6

THE TRANSPORT LAYER

Together with the network layer, the transport layer is the heart of the protocol hierarchy. The network layer provides end-to-end packet delivery using datagrams or virtual circuits. The transport layer builds on the network layer to provide data transport from a process on a source machine to a process on a destination machine with a desired level of reliability that is independent of the physical networks currently in use. It provides the abstractions that applications need to use the network. Without the transport layer, the whole concept of layered protocols would make little sense. In this chapter, we will study the transport layer in detail, including its services and choice of API design to tackle issues of reliability, connections and congestion control, protocols such as TCP and UDP, and performance.

6.1 THE TRANSPORT SERVICE

In the following sections, we will provide an introduction to the transport service. We will look at what kind of service is provided to the application layer. To make the issue of transport service more concrete, we will examine two sets of transport layer primitives. First comes a simple (but hypothetical) one to show the basic ideas. Then comes the interface commonly used in the Internet.

6.1.1 Services Provided to the Upper Layers

The ultimate goal of the transport layer is to provide efficient, reliable, and cost-effective data transmission service to its users, normally processes in the application layer. To achieve this, the transport layer makes use of the services provided by the network layer. The software and/or hardware within the transport layer that does the work is called the **transport entity**. The transport entity can be located in the operating system kernel, in a library package bound into network applications, in a separate user process, or even on the network interface card. The first two options are most common on the Internet. The (logical) relationship of the network, transport, and application layers is illustrated in Fig. 6-1.

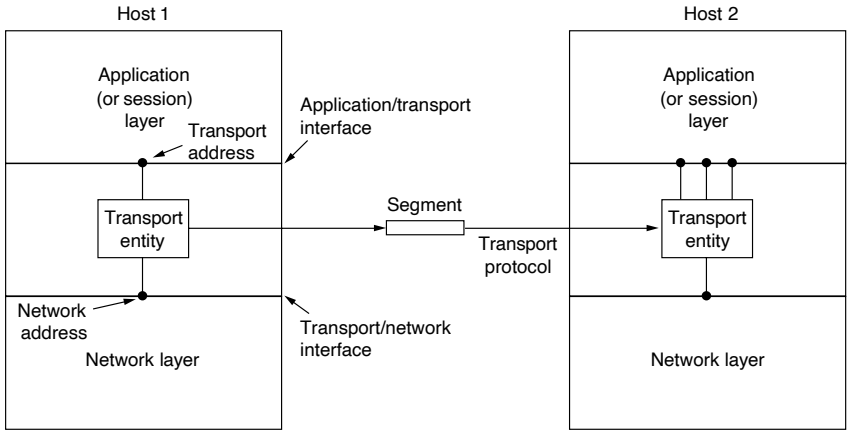


Figure 6-1. The network, transport, and application layers.

Just as there are two types of network service, connection-oriented and connectionless, there are also two types of transport service. The connection-oriented transport service is similar to the connection-oriented network service in many ways. In both cases, connections have three phases: establishment, data transfer, and release.

Addressing and flow control are also similar in both layers. Furthermore, the connectionless transport service is also very similar to the connectionless network service. However, note that it can be difficult to provide a connectionless transport service on top of a connection-oriented network service, since it is inefficient to set up a connection to send a single packet and then tear it down immediately afterwards.

The obvious question is this: if the transport layer service is so similar to the network layer service, why are there two distinct layers? Why is one layer not adequate? The answer is subtle, but really crucial. The transport code runs entirely on

the users' machines, but the network layer largely runs on the routers, which are operated by the carrier (at least for a wide area network). What happens if the network layer offers inadequate service? What if it frequently loses packets? What happens if routers crash from time to time?

Problems occur, that's what. The users have no real control over the network layer, so they cannot solve the problem of poor service by using better routers or putting more error handling in the data link layer because they don't own the routers. The only possibility is to put on top of the network layer another layer that improves the quality of the service. If, in a connectionless network, packets are lost or mangled, the transport entity can detect the problem and compensate for it by using retransmissions. If, in a connection-oriented network, a transport entity is informed halfway through a long transmission that its network connection has been abruptly terminated, with no indication of what has happened to the data currently in transit, it can set up a new network connection to the remote transport entity. Using this new network connection, it can send a query to its peer asking which data arrived and which did not, and knowing where it was, pick up from where it left off.

In essence, the existence of the transport layer makes it possible for the transport service to be more reliable than the underlying network, which may not be all that reliable. Furthermore, the transport primitives can be implemented as calls to library procedures to make them independent of the network primitives. The network service calls may vary considerably from one network to another (e.g., calls based on a connectionless Ethernet may be quite different from calls on a connection-oriented network). Hiding the network service behind a set of transport service primitives ensures that changing the network merely requires replacing one set of library procedures with another one that does the same thing with a different underlying service. Having applications be independent of the network layer is a good thing.

Thanks to the transport layer, application programmers can write code according to a standard set of primitives and have these programs work on a wide variety of networks, without having to worry about dealing with different network interfaces and levels of reliability. If all real networks were flawless and all had the same service primitives and were guaranteed never, ever to change, the transport layer might not be needed. However, in the real world it fulfills the key function of isolating the upper layers from the technology, design, and imperfections of the network.

For this reason, many people have made a qualitative distinction between layers one through four on the one hand and layer(s) above four on the other. The bottom four layers can be seen as the **transport service provider**, whereas the upper layer(s) are the **transport service user**. This distinction of provider versus user has a considerable impact on the design of the layers and puts the transport layer in a key position, since it forms the major boundary between the provider and user of the reliable data transmission service. It is the level that applications see.

6.1.2 Transport Service Primitives

To allow users to access the transport service, the transport layer must provide some operations to application programs, that is, a transport service interface. Each transport service has its own interface. In this section, we will first examine a simple (hypothetical) transport service and its interface to see the bare essentials. In the following section, we will look at a real example.

The transport service is similar to the network service, but there are also some important differences. The main difference is that the network service is intended to model the service offered by real networks, warts and all. Real networks can lose packets, so the network service is generally unreliable.

The connection-oriented transport service, in contrast, is reliable. Of course, real networks are not error-free, but that is precisely the purpose of the transport layer—to provide a reliable service on top of an unreliable network.

As an example, consider two processes on a single machine connected by a pipe in UNIX (or any other interprocess communication facility). They assume the connection between them is 100% perfect. They do not want to know about acknowledgements, lost packets, congestion, or anything at all like that. What they want is a 100% reliable connection. Process *A* puts data into one end of the pipe, and process *B* takes it out of the other. This is what the connection-oriented transport service is all about—hiding the imperfections of the network service so that user processes can just assume the existence of an error-free bit stream even when they are on different machines.

As an aside, the transport layer can also provide unreliable (datagram) service. However, there is relatively little to say about that besides ''it's datagrams,'' so we will mainly concentrate on the connection-oriented transport service in this chapter. Nevertheless, there are some applications, such as client-server computing and streaming multimedia, that build on a connectionless transport service, and we will say a little bit about that later on.

A second difference between the network service and transport service is whom the services are intended for. From the perspective of network endpoints, the network service is used only by the transport entities. Few users write their own transport entities, and thus few users or programs ever see the bare network service. In contrast, many programs (and thus programmers) see the transport primitives. Consequently, the transport service must be convenient and easy to use.

To get an idea of what a transport service might be like, consider the five primitives listed in Fig. 6-2. This transport interface is truly bare bones, but it gives the essential flavor of what a connection-oriented transport interface has to do. It allows application programs to establish, use, and then release connections, which is sufficient for many applications.

To see how these primitives might be used, consider an application with a server and a number of remote clients. To start with, the server executes a LISTEN primitive, typically by calling a library procedure that makes a system call that

Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	Request a release of the connection

Figure 6-2. The primitives for a simple transport service.

blocks the server until a client turns up. When a client wants to talk to the server, it executes a CONNECT primitive. The transport entity carries out this primitive by blocking the caller and sending a packet to the server. Encapsulated in the payload of this packet is a transport layer message for the server's transport entity.

A quick note on terminology is now in order. For lack of a better term, we will use the term **segment** for messages sent from transport entity to transport entity. TCP, UDP and other Internet protocols use this term. Some older protocols used the ungainly name **TPDU** (**Transport Protocol Data Unit**). That term is not used much any more now but you may see it in older papers and books.

Thus, segments (exchanged by the transport layer) are contained in packets (which are exchanged by the network layer). In turn, these packets are contained in frames (exchanged by the data link layer). When a frame arrives, the data link layer processes the frame header and, if the destination address matches for local delivery, passes the contents of the frame payload field up to the network entity. The network entity similarly processes the packet header and then passes the contents of the packet payload up to the transport entity. This nesting is illustrated in Fig. 6-3.

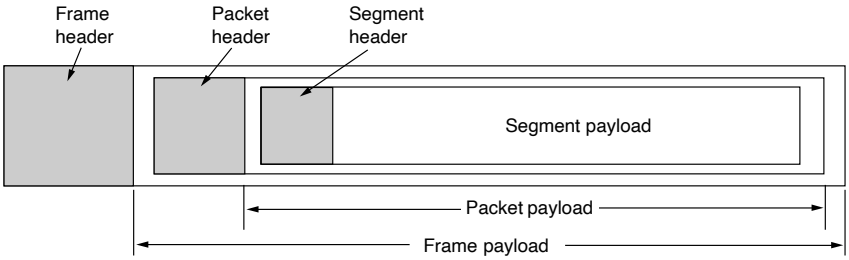


Figure 6-3. Nesting of segments, packets, and frames.

Getting back to our client-server example, the client's CONNECT call causes a CONNECTION REQUEST segment to be sent to the server. When it arrives, the transport entity checks to see that the server is blocked on a LISTEN (i.e., is ready to

handle requests). If so, it then unblocks the server and sends a CONNECTION ACCEPTED segment back to the client. When this segment arrives, the client is unblocked and the connection is established.

Data can now be exchanged using the SEND and RECEIVE primitives. In the simplest form, either party can do a (blocking) RECEIVE to wait for the other party to do a SEND. When the segment arrives, the receiver is unblocked. It can then process the segment and send a reply. As long as both sides can keep track of whose turn it is to send, this scheme works fine.

In the transport layer, even a simple unidirectional data exchange is more complicated than at the network layer. Every data packet sent will also be acknowledged (eventually). The packets bearing control segments are also acknowledged, implicitly or explicitly. These acknowledgements are managed by the transport entities, using the network layer protocol, and are not visible to the transport users. Similarly, the transport entities need to worry about timers and retransmissions. None of this machinery is visible to the transport users. To the transport users, a connection is a reliable bit pipe: one end stuffs bits in and they magically appear in the same order at the other end. This ability to hide complexity is the reason that layered protocols are such a powerful tool.

When a connection is no longer needed, it must be released to free up table space within the two transport entities. Disconnection has two variants: asymmetric and symmetric. In the asymmetric variant, either transport user can issue a DISCONNECT primitive, which results in a DISCONNECT segment being sent to the remote transport entity. Upon its arrival, the connection is released.

In the symmetric variant, each direction is closed separately, independently of the other one. When one side does a DISCONNECT, that means it has no more data to send but it is still willing to accept data from its partner. In this model, a connection is released when both sides have done a DISCONNECT.

A state diagram for connection establishment and release for these simple primitives is given in Fig. 6-4. Each transition is triggered by some event, either a primitive executed by the local transport user or an incoming packet. For simplicity, we assume here that each segment is separately acknowledged. We also assume that a symmetric disconnection model is used, with the client going first. Please note that this model is quite unsophisticated. We will look at more realistic models later on when we describe how TCP works.

6.1.3 Berkeley Sockets

Let us now briefly inspect another set of transport primitives, the socket primitives as they are used for TCP. Sockets were first released as part of the Berkeley UNIX 4.2BSD software distribution in 1983. They quickly became popular. The primitives are now widely used for Internet programming on many operating systems, especially UNIX-based systems, and there is a socket-style API for Windows called ''winsock.''

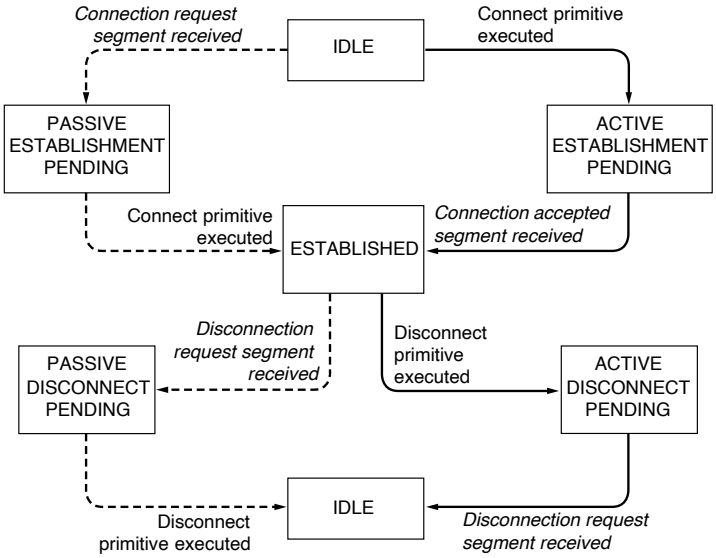


Figure 6-4. A state diagram for a simple connection management scheme. Transitions labeled in italics are caused by packet arrivals. The solid lines show the client's state sequence. The dashed lines show the server's state sequence.

The primitives are listed in Fig. 6-5. Roughly speaking, they follow the model of our first example but offer more features and flexibility. We will not look at the corresponding segments here. That discussion will come later.

Primitive	Meaning
SOCKET	Create a new communication endpoint
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Figure 6-5. The socket primitives for TCP.

The first four primitives in the list are executed in that order by servers. The SOCKET primitive creates a new endpoint and allocates table space for it within the transport entity. The parameters of the call specify the addressing format to be

used, the type of service desired (e.g., reliable byte stream), and the protocol. A successful SOCKET call returns an ordinary file descriptor for use in succeeding calls, the same way an OPEN call on a file does.

Newly created sockets do not have network addresses. These are assigned using the BIND primitive. Once a server has bound an address to a socket, remote clients can connect to it. The reason for not having the SOCKET call create an address directly is that some processes care about their addresses (e.g., they have been using the same address for years and everyone knows this address)..

Next comes the LISTEN call, which allocates space to queue incoming calls for the case that several clients try to connect at the same time. In contrast to LISTEN in our first example, in the socket model LISTEN is not a blocking call.

To block waiting for an incoming connection, the server executes an ACCEPT primitive. When a segment asking for a connection arrives, the transport entity creates a new socket with the same properties as the original one and returns a file descriptor for it. The server can then fork off a process or thread to handle the connection on the new socket and go back to waiting for the next connection on the original socket. ACCEPT returns a file descriptor, which can be used for reading and writing in the standard way, the same as for files.

Now let us look at the client side. Here, too, a socket must first be created using the SOCKET primitive, but BIND is not required since the address used does not matter to the server. The CONNECT primitive blocks the caller and starts the connection process. When it completes (i.e., when the appropriate segment is received from the server), the client process is unblocked and the connection is established. Both sides can now use SEND and RECEIVE to transmit and receive data over the full-duplex connection. The standard UNIX READ and WRITE system calls can also be used if none of the special options of SEND and RECEIVE are required.

Connection release with sockets is symmetric. When both sides have executed a CLOSE primitive, the connection is released.

Sockets have proved tremendously popular and are the de facto standard for abstracting transport services to applications. The socket API is often used with the TCP protocol to provide a connection-oriented service called a **reliable byte stream**, which is simply the reliable bit pipe that we described. However, other protocols could be used to implement this service using the same API. It should all be the same to the transport service users.

A strength of the socket API is that is can be used by an application for other transport services. For instance, sockets can be used with a connectionless transport service. In this case, CONNECT sets the address of the remote transport peer and SEND and RECEIVE send and receive datagrams to and from the remote peer. (It is also common to use an expanded set of calls, for example, SENDTO and RECVFROM, that emphasize messages and do not limit an application to a single transport peer.) Sockets can also be used with transport protocols that provide a message stream rather than a byte stream and that do or do not have congestion control. For example, **DCCP** (**Datagram Congestion Control Protocol**) is a

version of UDP with congestion control (Kohler et al., 2006). It is up to the transport users to understand what service they are getting.

However, sockets are not likely to be the final word on transport interfaces. For example, applications often work with a group of related streams, such as a Web browser that requests several objects from the same server. With sockets, the most natural fit is for application programs to use one stream per object. This structure means that congestion control is applied separately for each stream, not across the group, which is suboptimal. It punts to the application the burden of managing the set. Some protocols and interfaces have been devised that support groups of related streams more effectively and simply for the application. Two examples are **SCTP** (**Stream Control Transmission Protocol**) defined in RFC 4960 (Ford, 2007) and QUIC (discussed later). These protocols must change the socket API slightly to get the benefits of groups of related streams, and they also support features such as a mix of connection-oriented and connectionless traffic and even multiple network paths.

6.1.4 An Example of Socket Programming: An Internet File Server

As an example of the nitty-gritty of how real socket calls are made, consider the client and server code of Fig. 6-6. Here we have a very primitive Internet file server along with an example client that uses it. The code has many limitations (discussed below), but in principle the server code can be compiled and run on any UNIX system connected to the Internet. The client code can be compiled and run on any other UNIX machine on the Internet, anywhere in the world. The client code can be executed with appropriate parameters to fetch any file to which the server has access on its machine. The file is written to standard output, which, of course, can be redirected to a file or pipe.

Let us look at the server code first. It starts out by including some standard headers, the last three of which contain the main Internet-related definitions and data structures. Next comes a definition of *SERVER_PORT* as 8080. This number was chosen arbitrarily. Any number between 1024 and 65535 will work just as well, as long as it is not in use by some other process; ports below 1023 are reserved for privileged users.

The next two lines in the server define constants. The first one determines the chunk size in bytes used for the file transfer. The second one determines how many pending connections can be held before additional ones are discarded.

