDU generic TLD, 873
EGP (Exterior Gateway Protocol), 684
EHLO (extended hello), SMTP, 1269
EIA (Electronic Industries Alliance), 50
EIGRP (Enhanced Interior Gateway Routing Protocol), 682
Electronic Industries Alliance (EIA), 50
electronic mail. *See* email
email (electronic mail), 1215
access
 direct server, 1311
 models, 1287
 Telnet, 1311
 World Wide Web, 1313
access and retrieval protocols and methods, 1285
address books, 1230
addressing, 1226
 gatewaying, 1229
 historical, 1228
 multiple recipients, 1230
 notation, DNS, 941
 SMTP requirements, 1227
 standard DNS-based, 1226
 UUCP-style, 1229
aliases, 1230
communication, 1220
 early, using relaying, 1266
 model, 1222, **1223**
 modern, using DNS, 1266
 protocol roles, 1224
decoupling of sender and receiver, 1222
delayed delivery, 1222
DNS resource records for, 924
early days of, 1218
history of TCP/IP-based, 1219
messages
 difference between message and envelope, 1241
 formats and processing, 1233
 RFC 822, 1234
name resolution, DNS, 924
protocols, 1215
system overview, 1219
Unix-to-Unix Copy Protocol (UUCP) and, 1219
URL scheme (mailto), 1147
embedding, IPv6/IPv4 addresses, 392
encapsulated message type, MIME, 1257
Encapsulating Security Payload. *See* IPsec: ESP
encoding
 HTTP, 1373
 MIME. *See* MIME: encoding
encryption
 HTTP, 1390
 PPP. *See* PPP: ECP
encryption/hashing algorithms, IPsec, 453
END character, SLIP, 141
end-to-end
 encoding, HTTP, 1373
 headers, HTTP, 1385
 transport, Transport Layer, 107
Enhanced Interior Gateway Routing Protocol (EIGRP), 682
enhanced status code replies, SMTP, 1284
entities
 HTTP, 1370
 MIME, 1246
 SNMP, 1073
entity headers, HTTP. *See* HTTP: entity headers
envelope, analogy to IP datagram encapsulation, 330
EOR, Exclusive-OR Boolean logical function, 75
ephemeral ports, 703
error messages
 TFTP, 1213, **1214**
 ICMPv4. *See* ICMPv4: error messages
 ICMPv6. *See* ICMPv6: error messages
error reporting, BGP, 663, 673
escape character (ESC)
 DNS Master Files, 945
 SLIP, 142
 Telnet protocol commands, 1447
ESP (IPsec Encapsulating Security Payload). *See* IPsec: ESP
ETag, HTTP response header, 1364
/etc/hosts UNIX configuration file, 843, 1463

- Ethernet
- 100BASE-TX fast, 34
 - example link layer technology, 28
 - frames, 19
- ETSI (European Telecommunications Standards Institute), 50
- EUI-64 (64-bit extended unique identifier) format, IEEE, 388 modified, 388
- European Telecommunications Standards Institute (ETSI), 50
- Exclusive-OR Boolean logical function, 75
- expect, HTTP request header, 1362
- experimental
- IP addresses, Class E, 257
 - IPv6 network, 6BONE, 370
 - RFCs, 58
- expires, HTTP entity header, 1367
- explicit URL delimiting, 1156
- expressions, Boolean logic, 73
- extended hello (EHLO), SMTP command, 1269
- extension headers, IPv6, 407
- extensions
- DHCP, 1050, **1050**
 - DNS, for IPv6, and 949
 - FTP security and, 1176
 - Mobile IP Agent Advertisement messages, 488
 - NNTP command, 1422
 - SMTP, **1276**, 1281
- Exterior Gateway Protocol (EGP), 684
- exterior routing protocols, 593
- external BGP (EBGP), 652
- External Data Representation (XDR) Standard, NFS, 959
- external peers, in autonomous system, 652
- external performance limiters, networks, 40
- extranet, 30
- F**
- failure messages, PPP CHAP, 164
- family-related terminology, DNS, 861
- Fast Ethernet, 100BASE-TX, 34
- open standards and, 47
- FDDI (Fiber Distributed Data Interface), 340
- Fiber Distributed Data Interface (FDDI), 340
- fiber-optic communication, simplex operation, 42
- FidoNet addressing, 1228
- field compression, PPP, 185
- file and message transfer applications, 1163
- file concepts, 1164
- file system model, NFS, 968
- File Transfer Protocol. *See* FTP
- File Transfer Protocol, Trivial. *See* TFTP
- file, URL local file scheme syntax, 1149
- FIN (finish) messages, TCP, 747
- finite state machine. *See* FSM
- flag, unit of information measure, 63
- flat name
- architecture, 832, **832**
 - space, 832, **832**
- flow control, TCP, 793, 805
- footer, message, 19
- foreign agent care-of address, Mobile IP, 484, **484**
- format control, FTP, 1184
- format prefix, IPv6 addresses, 382
- forwarding agents, BOOTP, 991
- forwarding, 98
- four-message exchange, DHCPv6, 1066
- FQDNs (Fully Qualified Domain Names), DNS, 865
- fragment extension header, IPv6, **411**
- fragmentation
- IP, **342**
 - IPv6, 418–419
 - PPP MP, 196–199
- Fragmentation Needed and DF Set, 523
- fragments, URL, 1145
- frames
- Ethernet, 19
 - PPP LCP, 156
- Framework, Internet Standard Management. *See* SNMP Framework
- framework, SNMP. *See* SNMP Framework
- framing, SLIP, 141
- from, HTTP request header, 1362

- FSM (finite state machine), 746
 DHCP client, 1017
 IMAP, 1300, **1301**
 POP3, 1290
 TCP, 746
 FTP (File Transfer Protocol),
 1169–1171
 active data connection, **1179**
 anonymous, 1177
 authentication, 1175
 commands
 access control, 1185, *1186*
 protocol service, *1187*
 transfer parameter, *1187*
 compared to TFTP, 1201
 compressed mode, 1182
 connection
 active data, **1179**
 control, 1175
 establishment and user
 authentication, **1176**
 management, 1177
 normal, 1178
 passive data, 1178, **1180**
 control connection establishment,
 1175
 data
 communication, 1181
 representation, 1182
 data structures, 1185
 data types, 1183
 development, 1170
 format control, 1184
 IP NAT and, 447
 login sequence, 1175
 multiple-line text replies, 1192
 normal data connections, 1178
 operational model, 1172, **1173**
 passive data connection, 1178, **1180**
 PASV command, 1178
 PORT command, 1178
 process components and terminol-
 ogy, 1173
 protocol service commands, *1187*
 proxy FTP, 1174
 replies, 1188
 reply codes, 1188, **1190**, *1190*
 security extensions, 1176
 service commands, 1186
 session, example, 1196
 standardization, 1170
 stream mode, 1181
 Telnet NVT and, 1186
 third-party file transfer, 1174
 transfer parameter commands, *1187*
 transmission modes, 1181
 user
 commands, 1193, *1194*
 interfaces, 1193
 ftp, URL File Transfer Protocol syntax,
 1147
 full newsgroup retrieval, NNTP, 1418
 full-duplex communication, 42
 fully qualified domain names (FQDNs),
 865
 fundamental network characteristics, 10

G

- Garbage-Collection Timer, RIP, 606
 gates, logic, 75
 Gateway Information Protocol
 (GWINFO), 598
 gatewaying, email, 1228
 gateway, 353
 default, IP, 251
 protocols. *See* routing protocols
 Gateway-to-Gateway Protocol (GGP),
 592, 678
 general headers, HTTP. *See* HTTP:
 general headers
 generic TLDs, DNS, 870, **872**, 872
 geopolitical (country code) TLDs, DNS,
 874
 GET, HTTP method, 1349
 GetResponse messages, SNMP. *See*
 SNMP Protocol: messages
 GGP (Gateway-to-Gateway Protocol),
 678
 in TCP/IP core architecture, 592
 giga, 38
 Gigabit (1,000 Mbps) Ethernet, 40
 global address, IP NAT, 430
 global hierarchical domain
 architecture, DNS, **859**
 global routing prefix, IPv6, 386
 global unicast address format, IPv6, 383,
 384

Gopher Protocol, 1431
client/server operation, 1432
menus, 1433
selector strings, 1432
servers, 1432
URL syntax, 1148
World Wide Web and, 1434
.GOV generic TLD, 873
granularity, of IP address classes, 270, 316
gratuitous ARP, in Mobile IP, 499
GRE (Generic Routing Encapsulation), 496
group address, IP multicast, 268
group, multicast, 22
GWINFO (Gateway Information Protocol), 598

H

half-duplex
operation, 42
Telnet NVT, 1446
hardware
addresses, 104, 244
ARP, 214
client and server, 126–127
hashing/encryption algorithms, IPsec, 453
HEAD, HTTP method, 1350
header
caching, HTTP, 1385
message, 19
HELLO (HELLO Protocol), 679
Hello messages, OSPF, 641
HELO command, SMTP, 1268
HEMP (High-level Entity Management Protocol), 1070
HEMS (High-level Entity Management System), 1070
hexadecimal
IPv6 addresses, 378
notation, 68
numbers, 67
arithmetic, 72, 72
conversion, 69, 69, 70
representation of IPv6 addresses, 378
hierarchical
name registration, 835

routing and aggregation in IPv6, 421
topology, OSPF. *See* OSPF: hierarchical topology
hierarchies, Usenet, 1405
High-Level Data Link Control (HDLC) protocol, 144, 182
historic RFCs, 58
HMAC-MD5, Mobile IP authentication, 503
hold timer, BGP, 672
home agent, Mobile IP, 482
hop-by-hop
encoding, HTTP, 1373
headers and caching, HTTP, 1385
options, IPv6, 413
hops
in RIP, 601
in routing and address resolution, 205
routing protocol metric, 594
horizontal communication, OSI Reference Model, 93
host configuration protocols, 975
host ID, IP, 247
host name
history of, 842
lists, ARPAnet, 842
utility (hostname), 1462
Host Precedence Violation, 524
host redirection using ICMP Redirect message, 531
host table name system, 843
files, 842
modern networks and, 846
name registration and, 844
name resolution and, 844
weaknesses of, 845
Host Unreachable, 523
host Utility, 1475
Host, HTTP request header, 1362
host-host communication functions, ND, 583
hostname, TCP/IP host name utility, 1462
host-router discovery functions, ND, 582
HOSTS file, Windows, 843
Host-to-Host Transport Layer, 107
protocols, 132

HTML (Hypertext Markup Language), 1322
elements and tags, 1323, *1324*
text formatting tags, 1326

HTTP (Hypertext Transfer Protocol), 1315
agent-driven negotiation, 1378
authentication methods, 1389
cache
 benefits of, 1382
 control, 1384
 locations, 1383
 hop-by-hop headers and, 1385
 impact on communication, 1335
 intermediary, 1384
 Web client, 1383
 Web server, 1384
chunked transfers, 1375
client/server communication, 1333, **1334**
connections, 1329
content and transfer encodings, 1372
content negotiation, 1378
Continue (100) preliminary reply, 1356
cookies, 1391
data length issues, 1374
end-to-end
 encoding, 1373
 headers, 1385
entities, 1370
entity headers, 1365
 Allow, 1366
 Content-Encoding, 1366
 Content-Language, 1366
 Content-Length, 1366
 Content-Location, 1366
 Content-MD5, 1367
 Content-Range, 1367
 Content-Type, 1367
 Expires, 1367
 Last-Modified, 1367
features, 1381
general headers, 1358
 Cache-Control, 1358, *1359*
 Connection, 1360
 Date, 1360
 Pragma, 1360

Trailer, 1360
Transfer-Encoding, 1361
Upgrade, 1361
Via, 1361
Warning, 1359, *1360*

headers, 1357
 entity. *See* HTTP: entity headers
 general. *See* HTTP: general
 headers
 request. *See* HTTP: request headers
 response. *See* HTTP: response
 headers

hop-by-hop
 encoding, 1373
 headers and caching, 1385

intermediaries, 1334, **1335**
Internet media types, 1370
media types and subtypes, 1370
message trailers, 1375
messages
 generic format, 1342

Request messages
 format, 1343, **1344**
 headers, 1346
 request line, 1344
 request URI, 1345

Response messages
 format, 1346, **1347**
 headers, 1348
 status line, 1347

methods, 1349
 DELETE, 1351
 GET, 1349
 HEAD, 1350
 OPTIONS, 1350
 POST, 1350
 PUT, 1350
 TRACE, 1351
MIME and, 1371
operation, 1329
persistent connections, 1337
pipelining, 1337
privacy, 1388, 1390
proxying, 1386
 caching interaction, 1388
 comparing proxies and caches, 1387
 security and, 1386
quality values, 1378

HTTP, *continued*

reason phrases, 1353, 1354
request headers, 1361
 Accept, 1361
 Accept-Charset, 1362
 Accept-Encoding, 1362
 Accept-Language, 1362
 Authorization, 1362
 Expect, 1362
 From, 1362
 Host, 1362
 If-Match, 1362
 If-Modified-Since, 1362
 If-None-Match, 1363
 If-Range, 1363
 If-Unmodified-Since, 1363
 Max-Forwards, 1363
 Proxy-Authorization, 1363
 Range, 1363
 Referer, 1363
 TE, 1364
 User-Agent, 1364
request/response chain, 1334, 1335
resource paths and directory listings, 1328
response headers, 1364
 Accept-Ranges, 1364
 Age, 1364
 ETag, 1364
 Location, 1364
 Proxy-Authenticate, 1365
 Retry-After, 1365
 Server, 1365
 Vary, 1365
 WWW-Authenticate, 1365
security and privacy, 1388
server-driven negotiation, 1378
state management, 1391
status codes, 1352, 1354
transitory connections, 1336
uniform resource locators (URLs), 1326
URL syntax, 1327
 versions, 1330
http, URL World Wide Web/Hypertext Transfer Protocol syntax, 1146
hybrid routing protocol algorithms, 595
hypermedia, 1319, 1336, 1370
hypertext, 1318

I

IAB (Internet Architecture Board), 53
IAC (Interpret As Command), Telnet, 1447
IANA (Internet Assigned Numbers Authority)
 DNS root domain and, 869
 OUI for multicast address resolution, 223
IANA, 55
IBGP (Internal BGP), 652
IBM (International Business Machines, Inc.), 67
ICANN (Internet Corporation for Assigned Names and Numbers), 55–56, 254
ICMP (Internet Control Message Protocol), 507–508
 See also ICMPv4; ICMPv6
data encapsulation, 520
history, 508
messages
 classes, 512
 codes, 513
 common format, 518, 518, 519
 creation rules and RFC 1122, 516
 processing conventions, 517
 responses, limitations on, 516
 summary, 513, 513
 types, 512
 operation, 510, 511
 standards, 508
 versions, 508
ICMPv4
 See also ICMP; ICMPv6
error messages, 521
 Destination Unreachable, 343, 522, 522, 523
 Parameter Problem, 533, 533, 534
 Redirect, 530, 531, 531
 Source Quench, 525, 526, 526
 Time Exceeded, 349, 527, 528, 529, 529
informational messages, 535
 Address Mask Reply, 543, 544, 544
 Address Mask Request, 543, 544
 Echo (Reply), 536, 537
 Echo Request, 536, 536, 537

- ICMPv4, *continued*
- informational messages, *continued*
 - Information Reply, 536
 - Information Request, 536
 - Router Advertisement, 539
 - message format, 540, 541, 541
 - Router Solicitation, 539
 - message format, 542, 542, 542
 - router discovery, 581
 - Timestamp (Request), 537, 538, 539
 - Timestamp Reply, 537, 538, 539
 - Traceroute, 544, 545, 546
 - ICMPv6
 - error messages, 547
 - Destination Unreachable, 548, 548, 549
 - Packet Too Big, 550, 551, 551
 - Parameter Problem, 554, 555, 555
 - Redirect. *See ICMPv6:*
 - informational messages
 - Time Exceeded, 552, 553, 554
 - informational messages, 557
 - Echo Reply, 558, 559, 559
 - Echo (Request), 558, 559, 559
 - Neighbor Advertisement, 563
 - message format, 563, 563, 564
 - Neighbor Solicitation, 563
 - message format, 564, 565, 565
 - options, 571
 - MTU, 574, 574, 574
 - Prefix Information, 572, 572, 573
 - Redirected Header, 573, 574, 574
 - Source Link-Layer Address, 571, 571, 571
 - Target Link-Layer Address, 572, 572, 572
 - Redirect, 566, 566, 567
 - Router Advertisement, 560
 - message format, 560, 560, 561
 - Router Renumbering, 568, 569, 569
 - Router Solicitation, 560
 - message format, 562, 562, 562
 - ICMPv6 Packet Too Big messages, 417
 - ICS (Internet Connection Sharing), 252 - idempotence
 - HTTP methods and, 1351
 - SNMP message transmission and, 1115 - IEEE (Institute of Electrical and Electronics Engineers), 50
 - IEEE 802
 - 802.11 shared key authentication, 163
 - addresses and IP multicast address resolution, 223
 - networking architecture model, 87
 - Project, 51, 103 - IEEE EUI-64, 388
 - IEN (Internet Engineering Note), 57
 - IEN 2, 123
 - IESG (Internet Engineering Steering Group), 53
 - IETF (Internet Engineering Task Force), 53
 - ifconfig utility, UNIX, 1484
 - If-Match, 1362
 - If-Modified-Since, 1362
 - If-None-Match, 1363
 - If-Range, 1363
 - If-Unmodified-Since, 1363
 - IGRP (Interior Gateway Routing Protocol), 681
 - IKE (IPsec Internet Key Exchange), 471
 - image
 - media type, MIME, 1250
 - FTP Data Type, 1183
 - IMAP (Internet Message Access Protocol), 1297–1299
 - any state commands, 1304, 1304
 - authenticated state commands, 1307, 1308
 - authentication methods, 1306
 - command groups, 1303
 - command tagging, 1303
 - FSM, 1300, 1301
 - history, 1299
 - Not Authenticated state commands, 1306, 1306
 - operation, 1300
 - preattentation, 1302
 - response codes, 1305
 - result codes, 1305
 - selected state commands, 1309, 1310
 - session establishment and greeting, 1302

- IMAP**, *continued*
 session states, 1300
 standards, 1299
IMAP4 (Internet Message Access Protocol version 4). *See* IMAP
IN-ADDR.ARPA hierarchy, DNS, 920, 922
inbound NAT, 437, 438, 439
incremental zone transfers, DNS, 907
indirect datagram delivery (routing), 353
indirect network applications, 1170
industry groups, networking, 51
infinite DHCP leases, 1005
.INFO generic TLD, 873
information representation and groups, binary, 63
Information Sciences Institute (ISI), 57
Information Technology Industry Council (ITIC), 50
information (analog or digital), 62
informational messages
 ICMPv4. *See* ICMPv4: informational messages
 ICMPv6. *See* ICMPv6: informational messages
information-oriented design, SNMP, 1084
information, binary, 62
Institute of Electrical and Electronics Engineers (IEEE), 50
.INT generic TLD, 873
integrated architecture, IPsec, 455
interactive and remote application protocols, 1437
interfaces, OSI Reference Model, 91
Interior Gateway Routing Protocol (IGRP), 681
interior
 nodes, DNS tree, 860
 routing protocols, 593
intermediaries, HTTP, 1334
intermediate devices, message routing, 98
internal BGP (IBGP), 652
internal routers, AS architecture, 593
international networking standards organizations, 49
International Organization for Standardization (ISO), 49
International Telecommunication Union—Telecommunication Standardization Sector (ITU-T), 50
Internet Activities Board, (IAB), 53
Internet Architecture Board (IAB), 53
Internet Assigned Numbers Authority.
 See IANA
Internet Connection Sharing (ICS), 252
Internet Control Message Protocol. *See* ICMP
Internet Corporation for Assigned Names and Numbers. *See* ICANN
Internet Engineering Note number 2 (IEN 2), 123
Internet Engineering Notes (IENs), 57
Internet Engineering Steering Group (IESG), 53
Internet Engineering Task Force (IETF), 53
Internet Key Exchange, IPsec. *See* IPsec: IKE
Internet Layer, TCP/IP, 129
Internet media types, 1370
Internet Message Access Protocol. *See* IMAP
Internet minimum MTU, IP, 343
Internet Network Information Center (InterNIC), 56
Internet Protocol Control Protocol (IPCP), 162
Internet Protocol Mobility Support. *See* Mobile IP
Internet Protocol Version 4. *See* IP
Internet Protocol Version 6. *See* IPv6
Internet Protocol. *See* IP
Internet registration authorities and registries, 55
Internet Relay Chat Protocol. *See* IRC
Internet Research Steering Group (IRSG), 53
Internet Research Task Force (IRTF), 53
Internet Security Association and Key Management Protocol (ISAKMP), 472
Internet Society (ISOC), 52

Internet Standard Management Framework. *See* SNMP Framework

Internet standardization process, 58

Internet standards, 57

Internet standards organizations, 52, 54

Internet Stream Protocol, version 2, 240

internet, 28

internetwork, 28

internetwork datagram delivery, IP, 236

InterNIC (Internet Network Information Center), 56, 880

Interpret As Command (IAC) command, Telnet, 1447

interrupt handling

- TCP, 1449
- Telnet, 1449

interrupt-driven communication, SNMP, 1102

interrupts, 791

intranet, 30

inverse querying, DNS, 921

inverting bits with XOR, 77, 77

IP (Internet Protocol), 233, 236, 242, 256

- address
 - resolution and, 205
 - all ones and all zeros, 263
 - binary, hexadecimal, and dotted decimal representations, 246
- block for DHCP APIPA, 267
- categories, 249
- classless, 281
- concepts, 241
- configuration, 244
- division, 248
- dotted decimal notation, 245
- loopback, 266
- management of, 252
- multicast. *See* IP: multicast, addresses
- multihoming, 251, 253
- notation, 245
- number of, 251
- patterns with special meanings, 264
- private and public, 244
- reserved, private, and loopback, 267, 267

- size, 245
- special meanings, 264
- structure, 247
- subnet. *See* IP: subnetting
- address classes, 256, 256
- bit assignments and network/host ID sizes, 261
- bit patterns and address ranges, 260
- class determination algorithm, 258, 259
- determining class from first octet bit pattern, 260
- granularity of, 270, 316
- network and host capacities, 262, 262

address space, 246

CIDR, 315–316

- addressing block sizes, 322
- addressing example, 324
- IP routing with, 359
- notation, 281
- rationale for, 257

classes. *See* IP: address: classes

classless addressing. *See* CIDR

Classless Inter-Domain Routing. *See* CIDR

datagram

- Copied flag, 347
- delivery, 352, 352
- encapsulation, 330, 331
- format, 332, 332, 334
- fragmentation-related header fields, 346
- options, 336, 337, 337, 338
- size, 340
- Time to Live (TTL) field, 335
- Type of Service (TOS) field, 335, 336

feature protocols, 240

fragmentation, 341, 342, 344

- impact on TCP MSS selection, 778
- multiple-stage, 342
- process, 344, 345

functions, 238

history, 239

internetwork datagram delivery, 236

maximum transmission unit, 341

mobile. *See* Mobile IP, 486

IP (Internet Protocol), *continued*
MTU, 341, **341**
MTU Path Discovery, 343
multicast, 360
multicast address, 268
 resolution, 223
 administratively scoped, 269
 groups, 268
 types and ranges, 268, 268, **268**
 well-known, 269, 269
 group management, 361
multicast routing, 361
NAT (Network Address Translation).
 See IP NAT
packets. *See* IP: datagram
reassembly, 347
routing
 in a CIDR environment, 359
 datagram delivery and, 352
 next-hop, 355, **356**
 router discovery and, 540
 routers and, 239
 in a subnet environment, 359
routes and routing tables, 357, **358**
security. *See* IPsec
standards, 239
subnetting, 273–274
 address formulas, 309
 addresses, 287
 custom masks, 283, **285**
 default masks, 281, 282
 design trade-off, **300**
 determining host addresses, **312**
 determining addresses, **307**, **308**
 example, **277**, 297, **301**, **302**
 ID determination, 279, **279**, **280**
 identifiers, 286
 masks, 277–279
 multi-level, 294
 summary tables, 288
 three-level hierarchy, 276
Variable Length Subnet Masking. *See*
 IP: VLSM
versions, 239
VLSM, 292
 example, **295**
 network split into conventional
 subnets, **293**
 network split using VLSM, **294**

IP datagram expiration, example, **528**
IP encapsulation within IP, 496
IP layer parameters per host, 1046, *1047*
IP layer parameters per interface, 1047,
 1047
IP NAT (IP Network Address
 Translation), 426
address terminology, 430, **432**
advantages of, 428
bidirectional, 437, **438**, **439**
compatibility issues and special
 requirements, 429, 445
disadvantages of, 429
dynamic mappings, 433
FTP and, 447
inbound, 437, **438**, **439**
IPsec and, 447
 mappings, 433
outbound, 434, **435**, **436**
overlapping, 442, **445**, **446**
overloaded, 439, **441**, **442**
PAT, 439, **441**, **442**
port-based, 439, **441**, **442**
static mappings, 433
traditional, 434, **435**, **436**
twice NAT, 442, **445**, **446**
two-way, 437, **438**, **439**
unidirectional, 434, **435**, **436**
IP Next Generation (IPng). *See* IPv6
IP Security. *See* IPsec
IP supernetting, 315
IP within IP encapsulation, 496
IPv6.INT domain for IPv6 reverse
 resolution, 949
ipconfig
 utility (Windows), 1486
 role in DHCP client implementation,
 1056
IPCP (PPP Internet Protocol Control
 Protocol), 162
IPng. *See* IPv6
IPsec (IP Security), 449
 AH (Authentication Header), 461
 datagram placement and linking,
 462, **463**, 463
 format, **464**, 465, **465**
 architectures
 Bump in the Stack (BITS), 455,
 455