After the declarations of local variables, the server code begins. It starts out by initializing a data structure that will hold the server's IP address. This data structure will soon be bound to the server's socket. The call to *memset* sets the data structure to all 0s. The three assignments following it fill in three of its fields. The last of these contains the server's port. The functions *htonl* and *htons* have to do with converting values to a standard format so the code runs correctly on both little-endian machines (e.g., Intel x86) and big-endian machines (e.g., the SPARC).

```
/* This page contains a client program that can request a file from the server program
 * on the next page. The server responds by sending the whole file.
 */

#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 8080                    /* arbitrary, but client & server must agree */
#define BUF_SIZE 4096                       /* block transfer size */

int main(int argc, char **argv)
{
  int c, s, bytes;
  char buf[BUF_SIZE];                       /* buffer for incoming file */
  struct hostent *h;                        /* info about server */
  struct sockaddr_in channel;               /* holds IP address */

  if (argc != 3) {printf("Usage: client server-name file-name0); exit(-1);}
  h = gethostbyname(argv[1]);               /* look up host's IP address */
  if (!h) {printf("gethostbyname failed to locate %s0, argv[1]); exit(-1;}

  s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (s <0) {printf("socket call failed0); exit(-1);}
  memset(&channel, 0, sizeof(channel));
  channel.sin_family= AF_INET;
  memcpy(&channel.sin_addr.s_addr, h->h_addr, h->h_length);
  channel.sin_port= htons(SERVER_PORT);
  c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
  if (c < 0) {printf("connect failed0); exit(-1);}

  /* Connection is now established. Send file name including 0 byte at end. */
  write(s, argv[2], strlen(argv[2])+1);

  /* Go get the file and write it to standard output. */
  while (1) {
      bytes = read(s, buf, BUF_SIZE);       /* read from socket */
      if (bytes <= 0) exit(0);              /* check for end of file */
      write(1, buf, bytes);                 /* write to standard output */
  }
}
```

Figure 6-6. Client code using sockets. The server code is on the next page.

```
#include <sys/types.h>                  /* This is the server code */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 8080                /* arbitrary, but client & server must agree */
#define BUF_SIZE 4096                   /* block transfer size */
#define QUEUE_SIZE 10

int main(int argc, char *argv[])
{ int s, b, l, fd, sa, bytes, on = 1;
  char buf[BUF_SIZE];                   /* buffer for outgoing file */
  struct sockaddr_in channel;           /* holds IP address */

  /* Build address structure to bind to socket. */
  memset(&channel, 0, sizeof(channel)); /* zero channel */
  channel.sin_family = AF_INET;
  channel.sin_addr.s_addr = htonl(INADDR_ANY);
  channel.sin_port = htons(SERVER_PORT);

  /* Passive open. Wait for connection. */
  s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);   /* create socket */
  if (s < 0) {printf("socket call failed0); exit(-1);}
  setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));

  b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
  if (b < 0) {printf("bind failed0); exit(-1);}

  l = listen(s, QUEUE_SIZE);            /* specify queue size */
  if (l < 0) {printf("listen failed0); exit(-1);}

  /* Socket is now set up and bound. Wait for connection and process it. */
  while (1) {
      sa = accept(s, 0, 0);             /* block for connection request */
      if (sa < 0) {printf("accept failed0); exit(-1);}

      read(sa, buf, BUF_SIZE);          /* read file name from socket */

      /* Get and return the file. */
      fd = open(buf, O_RDONLY);         /* open the file to be sent back */
      if (fd < 0) {printf("open failed");

      while (1) {
            bytes = read(fd, buf, BUF_SIZE); /* read from file */
            if (bytes <= 0) break;      /* check for end of file */
            write(sa, buf, bytes);      /* write bytes to socket */
      }
      close(fd);                        /* close file */
      close(sa);                        /* close connection */
  }
}
```

Next, the server creates a socket and checks for errors (indicated by $s < 0$). In a production version of the code, the error message could be a trifle more explanatory. The call to *setsockopt* is needed to allow the port to be reused so the server can run indefinitely, fielding request after request. Now the IP address is bound to the socket and a check is made to see if the call to *bind* succeeded. The final step in the initialization is the call to *listen* to announce the server's willingness to accept incoming calls and tell the system to hold up to *QUEUE_SIZE* of them in case new requests arrive while the server is still processing the current one. If the queue is full and additional requests arrive, they are quietly discarded.

At this point, the server enters its main loop, which it never leaves. The only way to stop it is to kill it from outside. The call to *accept* blocks the server until some client tries to establish a connection with it. If the *accept* call succeeds, it returns a socket descriptor that can be used for reading and writing, analogous to how file descriptors can be used to read from and write to pipes. However, unlike pipes, which are unidirectional, sockets are bidirectional, so *sa* (the accepted socket) can be used for reading from the connection and also for writing to it. A pipe file descriptor is for reading or writing but not both.

After the connection is established, the server reads the file name from it. If the name is not yet available, the server blocks waiting for it. After getting the file name, the server opens the file and enters a loop that alternately reads blocks from the file and writes them to the socket until the entire file has been copied. Then the server closes the file and the connection and waits for the next connection to show up. It repeats this loop forever.

Now let us look at the client code. To understand how it works, it is necessary to understand how it is invoked. Assuming it is called *client*, a typical call is

```
client flits.cs.vu.nl /usr/tom/filename >f
```

This call only works if the server is already running on *flits.cs.vu.nl* and the file */usr/tom/filename* exists and the server has read access to it. If the call is successful, the file is transferred over the Internet and written to *f*, after which the client program exits. Since the server continues after a transfer, the client can be started again and again to get other files.

The client code starts with some includes and declarations. Execution begins by checking to see if it has been called with the right number of arguments, where *argc* = 3 means the program was called with its name plus two arguments. Note that *argv*[1] contains the name of the server (e.g., *flits.cs.vu.nl*) and is converted to an IP address by *gethostbyname*. This function uses DNS to look up the name. We will study DNS in Chap. 7.

Next, a socket is created and initialized. After that, the client attempts to establish a TCP connection to the server, using *connect*. If the server is up and running on the named machine and attached to *SERVER_PORT* and is either idle or has room in its *listen* queue, the connection will (eventually) be established. Using the connection, the client sends the name of the file by writing on the socket.

The number of bytes sent is one larger than the name proper, since the 0 byte terminating the name must also be sent to tell the server where the name ends.

Now the client enters a loop, reading the file block by block from the socket and copying it to standard output. When it is done, it just exits.

The procedure *fatal* prints an error message and exits. The server needs the same procedure, but it was omitted due to lack of space on the page. Since the client and server are compiled separately and normally run on different computers, they cannot share the code of *fatal*.

Just for the record, this server is not the last word in serverdom. Its error checking is meager and its error reporting is mediocre. Since it handles all requests strictly sequentially (because it has only a single thread), its performance is poor. It has clearly never heard about security, and using bare UNIX system calls is not the way to gain platform independence. It also makes some assumptions that are technically illegal, such as assuming that the file name fits in the buffer and is transmitted atomically. These shortcomings notwithstanding, it is a working Internet file server. For more information about using sockets, see Donahoo and Calvert (2008, 2009); and Stevens et al. (2004).

6.2 ELEMENTS OF TRANSPORT PROTOCOLS

The transport service is implemented by a **transport protocol** used between the two transport entities. In some ways, transport protocols resemble the data link protocols we studied in detail in Chap. 3. Both have to deal with error control, sequencing, and flow control, among other issues.

However, significant differences between the two also exist. These differences are due to major dissimilarities between the environments in which the two protocols operate, as shown in Fig. 6-7. At the data link layer, two routers communicate directly via a physical channel, whether wired or wireless, whereas at the transport layer, this physical channel is replaced by the entire network. This difference has many important implications for the protocols.

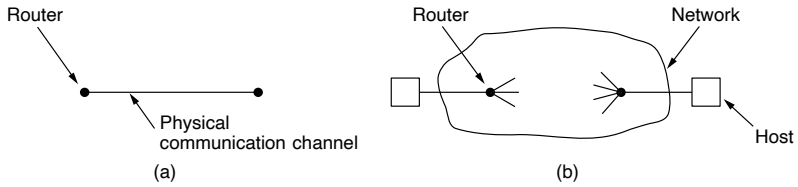


Figure 6-7. (a) Environment of the data link layer. (b) Environment of the transport layer.

For one thing, over point-to-point links such as wires or optical fiber, it is usually not necessary for a router to specify which router it wants to talk to—each

outgoing line leads directly to a particular router. In the transport layer, explicit addressing of destinations is required.

For another thing, the process of establishing a connection over the wire of Fig. 6-7(a) is simple: the other end is always there (unless it has crashed, in which case it is not there). Either way, there is not much to do. Even on wireless links, the process is not much different. Just sending a message is sufficient to have it reach all other destinations. If the message is not acknowledged due to an error, it can be resent. In the transport layer, initial connection establishment is complicated, as we will see.

Another (exceedingly annoying) difference between the data link layer and the transport layer is the potential existence of storage capacity in the network. When a router sends a packet over a link, it may arrive or be lost, but it cannot bounce around for a while, go into hiding in a far corner of the world, and suddenly emerge after other packets that were sent much later. If the network uses datagrams, which are independently routed inside, there is a nonnegligible probability that a packet may take the scenic route and arrive late and out of the expected order, or even that duplicates of the packet will arrive. The consequences of the network's ability to delay and duplicate packets can sometimes be disastrous and can require the use of special protocols to correctly transport information.

A final difference between the data link and transport layers is one of degree rather than of kind. Buffering and flow control are needed in both layers, but the presence in the transport layer of a large and varying number of connections with bandwidth that fluctuates as the connections compete with each other may require a different approach than we used in the data link layer. Some of the protocols discussed in Chap. 3 allocate a fixed number of buffers to each line, so that when a frame arrives a buffer is always available. In the transport layer, the larger number of connections that must be managed and variations in the bandwidth each connection may receive make the idea of dedicating many buffers to each one less attractive. In the following sections, we will examine all of these important issues, and others.

6.2.1 Addressing

When an application process wishes to set up a connection to a remote application process, it must specify which process on the remote endpoint to connect to. The method normally used is to define transport addresses to which processes can listen for connection requests. In the Internet, these endpoints are called **ports**. We will use the generic term **TSAP** (**Transport Service Access Point**) to mean a specific endpoint in the transport layer. The analogous endpoints in the network layer (i.e., network layer addresses) are not-surprisingly called **NSAPs** (**Network Service Access Points**). IP addresses are examples of NSAPs.

Figure 6-8 illustrates the relationship between the NSAPs, the TSAPs, and a transport connection using them. Application processes, both clients and servers,

can attach themselves to a local TSAP to establish a connection to a remote TSAP. These connections run through NSAPs on each host, as shown. The purpose of having TSAPs is that in some networks, each computer has a single NSAP, so some way is needed to distinguish multiple transport endpoints that share that NSAP.

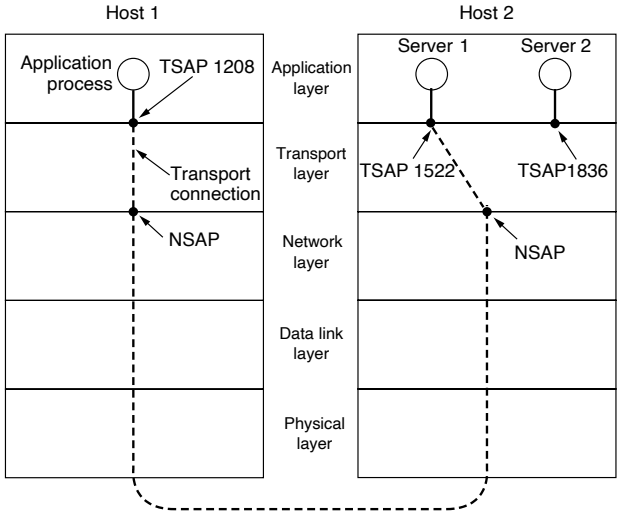


Figure 6-8. TSAPs, NSAPs, and transport connections.

A possible scenario for a transport connection is as follows:

1. A mail server process attaches itself to TSAP 1522 on host 2 to wait for an incoming call. How a process attaches itself to a TSAP is outside the networking model and depends entirely on the local operating system. A call such as our LISTEN might be used, for example.
2. An application process on host 1 wants to send an email message, so it attaches itself to TSAP 1208 and issues a CONNECT request. The request specifies TSAP 1208 on host 1 as the source and TSAP 1522 on host 2 as the destination. This action ultimately results in a transport connection being established between the application process and the server.
3. The application process sends over the mail message.
4. The mail server responds to say that it will deliver the message.
5. The transport connection is released.

Note that there may well be other servers on host 2 that are attached to other TSAPs and are waiting for incoming connections that arrive over the same NSAP.

The picture painted above is fine, except we have swept one little problem under the rug: how does the user process on host 1 know that the mail server is attached to TSAP 1522? One possibility is that the mail server has been attaching itself to TSAP 1522 for years and gradually all the network users have learned this. In this model, services have stable TSAP addresses that are listed in files in well-known places. For example, the */etc/services* file on UNIX systems lists which servers are permanently attached to which ports, including the fact that the mail server is found on TCP port 25.

While stable TSAP addresses work for a small number of key services that never change (e.g., the Web server), user processes, in general, often want to talk to other user processes that do not have TSAP addresses that are known in advance, or that may exist for only a short time.

To handle this situation, an alternative scheme can be used. In this scheme, there exists a special process called a **portmapper**. To find the TSAP address corresponding to a given service name, such as ''BitTorrent,'' a user sets up a connection to the portmapper (which listens to a well-known TSAP). The user then sends a message specifying the service name, and the portmapper sends back the TSAP address. Then the user releases the connection with the portmapper and establishes a new one with the desired service.

In this model, when a new service is created, it must register itself with the portmapper, giving both its service name (typically, an ASCII string) and its TSAP. The portmapper records this information in its internal database so that when queries come in later, it will know the answers.

The function of the portmapper is analogous to that of a directory assistance operator in the telephone system—it provides a mapping of names onto numbers. Just as in the telephone system, it is essential that the address of the well-known TSAP used by the portmapper is indeed well known. If you do not know the number of the information operator, you cannot call the information operator to find it out. If you think the number you dial for information is obvious, try it in a foreign country sometime.

Many of the server processes that can exist on a machine will be used only rarely. It is wasteful to have each of them active and listening to a stable TSAP address all day long. An alternative scheme is shown in Fig. 6-9 in a simplified form. It is known as the **initial connection protocol**. Instead of every conceivable server listening at a well-known TSAP, each machine that wishes to offer services to remote users has a special **process server** that acts as a proxy for less heavily used servers. This server is called *inetd* on UNIX systems. It listens to a set of ports at the same time, waiting for a connection request. Potential users of a service begin by doing a CONNECT request, specifying the TSAP address of the service they want. If no server is waiting for them, they get a connection to the process server, as shown in Fig. 6-9(a).

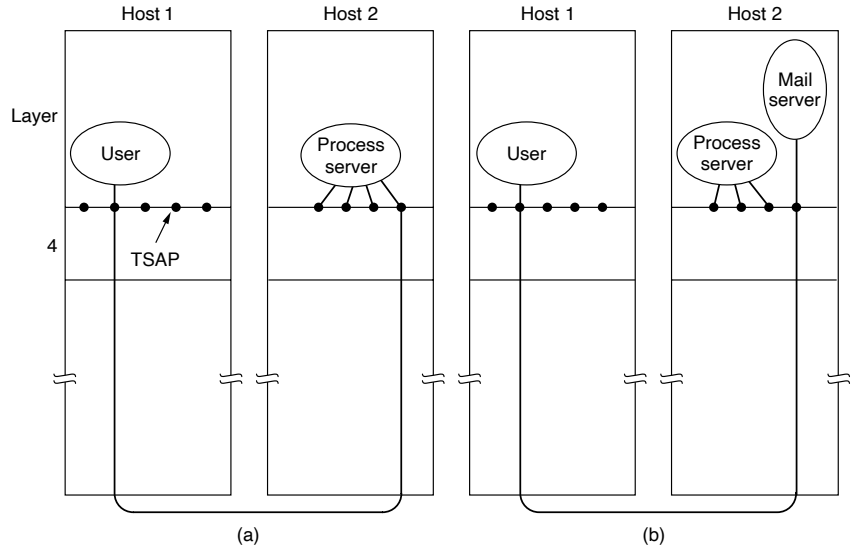


Figure 6-9. How a user process in host 1 establishes a connection with a mail server in host 2 via a process server.

After it gets the incoming request, the process server spawns the requested server, allowing it to inherit the existing connection with the user. The new server does the requested work, while the process server goes back to listening for new requests, as shown in Fig. 6-9(b). This method is only applicable when servers can be created on demand.

6.2.2 Connection Establishment

Establishing a connection sounds easy, but it is actually surprisingly tricky. At first glance, it would seem sufficient for one transport entity to just send a CONNECTION REQUEST segment to the destination and wait for a CONNECTION ACCEPTED reply. The problem occurs when the network can lose, delay, corrupt, and duplicate packets. This behavior causes serious complications.

Problem: Delayed and Duplicate Packets

Imagine a network that is so congested that acknowledgements hardly ever get back in time and each packet times out and is retransmitted two or three or more times. Suppose that the network uses datagrams inside and that every packet follows a different route. Some of the packets might get stuck in a traffic jam inside

the network and take a long time to arrive. That is, they may be delayed in the network and pop out much later, when the sender thought that they had been lost.

The worst possible nightmare is as follows. A user establishes a connection with a bank, sends messages telling the bank to transfer a large amount of money to the account of a not-entirely-trustworthy person. Unfortunately, the packets decide to take the scenic route to the destination and go off exploring a remote corner of the network. The sender then times out and sends them all again. This time the packets take the shortest route and are delivered quickly so the sender releases the connection.

Unfortunately, eventually the initial batch of packets finally come out of hiding and arrive at the destination in order, asking the bank to establish a new connection and transfer money (again). The bank has no way of telling that these are duplicates. It must assume that this is a second, independent transaction, and transfers the money again.

This scenario may sound unlikely, or even implausible but the point is this: protocols must be designed to be correct in all cases. Only the common cases need be implemented efficiently to obtain good network performance, but the protocol must be able to cope with the uncommon cases without breaking. If it cannot, we have built a fair-weather network that can fail without warning when the conditions get tough.

For the remainder of this section, we will study the problem of delayed duplicates, with emphasis on algorithms for establishing connections in a reliable way, so that nightmares like the one above cannot happen. The crux of the problem is that the delayed duplicates are thought to be new packets. We cannot prevent packets from being duplicated and delayed. But if and when this happens, the packets must be rejected as duplicates and not processed as fresh packets.

The problem can be attacked in various ways, none of them terribly satisfactory. One way is to use throwaway transport addresses. In this approach, each time a transport address is needed, a brand new one is generated. When a connection is released, the address is discarded and never used again. Delayed duplicate packets then never find their way to a transport process and can do no damage. However, this approach makes it more difficult to connect with a process in the first place.

Another option is to give each connection a unique identifier (i.e., a sequence number incremented for each connection established) chosen by the initiating party and put in each segment, including the one requesting the connection. After each connection is released, each transport entity can update a table listing obsolete connections as (peer transport entity, connection identifier) pairs. Whenever a connection request comes in, it can be checked against the table to see if it belongs to a previously released connection.

Unfortunately, this scheme has a basic flaw: it requires each transport entity to maintain a certain amount of history information effectively indefinitely. This history must persist at both the source and destination machines. Otherwise, if a

machine crashes and loses its memory, it will no longer know which connection identifiers have already been used by its peers.

Instead, we need to take a different tack to simplify the problem. Rather than allowing packets to live forever within the network, we devise a mechanism to kill off aged packets that are still hobbling about. With this restriction, the problem becomes somewhat more manageable.

Packet lifetime can be restricted to a known maximum using one (or more) of the following techniques:

1. Restricted network design.
2. Putting a hop counter in each packet.
3. Timestamping each packet.

The first technique includes any method that prevents packets from looping, combined with some way of bounding delay including congestion over the (now known) longest possible path. It is difficult, given that internets may range from a single city to international in scope. The second method consists of having the hop count initialized to some appropriate value and decremented each time the packet is forwarded. The network protocol simply discards any packet whose hop counter becomes zero. The third method requires each packet to bear the time it was created, with the routers agreeing to discard any packet older than some agreed-upon time. This latter method requires the router clocks to be synchronized, which itself is a nontrivial task, and in practice a hop counter is a close enough approximation to age.

In practice, we will need to guarantee not only that a packet is dead, but also that all acknowledgements to it are dead, too, so we will now introduce a period T, which is some small multiple of the true maximum packet lifetime. The maximum packet lifetime is a conservative constant for a network; for the Internet, it is somewhat arbitrarily taken to be 120 seconds. The multiple is protocol dependent and simply has the effect of making T longer. If we wait a time T secs after a packet has been sent, we can be sure that all traces of it are now gone and that neither it nor its acknowledgements will suddenly appear out of the blue to complicate matters.

With packet lifetimes bounded, it is possible to devise a practical and foolproof way to reject delayed duplicate segments. The method described below is due to Tomlinson (1975), as refined by Sunshine and Dalal (1978). Variants of it are widely used in practice, including in TCP.

The heart of the method is for the source to label segments with sequence numbers that will not be reused within T secs. The period, T, and the rate of packets per second determine the size of the sequence numbers. In this way, only one packet with a given sequence number may be outstanding at any given time. Duplicates of this packet may still occur, and they must be discarded by the destination.

However, it is no longer the case that a delayed duplicate of an old packet may beat a new packet with the same sequence number and be accepted by the destination in its stead.

To get around the problem of a machine losing all memory of where it was after a crash, one possibility is to require transport entities to be idle for T secs after a recovery. The idle period will let all old segments die off, so the sender can start again with any sequence number. However, in a complex internetwork, T may be large, so this strategy is unattractive.

Instead, Tomlinson proposed equipping each host with a time-of-day clock. The clocks at different hosts need not be synchronized. Each clock is assumed to take the form of a binary counter that increments itself at uniform intervals. Furthermore, the number of bits in the counter must equal or exceed the number of bits in the sequence numbers. Last, and most important, the clock is assumed to continue running even if the host goes down.

When a connection is set up, the low-order k-bits of the clock are used as the k-bit initial sequence number. Thus, unlike our protocols of Chap. 3, each connection starts numbering its segments with a different initial sequence number. The sequence space should be so large that by the time sequence numbers wrap around, old segments with the same sequence number are long gone. This linear relation between time and initial sequence numbers is shown in Fig. 6-10(a). The forbidden region shows the times for which segment sequence numbers are illegal leading up to their use. If any segment is sent with a sequence number in this region, it could be delayed and impersonate a different packet with the same sequence number that will be issued slightly later. For example, if the host crashes and restarts at time 70 seconds, it will use initial sequence numbers based on the clock to pick up after it left off; the host does not start with a lower sequence number in the forbidden region.

Once both transport entities have agreed on the initial sequence number, any sliding window protocol can be used for data flow control. This window protocol will correctly find and discard duplicates of packets after they have already been accepted. In reality, the initial sequence number curve (shown by the heavy line) is not linear, but a staircase, since the clock advances in discrete steps. For simplicity, we will ignore this detail.

To keep packet sequence numbers out of the forbidden region, we need to take care in two respects. We can get into trouble in two distinct ways. If a host sends too much data too fast on a newly opened connection, the actual sequence number versus time curve may rise more steeply than the initial sequence number versus time curve, causing the sequence number to enter the forbidden region. To prevent this from happening, the maximum data rate on any connection is one segment per clock tick. This also means that the transport entity must wait until the clock ticks before opening a new connection after a crash restart, lest the same number be used twice. Both of these points argue in favor of a short clock tick (1 μsec or less). However, the clock cannot tick too fast relative to the sequence number. For

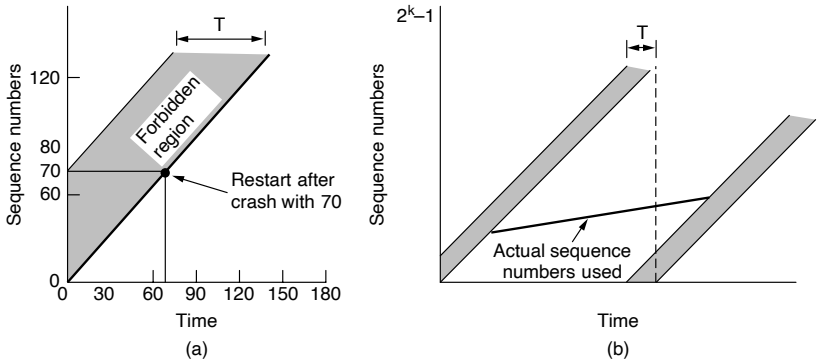


Figure 6-10. (a) Segments may not enter the forbidden region. (b) The resynchronization problem.

a clock rate of C and a sequence number space of size S, we must have $S/C > T$ so that the sequence numbers cannot wrap around too quickly.

Entering the forbidden region from underneath by sending too fast is not the only way to get into trouble. From Fig. 6-10(b), we see that at any data rate less than the clock rate, the curve of actual sequence numbers used versus time will eventually run into the forbidden region from the left as the sequence numbers wrap around. The greater the slope of the actual sequence numbers, the longer this event will be delayed. Avoiding this situation limits how slowly sequence numbers can advance on a connection (or how long the connections may last).

The clock-based method solves the problem of not being able to distinguish delayed duplicate segments from new segments. However, there is a practical snag for using it for establishing connections. Since we do not normally remember sequence numbers across connections at the destination, we still have no way of knowing if a CONNECTION REQUEST segment containing an initial sequence number is a duplicate of a recent connection. This snag does not exist during a connection because the sliding window protocol does remember the current sequence number.

Solution: Three-Way Handshake

To solve this specific problem, Tomlinson (1975) introduced the **three-way handshake**. This establishment protocol involves one peer checking with the other that the connection request is indeed current. The normal setup procedure when host 1 initiates is shown in Fig. 6-11(a). Host 1 chooses a sequence number, x, and sends a CONNECTION REQUEST segment containing it to host 2. Host 2 replies with an ACK segment acknowledging x and announcing its own initial sequence

number, *y*. Finally, host 1 acknowledges host 2's choice of an initial sequence number in the first data segment that it sends.

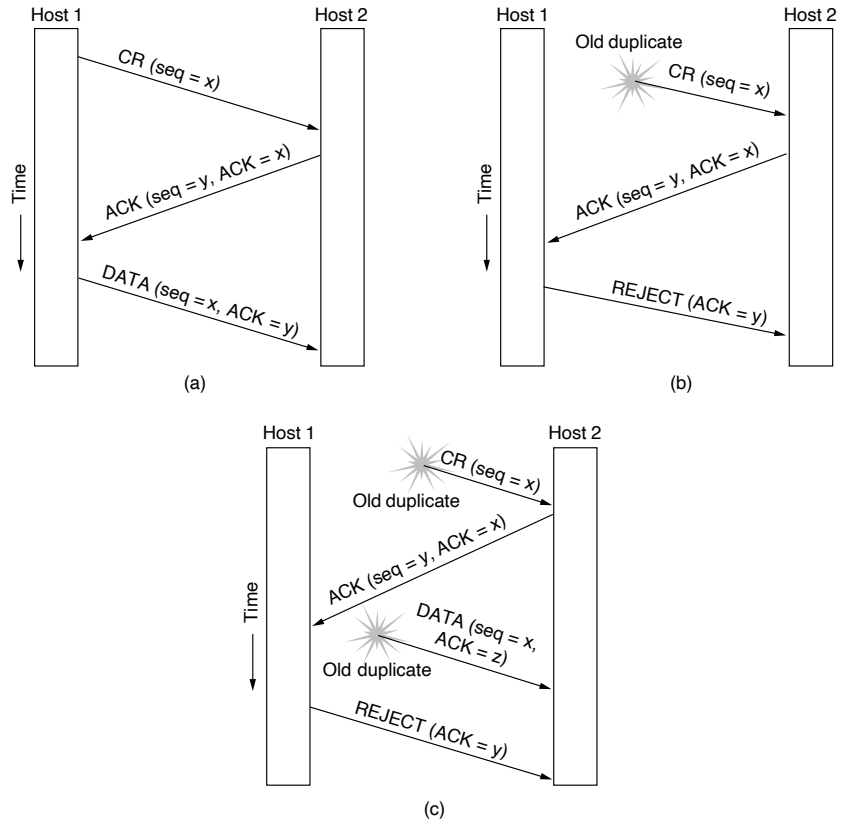


Figure 6-11. Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. (a) Normal operation. (b) Old duplicate CONNECTION REQUEST appearing out of nowhere. (c) Duplicate CONNECTION REQUEST and duplicate ACK.

Now let us see how the three-way handshake works in the presence of delayed duplicate control segments. In Fig. 6-11(b), the first segment is a delayed duplicate CONNECTION REQUEST from an old connection. This segment arrives at host 2 without host 1's knowledge. Host 2 reacts to this segment by sending host 1 an ACK segment, in effect asking for verification that host 1 was indeed trying to set up a new connection. When host 1 rejects host 2's attempt to establish a connection, host 2 realizes that it was tricked by a delayed duplicate and abandons the connection. In this way, a delayed duplicate does no damage.

The worst case is when both a delayed CONNECTION REQUEST and an ACK are floating around in the subnet. This case is shown in Fig. 6-11(c). As in the previous example, host 2 gets a delayed CONNECTION REQUEST and replies to it. At this point, it is crucial to realize that host 2 has proposed using *y* as the initial sequence number for host 2 to host 1 traffic, knowing full well that no segments containing sequence number *y* or acknowledgements to *y* are still in existence. When the second delayed segment finally arrives at host 2, the fact that *z* has been acknowledged rather than *y* tells host 2 that this, too, is an old duplicate. The important thing to realize here is that there is no combination of old segments that can cause the protocol to fail and have a connection set up by accident when no one wants it.

TCP always uses this three-way handshake to establish connections. Within a connection, a timestamp is used to extend the 32-bit sequence number so that it will not wrap within the maximum packet lifetime, even for gigabit-per-second connections. This mechanism is a fix to TCP that was needed as it was used on faster and faster links. It is described in RFC 1323 and called **PAWS** (**Protection Against Wrapped Sequence numbers**). Across connections, for the initial sequence numbers and before PAWS can come into play, TCP originally used the clock-based scheme just described. However, this turned out to have a security vulnerability. The clock made it easy for an attacker to predict the next initial sequence number and send packets that tricked the three-way handshake and established a forged connection. To close this hole, pseudorandom initial sequence numbers are used for connections in practice. However, it remains important that the initial sequence numbers not repeat for an interval even though they appear random to an observer. Otherwise, delayed duplicates can wreak havoc.

6.2.3 Connection Release

Releasing a connection is easier than establishing one. Nevertheless, there are more pitfalls than one might expect here. As we mentioned earlier, there are two styles of terminating a connection: asymmetric release and symmetric release. Asymmetric release is the way the telephone system works: when one party hangs up, the connection is broken. Symmetric release treats the connection as two separate unidirectional connections and requires each one to be released separately.

Asymmetric release is abrupt and may result in data loss. Consider the scenario of Fig. 6-12. After the connection is established, host 1 sends a segment that arrives properly at host 2. Then host 1 sends another segment. Unfortunately, host 2 issues a DISCONNECT before the second segment arrives. The result is that the connection is released and data are lost.

Clearly, a more sophisticated release protocol is needed to avoid data loss. One way is to use symmetric release, in which each direction is released independently of the other one. Here, a host can continue to receive data even after it has sent a DISCONNECT segment.

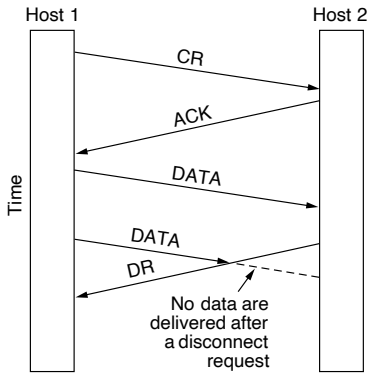


Figure 6-12. Abrupt disconnection with loss of data.

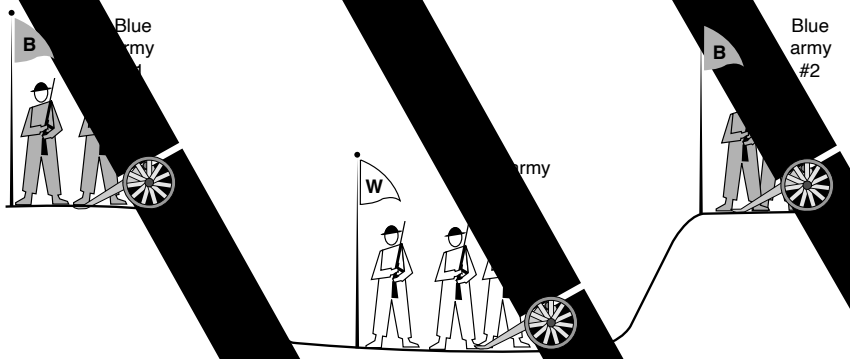
Symmetric release does the job when each process has a fixed amount of data to send and clearly knows when it has sent it. In other situations, determining that all the work has been done and the connection should be terminated is not so obvious. One can envision a protocol in which host 1 says ''I am done. Are you done too?'' If host 2 responds: ''I am done too. Goodbye, the connection can be safely released.''

Unfortunately, this protocol does not always work. There is a famous problem that illustrates this issue. It is called the **two-army problem**. Imagine that a white army is encamped in a valley, as shown in Fig. 6-13. On both of the surrounding hillsides are blue armies. The white army is larger than either of the blue armies alone, but together the blue armies are larger than the white army. If either blue army attacks by itself, it will be defeated, but if the two blue armies attack simultaneously, they will be victorious.

The blue armies want to synchronize their attacks. However, their only communication medium is to send messengers on foot down into the valley, where they might be captured and the message lost (i.e., they have to use an unreliable communication channel). The question is: does a protocol exist that allows the blue armies to win?

Suppose that the commander of blue army #1 sends a message reading: ''I propose we attack at dawn on March 29. How about it?'' Now suppose that the message arrives, the commander of blue army #2 agrees, and his reply gets safely back to blue army #1. Will the attack happen? Probably not, because commander #2 does not know if his reply got through. If it did not, blue army #1 will not attack, so it would be foolish for him to charge into battle.

Now let us improve the protocol by making it a three-way handshake. The initiator of the original proposal must acknowledge the response. Assuming no messages are lost, blue army #2 will get the acknowledgement, but the commander of



6-13. The two-army problem

blue army #1 will now hesi fter all, he does not kn s acknowledgement got through, and if it did knows that blue army # ot attack. We could now make a four-way han protocol, but that does either.

In fact, it can be proven that ocol exists that works. e that some protocol did exist. Either the last of the protocol is esse it is not. If it is not, we can remove it (and an unessential messages) are left with a protocol in which every mess essential. What happe e final message does not get through? We just t it was essential, so if , the attack does not take place. Since the sen he final message can ne ure of its arrival, he will not risk attacking. W the other blue army kn s, so it will not attack either.

To see the relevance of the two-army pro releasing connections, than to military affairs, just substitute ''disconn r ''attack.'' If neither si prepared to disconnect until it is convinced th other side is prepared disconnect too, the disconnection will never happe

In practice, we can avoid this quandary by fore he need for agreement and pushing the problem up to the transport user, let ch side independently decide when it is done. This is an easier problem to so gure 6-14 illustrates four scenarios of releasing using a three-way handshake. this protocol is not infallible, it is usually adequate.

In Fig. 6-14(a), we see the normal case in which one o ers sends a DR (DISCONNECTION REQUEST) segment to initiate the connect ase. When it arrives, the recipient sends back a DR segment and starts a timer case its DR is lost. When this DR arrives, the original sender sends back an gment and releases the connection. Finally, when the ACK segment arrives, iver also releases the connection. Releasing a connection means that the t entity

removes the information about the connection from its table of currently open connections and signals the connection's owner (the transport user) somehow. This action is different from a transport user issuing a DISCONNECT primitive.

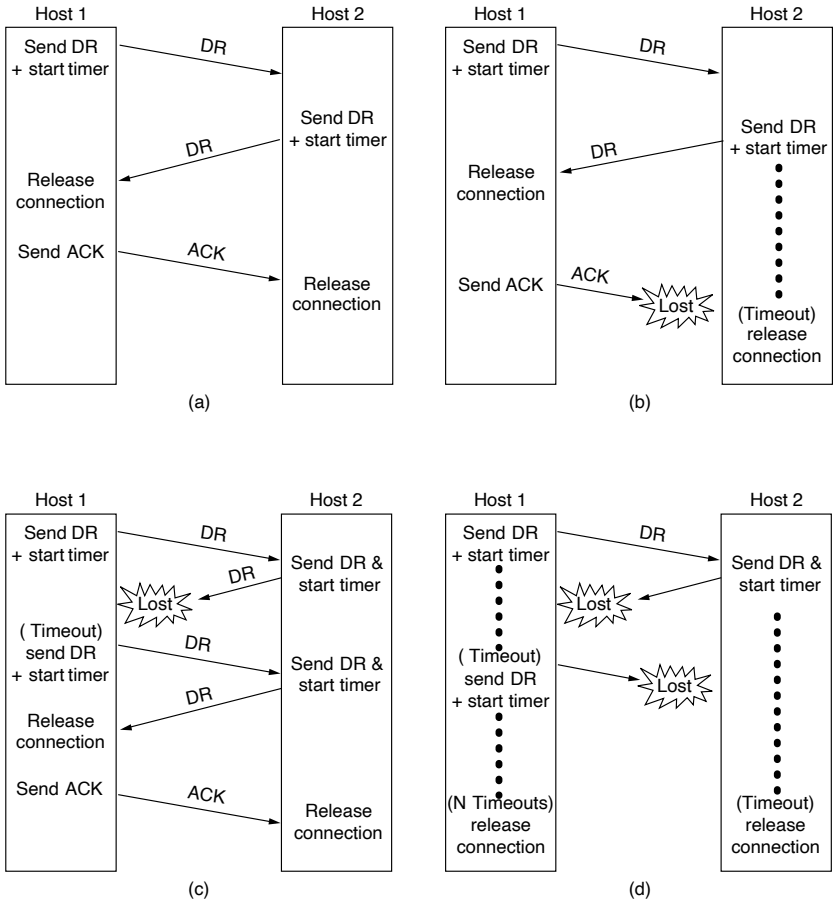


Figure 6-14. Four protocol scenarios for releasing a connection. (a) Normal case of three-way handshake. (b) Final ACK lost. (c) Response lost. (d) Response lost and subsequent DRs lost.

If the final ACK segment is lost, as shown in Fig. 6-14(b), the situation is saved by the timer. When the timer expires, the connection is released anyway.

Now consider the case of the second DR being lost. The user initiating the disconnection will not receive the expected response, will time out, and will start all over again. In Fig. 6-14(c), we see how this works, assuming that the second time no segments are lost and all segments are delivered correctly and on time.

Our last scenario, Fig. 6-14(d), is the same as Fig. 6-14(c) except that now we assume all the repeated attempts to retransmit the DR also fail due to lost segments. After *N* retries, the sender just gives up and releases the connection. Meanwhile, the receiver times out and also exits.

While this protocol usually suffices, in theory it can fail if the initial DR and *N* retransmissions are all lost. The sender will give up and release the connection, while the other side knows nothing at all about the attempts to disconnect and is still fully active. This situation results in a half-open connection. That is unacceptable.

We could have avoided this problem by not allowing the sender to give up after *N* retries and forcing it to go on forever until it gets a response. However, if the other side is allowed to time out, the sender will indeed go on forever, because no response will ever be forthcoming. If we do not allow the receiving side to time out, the protocol hangs in Fig. 6-14(d).

One way to kill off half-open connections is to have a rule saying that if no segments have arrived for a certain number of seconds, the connection is automatically disconnected. That way, if one side ever disconnects, the other side will detect the lack of activity and also disconnect. This rule also takes care of the case where the connection is broken (because the network can no longer deliver packets between the hosts) without either end disconnecting first.

Of course, if this rule is introduced, it is necessary for each transport entity to have a timer that is stopped and then restarted whenever a segment is sent. If this timer expires, a dummy segment is transmitted, just to keep the other side from disconnecting. On the other hand, if the automatic disconnect rule is used and too many dummy segments in a row are lost on an otherwise idle connection, first one side, then the other will automatically disconnect.

We will not belabor this point any more, but by now it should be clear that releasing a connection without data loss is not nearly as simple as it first appears. The lesson here is that the transport user must be involved in deciding when to disconnect—the problem cannot be cleanly solved by the transport entities themselves. To see the importance of the application, consider that while TCP normally does a symmetric close (with each side independently closing its half of the connection with a FIN packet when it has sent its data), many Web servers send the client a RST packet that causes an abrupt close of the connection that is more like an asymmetric close. This works only because the Web server knows the pattern of data exchange. First it receives a request from the client, which is all the data the client will send, and then it sends a response to the client.

When the Web server is finished with its response, all of the data has been sent in either direction. The server can send the client a warning and abruptly shut the connection. If the client gets this warning, it will release its connection state then and there. If the client does not get the warning, it will eventually realize that the server is no longer talking to it and release the connection state. The data has been successfully transferred in either case.