IPsec (IP Security), *continued*
architectures, *continued*
 Bump in the Wire (BITW), 456, **456**
 integrated, 455
Authentication Header (AH), 461
components, 452
core protocols, 453
Encapsulating Security Payload. *See*
 IPsec: ESP
end-host implementation, 454
ESP (Encapsulating Security Pay-
load), 450, 466
 authentication data, 466
 authentication field calculation
 and placement, 467
 datagram placement and linking,
 468
 fields, 466
 format, **469**, 470, **470**, **471**
 header, 466
 operations, 467
 trailer, 466–467
history, 450
IKE (Internet Key Exchange), 471
modes, 457
 transport, **458**
 tunnel, **459**
operation, 452
protocols and components, **453**
router implementation, 454
security constructs, 460
services and functions, 451
standards, **451**
support components, 453
transport mode, **458**
tunnel mode, **459**

IPv4. *See* IP

IPv6 (IP version 6), 363, 366
 6BONE experimental network, 370
 AAAA (IPv6 Address) resource
 record, DNS, 949
 address resolution for, 224
 address, 373
 allocations, 381, **382**
 anycast, 398
 binary, decimal, and hexadecimal
 representations, **378**
 broadcast, 375
 colon hexadecimal notation, 379

global unicast address format, 383,
384, **385**
 global routing prefix, 386
 historical routing prefix struc-
 ture, **386**
 routing prefix structure, **386**,
387
hexadecimal notation, 378
interface identifiers, 388
IPv4-compatible, 392, **393**
IPv4-mapped, 392, **393**
IPv6/IPv4 embedding, 392
link-local, 391
loopback, 391
mixed notation, 380
model, 374
multicast. *See* IPv6: multicast:
 addresses
physical address mapping, 388
prefix length representation, 381
private/unregistered/nonroutable,
 390
renumbering, 400
reserved, 390
site-local, 391
size, 376
solicited-node, 396, **397**
space, 376
special, 389
types, 375
unicast, 375
unspecified, 391
zero compression, 379

AH. *See* IPsec: AH

autoconfiguration, stateless, 399
changes and additions to, 368
datagrams, 402, **403**, **403**
 delivery, 420
 encapsulation, 401
 Hop Limit field, 552
 main header format, 404, **404**, **405**
 Next Header field, 405
 options, 412, 413, 414
 sizing, 416
 structure, 402
design goals, 367
development of RIPng for, 600
DHCP for. *See* DHCPv6
DNS changes for, 850, 948

IPv6 (IP version 6), *continued*
DNS extensions, 949
extension headers, 407
chaining using the Next Header field, 407, **408**
Fragment, 411, **411, 411**
No Next Header, 409
Routing, 410, **410, 410**
fragmentation, 418, **419**
global unicast address format, 383, **384, 385**
maximum transmission unit (MTU), 416
MTU, 416
multicast
addresses, 394, **394, 395, 396**
scopes, 395, **396**
ND (Neighbor Discovery protocol). *See ND*
Next Header values, 406
router renumbering, 568
routing, 420
standards, 366
stateless autoconfiguration, 399
transition from IPv4, 370
IPX/SPX protocol suite, 109
IRC (Internet Relay Chat Protocol), 1458
client/server operation and, 1459
logical network, 1459
messaging and channels, 1459
modern Internet and, 1460
IRSG (Internet Research Steering Group), 53
IRTF (Internet Research Task Force), 53
ISAKMP (Internet Security Association and Key Management Protocol), 472
ISDN and PPP Multilink Protocol, 175
ISI (Information Sciences Institute), 57
island networks, host table name system and, 846
ISO (International Organization for Standardization), 49
OSI Reference Model and, 82
ISO Standard 3166-1, 875
ISOC (Internet Society), 52
isochronous transmissions, QoS, 44

iterative name resolution, DNS, 912, **913**
ITIC (Information Technology Industry Council), 50
ITU-T (International Telecommunication Union—Telecommunication Standardization Sector), 50

K

Karn's algorithm, 804
keepalive messages
BGP, 672–673
TCP, 761
kilo, 38
kludge, 429

L

labels, DNS, 863, **864**
lag, 36
LAN (Local Area Network), 26
Last-Modified, HTTP entity header, 1367
latency, 35
QoS and, 43
Layer 1 (Physical Layer), 102
Layer 2 (Data Link Layer), 103
Layer 3 (Network Layer), 105
Layer 4 (Transport Layer), 106
Layer 5 (Session Layer), 109
Layer 6 (Presentation Layer), 110
Layer 7 (Application Layer), 111
layer stack, OSI Reference Model, 88
layer terminology, OSI Reference Model, 89
layering, importance in TCP/IP, 123
layers, abstraction level, 88
layers. *See OSI Reference Model*
LCP (PPP Link Control Protocol). *See PPP: LCP*
leases, DHCP. *See DHCP: leases*
line feed (LF) character, RFC 822 messages, 1235
line-delimiting issues, FTP ASCII data type, 1184
link configuration, PPP LCP, **157**

link layer parameters per interface, DHCP/BOOTP options, 1048
link state, OSPF. *See* OSPF
links, PPP. *See* PPP; LCP
link-state (shortest-path first), routing protocol algorithm, 595
listservs (list servers), email, 1231
load balancing, DNS, 904
local area networks (LANs), 26
local file URL scheme, 1149
local Internet registries, 57
local name mappings, 846
local resolution, DNS, 916
localhost, standard DNS domain name, 885
locality of reference, 211
 spatial, 915
 temporal, 915
Location, HTTP response header, 1364
Loc-RIB, BGP routing information base, 656
logic gates, 75
logical functions, Boolean, 73
loopback addresses
 IP, 266
 IPv6, 391
lower-layer core protocols, 135
LQR. *See* PPP: LQR

M

MAC address, 205
magic cookie, BOOTP, 989
Magic Number, PPP option, 158
Mail Box Protocol, 1219
Mail Exchange resource record, DNS, 892, 925, 939, 939, 947
mail forwarding, 1275
Mail Transfer Protocol (MTP), 1264
mail, electronic. *See* email; SMTP
mailing list expansion, SMTP, 1275
mailing lists, 1231
mailto, URL electronic mail scheme syntax, 1147
Majordomo, 1231
MAN (Metropolitan Area Network), 27
managed nodes, SNMP, 1072
managed objects. *See* MIBs: objects

management framework. *See* SNMP Framework
management information bases. *See* MIBs
manager, SNMP, 1073
manual allocation, DHCP, 1001
manual host configuration, problems with, 974
masks
 bit, 76
 bit masking for IP subnet, 277
 IP custom subnet, 283
Massachusetts Institute of Technology (MIT), 265, 443, 923
master files, DNS. *See* DNS: master files
mathematics of computing, 61
Max Repetitions parameter, SNMPv2/v3, 1107
Max-Access MIB object characteristic, 1088
Max-Forwards, HTTP request header, 1363
maximum segment lifetime (MSL), TCP, 766
maximum segment size (MSS), TCP, 777
maximum transfer unit (MTU), 341
maximum transmission unit (MTU), IPv6, 416
MD5 (Message Digest 5), 453
media types, HTTP Internet, 1370
mega, 38
Memex, 1318
menus, Gopher, 1433
message
 addressing, 20, 21
 formatting, 19, 19
 headers, 19
 media type, MIME, 1257
 nesting, 19
 routing, intermediate devices in, 98
 terminology, 17
 traceroute utility and, 1468
 transmission, 20, 21
Message Digest 5 (MD5), 453
methods, HTTP. *See* HTTP: methods
metrics
 general routing protocol, 594
 OSPF, 629

metrics, *continued*
RIP, 600
metropolitan area networks (MANs), 27
MIBs (management information bases),
 1084–1085
 See also SMI; SNMP Protocol
modules, 1096
object, 1087
 characteristics, 1087
 descriptors, 1091
 groups, 1094
 identifiers, 1091
 name hierarchy, 1092, **1092**
 recursive definition of, 1094
SMI, 1086
 data types, 1089
SNMP generic object groups, **1089**,
 1095
Microsoft
 APIPA and, 1059
 DHCP servers, 1061
 networking, 858
 Windows protocols, 12
.MIL generic TLD, 873
MIME (Multipurpose Internet Mail
 Extensions), 1242
 attachments in, 1243
 basic structures, 1246
 capabilities, 1243
 composite media types, 1253
 Content-Transfer-Encoding header,
 1257
 Content-Type header, 1248
 differences from HTTP constructs,
 1371
 discrete media types, 1246
 encapsulated, 1257
 encoding, 1257
 7-bit and 8-bit message, 1258
 base64, 1258, **1259, 1260**
 quoted-printable, 1258
 entities, 1246
 extension for non-ASCII mail mes-
 sage headers, 1261
 headers
 Content-Description, 1247
 Content-Disposition, 1248
 Content-ID, 1247
 Content-Length, 1248
Content-Location, 1248
Content-Transfer-Encoding, 1247
Content-Type, 1246
MIME-Version, 1246
media types, 1248
 application, 1251, **1252**
 octet-stream subtype, 1252
 audio, 1250, **1251**
 encapsulated, 1257
 image, 1250, **1250**
 message, 1257
 model, 1251, **1251**
 multipart, 1253, **1253, 1256**
 encoding, 1254
 subtypes, 1253
 text, 1250, **1250**
 video, 1251, **1251**
motivation for, 1242
multipart, **1253–1256**
non-ASCII extension, 1261
standards, 1244, **1245**
Usenet and, 1411
Minimal Encapsulation within IP,
 Mobile IP, 496
Minnesota, University of, 1431
miscellaneous TCP/IP troubleshooting
 protocols, 1489, **1490**
MIT (Massachusetts Institute of
 Technology), 265, 443, 923
mixed notation, IPv6 addresses, 380
MLP (Multilink Protocol), PPP. *See* PPP:
 MP
mnemonics, OSI Reference Model, 116
Mobile IP (Internet Protocol Mobility
 Support), 475–476, 480, **481**
 address, 483
 co-located care-of, 485
 foreign agent care-of, 484, **484**
agent
 discovery, 486
 registration, 492
ARP and, 498
 proxying by home agent, **500**
data encapsulation, 495
device roles, 481
efficiency, 500
functions, 482
Generic Routing Encapsulation
 (GRE), 496

- Mobile IP, *continued*
 inefficiency, 502
 limitations of, 479
 messages
 Agent Advertisement, 488, **489**,
 490, **490**
 Agent Solicitation, 487
 Registration Reply, **495**
 Registration Request, 493, **493**, **494**
 Minimal Encapsulation within IP, 496
 Mobility Agent Advertisement Extension, 488
 operation, 480, **481**
 registration, 492
 reverse tunneling, 498
 security and, 503
 transmission triangle, 497
 tunneling, 496, **497**
 Mobility Agent Advertisement extension, Mobile IP, **489**, **489**
 model media type, MIME, 1251
 models, networking, 83
 modems, 36
 moderated mailing lists, 1231
 moderated Usenet newsgroups, 1407
 modified EUI-64 format, 388
 modules, MIB, 1096
 More Fragments, IP datagram field, 346
 Mount Protocol, NFS, 968, **969**
 MP (Multilink Protocol). *See* PPP: MP,
 196
 MRU (Maximum Receive Unit) PPP
 option, 158
 MSL (maximum segment lifetime), TCP,
 766
 MSS (maximum segment size), TCP,
 729, 777
 MTP (Mail Transfer Protocol), 1264
 MTU (maximum transmission unit)
 IP, 341
 IPv6, 416
 MTU option, ICMPv6, 574, **574**
 multicast
 address resolution, 223
 IP. *See* IP: multicast
 IPv6. *See* IPv6: multicast
 messages, 22
 multihomed AS, BGP, 654
 multihomed devices, IP internetwork,
 253
 Multilink PPP architecture, **176**
 Multilink Protocol, PPP. *See* PPP: MP
 multipart media type, MIME, 1253
 multipart messages, MIME. *See* MIME:
 media types: multipart
 multiplexing, **698**, 699
 .MUSEUM generic TLD, 873
 MX (Mail Exchange) resource record,
 DNS, 892, 925, 939, 947, 1227
- N**
- “N” Notation, OSI, 89
 Nagle, John, 815
 Nagle’s Algorithm, TCP, 815
 name
 architecture, 831
 registration, 834
 DNS. *See* DNS: name registration
 methods, 835
 resolution, 837
 caching and, 839
 DNS. *See* DNS: name resolution
 host table, 844
 load balancing and, 840
 techniques, 836
 servers, DNS. *See* DNS: name servers
 spaces, 831
 DNS, **862**
 flat, **832**
 systems, 823, 826
 host table, 843
 .NAME generic TLD, 873
 Name Server resource record, DNS,
 892, 937, 946
 names, DNS. *See* DNS names
 NAT (Network Address Translation). *See*
 IP NAT
 National Committee for Information
 Technology (NCITS), 50
 national Internet registries, 57
 NCITS (National Committee for
 Information Technology), 50
 NCP (Network Control Protocol),
 ARPA, 720, 1170
 NCPs (Network Control Protocols),
 PPP. *See* PPP: NCPs

- ND (Neighbor Discovery Protocol), 575, 576, **578**
 duplicate address detection, 586
 functions compared to IPv4, 580
 host redirection, **587**
 host-host communication functions, 583
 host-router discovery functions, 582
 ICMPv6 messages used, 580
 neighbor unreachability detection, 585
 next-hop determination, 584
 operation, 578
 redirect function, 586
 standards, 577
 updating neighbors, 585
negative caching, 904
Neighbor Advertisement message, 563
Neighbor Discovery protocol. *See* ND
Neighbor Solicitation message, 563
Nelson, Ted, 1318
nesting, message, 19
.NET generic TLD, 873
NetBIOS, 109
NetBIOS/NetBEUI/NBF suite, 109
netiquette, 1400
netstat utility
 UNIX, 1480
 Windows, 1482
Network Address Translation. *See* IP
 NAT
network congestion avoidance, quality of service, 44
Network Control Protocol (NCP), ARPA, 720, 1170
Network Control Protocols (NCPs), PPP. *See* PPP: NCPs
network file and resource sharing, 953
Network File System. *See* NFS
Network Information Center (NIC), 869, 1477
Network Interface Layer, 128
 protocols, 137
Network Interface/Internet Layer
 protocols, TCP/IP, 132
Network Layer, OSI Reference Model, 105
network management. *See* SNMP
 Framework
network message formatting, **19**
Network News Transfer Protocol. *See* NNTP
network news. *See* Usenet
network segment, 28
Network Solutions, Inc. (NSI), 56
network standards and standards organizations, 45
network status utility (netstat), 1479
Network Time Protocol (NTP), 539
Network Unreachable, 523
Network Virtual Terminal (NVT), 1443, **1444**
networking, 6
 advantages, 7
 characteristics, 10
 client/server, 23, **25**
 costs, 8
 disadvantages, 8
 fundamentals, 3
 industry groups, 51
 layers, models, and architectures, 10
 Microsoft, 858
 nonperformance characteristics, 33
 peer-to-peer, 23
 performance, 32
 speed, 34
 standards, 48
 organizations, 49
 structural models, 23
networks, 28
newbies, 1400
news. *See* Usenet; NNTP
newsgroups, 1404
news, URL Network News/Usenet
 scheme syntax, 1148
Next Header
 field, IPv6, 405
 values, IPv6, **406**
NFS (Network File System), 953
 architecture and components, 953
 caching, 963
 client and server responsibilities, 963
 components, 957
 data definition, 959
 design goals, 955
 External Data Representation (XDR), 959
 file system model, 968

NFS (Network File System), *continued*
 Mount Protocol server procedures,
 969
 operation, 961
 remote procedure calls (RPCs), 961
 security in, 959
 server procedures and operations,
 964
 Version 2 and Version 3, 964, 964
 Version 4, 966, 966
 standards, 956
 versions, 956
 XDR, 959
 data types, 960, 960
nibble, binary information groups, 63
NIC (Network Information Center),
 843, 869
nick, 1459
nickname (DNS Registry Database
 Lookup Utility), 1477
NMS (SNMP Network Management
 Station), 1073
NNTP (Network News Transfer
 Protocol), 1397, 1411
 article propagation, 1415
 base commands, 1420, 1421
 client-server communication, 1416
 command extensions, 1422
 LIST, 1423
 newsreader, 1423, 1424
 transport, 1423, 1423
 command syntax, 1420
 interserver communication, 1413
 news access, 1418, 1419
 news article propagation, 1413
 news posting and access, 1416
 newsgroup header retrieval, 1418
 propagation methods, 1414
 reply codes, 1426, 1428
 SMTP, similarity to, 1412
 status responses, 1426
 streaming mode, 1416
 Telnet NVT and, 1420
No Next Header, IPv6 extension
 header, 409
No Operation, IPv4 option, 338
No Route to Destination, 549
nominal performance, 34
Non Repeaters Parameter, SNMPv2/v3
 Table Traversal, 1107
noncontiguous subnet masks, 280
noncore routers, 592
non-overlapping scopes, DHCP leases,
 1010, 1011
nonperformance networking
 characteristics, 33
No-Operation, TCP option, 773
Not Authenticated state commands,
 IMAP, 1306
NOT, Boolean logical function, 73
Notification messages, BGP. *See BGP:*
 messages: Notification
Notify feature, DNS, 898
NS (Name Server) resource record,
 DNS, 892, 937, 946
nslookup Utility, 1473, 1474
NTP (Network Time Protocol), 539
ntp, URL Network News Transfer
 Protocol scheme syntax, 1148
.NU country code TLD, 875
numbers
 base 8, 67
 base 16, 67
 base 10, 65
 base 2, 65
 binary, 65
 decimal, 65
 hexadecimal, 67
 octal, 65
NVT (Network Virtual Terminal),
 Telnet, 1443, 1444
nybble, binary information group, 63

0

OACK (option acknowledgment)
 messages, TFTP, 1208
OAKLEY, IPsec IKE, 472
object-oriented programming, basis for
 HTTP methods, 1349
objects, MIB. *See MIBs: objects*
obscuration, URL, 1157
octal numbers, 66
 conversion, 68
octet
 binary information grouping, 63
 versus byte, 64

octet-stream, MIME application media type subtype, 1252

offline access model, email, 1287

omission of URL syntax elements, 1144

One-Byte Padding Extension, Mobile IP Agent Advertisement message format, 488

online access model, email, 1287

opcodes, ARP, 216

Open message, BGP. *See* BGP: messages: Open

Open Shortest Path First. *See* OSPF

open standards, 47

Open Systems Interconnection Reference Model. *See* OSI Reference Model

option acknowledgment (OACK) messages, TFTP, 1208, 1214

option negotiation, Telnet, 1452 overloading, DHCP, 1044

OPTIONS, HTTP method, 1350

OR, Boolean logical function, 74, 74 setting groups of bits with, 76

organizational (generic) TLDs and authorities, DNS, 870

organizationally unique identifier (OUI), for IP multicast address resolution, 223

.ORG generic TLD, 873

OSI model. *See* OSI Reference Model

OSI protocol suite, 82

OSI Reference Model, 79, 117 analogy, 113, 114 Application Layer (Layer 7), 111 compared to TCP/IP model, 129 concepts, 87 data encapsulation, 95, 96 Data Link Layer (Layer 2), 103 addresses, contrasting to IP addresses, 244 MAC address, 104 sublayers, 103 entities, functions, facilities, and services, 90 history of, 82 importance of, 84 interfaces, 92 layer, 80, 101 abstraction level of, 88 relationships and terminology, 91 stack, 88 terminology, 89 lower layers, 88 mnemonics, 116, 116 “N” notation, 89 Network Layer (Layer 3), 105 network stacks, 90 OSI protocol suite, 82 other network architectures and protocol stacks, 86 PDU and SDU encapsulation, 97 Physical Layer (Layer 1), 102 Presentation Layer (Layer 6), 110 protocol data units (PDUs) and service data units (SDUs), 95 protocol stack, 90 protocols, 93, 94 routing, 98, 99 Session Layer (Layer 5), 109 Transport Layer (Layer 4), 106 upper layers, 88 WAN technologies and, 86

OSPF (Open Shortest Path First), 625–626 areas, 630 AS, 629 calculated OSPF routes, 637 common header format, 640 development, 626 features, 627 hierarchical topology, 630, 632 area border routers, 631 backbone routers, 631 router roles, 631 link-state database (LSDB), 628, 629 messages, 638 authentication, 639 Database Description messages, 641, 642, 642 header format, common, 639, 640 Hello messages, 641, 641, 641 Link State Acknowledgment messages, 644, 644, 644 Link State Advertisement messages, 644, 645, 645 Link State Request messages, 643, 643

- OSPF, *continued*
- messages, *continued*
 - Link State Update messages, 643, 643, **644**
 - messaging, 638
 - multicast version, 361
 - operation, 637
 - route
 - calculation, 637
 - determination using SPF trees, 633, **635**
 - SPF tree, **637**
 - topology, 628
 - OUI (organizationally unique identifier)
 - IP multicast address resolution, 223 - outbound NAT, 434, **435, 436**
 - overlapping/twice NAT, 442
 - overloaded IP NAT, 439, **441, 442**
- P**
- Packet Internet Groper, 1463
 - packet switching, 13, **14**
 - Packet Too Big messages, ICMPv6, 550, 551
 - packets, 18
 - IP, 330
 - PPP LCP, 156
 - padding, in IPv6 options, 414
 - Palo Alto Research Center (PARC),
 - Xerox, 598
 - PAP (Password Authentication Protocol), PPP. *See PPP: PAP*
 - parameter configuration, DHCP, **1034**
 - Parameter Problem messages
 - ICMPv4, 533, **534**
 - ICMPv6, 554, **555**
 - PARC (Xerox Palo Alto Research Center), 598
 - PARC Universal Protocol (PUP), 598
 - Partially Qualified Domain Names (PQDNs), DNS, 866
 - party-based security model,
 - SNMPv2/SNMPv3, 1111
 - passive data connections, FTP, 1178
 - passive opens, TCP, 751
 - PASV command, FTP, 1178
 - path attributes, BGP, 657, **659**
 - payload
 - encapsulation, 97
 - message, 19
 - PDU and SDU encapsulation, OSI Reference Model, **97**
 - PDUs, SNMP, 1103
 - peer-to-peer networking, 23, **24**
 - performance measurement units, 37
 - performance, network, 31
 - persistent connections, HTTP, 1337
 - Physical Layer, OSI Reference Model, 102
 - ping utility, 1463–1464
 - use, 1464–1465
 - options and parameters, 1466
 - ping6 utility (ping for IPv6), 1463
 - pipelining, HTTP, 1337
 - Pointer Indicates the Error, 534
 - Pointer resource record, DNS, 892, 938, 938, 947
 - Point-to-Point Protocol. *See PPP*
 - poisoned reverse, split horizon with, RIP, 612
 - poll-driven communication, SNMP, 1102
 - pool size selection, DHCP, 1009
 - POP (Post Office Protocol), 1288
 - See also POP3*
 - POP3 (Post Office Protocol version 3), 1288
 - authentication process, 1291
 - commands, 1291
 - finite state machine, 1290, **1291**
 - mail exchange process and commands, 1293, **1294, 1296**
 - operation, 1290
 - responses, 1290
 - session states, 1290, **1291**
 - update and session termination process and commands, 1295
 - user authentication process, **1292**
 - POP-before-SMTP authentication, 1278
 - PORT command, FTP, 1178
 - port mapper, RPC, 962
 - port numbers, IANA and management of, 707
 - Port Unreachable, 523, 549
 - port-based (overloaded) NAT, 439, **441, 442**

- ports, 698
- application use of well-known and registered ports, 707
 - BOOTP use of, 981
 - client/server application port mechanics, 703, **705**
 - demultiplexing, 699
 - destination, 699
 - ephemeral, 703
 - multiplexing, 699
 - process multiplexing/demultiplexing, **700**
 - server, 701
 - source, 699
- positive acknowledgment with retransmission (PAR), 732
- positive caching, 904
- Post Office Protocol. *See POP; POP3*
- POST, HTTP method, 1350
- Postel, Jonathan B. (Jon), 55, 123
- PPP (Point-to-Point Protocol), 139, 144
- advantages, 145
 - architecture, **145**
 - authentication protocols, 162
 - See also PPP: PAP; PPP: CHAP*
 - BACP (Bandwidth Allocation Control Protocol), 178, 179
 - BAP (Bandwidth Allocation Protocol), 178
 - messages, 179
 - operation, 179
- CCP (Compression Control Protocol), 169
- compression and decompression, **171**
 - configuration, 171
 - messages, 170
 - operation, 170
- CHAP (Challenge Handshake Authentication Protocol), 163, **164**
- frame formats, 194, **194, 195**
 - code values, 187
 - compression algorithms, 171
- Compression Control Protocol. *See PPP: CCP*
- control frames, 187, 187, 188, 190
 - option format, 188, **189**
 - data frames, **184**
- development and standardization, 144
- ECP (Encryption Control Protocol), 172
- configuration, 173
 - encryption algorithms, 173, **174**
 - encryption and decryption, 174
 - messages, 173
 - operation, 173
- Encryption Control Protocol. *See PPP: ECP*
- feature protocols, 167
 - field compression, 185
 - frame format, 182, **182, 183**
 - Protocol field ranges, 183, **184**
 - Protocol field values, 184
- function and architecture, 145
- functional groups, 147
- Internet Protocol Control Protocol (IPCP), **161**, 162
- LCP (Link Control Protocol), 155, **156**
- frame formats, 190, **191**
 - link configuration, 157, **157**
 - link maintenance, 159
 - link termination, 159
 - messages, 156
- Link Control Protocol. *See PPP: LCP*
- link phases, 148
- LQM (Link Quality Monitoring), 168
- LQR (Link Quality Reporting), 168
- MP (Multilink Protocol), 175
- architecture, 176, **176**
 - fragment frame formats, 196, 197, **197, 198**
 - fragmentation process, 196, 198, **199**
 - ISDN and, 175
 - operation, 177
- Multilink Protocol. *See PPP: MP*
- NCPs (Network Control Protocols), 159
- IPCP, **161**, 162
 - operation, 160
- Network Control Protocols. *See PPP: NCPs*
- operation, 147, **148**
 - options, 157