6.2.4 Error Control and Flow Control

Having examined connection establishment and release in some detail, let us now look at how connections are managed while they are in use. The key issues are error control and flow control. Error control is ensuring that the data is delivered with the desired level of reliability, usually that all of the data is delivered without any errors. Flow control is keeping a fast transmitter from overrunning a slow receiver.

Both of these issues have come up before, when we studied the data link layer. The solutions that are used at the transport layer are the same mechanisms that we studied in Chap. 3. As a very brief recap:

1. A frame carries an error-detecting code (e.g., a CRC or checksum) that is used to check if the information was correctly received.
2. A frame carries a sequence number to identify itself and is retransmitted by the sender until it receives an acknowledgement of successful receipt from the receiver. This is called **ARQ** (**Automatic Repeat reQuest**).
3. There is a maximum number of frames that the sender will allow to be outstanding at any time, pausing if the receiver is not acknowledging frames quickly enough. If this maximum is one packet the protocol is called **stop-and-wait**. Larger windows enable pipelining and improve performance on long, fast links.
4. The **sliding window** protocol combines these features and is also used to support bidirectional data transfer.

Given that these mechanisms are used on frames at the link layer, it is natural to wonder why they would be used on segments at the transport layer as well. However, there is little duplication between the link and transport layers in practice. Even though the same mechanisms are used, there are differences in function and degree.

For a difference in function, consider error detection. The link layer checksum protects a frame while it crosses a single link. The transport layer checksum protects a segment while it crosses an entire network path. It is an end-to-end check, which is not the same as having a check on every link. Saltzer et al. (1984) describe a situation in which packets were corrupted inside a router. The link layer checksums protected the packets only while they traveled across a link, not while they were inside the router. Thus, packets were delivered incorrectly even though they were correct according to the checks on every link.

This and other examples led Saltzer et al. to articulate what is called the **end-to-end argument**. According to this argument, the transport layer check that runs end-to-end is essential for correctness, and the link layer checks are not essential

but nonetheless valuable for improving performance (since without them a corrupted packet can be sent along the entire path unnecessarily).

As a difference in degree, consider retransmissions and the sliding window protocol. Most wireless links, other than satellite links, can have only a single frame outstanding from the sender at a time. That is, the bandwidth-delay product for the link is small enough that not even a whole frame can be stored inside the link. In this case, a small window size is sufficient for good performance. For example, 802.11 uses a stop-and-wait protocol, transmitting or retransmitting each frame and waiting for it to be acknowledged before moving on to the next frame. Having a window size larger than one frame would add complexity without improving performance. For wired and optical fiber links, such as (switched) Ethernet or ISP backbones, the error-rate is low enough that link-layer retransmissions can be omitted because the end-to-end retransmissions will repair the residual frame loss.

On the other hand, many TCP connections have a bandwidth-delay product that is much larger than a single segment. Consider a connection sending data across the U.S. at 1 Mbps with a round-trip time of 200 msec. Even for this slow connection, 200 Kbit of data will be stored at the receiver in the time it takes to send a segment and receive an acknowledgement. For these situations, a large sliding window must be used. Stop-and-wait will cripple performance. In our example it would limit performance to one segment every 200 msec, or 5 segments/sec no matter how fast the network really is.

Given that transport protocols generally use larger sliding windows, we will look at the issue of buffering data more carefully. Since a host may have many connections, each of which is treated separately, it may need a substantial amount of buffering for the sliding windows. The buffers are needed at both the sender and the receiver. Certainly they are needed at the sender to hold all transmitted but as yet unacknowledged segments. They are needed there because these segments may be lost and need to be retransmitted.

However, since the sender is buffering, the receiver may or may not dedicate specific buffers to specific connections, as it sees fit. The receiver may, for example, maintain a single buffer pool shared by all connections. When a segment comes in, an attempt is made to dynamically acquire a new buffer. If one is available, the segment is accepted; otherwise, it is discarded. Since the sender is prepared to retransmit segments lost by the network, no permanent harm is done by having the receiver drop segments, although some resources are wasted. The sender just keeps trying until it gets an acknowledgement.

The best trade-off between source buffering and destination buffering depends on the type of traffic carried by the connection. For low-bandwidth bursty traffic, such as that produced by a user typing at a remote computer, it is reasonable not to dedicate any buffers, but rather to acquire them dynamically at both ends, relying on buffering at the sender if segments must occasionally be discarded. On the other hand, for file transfer and most other high-bandwidth traffic, it is better if the

receiver does dedicate a full window of buffers, to allow the data to flow at maximum speed. This is the strategy that TCP uses.

There still remains the question of how to organize the buffer pool. If most segments are nearly the same size, it is natural to organize the buffers as a pool of identically sized buffers, with one segment per buffer, as in Fig. 6-15(a). However, if there is wide variation in segment size, from short requests for Web pages to large packets in peer-to-peer file transfers, a pool of fixed-sized buffers presents problems. If the buffer size is chosen to be equal to the largest possible segment, space will be wasted whenever a short segment arrives. If the buffer size is chosen to be less than the maximum segment size, multiple buffers will be needed for long segments, with the attendant complexity.

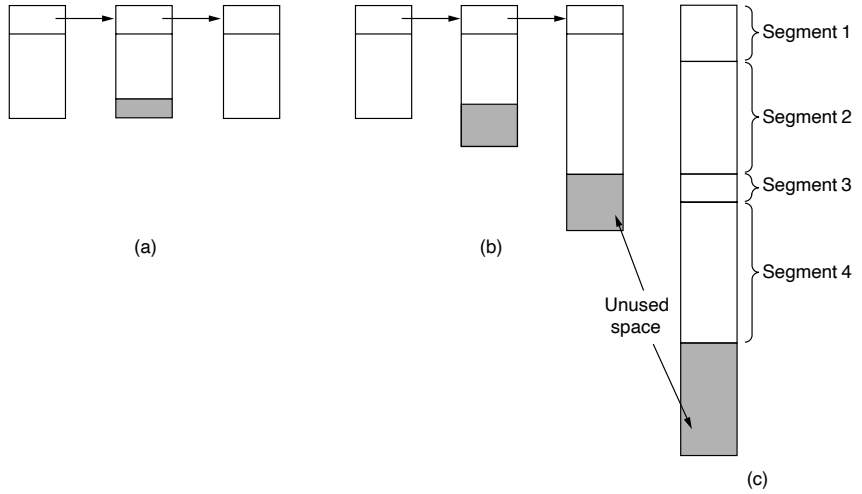


Figure 6-15. (a) Chained fixed-size buffers. (b) Chained variable-sized buffers. (c) One large circular buffer per connection.

Another approach to the buffer size problem is to use variable-sized buffers, as in Fig. 6-15(b). The advantage here is better memory utilization, at the price of more complicated buffer management. A third possibility is to dedicate a single large circular buffer per connection, as in Fig. 6-15(c). This system is simple and elegant and does not depend on segment sizes, but makes good use of memory only when the connections are heavily loaded.

As connections are opened and closed and as the traffic pattern changes, the sender and receiver need to dynamically adjust their buffer allocations. Consequently, the transport protocol should allow a sending host to request buffer space at the other end. Buffers could be allocated per connection, or collectively, for all connections running between the two hosts. Alternatively, the receiver, knowing

its buffer situation (but not knowing the offered traffic) could tell the sender "I have reserved *X* buffers for you." If the number of open connections should increase, it may be necessary for an allocation to be reduced, so the protocol should provide for this possibility.

A reasonably general way to manage dynamic buffer allocation is to decouple the buffering from the acknowledgements, in contrast to the sliding window protocols of Chap. 3. Dynamic buffer management means, in effect, a variable-sized window. Initially, the sender requests a certain number of buffers, based on its expected needs. The receiver then grants as many of these as it can afford. Every time the sender transmits a segment, it must decrement its allocation, stopping altogether when the allocation reaches zero. The receiver separately piggybacks both acknowledgements and buffer allocations onto the reverse traffic. TCP uses this scheme, carrying buffer allocations in a header field called *Window size*.

Figure 6-16 has an example of how dynamic window management might work in a datagram network with 4-bit sequence numbers. In this example, data flows in segments from host *A* to host *B* and acknowledgements and buffer allocations flow in segments in the reverse direction. Initially, *A* wants eight buffers, but it is granted only four of these. It then sends three segments, of which the third is lost. Segment 6 acknowledges receipt of all segments up to and including sequence number 1, thus allowing *A* to release those buffers, and furthermore informs *A* that it has permission to send three more segments starting beyond 1 (i.e., segments 2, 3, and 4). *A* knows that it has already sent number 2, so it thinks that it may send segments 3 and 4, which it proceeds to do. At this point it is blocked and must wait for more buffer allocation. Timeout-induced retransmissions (line 9), however, may occur while blocked, since they use buffers that have already been allocated. In line 10, *B* acknowledges receipt of all segments up to and including 4 but refuses to let *A* continue. Such a situation is impossible with the fixed-window protocols of Chap. 3. The next segment from *B* to *A* allocates another buffer and allows *A* to continue. This will happen when *B* has buffer space, likely because the transport user has accepted more segment data.

Problems with buffer allocation schemes of this kind can arise in datagram networks if control segments can get lost—which they most certainly can. Look at line 16. *B* has now allocated more buffers to *A*, but the allocation segment was lost. Oops. Since control segments are not sequenced or timed out, *A* is now deadlocked. To prevent this situation, each host should periodically send control segments giving the acknowledgement and buffer status on each connection. That way, the deadlock will be broken, sooner or later.

Until now we have assumed that the only limit imposed on the sender's data rate is the amount of buffer space available in the receiver. This is often not the case. Memory was once expensive but prices have fallen dramatically. Hosts may be equipped with sufficient memory that the lack of buffers is rarely a problem, even for wide area connections. Of course, this depends on the buffer size being set to be large enough, which is not always the case for TCP (Zhang et al., 2002).

	A	Message	B	Comments
1	→	< request 8 buffers>	→	A wants 8 buffers
2	←	<ack = 15, buf = 4>	←	B grants messages 0-3 only
3	→	<seq = 0, data = m0>	→	A has 3 buffers left now
4	→	<seq = 1, data = m1>	→	A has 2 buffers left now
5	→	<seq = 2, data = m2>	•••	Message lost but A thinks it has 1 left
6	←	<ack = 1, buf = 3>	←	B acknowledges 0 and 1, permits 2-4
7	→	<seq = 3, data = m3>	→	A has 1 buffer left
8	→	<seq = 4, data = m4>	→	A has 0 buffers left, and must stop
9	→	<seq = 2, data = m2>	→	A times out and retransmits
10	←	<ack = 4, buf = 0>	←	Everything acknowledged, but A still blocked
11	←	<ack = 4, buf = 1>	←	A may now send 5
12	←	<ack = 4, buf = 2>	←	B found a new buffer somewhere
13	→	<seq = 5, data = m5>	→	A has 1 buffer left
14	→	<seq = 6, data = m6>	→	A is now blocked again
15	←	<ack = 6, buf = 0>	←	A is still blocked
16	•••	<ack = 6, buf = 4>	←	Potential deadlock

Figure 6-16. Dynamic buffer allocation. The arrows show the direction of transmission. An ellipsis (...) indicates a lost segment.

When buffer space no longer limits the maximum flow, another bottleneck will appear: the carrying capacity of the network. If adjacent routers can exchange at most x packets/sec and there are k disjoint paths between a pair of hosts, there is no way that those hosts can exchange more than kx segments/sec, no matter how much buffer space is available at each end. If the sender pushes too hard (i.e., sends more than kx segments/sec), the network will become congested because it will be unable to deliver segments as fast as they are coming in.

What is needed is a mechanism that limits transmissions from the sender based on the network's carrying capacity rather than on the receiver's buffering capacity. Belsnes (1975) proposed using a sliding window flow-control scheme in which the sender dynamically adjusts the window size to match the network's carrying capacity.

This means that a dynamic sliding window can implement both flow control and congestion control. If the network can handle c segments/sec and the round-trip time (including transmission, propagation, queueing, processing at the receiver, and return of the acknowledgement) is r, the sender's window should be cr. With a window of this size, the sender normally operates with the pipeline full. Any small decrease in network performance will cause it to block. Since the network capacity available to any given flow varies over time, the window size should be adjusted frequently, to track changes in the carrying capacity. As we will see later, TCP uses a similar scheme.

6.2.5 Multiplexing

Multiplexing, or sharing several conversations over connections, virtual circuits, and physical links plays a role in several layers of the network architecture. In the transport layer, the need for multiplexing can arise in a number of ways. For example, if only one network address is available on a host, all transport connections on that machine have to use it. When a segment comes in, some way is needed to tell which process to give it to. This situation, called **multiplexing**, is shown in Fig. 6-17(a). In this figure, four distinct transport connections all use the same network connection (e.g., IP address) to the remote host.

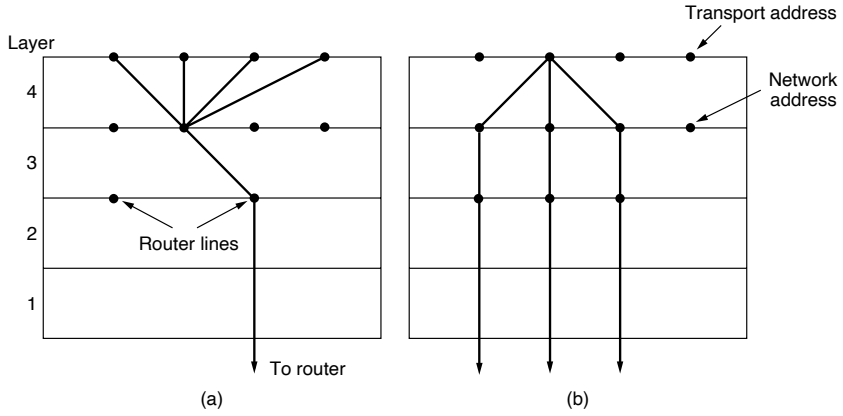


Figure 6-17. (a) Multiplexing. (b) Inverse multiplexing.

Multiplexing can also be useful in the transport layer for another reason. Suppose, for example, that a host has multiple network paths that it can use. If a user needs more bandwidth or more reliability than one of the network paths can provide, a way out is to have a connection that distributes the traffic among multiple network paths on a round-robin basis, as indicated in Fig. 6-17(b). This modus operandi is called **inverse multiplexing**. With k network connections open, the effective bandwidth might be increased by a factor of k. An example of inverse multiplexing is SCTP which can run a connection using multiple network interfaces. In contrast, TCP uses a single network endpoint. Inverse multiplexing is also found at the link layer, when several low-rate links are used in parallel as one fast link.

6.2.6 Crash Recovery

If hosts and routers are subject to crashes or connections are long-lived (e.g., large software or media downloads), recovery from these crashes becomes an issue. If the transport entity is entirely within the hosts, recovery from network

and router crashes is straightforward. The transport entities expect lost segments all the time and know how to cope with them by using retransmissions.

A more troublesome problem is how to recover from host crashes. In particular, it may be desirable for clients to be able to continue working when servers crash and quickly reboot. To illustrate the difficulty, let us assume that one host, the client, is sending a long file to another host, the file server, using a simple stop-and-wait protocol. The transport layer on the server just passes the incoming segments to the transport user, one by one. Partway through the transmission, the server crashes. When it comes back up, its tables are reinitialized, so it no longer knows precisely where it was.

In an attempt to recover its previous status, the server might send a broadcast segment to all other hosts, announcing that it has just crashed and requesting that its clients inform it of the status of all open connections. Each client can be in one of two states: one segment outstanding, *S1*, or no segments outstanding, *S0*. Based on only this state information, the client must decide whether to retransmit the most recent segment.

At first glance, it would seem obvious: the client should retransmit if and only if it has an unacknowledged segment outstanding (i.e., is in state *S1*) when it learns of the crash. However, a closer inspection reveals difficulties with this naive approach. Consider, for example, the situation in which the server's transport entity first sends an acknowledgement and then, when the acknowledgement has been sent, writes to the application process. Writing a segment onto the output stream and sending an acknowledgement are two distinct events that cannot be done simultaneously. If a crash occurs after the acknowledgement has been sent but before the write has been fully completed, the client will receive the acknowledgement and thus be in state *S0* when the crash recovery announcement arrives. The client will therefore not retransmit, (incorrectly) thinking that the segment has arrived. This decision by the client leads to a missing segment.

At this point you may be thinking: ''That problem can be solved easily. All you have to do is reprogram the transport entity to first do the write and then send the acknowledgement.'' Try again. Imagine that the write has been done but the crash occurs before the acknowledgement can be sent. The client will be in state *S1* and thus retransmit, leading to an undetected duplicate segment in the output stream to the server application process.

No matter how the client and server are programmed, there are always situations where the protocol fails to recover properly. The server can be programmed in one of two ways: acknowledge first or write first. The client can be programmed in one of four ways: always retransmit the last segment, never retransmit the last segment, retransmit only in state *S0*, or retransmit only in state *S1*. This gives eight combinations, but as we shall see, for each combination there is some set of events that makes the protocol fail.

Three events are possible at the server: sending an acknowledgement (*A*), writing to the output process (*W*), and crashing (*C*). The three events can occur in six

different orderings: *AC*(*W*), *AWC*, *C*(*AW*), *C*(*WA*), *WAC*, and *WC*(*A*), where the parentheses are used to indicate that neither *A* nor *W* can follow *C* (i.e., once it has crashed, it has crashed). Figure 6-18 shows all eight combinations of client and server strategies and the valid event sequences for each one. Notice that for each strategy there is some sequence of events that causes the protocol to fail. For example, if the client always retransmits, the *AWC* event will generate an undetected duplicate, even though the other two events work properly.

	Strategy used by receiving host					
	First ACK, then write			First write, then ACK		
Strategy used by sending host	AC(W)	AWC	C(AW)	C(WA)	W AC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

OK = Protocol functions correctly
DUP = Protocol generates a duplicate message
LOST = Protocol loses a message

Figure 6-18. Different combinations of client and server strategies.

Making the protocol more elaborate does not help. Even if the client and server exchange several segments before the server attempts to write, so that the client knows exactly what is about to happen, the client has no way of knowing whether a crash occurred just before or just after the write. The conclusion is inescapable: under our ground rules of no simultaneous events—that is, separate events happen one after another not at the same time—host crash and recovery cannot be made transparent to higher layers.

Put in more general terms, this result can be restated as ''recovery from a layer *N* crash can only be done by layer *N* + 1,'' and then only if the higher layer retains enough status information to reconstruct where it was before the problem occurred. This is consistent with the case mentioned above that the transport layer can recover from failures in the network layer, provided that each end of a connection keeps track of where it is.

This problem gets us into the issue of what a so-called end-to-end acknowledgement really means. In principle, the transport protocol is end-to-end and not chained like the lower layers. Now consider the case of a user entering requests for transactions against a remote database. Suppose that the remote transport entity is programmed to first pass segments to the next layer up and then acknowledge.

Even in this case, the receipt of an acknowledgement back at the user's machine does not necessarily mean that the remote host stayed up long enough to actually update the database. A truly end-to-end acknowledgement, whose receipt means that the work has actually been done and lack thereof means that it has not, is probably impossible to achieve. This point is discussed in more detail by Saltzer et al. (1984).

6.3 CONGESTION CONTROL

If the transport entities on many machines send too many packets into the network too quickly, the network will become congested, with performance degraded as packets are delayed and lost. Controlling congestion to avoid this problem is the combined responsibility of the network and transport layers. Congestion occurs at routers, so it is detected at the network layer. However, congestion is ultimately caused by traffic sent into the network by the transport layer. The only effective way to control congestion is for the transport protocols to send packets into the network more slowly.

In Chap. 5, we studied congestion control mechanisms in the network layer. In this section, we will study the other half of the problem, congestion control mechanisms in the transport layer. After describing the goals of congestion control, we will describe how hosts can regulate the rate at which they send packets into the network. The Internet relies heavily on the transport layer for congestion control, and specific algorithms are built into TCP and other protocols.

6.3.1 Desirable Bandwidth Allocation

Before we describe how to regulate traffic, we must understand what we are trying to achieve by running a congestion control algorithm. That is, we must specify the state in which a good congestion control algorithm will operate the network. The goal is more than to simply avoid congestion. It is to find a good allocation of bandwidth to the transport entities that are using the network. A good allocation will deliver good performance because it uses all the available bandwidth but avoids congestion, it will be fair across competing transport entities, and it will quickly track changes in traffic demands. We will make each of these criteria more precise in turn.

Efficiency and Power

An efficient allocation of bandwidth across transport entities will use all of the network capacity that is available. However, it is not quite right to think that if there is a 100-Mbps link, five transport entities should get 20 Mbps each. They should usually get less than 20 Mbps for good performance. The reason is that the

traffic is often bursty. Recall that in Sec. 5.3 we described the **goodput** (or rate of useful packets arriving at the receiver) as a function of the offered load. This curve and a matching curve for the delay as a function of the offered load are given in Fig. 6-19.

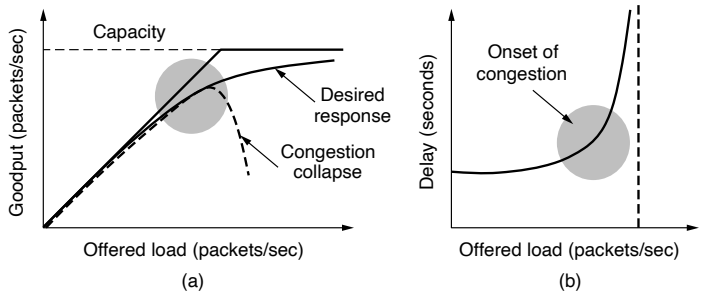


Figure 6-19. (a) Goodput and (b) delay as a function of offered load.

As the load increases in Fig. 6-19(a) goodput initially increases at the same rate, but as the load approaches the capacity, goodput rises more gradually. This falloff is because bursts of traffic can occasionally mount up and cause some losses at buffers inside the network. If the transport protocol is poorly designed and retransmits packets that have been delayed but not lost, the network can enter congestion collapse. In this state, senders are furiously sending packets, but increasingly little useful work is being accomplished.

The corresponding delay is given in Fig. 6-19(b) Initially the delay is fixed, representing the propagation delay across the network. As the load approaches the capacity, the delay rises, slowly at first and then much more rapidly. This is again because of bursts of traffic that tend to mound up at high load. The delay cannot really go to infinity, except in a model in which the routers have infinite buffers. Instead, packets will be lost after experiencing the maximum buffering delay.

For both goodput and delay, performance begins to degrade at the onset of congestion. Intuitively, we will obtain the best performance from the network if we allocate bandwidth up until the delay starts to climb rapidly. This point is below the capacity. To identify it, Kleinrock (1979) proposed the metric of **power**, where

$$power = \frac{load}{delay}$$

Power will initially rise with offered load, as delay remains small and roughly constant, but will reach a maximum and fall as delay grows rapidly. The load with the highest power represents an efficient load for the transport entity to place on the network. The network should try to stay close it as best it can.

Max-Min Fairness

In the discussion above, we did not talk about how to divide bandwidth between different transport senders. This sounds like a simple question—give all the senders an equal fraction of the bandwidth—but it is more complicated than that.

Perhaps the first consideration is to ask what this problem has to do with congestion control. After all, if the network gives a sender some amount of bandwidth to use, the sender should just use that much bandwidth. However, it is often the case that networks do not have a strict bandwidth reservation for each flow or connection. They may for some flows if quality of service is supported, but many connections will seek to use whatever bandwidth is available or be lumped together by the network under a common allocation. For example, IETF's differentiated services separates traffic into two classes and connections compete for bandwidth within each class. IP routers often have all connections competing for the same bandwidth. In this situation, it is the congestion control mechanism that is allocating bandwidth to the competing connections.

A second consideration is what a fair portion means for flows in a network. It is simple enough if N flows use a single link, in which case they can all have $1/N$ of the bandwidth (although efficiency will dictate that they use slightly less if the traffic is bursty). But what happens if the flows have different, but overlapping, network paths? For example, one flow may cross three links, and the other flows may cross one link. The three-link flow consumes more network resources. It might be fairer in some sense to give it less bandwidth than the one-link flows. It should certainly be possible to support more one-link flows by reducing the bandwidth of the three-link flow. This point demonstrates an inherent tension between fairness and efficiency.

However, we will adopt a notion of fairness that does not depend on the length of the network path. Even with this simple model, giving connections an equal fraction of bandwidth is a bit complicated because different connections will take different paths through the network and these paths will themselves have different capacities. In this case, it is possible for a flow to be bottlenecked on a downstream link and take a smaller portion of an upstream link than other flows; reducing the bandwidth of the other flows would slow them down but would not help the bottlenecked flow at all.

The form of fairness that is often desired for network usage is **max-min fairness**. An allocation is max-min fair if the bandwidth given to one flow cannot be increased without decreasing the bandwidth given to another flow with an allocation that is no larger. That is, increasing the bandwidth of a flow will only make the situation worse for flows that are less well off.

Let us see an example. A max-min fair allocation is shown for a network with four flows, A, B, C, and D, in Fig. 6-20. Each of the links between routers has the same capacity, taken to be 1 unit, though in the general case the links will have different capacities. Three flows compete for the bottom-left link between routers *R4*

and *R5*. Each of these flows therefore gets 1/3 of the link. The remaining flow, *A*, competes with *B* on the link from *R2* to *R3*. Since *B* has an allocation of 1/3, *A* gets the remaining 2/3 of the link. Notice that all of the other links have spare capacity. However, this capacity cannot be given to any of the flows without decreasing the capacity of another, lower flow. For example, if more of the bandwidth on the link between *R2* and *R3* is given to flow *B*, there will be less for flow *A*. This is reasonable as flow *A* already has more bandwidth. However, the capacity of flow *C* or *D* (or both) must be decreased to give more bandwidth to *B*, and these flows will have less bandwidth than *B*. Thus, the allocation is max-min fair.

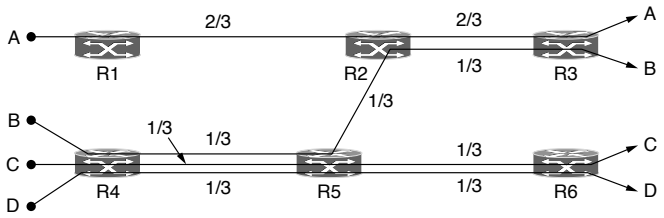


Figure 6-20. Max-min bandwidth allocation for four flows.

Max-min allocations can be computed given a global knowledge of the network. An intuitive way to think about them is to imagine that the rate for all of the flows starts at zero and is slowly increased. When the rate reaches a bottleneck for any flow, that flow stops increasing. The other flows continue to increase, sharing equally in the available capacity, until they too reach their respective bottlenecks.

A third consideration is the level over which to consider fairness. A network could be fair at the level of connections, connections between a pair of hosts, or all connections per host. We examined this issue when we were discussing WFQ (Weighted Fair Queueing) in Sec. 5.4 and concluded that each of these definitions has its problems. For example, defining fairness per host means that a busy server will fare no better than a mobile phone, while defining fairness per connection encourages hosts to open more connections. Given that there is no clear answer, fairness is often considered per connection, but precise fairness is usually not a concern. It is more important in practice that no connection be starved of bandwidth than that all connections get precisely the same amount of bandwidth. In fact, with TCP it is possible to open multiple connections and compete for bandwidth more aggressively. This tactic is used by bandwidth-hungry applications such as BitTorrent for peer-to-peer file sharing.

Convergence

A final criterion is that the congestion control algorithm converge quickly to a fair and efficient allocation of bandwidth. The discussion of the desirable operating point above assumes a static network environment. However, connections are

always coming and going in a network, and the bandwidth needed by a given connection will vary over time too, for example, as a user browses Web pages and occasionally downloads large videos.

Because of the variation in demand, the ideal operating point for the network varies over time. A good congestion control algorithm should rapidly converge to the ideal operating point, and it should track that point as it changes over time. If the convergence is too slow, the algorithm will never be close to the changing operating point. If the algorithm is not stable, it may fail to converge to the right point in some cases, or even oscillate around the right point.

An example of a bandwidth allocation that changes over time and converges quickly is shown in Fig. 6-21. Initially, flow 1 has all of the bandwidth. One second later, flow 2 starts. It needs bandwidth as well. The allocation quickly changes to give each of these flows half the bandwidth. At 4 seconds, a third flow joins. However, this flow uses only 20% of the bandwidth, which is less than its fair share (which is a third). Flows 1 and 2 quickly adjust, dividing the available bandwidth to each have 40% of the bandwidth. At 9 seconds, the second flow leaves, and the third flow remains unchanged. The first flow quickly captures 80% of the bandwidth. At all times, the total allocated bandwidth is approximately 100%, so that the network is fully used, and competing flows get equal treatment (but do not have to use more bandwidth than they need).

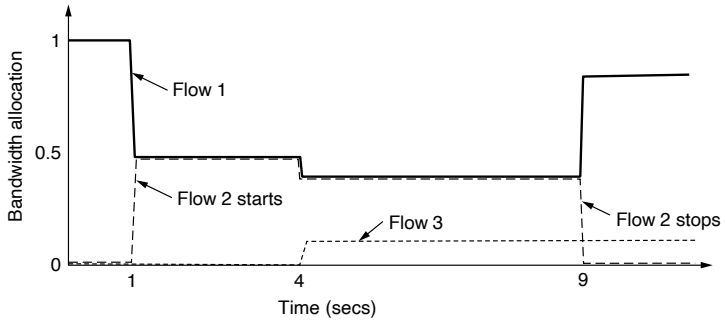


Figure 6-21. Changing bandwidth allocation over time.

6.3.2 Regulating the Sending Rate

Now it is time for the main course. How do we regulate the sending rates to obtain a desirable bandwidth allocation? The sending rate may be limited by two factors. The first is flow control, in the case that there is insufficient buffering at the receiving end. The second is congestion, in the case that there is insufficient capacity in the network. In Fig. 6-22, we see this problem illustrated hydraulically.

In Fig. 6-22(a), we see a thick pipe leading to a small-capacity receiver. This is a flow-control limited situation. As long as the sender does not send more water than the bucket can contain, no water will be lost. In Fig. 6-22(b), the limiting factor is not the bucket capacity, but the internal carrying capacity of the network. If too much water comes in too fast, it will back up and some will be lost (in this case, by overflowing the funnel).

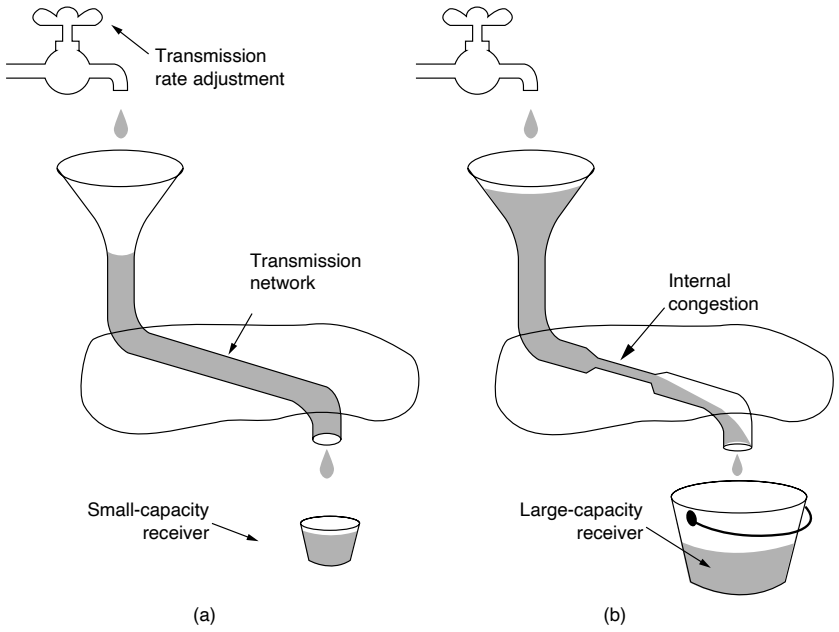


Figure 6-22. (a) A fast network feeding a low-capacity receiver. (b) A slow network feeding a high-capacity receiver.

These cases may appear similar to the sender, as transmitting too fast causes packets to be lost. However, they have different causes and call for different solutions. We have already talked about a flow-control solution with a variable-sized window. Now we will consider a congestion control solution. Since either of these problems can occur, the transport protocol will in general need to run both solutions and slow down if either problem occurs.

The way that a transport protocol should regulate the sending rate depends on the form of the feedback returned by the network. Different network layers may return different kinds of feedback. The feedback may be explicit or implicit, and it may be precise or imprecise.

An example of an explicit, precise design is when routers tell the sources the rate at which they may send. Designs in the literature such as XCP (eXplicit Congestion Protocol) operate in this manner (Katabi et al., 2002). An explicit, imprecise design is the use of ECN (Explicit Congestion Notification) with TCP. In this design, routers set bits on packets that experience congestion to warn the senders to slow down, but they do not tell them how much to slow down.

In other designs, there is no explicit signal. FAST TCP measures the round-trip delay and uses that metric as a signal to avoid congestion (Wei et al., 2006). Finally, in the form of congestion control most prevalent in the Internet today, TCP with drop-tail or RED routers, packet loss is inferred and used to signal that the network has become congested. There are many variants of this form of TCP, including TCP CUBIC, which is used in Linux (Ha et al., 2008). Combinations are also possible. For example, Windows includes Compound TCP that uses both packet loss and delay as feedback signals (Tan et al., 2006). These designs are summarized in Fig. 6-23.

Protocol	Signal	Explicit?	Precise?
XCP	Rate to use	Yes	Yes
TCP with ECN	Congestion warning	Yes	No
FAST TCP	End-to-end delay	No	Yes
Compound TCP	Packet loss & end-to-end delay	No	Yes
CUBIC TCP	Packet loss	No	No
TCP	Packet loss	No	No

Figure 6-23. Signals of some congestion control protocols.

If an explicit and precise signal is given, the transport entity can use that signal to adjust its rate to the new operating point. For example, if XCP tells senders the rate to use, the senders may simply use that rate. In the other cases, however, some guesswork is involved. In the absence of a congestion signal, the senders should increase their rates. When a congestion signal is given, the senders should decrease their rates. The way in which the rates are increased or decreased is given by a **control law**. These laws have a major effect on performance.

Chiu and Jain (1989) studied the case of binary congestion feedback and concluded that **AIMD** (**Additive Increase Multiplicative Decrease**) is the appropriate control law to arrive at the efficient and fair operating point. To argue this case, they constructed a graphical argument for the simple case of two connections competing for the bandwidth of a single link. The graph in Fig. 6-24 shows the bandwidth allocated to user 1 on the *x*-axis and to user 2 on the *y*-axis. When the allocation is completely fair, both users will receive the same amount of bandwidth. This is shown by the dotted fairness line. When the allocations sum to 100%, the capacity of the link, the allocation is efficient. This is shown by the dotted efficiency line. A congestion signal is given by the network to both users when

the sum of their allocations crosses this line. The intersection of these lines is the desired operating point, when both users have the same bandwidth and all of the network bandwidth is used.

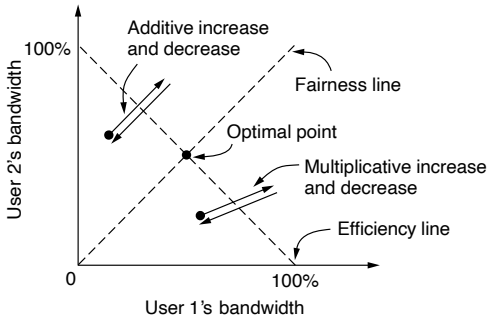


Figure 6-24. Additive and multiplicative bandwidth adjustments.

Consider what happens from some starting allocation if both user 1 and user 2 additively increase their respective bandwidths over time. For example, the users may each increase their sending rate by 1 Mbps every second. Eventually, the operating point crosses the efficiency line and both users receive a congestion signal from the network. At this stage, they must reduce their allocations. However, an additive decrease would simply cause them to oscillate along an additive line. This situation is shown in Fig. 6-24. The behavior will keep the operating point close to efficient, but it will not necessarily be fair.

Similarly, consider the case when both users multiplicatively increase their bandwidth over time until they receive a congestion signal. For example, the users may increase their sending rate by 10% every second. If they then multiplicatively decrease their sending rates, the operating point of the users will simply oscillate along a multiplicative line. This behavior is also shown in Fig. 6-24. The multiplicative line has a different slope than the additive line. (It points to the origin, while the additive line has an angle of 45 degrees.) But it is otherwise no better. In neither case will the users converge to the optimal sending rates that are both fair and efficient.

Now consider the case that the users additively increase their bandwidth allocations and then multiplicatively decrease them when congestion is signaled. This behavior is the AIMD control law, and it is shown in Fig. 6-25. It can be seen that the path traced by this behavior does converge to the optimal point that is both fair and efficient. This convergence happens no matter what the starting point, making AIMD broadly useful. By the same argument, the only other combination, multiplicative increase and additive decrease, would diverge from the optimal point.

AIMD is the control law that is used by TCP, based on this argument and another stability argument (that it is easy to drive the network into congestion and

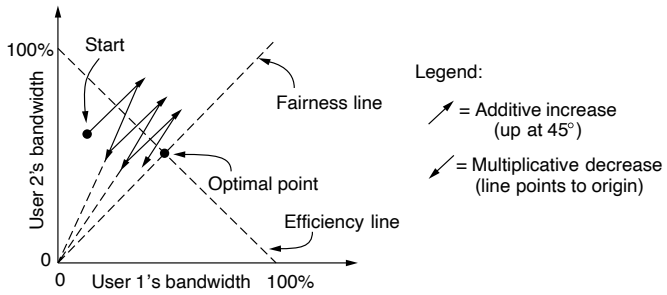


Figure 6-25. Additive Increase Multiplicative Decrease (AIMD) control law.

difficult to recover, so the increase policy should be gentle and the decrease policy aggressive). It is not quite fair, since TCP connections adjust their window size by a given amount every round-trip time. Different connections will have different round-trip times. This leads to a bias in which connections to closer hosts receive more bandwidth than connections to distant hosts, all else being equal.

In Sec. 6.5, we will describe in detail how TCP implements an AIMD control law to adjust the sending rate and provide congestion control. This task is more difficult than it sounds because rates are measured over some interval and traffic is bursty. Instead of adjusting the rate directly, a strategy that is often used in practice is to adjust the size of a sliding window. TCP uses this strategy. If the window size is W and the round-trip time is RTT, the equivalent rate is W/RTT. This strategy is easy to combine with flow control, which already uses a window, and has the advantage that the sender paces packets using acknowledgements and hence slows down in one RTT if it stops receiving reports that packets are leaving the network.

As a final issue, there may be many different transport protocols that send traffic into the network. What will happen if the different protocols compete with different control laws to avoid congestion? Unequal bandwidth allocations, that is what. Since TCP is the dominant form of congestion control in the Internet, there is significant community pressure for new transport protocols to be designed so that they compete fairly with it. The early streaming media protocols caused problems by excessively reducing TCP throughput because they did not compete fairly. This led to the notion of **TCP-friendly** congestion control in which TCP and non-TCP transport protocols can be freely mixed with no ill effects (Floyd et al., 2000).

6.3.3 Wireless Issues

Transport protocols such as TCP that implement congestion control should be independent of the underlying network and link layer technologies. That is a good theory, but in practice there are issues with wireless networks. The main issue is that packet loss is often used as a congestion signal, including by TCP as we have

just discussed. Wireless networks lose packets all the time due to transmission errors. They just are not as reliable as wired networks.