PPP, *continued*

- PAP (Password Authentication Protocol), 162, **163**
- frame formats, 192, **193**, **193**
- phases, 148, **151**, **152**
- PPP over ATM (PPPoA), 146
- PPP over Ethernet (PPPoE), 146
- SLIP versus, 140
- standards, 151, **153**
- PPPoA (PPP over ATM), 146
- PPPoE (PPP over Ethernet), 146
- PQDNs (Partially Qualified Domain Names), DNS, 866
- Pragma, HTTP general header, 1360
- preauthentication, IMAP, 1302
- Precedence Cutoff in Effect, 524
- prefix discovery, ND, 579
- Prefix Information option, ICMPv6, 572, **572**
- prefix length
 - CIDR, 319
 - representation, IPv6 addresses, 381
- Prefix-Lengths extension, Mobile IP
 - Agent Advertisement messages, 488, **490**, **490**
- Presentation Layer (Layer 6), OSI Reference Model, 110
- PRI (primary rate interface), ISDN, 175
- primary DNS namer server, 893
- privacy, in HTTP, 1390
- private addresses
 - IP, 265
 - IPv6, 391
- private name registration, DNS, 884
- private/dynamic port numbers,
 - TCP/UDP, 703
- .PRO generic TLD, 873
- probe segments, TCP, 811
- procedures and operations, NFS, 964
- processes
 - TCP/IP client and server, 697
 - multiplexing and demultiplexing, **698**
- processing conventions and rules, ICMP, 515
- propagation methods, NNTP, 1414
- proprietary standards, 46
- Protocol field
 - IP, use by TCP and UDP, 699
 - PPP, 183

protocol. *See also individual protocol names.*

- administration and troubleshooting, 1461
- Application Layer, 821
- connection-oriented, 15
- connectionless, 15
- general file transfer, 1167
- host configuration, 971
- interactive, 1435
- lower-layer core, 135
- network file and resource sharing, 951
- Network Interface Layer, 137
- Network Interface/Internet Layer connection, 201
- operations, SNMP, 1102
- OSI Reference Model, 93
- roles, email, 1224
- routing, 589
- stacks, 86
- suites, 12
 - IPX/SPX, 109
 - OSI, 82
 - TCP/IP, 119
 - TCP/IP, 131
 - Transport Layer, 687
- Protocol Unreachable, 11
- proxies, HTTP. *See* HTTP: proxying
- proxy ARP, 221
- Proxy FTP (FTP Third-Party File Transfer), 1174
- Proxy-Authenticate, HTTP response header, 1365
- Proxy-Authorization HTTP request header, 1363
- proxying, ARP, 221
- proxying, HTTP. *See* HTTP: proxying
- pseudo header
 - TCP, 774
 - UDP, 714
- PTR (Pointer) resource record, DNS, 938, 947
- public cache, 1384
- PUP (PARC Universal Protocol), 598
- push function, TCP, 790
- PUT, HTTP method, 1350

Q

q (quality) values, HTTP, 1380
QoS (quality of service), 43
 ATM and, 44
 IP Datagram Type of Service (TOS)
 field and, 335
 latency and, 43
 quadruple, TCP socket identifier, 750
 quality of service. *See* QoS
 Quality Protocol, PPP option, 158
 quality values, HTTP, 1378
 Question Section format, DNS, 935
 quoted-printable encoding, MIME, 1258

R

r (Berkeley Remote) Commands, 1454
Range, HTTP request header, 1363
RARP (Reverse Address Resolution
Protocol), 228, **229**
 bootstrapping and, 228
 limitations of, 231
 servers, 228
rate, signaling, 38
rated or nominal speed, 34
rcp (Remote Copy), 1457
RCV (receive) pointers, TCP, 781, 782
RData field formats, DNS, 936
read process, TFTP, 1205, **1206**
Read Request (RRQ) messages, TFTP,
 1211
reallocation, DHCP leases, **1028**
reason phrases, HTTP, 1353
reassembly, IP message, 347
rebinding process, DHCP, 1028
Rebinding Timer (T2), DHCP, 1008
receive (RCV) Pointers, TCP, 781
receive categories, TCP, 780
receiver SWS avoidance, TCP, 815
Record Route, IP option, 338
recursive DNS name resolution, 913,
 914
Redirect function, ND, 586
Redirect messages
 ICMPv4, 530, **531**
 ICMPv6, 566, **566**
Redirected Header option, ICMPv6,
 573, **574**

redundancy

 DNS name servers, 894
 DNS root name server, 899
reference model. *See* OSI Reference
 Model
reference, locality of, 211
Referer, HTTP request header, 1363
 misspelled as Referer in HTTP, 1363
Refresh field, DNS SOA resource
 record, 897
regional Internet registries (RIRs), 56,
 254, 321
registered port numbers, 707
registration authorities, Internet, 55
Registration Reply messages, Mobile IP,
 491, **495**, **495**
Registration Request messages, Mobile
 IP, 491, **493**, **494**
registration, DNS. *See* DNS: name
 registration
registration
 host table name, 844
 name, 834
 Mobile IP, 492
registries, national Internet, 57
relative domain names, DNS, 865
relative URLs. *See* URLs: relative
relaying
 BOOTP. *See* BOOTP relay agent
 DHCP, 1057
 email, 1266
reliability features, TCP, 793
remote (r) commands, Berkeley, 1454
remote copy (rcp), 1457
remote login (rlogin), 1455
Remote Network Monitoring (RMON),
 1133, **1135**, **1135**
 alarms, events, and statistics, 1136
 MIB hierarchy and object groups,
 1134
 standards, 1134
Remote Procedure Calls (RPCs), 109
remote server email access, 1311
remote shell (rsh), 1456
remote uptime (ruptime), 1457
remote who (rwho), 1457
renewal and rebinding, DHCP leases,
 1032
renewal process, DHCP, 1028

Renewal Timer (T1), DHCP, 1008
renumbering
 IPv6 device, 400
 IPv6 router, 568
replay attacks, 453, 503
Reply messages, ARP, 214
Reply Messages, Application of ICMPv6
 Echo and Echo, 559
Request for Comment. *See* RFCs
request headers, HTTP, 1361
Request messages
 ARP, 214
 HTTP, 1343, **1344**
 RIP, 604
request/response chain, HTTP, 1334
requirements analysis
 for IP subnetting, 298
 for a network, 32
reregistration, Mobile IP, 491
Réseaux IP Européens Network Coordination Center (RIPE NCC)
 regional Internet registry, 57
reserved addresses
 IP, 265
 IPv6, 390
reserved port numbers, TCP/UDP, 702
reset function, TCP, 760
Reset-Ack and Reset-Request messages
 PPP CCP, 170
 PPP ECP, 173
resetting bits, 63
resolution
 address. *See* address resolution
 conflict, DNS, 880
 DNS. *See* DNS: name resolution
 URN, 1161
resolvers, DNS. *See* DNS: name resolvers
resource paths, HTTP, 1328
resource records, DNS. *See* DNS:
 resource records
response headers, HTTP, 1364
Response messages
 HTTP, 1346, **1347**
 PPP CHAP, 164
 RIP, 604
retransmissions
 BOOTP, 982
 DHCP, 1037
 TCP, 794
Retry field, DNS Start of Authority resource record, 897
Retry-After, HTTP response header, 1365
reverse address resolution, 227
Reverse Address Resolution Protocol (RARP), 228
reverse name resolution, DNS, 920
 IPv6, 949
reverse tunneling, Mobile IP, 498
RFC 1497 vendor extensions,
 DHCP/BOOTP options, *1045*
RFC 822 email message format, 1234
 development of, 1235
 header field, 1238
 groups, 1237, 1238, *1239*
 structure, 1237
 message, 1235
 structure, 1236
 processing and interpretation, 1241
RFC Editor, 57
RFCs (Requests for Comments), 57
 categories, 58
 Best Current Practice, 58
 Experimental, 58
 Historic, 58
 Informational, 58
 Internet Draft (ID), 58
 Internet Standard, 59
 Proposed Standard/Draft
 Standard/Standard, 58
 master list, 58
RFCs by number. *See* “RFCs by Number” following this index
RIBs (routing information bases), BGP, 656
RIP (Routing Information Protocol), 598, 604
 30-second timer, 605
 algorithm issues, 607
 counting to infinity, 607, **608**
 routing loops, 607
 slow convergence, 607
 small infinity, 609
ASes (autonomous systems), **601**
 default routes, 604
 development of RIP-2 and RIPng, 600
 Garbage-Collection Timer, 606
 message types, 604

- RIP, *continued*
- metric, issues with, 610
 - problems, 606
 - RIP-1 (RIP version 1), 614
 - messaging, 615, 615, **616**
 - version-specific features, 617
 - RIP-2 (RIP version 2), 617
 - messaging, 618, 619, 619, **620**
 - version-specific features, 617
 - RIPng (RIP for IPv6), 620
 - messaging, 621, 622, **622**
 - version-specific features, 620
 - route determination and information propagation, 601, **603**
 - routing information and route distance metric, 600
 - special features, 610
 - hold down, 613
 - split horizon, 611
 - split horizon with poisoned reverse, 612, **613**
 - standardization, 598
 - timeout timer, 605
 - triggered updates, 606, 612
 - update messaging, 605
 - version-specific features, 614
- RIPE NCC (Réseaux IP Européens Network Coordination Center)
- regional Internet registry, 57
- RIPng. *See* RIP: RIPng
- IPv6. *See* RIP: RIPng
- RIRs (regional Internet registries), 56, 254, 321
- rlogin (Berkeley remote login), 1455
- RMON (Remote Network Monitoring), 1133, **1135**, 1135
- robots, mailing list, 1231
- robustness principle, TCP, 726
- roles
- DHCP client/server, 1015
 - DNS name servers, 893
 - email communication, 1224
 - Mobile IP, 481
 - TCP/IP hardware and software, 127
- root domain, DNS, 860
- root name servers, DNS, 899
- round-trip time (RTT) calculation, TCP, 804
- routable protocols, vs. routing protocols, 358
- route
- aggregation, 359
 - determination
 - BGP, 659
 - OSPF, 633, **635**
 - RIP, 601
 - propagation, RIP, 601, **603**
- Route Tracing Utility. *See* traceroute
- routed, RIP daemon for UNIX, 598
- Router Advertisement and Router Solicitation messages
- ICMPv4, 539
 - ICMPv6, 560
- router discovery
- IP, 540
 - IPv6, 581
- router loops, 335, 527, 552, **553**, 607
- router renumbering, IPv6, 568, 569
- routers
- core, 592
 - default, 530
 - IP routing and, 239
 - IPv6, 420
 - noncore, 592
- OSI Reference Model Network Layer and, 106
- OSPF, 631
- used for proxy ARP, 221
- routes and routing tables, IP, 357, **358**
- routing
- classless, benefits of, 317
 - IP, 355
 - IP classless, 281
 - IP multicast, 361
 - IPv6, 420
 - OSI Reference Model and, 98
 - software, 252
 - source, 338
- routing entries, 357
- Routing extension header, IPv6, 410, 410
- routing information bases (RIBs), 656
- Routing Information Protocol. *See* RIP
- routing
- loops, 335, 527, 552, **553**, 607
 - policies, BGP, 654

- routing, *continued*
- prefix
 - IPv6 unicast addresses, 385
 - structure, IPv6 global unicast address format, 386
 - protocols, 589
 - algorithms and metrics, 594
 - architectures and concepts, 591
 - exterior and interior, 593
 - static and dynamic, 595
 - tables, IP, 357
 - RPCs (Remote Procedure Calls), 109, 962
 - RR. *See* DNS: resource records
 - RRQ (Read Request) messages, TFTP, 1211
 - rsh (Berkeley Remote Shell), 1456
 - RTT (round-trip time) calculation, TCP, 804
 - ruptime (Berkeley remote uptime), 1457
 - rwho (Berkeley remote who), 1457
- ## S
- SACK (selective acknowledgment), TCP, 798
 - SAD (Security Association Database), IPsec, 460
 - safe and idempotent HTTP methods, 1351
 - SAs (Security Associations), IPsec, 460
 - scalability, TCP/IP, 124
 - schemes and scheme-specific syntaxes, URL, 1146
 - scopes (lease address ranges), DHCP, 1009, **1010**, 1010
 - scopes, IPv6 multicast, 395
 - SDLC (Synchronous Data Link Control) protocol, 144
 - SDOs (Standards Developing Organizations), 50
 - secondary DNS name server, 893
 - second-level DNS domains, 860
 - Secure Hash Algorithm 1 (SHA-1), 453
 - secure login (slogin), 1456
 - secure shell (ssh), 1457
 - Secure Sockets Layer (SSL), 111, 1390
 - security
 - DHCP, 1064
 - HTTP, 1388
 - IP. *See* IPsec
 - Mobile IP, 503
 - NFS, 959
 - SMTP, 1277
 - SNMP, 1075, 1110
 - security associations (SAs), IPsec, 460
 - security extensions, FTP, 1176
 - security methods, SNMPv2/v3, 1111
 - security parameter index (SPI), IPsec, 461
 - Security, IP option, 338
 - segment retransmission timers, TCP, 794
 - segment
 - coaxial cable, 29
 - network, 28
 - TCP. *See* TCP segments
 - Selected state, IMAP4, 1309, *1310*
 - selective acknowledgment (SACK), TCP, 798
 - Selective Acknowledgment Permitted, TCP option, 773
 - Selective Acknowledgment, TCP Option, 773
 - selector string, Gopher, 1432
 - selectors, IPsec, 461
 - send (SND) pointers, TCP, 781
 - send window and usable window, 737
 - sender and receiver in TCP/IP email, decoupling, 1222
 - sender SWS avoidance and Nagle's algorithm, 815
 - Sequence Number field
 - PPP MP, 197
 - TCP, 783
 - sequence number synchronization, TCP, 737, 758
 - Serial Line Internet Protocol. *See* SLIP
 - Server, HTTP response header, 1365
 - server-driven negotiation, HTTP, 1378
 - servers, 23
 - BOOTP, 980
 - conflict detection, DHCP, 1061
 - data transfer process (Server-DTP), FTP, 1174
 - DNS name. *See* DNS: name servers