With the AIMD control law, high throughput requires very small levels of packet loss. Analyses by Padhye et al. (1998) show that the throughput goes up as the inverse square root of the packet loss rate. What this means in practice is that the loss rate for fast TCP connections is very small; 1% is a moderate loss rate, and by the time the loss rate reaches 10% the connection has effectively stopped working. However, for wireless networks such as 802.11 LANs, frame loss rates of at least 10% are common. This difference means that, absent protective measures, congestion control schemes that use packet loss as a signal will unnecessarily throttle connections that run over wireless links to very low rates.

To function well, the only packet losses that the congestion control algorithm should observe are losses due to insufficient bandwidth, not losses due to transmission errors. One solution to this problem is to mask the wireless losses by using retransmissions over the wireless link. For example, 802.11 uses a stop-and-wait protocol to deliver each frame, retrying transmissions multiple times if need be before reporting a packet loss to the higher layer. In the normal case, each packet is delivered despite transient transmission errors that are not visible to the higher layers.

Fig. 6-26 shows a path with a wired and wireless link for which the masking strategy is used. There are two aspects to note. First, the sender does not necessarily know that the path includes a wireless link, since all it sees is the wired link to which it is attached. Internet paths are heterogeneous and there is no general method for the sender to tell what kind of links comprise the path. This complicates the congestion control problem, as there is no easy way to use one protocol for wireless links and another protocol for wired links.

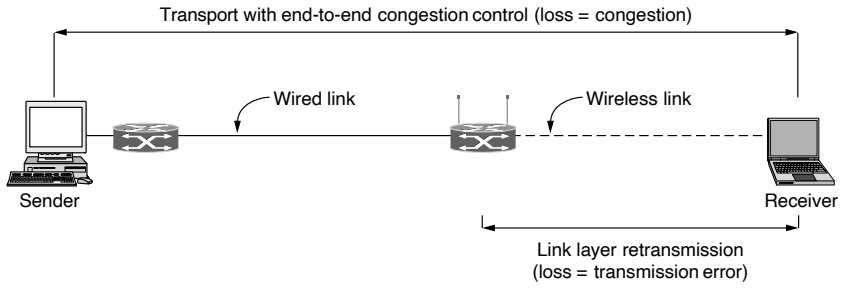


Figure 6-26. Congestion control over a path with a wireless link.

The second aspect is a puzzle. The figure shows two mechanisms that are driven by loss: link layer frame retransmissions, and transport layer congestion control. The puzzle is how these two mechanisms can co-exist without getting confused. After all, a loss should cause only one mechanism to take action because it is either a transmission error or a congestion signal. It cannot be both. If both

mechanisms take action (by retransmitting the frame and slowing down the sending rate) then we are back to the original problem of transports that run far too slowly over wireless links. Consider this puzzle for a moment and see if you can solve it.

The solution is that the two mechanisms act at different timescales. Link layer retransmissions happen on the order of microseconds to milliseconds for wireless links such as 802.11. Loss timers in transport protocols fire on the order of milliseconds to seconds. The difference is three orders of magnitude. This allows wireless links to detect frame losses and retransmit frames to repair transmission errors long before packet loss is inferred by the transport entity.

The masking strategy is sufficient to let most transport protocols run well across most wireless links. However, it is not always a fitting solution. Some wireless links have long round-trip times, such as satellites. For these links other techniques must be used to mask loss, such as FEC (Forward Error Correction), or the transport protocol must use a non-loss signal for congestion control.

A second issue with congestion control over wireless links is variable capacity. That is, the capacity of a wireless link changes over time, sometimes abruptly, as nodes move and the signal-to-noise ratio varies with the changing channel conditions. This is unlike wired links whose capacity is fixed. The transport protocol must adapt to the changing capacity of wireless links, otherwise it will either congest the network or fail to use the available capacity.

One possible solution to this problem is simply not to worry about it. This strategy is feasible because congestion control algorithms must already handle the case of new users entering the network or existing users changing their sending rates. Even though the capacity of wired links is fixed, the changing behavior of other users presents itself as variability in the bandwidth that is available to a given user. Thus it is possible to simply run TCP over a path with an 802.11 wireless link and obtain reasonable performance.

However, when there is much wireless variability, transport protocols designed for wired links may have trouble keeping up and deliver poor performance. The solution in this case is a transport protocol that is designed for wireless links. A particularly challenging setting is a wireless mesh network in which multiple, interfering wireless links must be crossed, routes change due to mobility, and there is lots of loss. Research in this area is ongoing. See Li et al. (2009) for an example of wireless transport protocol design.

6.4 THE INTERNET TRANSPORT PROTOCOLS: UDP

The Internet has two main protocols in the transport layer, a connectionless protocol and a connection-oriented one. The protocols complement each other. The connectionless protocol is UDP. It does almost nothing beyond sending packets between applications, letting applications build their own protocols on top as

needed. The connection-oriented protocol is TCP. It does almost everything. It makes connections and adds reliability with retransmissions, along with flow control and congestion control, all on behalf of the applications that use it.

In the following sections, we will study UDP and TCP. We will start with UDP because it is simplest. We will also look at two uses of UDP. Since UDP is a transport layer protocol that typically runs in the operating system and protocols that use UDP typically run in user space, these uses might be considered applications. However, the techniques they use are useful for many applications and are better considered to belong to a transport service, so we will cover them here.

6.4.1 Introduction to UDP

The Internet protocol suite supports a connectionless transport protocol called **UDP** (**User Datagram Protocol**). UDP provides a way for applications to send encapsulated IP datagrams without having to establish a connection. UDP is described in RFC 768.

UDP transmits **segments** consisting of an 8-byte header followed by the payload. The header is shown in Fig. 6-27. The two **ports** serve to identify the endpoints within the source and destination machines. When a UDP packet arrives, its payload is handed to the process attached to the destination port. This attachment occurs when the BIND primitive or something similar is used, as we saw in Fig. 6-6 for TCP (the binding process is the same for UDP). Think of ports as mailboxes that applications can rent to receive packets. We will have more to say about them when we describe TCP, which also uses ports. In fact, the main value of UDP over just using raw IP is the addition of the source and destination ports. Without the port fields, the transport layer would not know what to do with each incoming packet. With them, it delivers the embedded segment to the correct application.

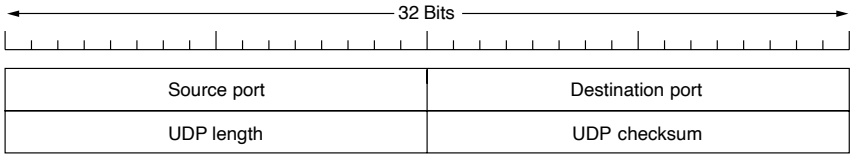


Figure 6-27. The UDP header.

The source port is primarily needed when a reply must be sent back to the source. By copying the *Source port* field from the incoming segment into the *Destination port* field of the outgoing segment, the process sending the reply can specify which process on the sending machine is to get it.

The *UDP length* field includes the 8-byte header and the data. The minimum length is 8 bytes, to cover the header. The maximum length is 65,515 bytes, which

is lower than the largest number that will fit in 16 bits because of the size limit on IP packets.

An optional *Checksum* is also provided for extra reliability. It checksums the header, the data, and a conceptual IP pseudoheader. When performing this computation, the *Checksum* field is set to zero and the data field is padded out with an additional zero byte if its length is an odd number. The checksum algorithm is simply to add up all the 16-bit words in one's complement and to take the one's complement of the sum. As a consequence, when the receiver performs the calculation on the entire segment, including the *Checksum* field, the result should be 0. If the checksum is not computed, it is stored as a 0, since by a happy coincidence of one's complement arithmetic a true computed 0 is stored as all 1s. However, turning it off is foolish unless the quality of the data does not matter (e.g., for digitized speech).

The pseudoheader for the case of IPv4 is shown in Fig. 6-28. It contains the 32-bit IPv4 addresses of the source and destination machines, the protocol number for UDP (17), and the byte count for the UDP segment (including the header). It is different but analogous for IPv6. Including the pseudoheader in the UDP checksum computation helps detect misdelivered packets, but including it also violates the protocol hierarchy since the IP addresses in it belong to the IP layer, not to the UDP layer. TCP uses the same pseudoheader for its checksum.

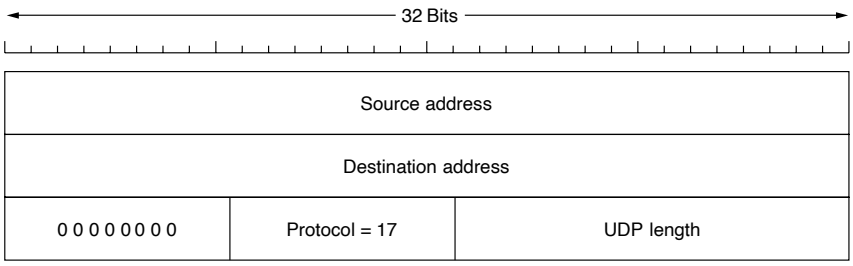


Figure 6-28. The IPv4 pseudoheader included in the UDP checksum.

It is probably worth mentioning explicitly some of the things that UDP does *not* do. It does not do flow control, congestion control, or retransmission upon receipt of a bad segment. All of that is up to the user processes. What it does do is provide an interface to the IP protocol with the added feature of demultiplexing multiple processes using the ports and optional end-to-end error detection. That is all it does.

For applications that need to have precise control over the packet flow, error control, or timing, UDP provides just what the doctor ordered. One area where it is especially useful is in client-server situations. Often, the client sends a short request to the server and expects a short reply back. If either the request or the reply

is lost, the client can just time out and try again. Not only is the code simple, but fewer messages are required (one in each direction) than with a protocol requiring an initial setup like TCP.

An application that uses UDP this way is DNS (Domain Name System), which we will study in Chap. 7. In brief, a program that needs to look up the IP address of some host name, for example, *www.cs.berkeley.edu*, can send a UDP packet containing the host name to a DNS server. The server replies with a UDP packet containing the host's IP address. No setup is needed in advance and no release is needed afterward. Just two messages go over the network.

6.4.2 Remote Procedure Call

In a certain sense, sending a message to a remote host and getting a reply back is a lot like making a function call in a programming language. In both cases, you start with one or more parameters and you get back a result. This observation has led people to try to arrange request-reply interactions on networks to be cast in the form of procedure calls. Such an arrangement makes network applications much easier to program and more familiar to deal with. For example, just imagine a procedure named *get_IP_address*(*host_name*) that works by sending a UDP packet to a DNS server and waiting for the reply, timing out and trying again if one is not forthcoming quickly enough. In this way, all the details of networking can be hidden from the programmer.

The key work in this area was done by Birrell and Nelson (1984). In a nutshell, what Birrell and Nelson suggested was allowing programs to call procedures located on remote hosts. When a process on machine 1 calls a procedure on machine 2, the calling process on 1 is suspended and execution of the called procedure takes place on 2. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing is visible to the application programmer. This technique is known as **RPC** (**Remote Procedure Call**) and has become the basis for many networking applications. Traditionally, the calling procedure is known as the client and the called procedure is known as the server, and we will use those names here too.

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In the simplest form, to call a remote procedure, the client program must be bound with a small library procedure, called the **client stub**, that represents the server procedure in the client's address space. Similarly, the server is bound with a procedure called the **server stub**. These procedures hide the fact that the procedure call from the client to the server is not local.

The actual steps in making an RPC are shown in Fig. 6-29. Step 1 is the client calling the client stub. This call is a local procedure call, with the parameters pushed onto the stack in the normal way. Step 2 is the client stub packing the parameters into a message and making a system call to send the message. Packing the parameters is called **marshaling**. Step 3 is the operating system sending the

message from the client machine to the server machine. Step 4 is the operating system passing the incoming packet to the server stub. Finally, step 5 is the server stub calling the server procedure with the unmarshaled parameters. The reply traces the same path in the other direction.

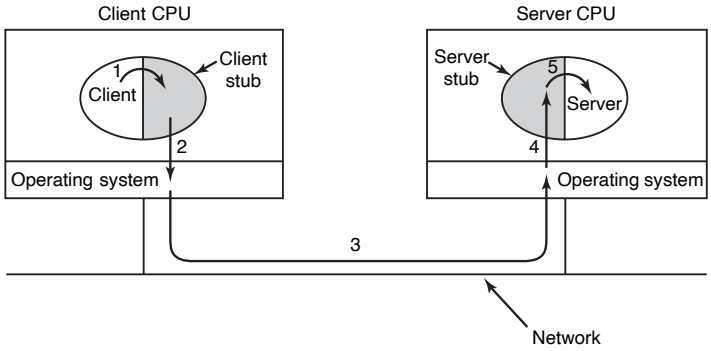


Figure 6-29. Steps in making a remote procedure call. The stubs are shaded.

The key item to note here is that the client procedure, written by the user, just makes a normal (i.e., local) procedure call to the client stub, which has the same name as the server procedure. Since the client procedure and client stub are in the same address space, the parameters are passed in the usual way. Similarly, the server procedure is called by a procedure in its address space with the parameters it expects. To the server procedure, nothing is unusual. In this way, instead of I/O being done on sockets, network communication is done by faking a normal procedure call.

Despite the conceptual elegance of RPC, there are a few snakes hiding under the grass. A big one is the use of pointer parameters. Normally, passing a pointer to a procedure is not a problem. The called procedure can use the pointer in the same way the caller can because both procedures live in the same virtual address space. With RPC, passing pointers is impossible because the client and server are in different address spaces.

In some cases, tricks can be used to make it possible to pass pointers. Suppose that the first parameter is a pointer to an integer, k. The client stub can marshal k and send it along to the server. The server stub then creates a pointer to k and passes it to the server procedure, just as it expects. When the server procedure returns control to the server stub, the latter sends k back to the client, where the new k is copied over the old one, just in case the server changed it. In effect, the standard calling sequence of call-by-reference has been replaced by call-by-copy-restore. Unfortunately, this trick does not always work, for example, if the pointer points to a graph or other complex data structure. For this reason, some restrictions must be placed on parameters to procedures called remotely, as we shall see.

A second problem is that in weakly typed languages, like C, it is perfectly legal to write a procedure that computes the inner product of two vectors (arrays), without specifying how large either one is. Each could be terminated by a special value known only to the calling and called procedures. Under these circumstances, it is essentially impossible for the client stub to marshal the parameters: it has no way of determining how large they are.

A third problem is that it is not always possible to deduce the types of the parameters, not even from a formal specification or the code itself. An example is *printf*, which may have any number of parameters (at least one), and the parameters can be an arbitrary mixture of integers, shorts, longs, characters, strings, floating-point numbers of various lengths, and other types. Trying to call *printf* as a remote procedure would be practically impossible because C is so permissive. However, a rule saying that RPC can be used provided that you do not program in C (or C++) would not be popular with a lot of programmers.

A fourth problem relates to the use of global variables. Normally, the calling and called procedure can communicate by using global variables (although it is not good practice), in addition to communicating via parameters. But if the called procedure is moved to a remote machine, the code will fail because the global variables are no longer shared.

These problems are not meant to suggest that RPC is hopeless. In fact, it is widely used, but some restrictions are needed to make it work well in practice.

In terms of transport layer protocols, UDP is a good base on which to implement RPC. Both requests and replies may be sent as a single UDP packet in the simplest case and the operation can be fast. However, an implementation must include other machinery as well. Because the request or the reply may be lost, the client must keep a timer to retransmit the request. Note that a reply serves as an implicit acknowledgement for a request, so the request need not be separately acknowledged. Sometimes the parameters or results may be larger than the maximum UDP packet size, in which case some protocol is needed to deliver large messages in pieces and reassemble them correctly. If multiple requests and replies can overlap (as in the case of concurrent programming), an identifier is needed to match the request with the reply.

A higher-level concern is that the operation may not be idempotent (i.e., safe to repeat). The simple case is idempotent operations such as DNS requests and replies. The client can safely retransmit these requests again and again if no replies are forthcoming. It does not matter whether the server never received the request, or it was the reply that was lost. The answer, when it finally arrives, will be the same (assuming the DNS database is not updated in the meantime). However, not all operations are idempotent, for example, because they have important side effects such as incrementing a counter. RPC for these operations requires stronger semantics so that when the programmer calls a procedure it is not executed multiple times. In this case, it may be necessary to set up a TCP connection and send the request over it rather than using UDP.

6.4.3 Real-Time Transport Protocols

Client-server RPC is one area in which UDP is widely used. Another one is for real-time multimedia applications. In particular, as Internet radio, Internet telephony, music-on-demand, videoconferencing, video-on-demand, and other multimedia applications became more commonplace, people have discovered that each application was reinventing more or less the same real-time transport protocol. It gradually became clear that having a generic real-time transport protocol for multiple applications would be a good idea.

Thus was **RTP** (**Real-time Transport Protocol**) born. It is described in RFC 3550 and is now in widespread use for multimedia applications. We will describe two aspects of real-time transport. The first is the RTP protocol for transporting audio and video data in packets. The second is the processing that takes place, mostly at the receiver, to play out the audio and video at the right time. These functions fit into the protocol stack as shown in Fig. 6-30.

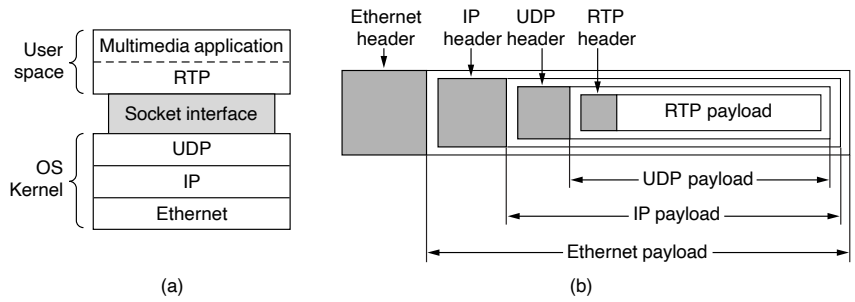


Figure 6-30. (a) The position of RTP in the protocol stack. (b) Packet nesting.

RTP normally runs in user space over UDP (in the operating system). It operates as follows. The multimedia application consists of multiple audio, video, text, and possibly other streams. These are fed into the RTP library, which is in user space along with the application. This library multiplexes the streams and encodes them in RTP packets, which it stuffs into a socket. On the operating system side of the socket, UDP packets are generated to wrap the RTP packets and handed to IP for transmission over a link such as Ethernet. The reverse process happens at the receiver. The multimedia application eventually receives multimedia data from the RTP library. It is responsible for playing out the media. The protocol stack for this situation is shown in Fig. 6-30(a). The packet nesting is shown in Fig. 6-30(b).

As a consequence of this design, it is a little hard to say which layer RTP is in. Since it runs in user space and is linked to the application program, it certainly looks like an application protocol. On the other hand, it is a generic, application-independent protocol that just provides transport facilities, so it also looks like

a transport protocol. Probably the best description is that it is a transport protocol that just happens to be implemented in the application layer, which is why we are covering it in this chapter.

RTP—The Real-time Transport Protocol

The basic function of RTP is to multiplex several real-time data streams onto a single stream of UDP packets. The UDP stream can be sent to a single destination (unicasting) or to multiple destinations (multicasting). Because RTP just uses normal UDP, its packets are not treated specially by the routers unless some normal IP quality-of-service features are enabled. In particular, there are no special guarantees about delivery, and packets may be lost, delayed, corrupted, etc.

The RTP format contains several features to help receivers work with multimedia information. Each packet sent in an RTP stream is given a number one higher than its predecessor. This numbering allows the destination to determine if any packets are missing. If a packet is missing, the best action for the destination to take is up to the application. It may be to skip a video frame if the packets are carrying video data, or to approximate the missing value by interpolation if the packets are carrying audio data. Retransmission is not a practical option since the retransmitted packet would probably arrive too late to be useful. As a consequence, RTP has no acknowledgements, and no mechanism to request retransmissions.