servers, *continued*
hardware, 126
implementations, DHCP, 1054
procedures and operations, NFS, 964
processing, SMTP, 1241
RARP, 228
responsibilities, DHCP, 1014
roles, 23
software, 23, 126
software features, DHCP, 1054
structure, Usenet, 1413
support functions, DNS, 889
TCP port assignments, 742
UDP port assignments, 716
Web, 1321
service commands, FTP, 1186
services, TCP/IP, 125
Session Layer (Layer 5), OSI Reference Model, 109
SetRequest messages, SNMP, 1107
setting bits using OR bit mask, 76, 76
SGML (Standard Generalized Markup Language), 1322
SGMP (Simple Gateway Monitoring Protocol), 1070, 1100
SHA-1 (Secure Hash Algorithm 1), 453
shared cache, 1384
shared key authentication, IEEE 802.11, 163
shortcuts for computing IP subnet host addresses, 313
shortest path first (link-state) routing protocol algorithm, 595
shrinking TCP window, problems with, 809, **810**
signaling rate, 38
Silly Window Syndrome (SWS), TCP, 812
Simple Gateway Monitoring Protocol (SGMP), 1070, 1100
Simple Mail Transfer Protocol. *See* SMTP
Simple Network Management Protocol. *See* SNMP Framework; SNMP Protocol
simple structure, MIME, 1246
simplex network operation, 41
simultaneous connection termination, TCP, 766
simultaneous open connection establishment, TCP, 755
site-local addresses, IPv6, 391
SIZE extension, SMTP, 1281
sizes of networks, 25
SKEME, IKE, 472
slash (CIDR) notation, 303, 319, **320**
sliding the TCP send window, 738, **739**
sliding window system, TCP. *See* TCP: sliding window system
SLIP (Serial Line Internet Protocol), 141
compressed (CSLIP), 143
data framing method and general operation, 141
escape character (ESC) in, 142
operation, **143**
PPP vs., 140
problems, 142
slogin (secure login), 1456
SMI (Structure of Management Information), 1083–1084
See also MIBs; SNMP Framework
data types, 1089, **1089**
defining MIB objects, 1086
textual conventions, 1090
smoothing factor, TCP RTT, 804
SMTP (Simple Mail Transfer Protocol), 1263–1264
commands, 1279, **1279**
communication and message transport methods, 1265
connection establishment, 1268
using SMTP extensions, 1269
connection termination, 1270
enhanced status code replies, 1284
extensions, 1276, **1276**
AUTH, 1281
mail transaction process, 1271, **1273**
reply codes, 1281, **1283**
security issues, 1277
SIZE extension, 1281
special features, 1275
address debugging, 1275
mail forwarding, 1275
mail gatewaying, 1275
mail relaying, 1275
mailing list expansion, 1275
turning, 1275

- SMTP, *continued***
- special requirements for email addresses, 1227
 - standards, 1264
 - Telnet NVT and, 1279
 - terminology, 1267
 - transaction session establishment and termination, **1269**
 - SND (send) pointers, TCP, 781–782
 - sneakernet, 7
 - SNMP (Simple Network Management Protocol)**, 1099
 - See also* SNMP Framework; SNMP Protocol
 - SNMP Framework**, 1070
 - See also* SNMP Protocol
 - architecture, 1076
 - components, 1075
 - design goals, 1071
 - device types, 1072
 - early development of, 1070
 - entities, 1073
 - information-oriented design of, 1084
 - managed node entities, 1073
 - management information base (MIB), **1089**
 - generic object groups, 1095
 - network management station entities, 1073
 - operational model, 1073, **1074**
 - SNMP agents, 1073
 - SNMP applications, 1073
 - SNMP management information base (MIB), 1073
 - SNMP manager, 1073
 - standards, 1079
 - SNMPsec, **1080**
 - SNMPv1, **1080**
 - SNMPv2c, **1081**
 - SNMPv2p, **1080**
 - SNMPv2u, **1081**
 - SNMPv3, **1082**
 - versions, 1076
 - SNMPsec, 1077
 - SNMPv1.5, 1078
 - SNMPv1, 1077
 - SNMPv2 (SNMPv2u), User-Based, 1078
 - SNMPv2* (SNMP “asterisk”), 1078

- SNMPv2c (community-based SNMPv2), 1078
- SNMPv2, 1078
- SNMPv3, 1079
- SNMP Protocol**, 1099, 1100
 - See also* SNMP Framework
 - communication methods, 1102
 - information notification process, 1109
 - information poll process, 1104, **1105**
 - messages
 - format, 1116, **1118**
 - generation, 1114
 - vs. PDUs, 1117
 - SNMPv1 message format, 1119, **1119, 1119**
 - SNMPv1 PDU formats, 1120, **1121, 1122**
 - SNMPv1 Trap-PDU format, **1121**
 - SNMPv2 common PDU format, **1126, 1128**
 - SNMPv2 GetBulkRequest-PDU Format, **1129**
 - SNMPv2c message format, 1124, **1124, 1124**
 - SNMPv2p message format, 1123, **1123, 1123**
 - SNMPv2u message format, 1124, **1125, 1125**
 - SNMPv3 message format, 1129, **1130, 1130**
 - object modification process, 1107, **1108**
 - PDU
 - classes, 1103, **1103**
 - format, 1117
 - protocol operations, 1102
 - and transport mappings, 1101
 - security issues and methods, 1110
 - table traversal, 1106
 - transport mappings, 1114
 - variable binding format, **1118**
 - SOA (Start Of Authority) resource record, DNS, 938, 946
 - socket identifier quadruple, TCP, 750
 - sockets, 109, 706
 - software
 - client, 23, 126
 - roles, client/server, 127

- software, *continued*
 - routing, 252
 - server, 23, 126
 - solicited-node multicast addresses, 396
 - sonar, 1463
 - Source Host Isolated, 524
 - Source Link-Layer Address option, ICMPv6, 571, 571
 - source port and destination port numbers, TCP/UDP, 699
 - Source Quench messages, ICMPv4, 525, 526
 - Source Route Failed, 524
 - source routing, 338
 - spam, 1157
 - spatial locality of reference, 915
 - SPD (Security Policy Database), IPsec, 460
 - speed
 - rated or nominal, 34
 - network, 34
 - SPF (shortest path first), 626
 - SPF trees, OSPF, 633, **637**
 - SPI (Security Parameter Index), IPsec, 461
 - split horizon with poisoned reverse, RIP, 611–**613**
 - ssh (secure shell), 1457
 - SSL (Secure Sockets Layer), 111, 1390
 - stack, OSI Reference Model layer, 88
 - standardization process, Internet, 58
 - standards
 - See also individual protocols*
 - de facto*, 48
 - networking, 48
 - open, 47
 - proprietary, 46
 - standards developing organizations (SDOs), 50
 - standards
 - organizations, Internet, **54**
 - track, RFC process, 58
 - Stanford University, 843, 1298
 - Start Of Authority resource record, DNS, 892, 938, 938, 946
 - state management using cookies, HTTP, 1390
 - stateful autoconfiguration, 1065
 - stateless autoconfiguration, 399, 1065
 - static and dynamic
 - address mappings, IP NAT, 433
 - ARP cache entries, 219
 - routing protocols, 595
 - stream mode, FTP, 1181
 - streaming mode, NNTP, 1416
 - Strict Source Route, IP option, 338
 - Structure of Management Information.
 - See SMI*
 - structured name space, DNS, 832
 - stub AS, BGP, 654
 - stub resolver, DNS, 911
 - subdomains, DNS, 860
 - sublayers, 89
 - physical layer, 103
 - subnet, 28
 - subnetting. *See IP: subnetting*
 - subnetwork, 28
 - subprotocols, 87
 - Success message, PPP CHAP, 164
 - suites, protocol, 12
 - supernetting. *See CIDR*
 - support protocols, IP, 505
 - SWS (Silly Window Syndrome), TCP, 812
 - symbolic names for addressing, 826
 - symmetric operation of Telnet, 1440
 - SYN field and control messages, TCP, 753
 - Synchronous Data Link Control (SDLC) protocol, 144

T

- T1 (Renewal Timer), DHCP, 1008
- T2 (Rebinding Timer), DHCP, 1008
- table name registration, 835
- table traversal, SNMP, 1106
- table-based name resolution, 837
- tables, routing, 357
- tags, HTML, 1323
- TANSTAAFL, 8
- Target Hardware Address, ARP, 214
- Target Link-Layer Address option, ICMPv6, 572, 572
- Target Protocol Address, ARP, 214
- TCB (Transmission Control Block), TCP, 751

TCP (Transmission Control Protocol),
 720, 727
 acknowledgment header flag (ACK),
 747
 acknowledgment mechanics, 780
 adaptive retransmission, 803
 aggressive retransmission, 800
 applications and server port assignments, 742
 characteristics, 724
 congestion collapse, 817
 congestion
 avoidance, 818
 fast recovery, 819
 fast retransmit, 819
 handling, 816
 slow start, 818
 connections, 745
 active and passive opens, 751
 establishment, 752, 754, 755
 parameter exchange, 759
 sequence number synchronization, 757
 SYN and ACK messages, 753
 keepalive messages, 761
 management, 760
 preparation, 750
 reset segments, 761
 termination, 762, 764, 765
 transmission control blocks
 (TCBs), 751
 data handling and processing, 728
 data stream processing and segment packaging, 730
 finite state machine (FSM), 746, 748, 750
 functions, 722
 history, 720
 maximum segment lifetime (MSL), 766
 maximum segment size. *See* TCP: MSS messages. *See* TCP: segments
 MSS (maximum segment size), 729, 777
 default, 778
 negotiation, 779
 nondefault value specification, 779
 selection, 778
 MSS (maximum segment size)
 parameter, 759
operation, 721
options, 773
ports and connection identification, 741
probe segments, 811
pseudo header for checksum calculations, 774, 775, 775
push function, 790
reliability and flow control features, 793
reset function, 760
retransmission timer calculations, 803
retransmissions and retransmission timers, 794
robustness principle, 726
RTT calculation, adaptive retransmission and, 803
segments, 771
 checksum calculation, 774, 776
 control bits, 772
 format, 770, 772
 packaging, 728
selective acknowledgment (SACK), 798, 802
send window. *See* TCP: sliding window system
sequence number synchronization, 758, 759
sequence numbers, 729
silly window syndrome, 812, 814
 avoidance algorithms, 815
sliding window system, 731
categories and send window
 terminology, 738
closing send window, 808
mechanics, 780
receive categories and pointers, 783
send (SND) and receive (RCV)
 pointers, 781
send window and usable window, 737
sliding the send window, 738
transmit and receive categories, 780, 782
window management issues, 809
window size adjustment and flow control, 805, 807
socket identifier quadruple, 750

TCP, *continued*
standards, 721
SWS. *See* TCP: sliding window system;
TCP: silly window syndrome
three-way handshake. *See* TCP: connections: establishment
TIME-WAIT state, 765
transmission round-trip time (RTT),
803
UDP vs., 689
urgent function, 791
Telnet Interrupt Handling, 1449
usable window. *See* TCP: sliding window system
TCP parameters, DHCP/BOOTP
options, 1048, 1048
TCP/IP, 119, 122
See also specific protocols and technologies
architecture, 128
client processes, 697
client/server operation, 126, 697
factors in success of, 123
history, 122
model, 128
vs. OSI Reference Model, 129
ports. *See* ports
protocols, 131, 132, 134
role of host configuration protocols
in, 975
server processes, 697
services, 125
sockets, 109
transactional roles, 127
TCP/UDP ports, 698, 702
TE, HTTP request header, 1364
Telecommunications Industry
Association (TIA), 50
Telecommunications Standardization
Sector of the International
Telecommunication Union
(ITU-T), 82
Telnet Protocol, 1438–1439
accessing other servers with, 1442
applications, 1440
commands, 1446, 1448
communication and the Network Virtual Terminal (NVT), 1443, 1444
ASCII control codes, 1445, 1445
connections and client/server operation, 1441
history, 1438
interrupt handling, 1449
options, 1451, 1451
negotiation, 1452
subnegotiation, 1453
sessions and client/server communication, 1441
symmetric operation of, 1440
URL scheme syntax (telnet), 1149
telnet, URL Telnet scheme syntax, 1149
temporal locality of reference, 915
Terminate-Ack and Terminate-Request
messages
PPP CCP, 171
PPP ECP, 173
PPP LCP, 159
PPP NCPs, 160
text formatting tags, HTML, 1326
text media type, MIME, 1250
text representation, DNS master file,
891
Text resource record, DNS, 892, 939,
940, 947
textual conventions, SMI, 1090
text, MIME text media type, 1250
TFTP (Trivial File Transfer Protocol),
1199–1200
bootstrapping and, 1200
connection
establishment, 1203
termination, 1202
data block numbering, 1205
FTP and, 1201
identification, 1203
initial message exchange, 1204
messages, 1211
Acknowledgment, 1213, 1213,
1213
Data, 1212, 1212, 1213
Error, 1213, 1213, 1214
Option Acknowledgment, 1214,
1214, 1214
Read Request and Write Request,
1211, 1212, 1212
messaging, 1204
operation, 1202, 1204

TFTP, *continued*
 options, 1211, 1211
 negotiation, 1208
 read process, 1205, 1206
 with option negotiation, 1210
 transfer identifier (TID), 1203
 write process, 1206, 1207
third-party
 cookies, 1392
 file transfer (Proxy FTP), 1174
threading, Usenet article, 1419
three-way handshake
 PPP CHAP, 163
 TCP, 753
throughput, 35
TIA (Telecommunications Industry Association), 50
TID (transfer identifier), TFTP, 1203
Time Exceeded messages
 ICMPv4, 527, 529
 ICMPv6, 552, 553
Time to Live (TTL)
 field, IP Datagram, 335
 interval, DNS, 902
Time-Remaining messages, PPP LCP, 159
Timestamp
 IP option, 338
 (Request) messages, ICMPv4, 537, 538
 Reply messages, ICMPv4, 537, 538
TIME-WAIT state, TCP, 765
TLDs (top-level domains), DNS, 860
TLV encoding of BOOTP vendor information fields, 990
.TO country code TLD, 875
Token Ring, 28
top-level domains (TLDs), DNS, 860
TOS (Type of Service) field, IP, 335
Total Length field, use in IP fragmentation, 346
TRACE, HTTP method, 1351
Traceroute messages, ICMPv4, 544, 545
traceroute utility, 1467
 operation, 1468, 1469
 options and parameters, 1470
 traceroute message and, 1468
traceroute, 1467
tracert, abbreviation, 1467
use, 1469
Traceroute, IP option, 338
Traditional NAT, 434, 435, 436
traffic
 flow and types, BGP, 653
 prioritization and shaping, quality of service, 44
trailer
 calculation and placement, ESP, 467
 footer, alias 19
Trailer, ESP, 466
Trailer, HTTP general header, 1360
trailers, HTTP, 1375
transaction identifier (XID), DHCP, 1036
transactional roles, TCP/IP, 127
transfer encodings, HTTP, 1372
transfer identifier (TID), TFTP, 1203
Transfer-Encoding, HTTP general Header, 1361
transistors and binary information, 62
transit
 AS, BGP, 655
 traffic, BGP, 654
transition from IPv4 to IPv6, 370
transitory connections, HTTP, 1336
Transmission Control Blocks (TCBs), TCP, 751
Transmission Control Program, original meaning of TCP, 122
Transmission Control Protocol. *See* TCP
transmission round-trip time (RTT), TCP, 803
transmission triangle, Mobile IP, 497
transport extensions, NNTP, 1423
Transport Layer (Layer 4), OSI Reference Model, 106
transport mappings, SNMP, 1114
transport mode, IPsec, 457, 458
Trap and Trapv2 messages, SNMP, 1109
tree, DNS. *See* DNS: names
triggered updates, RIP, 606, 612
Trivial File Transfer Protocol. *See* TFTP
troubleshooting utilities and protocols, TCP/IP, 1461
 miscellaneous, 1490
true and false Boolean values, 73

Truscott, Tom, 1398
truth tables, Boolean logic, 73
TTL (Time to Live)
 Field, IP, router loops and, 335
 DNS, 903
tunnel mode, IPsec, 457, **459**
tunneling, Mobile IP, 496, **497**
turning, SMTP, 1275
.TV country code TLD, 875
Two-Way NAT, 437, **438**, **439**
TXT (text) resource record, DNS, 939,
 947
Type of Service (TOS) field, IP
 datagram, 335, **336**

U

UDP (User Datagram Protocol),
 711–712
 applications and server port use, 717
 messages, 714, **714**, **715**
 operation, 713
 pseudo header, 714, **715**
 TCP vs., 689
UDRP (Uniform Domain Name Dispute
 Resolution Policy), DNS, 881
unary operator, NT, 73
UNC (University of North Carolina),
 1399
unicast, 21, 22
 address format, IPv6. *See* IPv6: global
 unicast address format
unidirectional IP NAT, 434, **435**, **436**
Uniform Domain Name Dispute
 Resolution Policy, DNS, 880
Uniform Resource Identifiers. *See* URIs;
 URLs; URNs
unintentional cookies, 1392
unit prefixes, 38
United States ASCII (US-ASCII)
 character set, 1444
United States Department of Defense
 (DoD), 371
units, 38
University of Minnesota, 1431
University of North Carolina (UNC),
 1399
UNIX configuration file /etc/hosts,
 843, 1463

Unix-to-Unix Copy Protocol (UUCP)
 email and, 1219
 Usenet and, 1398
unmoderated Usenet newsgroups, 1407
Unrecognized IPv6 Option
 Encountered, 556
Unrecognized Next Header Type
 Encountered, 556
unreliable protocol, **732**
unsafe characters in URLs, 1145
unspecified address, IPv6, 391
Update messages, BGP. *See* BGP:
 messages: Update
Upgrade, HTTP general Header, 1361
urgent function, TCP, 791
URIs (Uniform Resource Identifiers),
 1140
 See also URLs; URNs
 categories (URLs and URNs), 1141
 standards, 1142
URLs (Uniform Resource Locators)
 See also URIs; URNs
 abbreviation, 1156
 common Internet scheme syntax,
 1143
 example, **1144**
 explicit delimiting and redirectors,
 1156
 fragments, 1145
 general syntax, 1142
 length and complexity issues, 1154
 obscuration and obfuscation, 1156
 parameter strings, 1155
 problems with, 1159
 relative, **1152**
 interpretation rules, 1151
 syntax and base URLs, 1150
 schemes and scheme-specific syn-
 taxes, 1146
 special encodings, 1145, **1145**
 special syntax rules, 1149
 unsafe characters, 1145
 wrapping and delimiting, 1155
URNs (Uniform Resource Names),
 1159–1160
 See also URIs; URLs
 namespaces, 1160
 syntax, 1160
 resolution and implementation diffi-
 culties, 1161

usable window, TCP, 737
US-ASCII (United States ASCII)
 character set, 1235, 1444
Usenet
 See also NNTP
 access, 1417
 addressing, 1404
 article propagation, 1414, 1416
 Big Eight newsgroup hierarchies,
 1405
 headers, 1409, 1410
 history, 1398
 message format, 1408
 MIME messages and, 1411
 model, 1401–**1403**
 posting, 1417
 newsgroup hierarchies, 1405–**1406**
 newsgroups, 1404
 operation, 1399
 overview, 1398
 reading, 1417
 server structure, 1413
 threading, 1419
 transport methods, 1400
user (registered) port numbers, 703
user commands, FTP, 1194
User Datagram Protocol. *See* UDP
User-Agent, HTTP request header, 1364
user-based security model (USM),
 SNMPv2/SNMPv3, 1112
USM (User-Based Security Model),
 SNMPv2/SNMPv3, 1112
UUCP (Unix-to-Unix Copy Protocol)
 email and, 1219
 Usenet and, 1398
UUCP-style email addressing, 1229

V

VACM (View-Based Access Control
 Model), SNMPv2/v3, 1112
Variable Length Subnet Masking
 (VLSM), IP. *See* IP: VLSM
Vary, HTTP response header, 1365
vendor information
 extensions, BOOTP, 989
 fields, BOOTP, 990
vertical communication, OSI Reference
 Model layers, 91

Via, HTTP general header, 1361
video media type, MIME, 1251
View-Based Access Control Model
 (VACM), SNMPv2/v3, 1112
virtual private networking (VPN), 30,
 496
VLSM, IP. *See* IP: VLSM
VPN (virtual private networking), 30,
 496

W

WAN (Wide Area Network), 26
Warning, HTTP general header, 1359
Web. *See* World Wide Web
WECA (Wireless Ethernet Compatibility
 Alliance), 51
well-known
 (privileged) TCP/UDP port num-
 bers, 702
IP multicast addresses, 269, 269
IPv6 multicast addresses, 396
white space, in DNS Master Files, 945
whois (DNS registry database lookup
 utility), 1477
Wide Area Network (WANs), 26
window, TCP. *See* TCP: sliding window
 system
Windows winipcfg utility, **1489**
Windows Calculator program, 68
Windows protocols, Microsoft, 12
Windows Sockets (Winsock), 706
winipcfg utility for Windows, 1488, **1489**
Winsock, 706
Wireless Ethernet Compatibility
 Alliance (WECA), 51
wireless
 LANs (WLANs), 26
 MANs (WMANs), 27
 PANs (WPANs), 27
 WANs (WWANs), 26
word, binary unit, 64
World Wide Web, 1318
 See also HTTP
 addressing through HTTP URLs,
 1326
 browsers, 1321
 components, 1320
 email access, 1313

World Wide Web, *continued*
 history, 1318
 major functional components, 1320
 media and the Hypertext Markup
 Language (HTML), 1322
 servers, 1321
WPANs (wireless PANs), 27
wrapper protocol, UDP, 714
Write Request (WRQ) messages, TFTP,
 1211
WWANs (wireless WANs), 26
WWW-Authenticate, HTTP response
 header, 1365

X

X3 (Accredited Standards Committee
 X3, Information Technology),
 50
Xanadu, 1318
XDR (External Data Representation),
 959
 data types, 960, *960*
Xerox Network System (XNS), 598
Xerox Palo Alto Research Center
 (PARC), 598
XID (transaction identifier), DHCP,
 1036
XNS (Xerox Network System), 598
XOR (Exclusive OR)
 Boolean logical function, 75
 inverting bits with, 77
 truth table, 75

Z

zero compression, IPv6 Addresses, 379
zones of authority, DNS, **883**
zones, DNS. *See* DNS: zones

RFCs BY NUMBER

RFC 95: 1219	RFC 865: 1046	RFC 1081: 1289
RFC 97: 1439	RFC 879: 722, 778	RFC 1084: 980
RFC 114: 1170	RFC 881: 848	RFC 1094: 956
RFC 155: 1219	RFC 882: 849	RFC 1105: 648, 649
RFC 172: 1170	RFC 883: 849	RFC 1122: 722
RFC 196: 1219	RFC 887: 1046	DHCP messaging and: 1037
RFC 226: 842	RFC 891: 679	ICMP message cre- ation rules and: 516
RFC 265: 1170	RFC 894: 1048	ICMP standards and: 509
RFC 354: 1170	RFC 896: 722, 816	RFC 1131: 627
RFC 542: 1170	RFC 903: 228	RFC 1134: 144
RFC 606: 843	RFC 904: 684	RFC 1146: 722
RFC 608: 843	RFC 918: 1288	RFC 1155: 1080
RFC 675: 122, 720	RFC 937: 1289	RFC 1156: 1080
RFC 733: 1235	RFC 943: 1477	RFC 1157: 1080
RFC 760: 239	RFC 950: 274, 285, 305, 509	RFC 1163: 649
RFC 765: 1170	RFC 951: 978	RFC 1171: 144
RFC 768: 712	RFC 959: 1171	RFC 1176: 1299
RFC 772: 1219, 1264	RFC 977: 1400, 1412	RFC 1179: 1046
RFC 780: 1264	RFC 1001: 1049	RFC 1183: 850
RFC 783: 1201	RFC 1014: 959	RFC 1190: 240
RFC 788: 1264	RFC 1021: 1070	RFC 1203: 1300
RFC 791: 239	RFC 1022: 1070	RFC 1213: 1080
RFC 792: 508	RFC 1023: 1070	RFC 1247: 627
RFC 793: 720	RFC 1024: 1070	RFC 1256: 509, 540
RFC 799: 843, 848	RFC 1028: 1070, 1100	RFC 1267: 649
RFC 810: 843	RFC 1032: 849	RFC 1271: 1134
RFC 812: 1477	RFC 1033: 849	RFC 1322: 658
RFC 813: 722	RFC 1034: 849	RFC 1323: 722
RFC 821: 1264	RFC 1035: 849	RFC 1332: 153
RFC 822. <i>See</i> RFC 822 email message format in main index	RFC 1036: 1408	RFC 1334: 153
RFC 823: 678	RFC 1048: 979	RFC 1341: 1244
RFC 826: 213	RFC 1052: 1070	RFC 1342: 1244
RFC 827: 684	RFC 1055: 141	RFC 1350: 1201
RFC 854: 1439	RFC 1058: 598	RFC 1351: 1080
RFC 855: 1450	RFC 1064: 1299	
	RFC 1072: 801	