Each RTP payload may contain multiple samples, and they may be coded any way that the application wants. To allow for interworking, RTP defines several profiles (e.g., a single audio stream), and for each profile, multiple encoding formats may be allowed. For example, a single audio stream may be encoded as 8-bit PCM samples at 8 kHz using delta encoding, predictive encoding, GSM encoding, MP3 encoding, and so on. RTP provides a header field in which the source can specify the encoding but is otherwise not involved in how encoding is done.

Another facility many real-time applications need is timestamping. The idea here is to allow the source to associate a timestamp with the first sample in each packet. The timestamps are relative to the start of the stream, so only the differences between timestamps are significant. The absolute values have no meaning. As we will describe shortly, this mechanism allows the destination to do a small amount of buffering and play each sample the right number of milliseconds after the start of the stream, independently of when the packet containing the sample arrived.

Not only does timestamping reduce the effects of variation in network delay, but it also allows multiple streams to be synchronized with each other. For example, a digital television program might have a video stream and two audio streams. The two audio streams could be for stereo broadcasts or for handling films with an original language soundtrack and a soundtrack dubbed into the local language, giving the viewer a choice. Each stream comes from a different physical

device, but if they are timestamped from a single counter, they can be played back synchronously, even if the streams are transmitted and/or received somewhat erratically.

The RTP header is illustrated in Fig. 6-31. It consists of three 32-bit words and potentially some extensions. The first word contains the *Version* field, which is already at 2. Let us hope this version is very close to the ultimate version since there is only one code point left (although 3 could be defined as meaning that the real version was in an extension word).

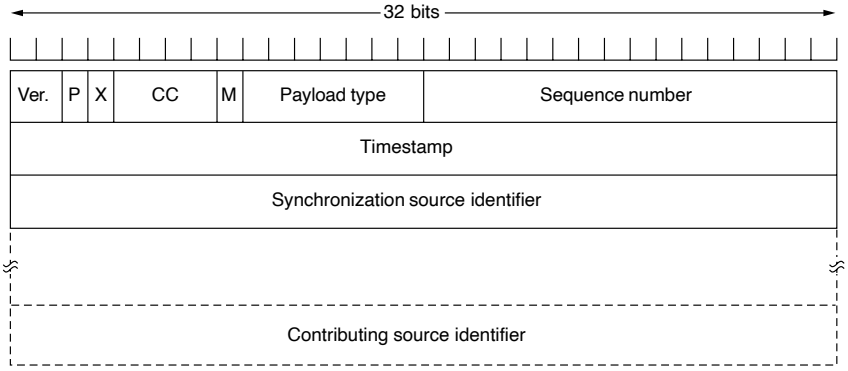


Figure 6-31. The RTP header.

The *P* bit indicates that the packet has been padded to a multiple of 4 bytes. The last padding byte tells how many bytes were added. The *X* bit indicates that an extension header is present. The format and meaning of the extension header are not defined. The only thing that is defined is that the first word of the extension gives the length. This is an escape hatch for any unforeseen requirements.

The *CC* field tells how many contributing sources are present, from 0 to 15 (see below). The *M* bit is an application-specific marker bit. It can be used to mark the start of a video frame, the start of a word in an audio channel, or something else that the application understands. The *Payload type* field tells which encoding algorithm has been used (e.g., uncompressed 8-bit audio, MP3, etc.). Since every packet carries this field, the encoding can change during transmission. The *Sequence number* is just a counter that is incremented on each RTP packet sent. It is used to detect lost packets.

The *Timestamp* is produced by the stream's source to note when the first sample in the packet was made. This value can help reduce timing variability which is called **jitter**, at the receiver by decoupling the playback from the packet arrival time. The *Synchronization source identifier* tells which stream the packet belongs to. It is the method used to multiplex and demultiplex multiple data streams onto a

single stream of UDP packets. Finally, the *Contributing source identifiers*, if any, are used when mixers are present in the studio. In that case, the mixer is the synchronizing source, and the streams being mixed are listed here.

RTCP—The Real-time Transport Control Protocol

RTP has a little sister protocol (little sibling protocol?) called **RTCP** (**Real-time Transport Control Protocol**). It is defined along with RTP in RFC 3550 and handles feedback, synchronization, and the user interface. It does not transport any media samples.

The first function can be used to provide feedback on delay, variation in delay or jitter, bandwidth, congestion, and other network properties to the sources. This information can be used by the encoding process to increase the data rate (and give better quality) when the network is functioning well and to cut back the data rate when there is trouble in the network. By providing continuous feedback, the encoding algorithms can be continuously adapted to provide the best quality possible under the current circumstances. For example, if the bandwidth increases or decreases during the transmission, the encoding may switch from MP3 to 8-bit PCM to delta encoding as required. The *Payload type* field is used to tell the destination what encoding algorithm is used for the current packet, making it possible to vary it on demand.

An issue with providing feedback is that the RTCP reports are sent to all participants. For a multicast application with a large group, the bandwidth used by RTCP would quickly grow large. To prevent this from happening, RTCP senders scale down the rate of their reports to collectively consume no more than, say, 5% of the media bandwidth. To do this, each participant needs to know the media bandwidth, which it learns from the sender, and the number of participants, which it estimates by listening to other RTCP reports.

RTCP also handles interstream synchronization. The problem is that different streams may use different clocks, with different granularities and different drift rates. RTCP can be used to keep them in sync.

Finally, RTCP provides a way for naming the various sources (e.g., in ASCII text). This information can be displayed on the receiver's screen to indicate who is talking at the moment.

More information about RTP can be found in Perkins (2003).

Playout with Buffering and Jitter Control

Once the media information reaches the receiver, it must be played out at the right time. In general, this will not be the time at which the RTP packet arrived at the receiver because packets will take slightly different amounts of time to transit the network. Even if the packets are injected with exactly the right intervals between them at the sender, they will reach the receiver with different relative times.

Even a small amount of packet jitter can cause distracting media artifacts, such as jerky video frames and unintelligible audio, if the media is simply played out as it arrives.

The solution to this problem is to **buffer** packets at the receiver before they are played out to reduce the jitter. As an example, in Fig. 6-32 we see a stream of packets being delivered with a substantial amount of jitter. Packet 1 is sent from the server at $t = 0$ sec and arrives at the client at $t = 1$ sec. Packet 2 undergoes more delay and takes 2 sec to arrive. As the packets arrive, they are buffered on the client machine.

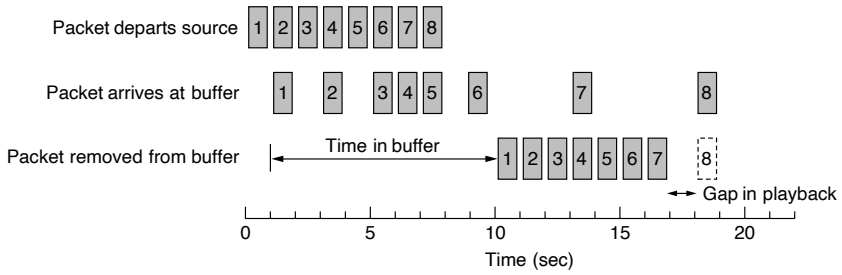


Figure 6-32. Smoothing the output stream by buffering packets.

At $t = 10$ sec, playback begins. At this time, packets 1 through 6 have been buffered so that they can be removed from the buffer at uniform intervals for smooth play. In the general case, it is not necessary to use uniform intervals because the RTP timestamps tell when the media should be played.

Unfortunately, we can see that packet 8 has been delayed so much that it is not available when its play slot comes up. There are two options. Packet 8 can be skipped and the player can move on to subsequent packets. Alternatively, playback can stop until packet 8 arrives, creating an annoying gap in the music or movie. In a live media application like a voice-over-IP call, the packet will typically be skipped. Live applications do not work well on hold. In a streaming media application, the player might pause. This problem can be alleviated by delaying the starting time even more, by using a larger buffer. For a streaming audio or video player, buffers of about 10 seconds are often used to ensure that the player receives all of the packets (that are not dropped in the network) in time. For live applications like videoconferencing, short buffers are needed for responsiveness.

A key consideration for smooth playout is the **playback point**, or how long to wait at the receiver for media before playing it out. Deciding how long to wait depends on the jitter. The difference between a low-jitter and high-jitter connection is shown in Fig. 6-33. The average delay may not differ greatly between the two, but if there is high jitter the playback point may need to be much further out to capture 99% of the packets than if there is low jitter.

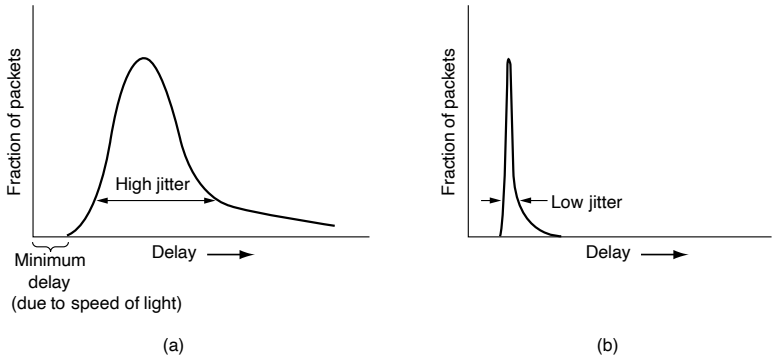


Figure 6-33. (a) High jitter. (b) Low jitter.

To pick a good playback point, the application can measure the jitter by looking at the difference between the RTP timestamps and the arrival time. Each difference gives a sample of the delay (plus an arbitrary, fixed offset). However, the delay can change over time due to other, competing traffic and changing routes. To accommodate this change, applications can adapt their playback point while they are running. However, if not done well, changing the playback point can produce an observable glitch to the user. One way to avoid this problem for audio is to adapt the playback point between **talkspurts**, in the gaps in a conversation. No one will notice the difference between a short and slightly longer silence. RTP lets applications set the *M* marker bit to indicate the start of a new talkspurt for this purpose.

If the absolute delay until media is played out is too long, live applications will suffer. Nothing can be done to reduce the propagation delay if a direct path is already being used. The playback point can be pulled in by simply accepting that a larger fraction of packets will arrive too late to be played. If this is not acceptable, the only way to pull in the playback point is to reduce the jitter by using a better quality of service, for example, the expedited forwarding differentiated service. That is, a better network is needed.

6.5 THE INTERNET TRANSPORT PROTOCOLS: TCP

UDP is a simple protocol and it has some very important uses, such as client-server interactions and multimedia, but for most Internet applications, reliable, sequenced delivery is needed, so UDP will not do. UDP cannot provide this, so another protocol is required. It is called TCP and is the main workhorse of the Internet. Let us now study it in detail.

6.5.1 Introduction to TCP

TCP (**Transmission Control Protocol**) was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork. An internetwork differs from a single network because different parts may have wildly different topologies, bandwidths, delays, packet sizes, and other parameters. TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures.

TCP was formally defined in RFC 793 in September 1981. As time went on, many improvements have been made, and various errors and inconsistencies have been fixed. To give you a sense of the extent of TCP, the important RFCs are now RFC 793 plus: clarifications and bug fixes in RFC 1122; extensions for high-performance in RFC 1323; selective acknowledgements in RFC 2018; congestion control in RFC 2581; repurposing of header fields for quality of service in RFC 2873; improved retransmission timers in RFC 2988; and explicit congestion notification in RFC 3168. The full collection is even larger, which led to a guide to the many RFCs, published of course as another RFC document, RFC 4614.

Each machine supporting TCP has a TCP transport entity, either a library procedure, a user process, or most commonly part of the kernel. In all cases, it manages TCP streams and interfaces to the IP layer. A TCP entity accepts user data streams from local processes, breaks them up into pieces not exceeding 64 KB (in practice, often 1460 data bytes in order to fit in a single Ethernet frame with the IP and TCP headers), and sends each piece as a separate IP datagram. When datagrams containing TCP data arrive at a machine, they are given to the TCP entity, which reconstructs the original byte streams. For simplicity, we will sometimes use just ''TCP'' to mean the TCP transport entity (a piece of software) or the TCP protocol (a set of rules). From the context it will be clear which is meant. For example, in ''The user gives TCP the data,'' the TCP transport entity is clearly intended.

The IP layer gives no guarantee that datagrams will be delivered properly, nor any indication of how fast datagrams may be sent. It is up to TCP to send datagrams fast enough to make use of the capacity but not cause congestion, and to time out and retransmit any datagrams that are not delivered. Datagrams that do arrive may well do so in the wrong order; it is also up to TCP to reassemble them into messages in the proper sequence. In short, TCP must furnish good performance with the reliability that most applications want and that IP does not provide.

6.5.2 The TCP Service Model

TCP service is obtained by both the sender and the receiver creating end points, called **sockets**, as discussed in Sec. 6.1.3. Each socket has a socket number (address) consisting of the IP address of the host and a 16-bit number local to that

host, called a **port**. A port is the TCP name for a TSAP. For TCP service to be obtained, a connection must be explicitly established between a socket on one machine and a socket on another machine. The socket calls are listed in Fig. 6-5.

A socket may be used for multiple connections at the same time. In other words, two or more connections may terminate at the same socket. Connections are identified by the socket identifiers at both ends, that is, (*socket1*, *socket2*). No virtual circuit numbers or other identifiers are used.

Port numbers below 1024 are reserved for standard services that can usually only be started by privileged users (e.g., root in UNIX systems). They are called **well-known ports**. For example, any process wishing to remotely retrieve mail from a host can connect to the destination host's port 143 to contact its IMAP daemon. The list of well-known ports is given at *www.iana.org*. Over 700 have been assigned. A few of the better-known ones are listed in Fig. 6-34.

Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

Figure 6-34. Some assigned ports.

Other ports from 1024 through 49151 can be registered with IANA for use by unprivileged users, but applications can and do choose their own ports. For example, the BitTorrent peer-to-peer file-sharing application (unofficially) uses ports 6881–6887, but may run on other ports as well.

It would certainly be possible to have the FTP daemon attach itself to port 21 at boot time, the SSH daemon attach itself to port 22 at boot time, and so on. However, doing so would clutter up memory with daemons that were idle most of the time. Instead, what is commonly done is to have a single daemon, called **inetd** (**Internet daemon**) in UNIX, attach itself to multiple ports and wait for the first incoming connection. When that occurs, *inetd* forks off a new process and executes the appropriate daemon in it, letting that daemon handle the request. In this way, the daemons other than *inetd* are only active when there is work for them to do. Inetd learns which ports it is to use from a configuration file. Consequently, the system administrator can set up the system to have permanent daemons on the busiest ports (e.g., port 80) and *inetd* on the rest.

All TCP connections are full duplex and point-to-point. Full duplex means that traffic can go in both directions at the same time. Point-to-point means that each connection has exactly two end points. TCP does not support multicasting or broadcasting.

A TCP connection is a byte stream, not a message stream. Message boundaries are not preserved end to end. For example, if the sending process does four 512-byte writes to a TCP stream, these data may be delivered to the receiving process as four 512-byte chunks, two 1024-byte chunks, one 2048-byte chunk (see Fig. 6-35), or some other way. There is no way for the receiver to detect the unit(s) in which the data were written, no matter how hard it tries.

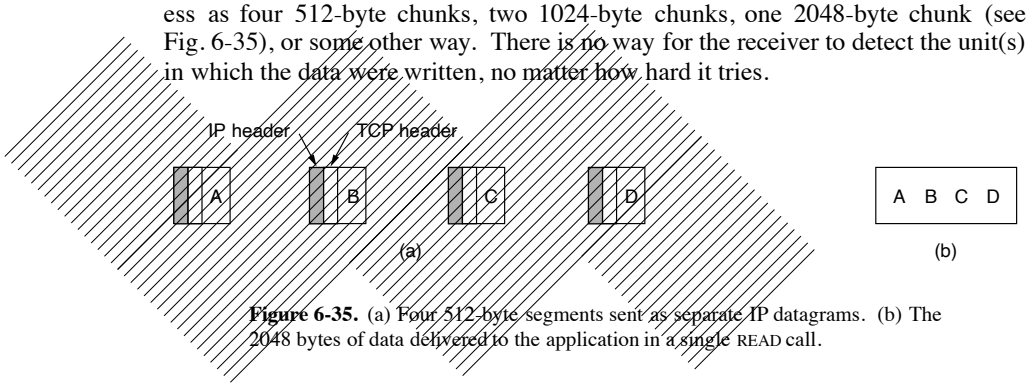


Figure 6-35. (a) Four 512-byte segments sent as separate IP datagrams. (b) The 2048 bytes of data delivered to the application in a single READ call.

Files in UNIX have this property too. The reader of a file cannot tell whether the file was written a block at a time, a byte at a time, or all in one blow. As with a UNIX file, the TCP software has no idea of what the bytes mean and no interest in finding out. A byte is just a byte.

When an application passes data to TCP, TCP may send it immediately or buffer it (in order to collect a larger amount to send at once), at its discretion. However, sometimes the application really wants the data to be sent immediately. For example, suppose a user of an interactive game wants to send a stream of updates. It is essential that the updates be sent immediately, not buffered until there is a collection of them. To force data out, TCP has the notion of a PUSH flag that is carried on packets. The original intent was to let applications tell TCP implementations via the PUSH flag not to delay the transmission. However, applications cannot literally set the PUSH flag themselves when they send data. Instead, different operating systems have evolved different options to expedite transmission (e.g., TCP_NODELAY in Windows and Linux).

For Internet archaeologists, we will also mention one interesting feature of TCP service that remains in the protocol but is rarely used: **urgent data**. When an application has high-priority data that should be processed immediately, for example, if an interactive user hits the CTRL-C key to break off a remote computation that has already begun, the sending application can put some control information in the data stream and give it to TCP along with the URGENT flag. This event causes TCP to stop accumulating data and transmit everything it has for that connection immediately, with no delay.

When the urgent data are received at the destination, the receiving application is interrupted (e.g., given a signal in UNIX terms) so it can stop whatever it was doing and read the data stream to find the urgent data. The end of the urgent data is marked so the application knows when it is over. The start of the urgent data is not marked. It is up to the application to figure that out.

This scheme provides a crude signaling mechanism and leaves everything else up to the application. However, while urgent data is potentially useful, it found no compelling application early on and fell into disuse. Its use is now discouraged because of implementation differences, leaving applications to handle their own signaling. Perhaps future transport protocols will provide better signaling.

6.5.3 The TCP Protocol

In this section, we will give a general overview of the TCP protocol. In the next one, we will go over the protocol header, field by field.

A key feature of TCP, and one that dominates the protocol design, is that every byte on a TCP connection has its own 32-bit sequence number. When the Internet began, the lines between routers were mostly 56-kbps leased lines, so a host blasting away at full speed took over 1 week to cycle through the sequence numbers. At modern network speeds, the sequence numbers can be consumed at an alarming rate, as we will see later. Separate 32-bit sequence numbers are carried on packets for the sliding window position in one direction and for acknowledgements in the reverse direction, as discussed below.

The sending and receiving TCP entities exchange data in the form of segments. A **TCP segment** consists of a fixed 20-byte header (plus an optional part) followed by zero or more data bytes. The TCP software decides how big segments should be. It can accumulate data from several writes into one segment or can split data from one write over multiple segments. Two limits restrict the segment size. First, each segment, including the TCP header, must fit in the 65,515-byte IP payload. Second, each link has an **MTU** (**Maximum Transfer Unit**). Each segment must fit in the MTU at the sender and receiver so that it can be sent and received in a single, unfragmented packet. In practice, the MTU is generally 1500 bytes (the Ethernet payload size) and thus defines the upper bound on segment size.