RFC 1352: 1080	RFC 1701: 496	RFC 1978: 171
RFC 1353: 1080	RFC 1717: 175	RFC 1979: 171
RFC 1377: 153	RFC 1723: 600	RFC 1981: 417, 551
RFC 1378: 153	RFC 1730: 1300	RFC 1989: 154
RFC 1388: 600, 617	RFC 1731: 1300	RFC 1990: 154, 175
RFC 1393: 338, 509, 545	RFC 1733: 1287	RFC 1993: 171
RFC 1395: 980	RFC 1734: 1293	RFC 1994: 153
RFC 1425: 1265	RFC 1737: 1142, 1160	RFC 1995: 850, 907
RFC 1436: 1432	RFC 1738: 1142	RFC 1996: 850, 906
RFC 1441: 1080	RFC 1757: 1134	RFC 2001: 817, 818
RFC 1442: 1080	RFC 1771: 649	RFC 2002: 478
RFC 1443: 1080	RFC 1782: 1202	RFC 2003: 496
RFC 1444: 1080	RFC 1783: 1202	RFC 2004: 496
RFC 1445: 1081	RFC 1784: 1202	RFC 2012: 1097
RFC 1446: 1081	RFC 1794: 850	RFC 2014: 53
RFC 1447: 1081	RFC 1808: 1142	RFC 2018: 722, 801
RFC 1448: 1081	RFC 1812: 286, 509	RFC 2026: 59
RFC 1449: 1081	RFC 1813: 956	RFC 2043: 153
RFC 1450: 1081	RFC 1832: 959	RFC 2045: 1245
RFC 1451: 1081	RFC 1847: 1254	RFC 2046: 1245
RFC 1452: 1081	RFC 1869: 1265	RFC 2047: 1245
RFC 1459: 1458	RFC 1870: 1276	RFC 2048: 1245
RFC 1497: 980, 1043	RFC 1885: 509	RFC 2049: 1245
RFC 1513: 1136	RFC 1886: 850, 949	RFC 2060: 1300
RFC 1521: 1244	RFC 1891: 1276	RFC 2068: 1331
RFC 1522: 1244	RFC 1893: 1276	RFC 2077: 1249, 1251
RFC 1531: 999	RFC 1896: 1250	RFC 2080: 600, 620
RFC 1532: 980	RFC 1901: 1081	RFC 2097: 153
RFC 1533: 980	RFC 1902: 1081	RFC 2109: 1391
RFC 1534: 1063	RFC 1903: 1081	RFC 2118: 171
RFC 1541: 999	RFC 1904: 1081	RFC 2125: 154, 178
RFC 1542: 980	RFC 1905: 1081	RFC 2131: 1000
RFC 1546: 398	RFC 1906: 1081	RFC 2132: 1043
RFC 1552: 153	RFC 1907: 1081	RFC 2136: 850, 907, 908
RFC 1570: 153, 159	RFC 1908: 1081	RFC 2141: 1142, 1160
RFC 1579: 1180	RFC 1909: 1081	RFC 2178: 627
RFC 1583: 627	RFC 1910: 1081	RFC 2181: 850
RFC 1590: 1244	RFC 1918: 266	RFC 2183: 1244, 1248
RFC 1597: 266	RFC 1939: 1289	RFC 2192: 1142
RFC 1618: 154	RFC 1945: 1244, 1330	RFC 2224: 1142
RFC 1630: 1142	RFC 1952: 1374	RFC 2228: 1176
RFC 1631: 427	RFC 1962: 153	RFC 2290: 154
RFC 1635: 1177	RFC 1967: 171	RFC 2302: 1250
RFC 1651: 1265	RFC 1968: 153	RFC 2308: 850
RFC 1652: 1276	RFC 1969: 174	RFC 2318: 1250
RFC 1654: 649	RFC 1970: 577	RFC 2328: 627
RFC 1661: 153	RFC 1973: 154	RFC 2364: 154
RFC 1662: 153, 182	RFC 1974: 171	RFC 2368: 1142
RFC 1700: 707	RFC 1977: 171	RFC 2373: 367, 374

RFC 2374: 367, 374, 385	RFC 2980: 1400
RFC 2384: 1142	RFC 2988: 722, 803
RFC 2387: 1254	RFC 3003: 1251
RFC 2396: 1142	RFC 3010: 956
RFC 2401: 451, 454	RFC 3021: 322
RFC 2402: 451	RFC 3118: 1064
RFC 2403: 451	RFC 3160: 53
RFC 2404: 452	RFC 3189: 1251
RFC 2406: 452	RFC 3220: 478
RFC 2408: 452	RFC 3330: 1059
RFC 2409: 452, 471	RFC 3344: 478
RFC 2412: 452	RFC 3363: 950
RFC 2419: 174	RFC 3364: 950
RFC 2420: 174	RFC 3401 to RFC 3405: 1162
RFC 2453: 600, 617	RFC 3406: 1162
RFC 2460: 366, 374	RFC 3410: 1082
RFC 2461: 367, 509, 577	RFC 3411: 1082
RFC 2462: 399	RFC 3412: 1082
RFC 2463: 367, 509, 516	RFC 3413: 1082
RFC 2472: 153	RFC 3414: 1082
RFC 2474: 335	RFC 3415: 1082
RFC 2483: 1162	RFC 3416: 1082, 1101
RFC 2516: 154	RFC 3417: 1082, 1101
RFC 2554: 1276	RFC 3418: 1082
RFC 2557: 1244, 1248	RFC 3425: 921
RFC 2576: 1082	RFC 3501: 1300
RFC 2578: 1082	RFC 3513: 367, 374, 382
RFC 2579: 1082	RFC 3530: 957
RFC 2580: 1082	RFC 3587: 367, 374, 386
RFC 2581: 722	RFC 3927: 1059
RFC 2615: 154	
RFC 2616: 1331	
RFC 2617: 1331	
RFC 2671: 929	
RFC 2694: 437	
RFC 2717: 1142	
RFC 2718: 1142	
RFC 2774: 1332	
RFC 2810: 1458	
RFC 2819: 1134	
RFC 2821: 1235	
RFC 2822: 1235	
RFC 2850: 53	
RFC 2854: 1250	
RFC 2870: 899	
RFC 2874: 949	
RFC 2894: 400, 509, 568	
RFC 2920: 1276	
RFC 2965: 1391	

More No-Nonsense Books from  NO STARCH



SILENCE ON THE WIRE

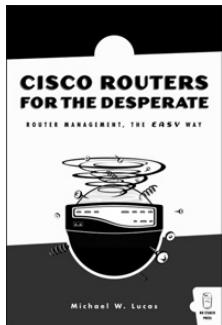
A Field Guide to Passive Reconnaissance and Indirect Attacks

by MICHAL ZALEWSKI

Author Michal Zalewski has long been known and respected in the hacking and security communities for his intelligence, curiosity, and creativity, and this book is truly unlike anything else out there. *Silence on the Wire* is no humdrum technical white paper or how-to manual for protecting one's network. Rather, Zalewski's book is a fascinating narrative that explores a variety of unique, uncommon, and often quite elegant security challenges that defy classification and eschew the traditional attacker-victim model.

FEBRUARY 2005, 312 PP., \$39.95 (\$53.95 CAN)

ISBN 1-59327-046-1



CISCO ROUTERS FOR THE DESPERATE™

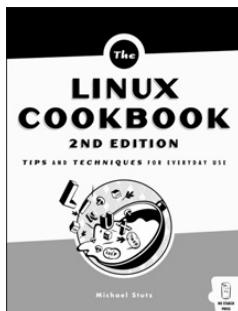
Router Management, The Easy Way

by MICHAEL W. LUCAS

A brief, meaty introduction to Cisco routers that will make competent system administrators comfortable with the Cisco environment, teach them how to troubleshoot problems, and take them through the basic tasks of router maintenance and integration into an existing network. Designed to be read once and then left on top of the router until something breaks, this is a book for those who don't need to know a huge amount about routers but must still provide reliable network services.

NOVEMBER 2004, 144 PP., \$19.95 (\$27.95 CAN)

ISBN 1-59327-049-6



THE LINUX COOKBOOK, 2ND EDITION

Tips and Techniques for Everyday Use

by MICHAEL STUTZ

The second edition of our acclaimed *Linux Cookbook* contains step-by-step recipes that show you how to do just about anything with Linux. Organized by general task (such as managing files and manipulating graphics), this edition includes hundreds of new recipes as well as new sections on package management, file conversion, multimedia, working with sound files (including OGG and MP3), Vi text editing, and advanced text manipulation. A perfect introduction to the Linux command line or complete desktop reference for any of the major Linux distributions.

AUGUST 2004, 824 PP., \$39.95 (\$55.95 CAN)

ISBN 1-59327-031-3

THE BOOK OF™ POSTFIX

State-of-the-Art Message Transport

by RALF HILDEBRANDT and PATRICK KOETTER

Developed with security and speed in mind, Postfix has become a popular alternative to Sendmail. *The Book of Postfix* is a complete guide to Postfix whether used by the home user, as a mailrelay or virus scanning gateway, or as a company mailserver. Practical examples show how to deal with daily challenges like protecting mail users from spam and viruses, managing multiple domains, and offering roaming access.

MARCH 2005, 496 PP., \$44.95 (\$62.95 CAN)

ISBN 1-59327-001-1



ENDING SPAM

Bayesian Content Filtering and the Art of Statistical Language Classification

by JONATHAN A. ZDZIARSKI

Through considerable research, creative minds have invented clever new ways to fight spam in all its nefarious forms. This landmark title describes, in depth, how statistical filtering is being used by next generation spam filters to identify and filter spam. Zdziarski explains how spam filtering works and how language classification and machine learning combine to produce remarkably accurate spam filters. Readers gain a complete understanding of the mathematical approaches used in today's spam filters, decoding, tokenization, the use of various algorithms (including Bayesian analysis and Markovian discrimination), and the benefits of using open-source solutions to end spam. Interviews with the creators of many of the best spam filters provide further insight into the anti-spam crusade.

JULY 2005, 312 PP., \$39.95 (\$53.95 CAN)

ISBN 1-59327-052-6



PHONE:

800.420.7240 OR

415.863.9900

MONDAY THROUGH FRIDAY,

9 A.M. TO 5 P.M. (PST)

EMAIL:

SALES@NOSTARCH.COM

WEB:

HTTP://WWW.NOSTARCH.COM

FAX:

415.863.9950

24 HOURS A DAY,

7 DAYS A WEEK

MAIL:

NO STARCH PRESS

555 DE HARO ST, SUITE 250

SAN FRANCISCO, CA 94107

USA

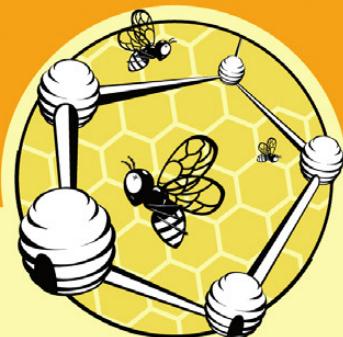
UPDATES

Visit <http://www.nostarch.com/tcpip.htm> for updates, errata, and other information.

COLOPHON

The TCP/IP Guide is set in New Baskerville, TheSansMono Condensed, Futura, and Dogma. The book was printed and bound at Malloy Incorporated in Ann Arbor, Michigan. The paper is 45# Utopia Filmcote.

COVERS IPv4 AND IPv6



**AN UP-TO-DATE,
ACCESSIBLE, AND
WELL-ILLUSTRATED
TCP/IP REFERENCE**

"It's informative and easy to read, even when discussing rather nasty protocols, and when it covers something, it generally covers it quite completely."—login:

"Keeps things interesting and flowing well enough that working one's way through...is actually entertaining instead of torture."—Slashdot

"The most comprehensive guide to TCP/IP protocols we have ever come across. It also is the most readable. This is a book that will be staying on our shelves, and we highly recommend it."—Network World

From Charles M. Kozierok, the creator of the highly regarded www.PCGuide.com, comes *The TCP/IP Guide*. This completely up-to-date, encyclopedic reference on the TCP/IP protocol suite will appeal to newcomers and the seasoned professional alike. Kozierok details the core protocols that make TCP/IP internetworks function and the most important classic TCP/IP applications, integrating IPv6 coverage throughout. Over 350 illustrations and hundreds of tables help to explain the finer points of this complex topic. The book's personal, user-friendly writing style lets readers of all levels understand the dozens of protocols and technologies

that run the Internet, with full coverage of PPP, ARP, IP, IPv6, IP NAT, IPSec, Mobile IP, ICMP, RIP, BGP, TCP, UDP, DNS, DHCP, SNMP, FTP, SMTP, NNTP, HTTP, Telnet, and much more.

The *TCP/IP Guide* is a must-have addition to the libraries of internetworking students, educators, networking professionals, and those working toward certification.

ABOUT THE AUTHOR

Charles M. Kozierok is the author and publisher of *The PC Guide* (www.PCGuide.com), an extensive online reference work on personal computers, as well as several other educational websites, including *The TCP/IP Guide* (www.TCPIPGuide.com). He holds master's degrees from MIT in management and in electrical engineering and computer science (EECS), and worked in various technical and managerial roles before dedicating himself full-time to writing and educational pursuits. He lives in rural Vermont with his wife and three sons.



THE FINEST IN GEEK ENTERTAINMENT™
www.nostarch.com

\$99.95 (\$124.95 CAN)

ISBN: 978-1-59327-047-6



9 781593 270476



SHelfe IN:
NETwORkING