However, it is still possible for IP packets carrying TCP segments to be fragmented when passing over a network path for which some link has a small MTU. If this happens, it degrades performance and causes other problems (Kent and Mogul, 1987). Instead, modern TCP implementations perform **path MTU discovery** by using the technique outlined in RFC 1191 . We describedit in Sec. 5.5.6. This technique uses ICMP error messages to find the smallest MTU for any link on the path. TCP then adjusts the segment size downwards to avoid fragmentation.

The basic protocol used by TCP entities is the sliding window protocol with a dynamic window size. When a sender transmits a segment, it also starts a timer. When the segment arrives at the destination, the receiving TCP entity sends back a

segment (with data if any exist, and otherwise without) bearing an acknowledgement number equal to the next sequence number it expects to receive and the remaining window size. If the sender's timer goes off before the acknowledgement is received, the sender transmits the segment again.

Although this protocol sounds simple, there are many sometimes subtle ins and outs, which we will cover below. Segments can arrive out of order, so bytes 3072–4095 can arrive but cannot be acknowledged because bytes 2048–3071 have not turned up yet. Segments can also be delayed so long in transit that the sender times out and retransmits them. The retransmissions may include different byte ranges than the original transmission, requiring careful administration to keep track of which bytes have been correctly received so far. However, since each byte in the stream has its own unique offset, it can be done.

TCP must be prepared to deal with these problems and solve them in an efficient way. A considerable amount of effort has gone into optimizing the performance of TCP streams, even in the face of network problems. A number of the algorithms used by many TCP implementations will be discussed below.

6.5.4 The TCP Segment Header

Figure 6-36 shows the layout of a TCP segment. Every segment begins with a fixed-format, 20-byte header. The fixed header may be followed by header options. After the options, if any, up to 65,535 – 20 – 20 = 65,495 data bytes may follow, where the first 20 refer to the IP header and the second to the TCP header. Segments without any data are legal and are commonly used for acknowledgements and control messages.

Let us dissect the TCP header field by field. The *Source port* and *Destination port* fields identify the local end points of the connection. A TCP port plus its host's IP address forms a 48-bit unique end point. The source and destination end points together identify the connection. This connection identifier is called a **5 tuple** because it consists of five pieces of information: the protocol (TCP), source IP and source port, and destination IP and destination port.

The *Sequence number* and *Acknowledgement number* fields perform their usual functions. Note that the latter specifies the next in-order byte expected, not the last byte correctly received. It is a **cumulative acknowledgement** because it summarizes the received data with a single number. It does not go beyond lost data. Both are 32 bits because every byte of data is numbered in a TCP stream.

The *TCP header length* tells how many 32-bit words are contained in the TCP header. This information is needed because the *Options* field is of variable length, so the header is, too. Technically, this field really indicates the start of the data within the segment, measured in 32-bit words, but that number is just the header length in words, so the effect is the same.

Next comes a 4-bit field that is not used. The fact that these bits have remained unused for 30 years (as only 2 of the original reserved 6 bits have been

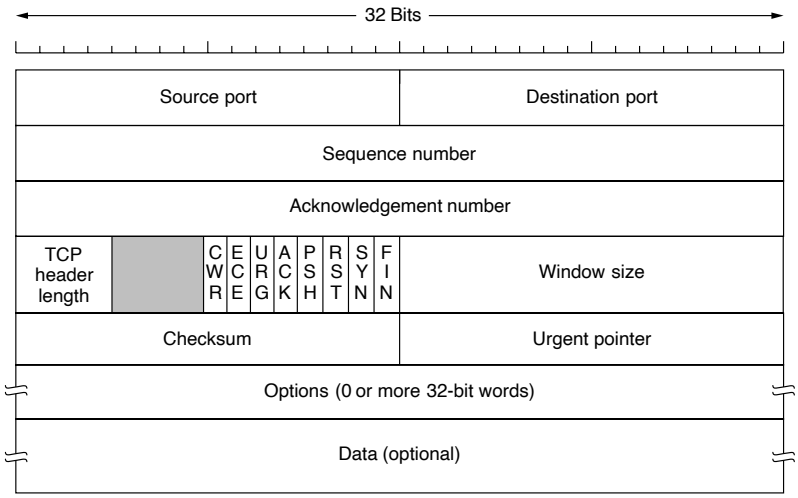


Figure 6-36. The TCP header.

reclaimed) is testimony to how well thought out TCP is. Lesser protocols would have needed these bits to fix bugs in the original design.

Now come eight 1-bit flags. *CWR* and *ECE* are used to signal congestion when ECN (Explicit Congestion Notification) is used, as specified in RFC 3168. *ECE* is set to signal an *ECN-Echo* to a TCP sender to tell it to slow down when the TCP receiver gets a congestion indication from the network. *CWR* is set to signal *Congestion Window Reduced* from the TCP sender to the TCP receiver so that it knows the sender has slowed down and can stop sending the *ECN-Echo*. We discuss the role of ECN in TCP congestion control in Sec. 6.5.10.

URG is set to 1 if the *Urgent pointer* is in use. The *Urgent pointer* is used to indicate a byte offset from the current sequence number at which urgent data are to be found. This facility is in lieu of interrupt messages. As we mentioned above, this facility is a bare-bones way of allowing the sender to signal the receiver without getting TCP itself involved in the reason for the interrupt, but it is seldom used.

The *ACK* bit is set to 1 to indicate that the *Acknowledgement number* is valid. This is the case for nearly all packets. If *ACK* is 0, the segment does not contain an acknowledgement, so the *Acknowledgement number* field is ignored.

The *PSH* bit indicates PUSHed data. The receiver is hereby kindly requested to deliver the data to the application upon arrival and not buffer it until a full buffer has been received (which it might otherwise do for efficiency).

The *RST* bit is used to abruptly reset a connection that has become confused due to a host crash or for some other reason. It is also used to reject an invalid

segment or refuse an attempt to open a connection. In general, if you get a segment with the *RST* bit on, you have a problem on your hands.

The *SYN* bit is used to establish connections. The connection request has *SYN* = 1 and *ACK* = 0 to indicate that the piggyback acknowledgement field is not in use. The connection reply does bear an acknowledgement, however, so it has *SYN* = 1 and *ACK* = 1. In essence, the *SYN* bit is used to denote both CONNECTION REQUEST and CONNECTION ACCEPTED, with the *ACK* bit used to distinguish between those two possibilities.

The *FIN* bit is used to release a connection. It specifies that the sender has no more data to *transmit*. However, after closing a connection, the closing process may continue to *receive* data indefinitely. Both *SYN* and *FIN* segments have sequence numbers and are thus guaranteed to be processed in the correct order.

Flow control in TCP is handled using a variable-sized sliding window. The *Window size* field tells how many bytes may be sent starting at the byte acknowledged. A *Window size* field of 0 is legal and says that the bytes up to and including *Acknowledgement number* – 1 have been received, but that the receiver has not had a chance to consume the data and would like no more data for the moment, thank you. The receiver can later grant permission to send by transmitting a segment with the same *Acknowledgement number* and a nonzero *Window size* field.

In the protocols of Chap. 3, acknowledgements of frames received and permission to send new frames were tied together. This was a consequence of a fixed window size for each protocol. In TCP, acknowledgements and permission to send additional data are completely decoupled. In effect, a receiver can say: ''I have received bytes up through *k* but I do not want any more just now, thank you.'' This decoupling (in fact, a variable-sized window) gives additional flexibility. We will study it in detail below.

A *Checksum* is also provided for extra reliability. It checksums the header, the data, and a conceptual pseudoheader in exactly the same way as UDP, except that the pseudoheader has the protocol number for TCP (6) and the checksum is mandatory. Please see Sec. 6.4.1 for details.

The *Options* field provides a way to add extra facilities not covered by the regular header. Many options have been defined and several are commonly used. The options are of variable length, fill a multiple of 32 bits by using padding with zeros, and may extend to 40 bytes to accommodate the longest TCP header that can be specified. Some options are carried when a connection is established to negotiate or inform the other side of capabilities. Other options are carried on packets during the lifetime of the connection. Each option has a Type-Length-Value encoding.

A widely used option is the one that allows each host to specify the **MSS** (**Maximum Segment Size**) it is willing to accept. Using large segments is more efficient than using small ones because the 20-byte header can be amortized over more data, but small hosts may not be able to handle big segments. During connection setup, each side can announce its maximum and see its partner's. If a host

does not use this option, it defaults to a 536-byte payload. All Internet hosts are required to accept TCP segments of 536 + 20 = 556 bytes. The maximum segment size in the two directions need not be the same.

For lines with high bandwidth, high delay, or both, the 64-KB window corresponding to a 16-bit field is a problem. For example, on an OC-12 line (of roughly 600 Mbps), it takes less than 1 msec to output a full 64-KB window. If the round-trip propagation delay is 50 msec (which is typical for a transcontinental fiber), the sender will be idle more than 98% of the time waiting for acknowledgements. A larger window size would allow the sender to keep pumping data out. The **window scale** option allows the sender and receiver to negotiate a window scale factor at the start of a connection. Both sides use the scale factor to shift the *Window size* field up to 14 bits to the left, thus allowing windows of up to 2^{30} bytes. Most TCP implementations support this option.

The **timestamp** option carries a timestamp sent by the sender and echoed by the receiver. It is included in every packet, once its use is established during connection setup, and used to compute round-trip time samples that are used to estimate when a packet has been lost. It is also used as a logical extension of the 32-bit sequence number. On a fast connection, the sequence number may wrap around quickly, leading to possible confusion between old and new data. The PAWS scheme described earlier discards arriving segments with old timestamps to prevent this problem.

Finally, the **SACK** (**Selective ACKnowledgement**) option lets a receiver tell a sender the ranges of sequence numbers that it has received. It supplements the *Acknowledgement number* and is used after a packet has been lost but subsequent (or duplicate) data has arrived. The new data is not reflected by the *Acknowledgement number* field in the header because that field gives only the next in-order byte that is expected. With SACK, the sender is explicitly aware of what data the receiver has and hence can determine what data should be retransmitted. SACK is defined in RFC 2108 and RFC 2883 and is increasingly used. We describe the use of SACK along with congestion control in Sec. 6.5.10.

6.5.5 TCP Connection Establishment

Connections are established in TCP by means of the three-way handshake discussed in Sec. 6.2.2. To establish a connection, one side, say, the server, passively waits for an incoming connection by executing the LISTEN and ACCEPT primitives in that order, either specifying a specific source or nobody in particular.

The other side, say, the client, executes a CONNECT primitive, specifying the IP address and port to which it wants to connect, the maximum TCP segment size it is willing to accept, and optionally some user data (e.g., a password). The CONNECT primitive sends a TCP segment with the *SYN* bit on and *ACK* bit off and waits for a response from the other end.

When this segment arrives at the destination, the TCP entity there checks to see if there is a process that has done a LISTEN on the port given in the *Destination port* field. If not, it sends a reply with the *RST* bit on to reject the connection.

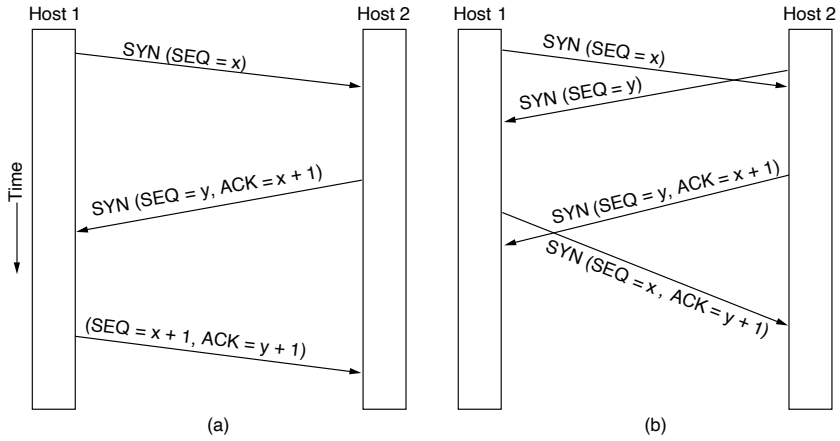


Figure 6-37. (a) TCP connection establishment in the normal case. (b) Simultaneous connection establishment on both sides.

If some process is listening to the port, that process is given the incoming TCP segment. It can either accept or reject the connection. If it accepts, an acknowledgement segment is sent back. The sequence of TCP segments sent in the normal case is shown in Fig. 6-37(a). Note that a *SYN* segment consumes 1 byte of sequence space so that it can be acknowledged unambiguously.

In the event that two hosts simultaneously attempt to establish a connection between the same two sockets, the sequence of events is as illustrated in Fig. 6-37(b). The result of these events is that just one connection is established, not two, because connections are identified by their end points. If the first setup results in a connection identified by (x, y) and the second one does too, only one table entry is made, namely, for (x, y).

Recall that the initial sequence number chosen by each host should cycle slowly, rather than be a constant such as 0. This rule is to protect against delayed duplicate packets, as we discussed in Sec 6.2.2. Originally, this was accomplished with a clock-based scheme in which the clock ticked every 4 μsec.

However, a vulnerability with implementing the three-way handshake is that the listening process must remember its sequence number as soon it responds with its own *SYN* segment. This means that a malicious sender can tie up resources on a host by sending a stream of *SYN* segments and never following through to complete the connection. This attack is called a **SYN flood**, and it crippled many Web servers in the 1990s. Now ways are known for defending against this attack.

One way to defend against this attack is to use **SYN cookies**. Instead of remembering the sequence number, a host chooses a cryptographically generated sequence number, puts it on the outgoing segment, and forgets it. If the three-way handshake completes, this sequence number (plus 1) will be returned to the host. It can then regenerate the correct sequence number by running the same cryptographic function, as long as the inputs to that function are known, for example, the other host's IP address and port, and a local secret. This procedure allows the host to check that an acknowledged sequence number is correct without having to remember the sequence number separately. There are some caveats, such as the inability to handle TCP options, so SYN cookies may be used only when the host is subject to a SYN flood. However, they are an interesting twist on connection establishment. For more information, see RFC 4987 and Lemon (2002).

6.5.6 TCP Connection Release

Although TCP connections are full duplex, to understand how connections are released it is best to think of them as a pair of simplex connections. Each simplex connection is released independently of its sibling. To release a connection, either party can send a TCP segment with the *FIN* bit set, which means that it has no more data to transmit. When the *FIN* is acknowledged, that direction is shut down for new data. Data may continue to flow indefinitely in the other direction, however. When both directions have been shut down, the connection is released. Normally, four TCP segments are needed to release a connection: one *FIN* and one *ACK* for each direction. However, it is possible for the first *ACK* and the second *FIN* to be contained in the same segment, reducing the total count to three.

Just as with telephone calls in which both people say goodbye and hang up the phone simultaneously, both ends of a TCP connection may send *FIN* segments at the same time. These are each acknowledged in the usual way, and the connection is shut down. There is, in fact, no essential difference between the two hosts releasing sequentially or simultaneously.

To avoid the two-army problem (discussed in Sec. 6.2.3), timers are used. If a response to a *FIN* is not forthcoming within two maximum packet lifetimes, the sender of the *FIN* releases the connection. The other side will eventually notice that nobody seems to be listening to it anymore and will time out as well. While this solution is not perfect, given the fact that a perfect solution is theoretically impossible, it will have to do. In practice, problems rarely arise.

6.5.7 TCP Connection Management Modeling

The steps required to establish and release connections can be represented in a finite state machine with the 11 states listed in Fig. 6-38. In each state, certain events are legal. When a legal event happens, some action may be taken. If some other event happens, an error is reported.

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIME WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

Figure 6-38. The states used in the TCP connection management finite state machine.

Each connection starts in the *CLOSED* state. It leaves that state when it does either a passive open (LISTEN) or an active open (CONNECT). If the other side does the opposite one, a connection is established and the state becomes *ESTABLISHED*. Connection release can be initiated by either side. When it is complete, the state returns to *CLOSED*.

The finite state machine itself is shown in Fig. 6-39. The common case of a client actively connecting to a passive server is shown with heavy lines—solid for the client, dotted for the server. The lightface lines are unusual event sequences. Each line in Fig. 6-39 is marked by an *event/action* pair. The event can either be a user-initiated system call (CONNECT, LISTEN, SEND, or CLOSE), a segment arrival (*SYN*, *FIN*, *ACK*, or *RST*), or, in one case, a timeout of twice the maximum packet lifetime. The action is the sending of a control segment (*SYN*, *FIN*, or *RST*) or nothing, indicated by –. Comments are shown in parentheses.

One can best understand the diagram by first following the path of a client (the heavy solid line), then later following the path of a server (the heavy dashed line). When an application program on the client machine issues a CONNECT request, the local TCP entity creates a connection record, marks it as being in the *SYN SENT* state, and shoots off a *SYN* segment. Note that many connections may be open (or being opened) at the same time on behalf of multiple applications, so the state is per connection and recorded in the connection record. When the *SYN+ACK* arrives, TCP sends the final *ACK* of the three-way handshake and switches into the *ESTABLISHED* state. Data can now be sent and received.

When an application is finished, it executes a CLOSE primitive, which causes the local TCP entity to send a *FIN* segment and wait for the corresponding *ACK* (dashed box marked ''active close''). When the *ACK* arrives, a transition is made to the state *FIN WAIT 2* and one direction of the connection is closed. When the

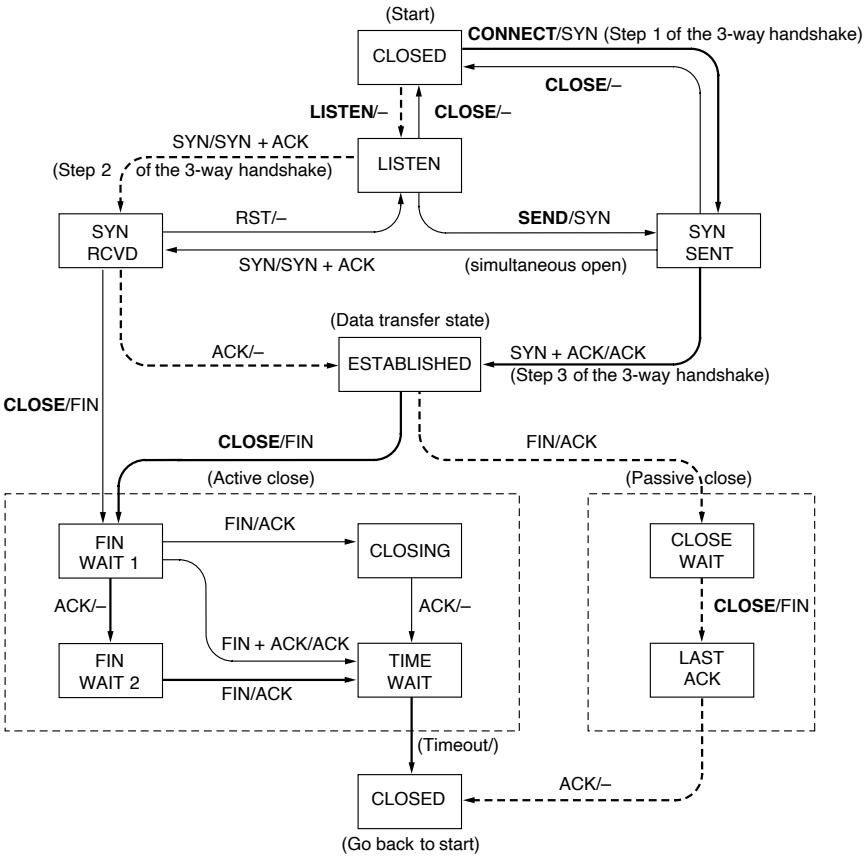


Figure 6-39. TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. Each transition is labeled with the event causing it and the action resulting from it, separated by a slash.

other side closes, too, a *FIN* comes in, which is acknowledged. Now both sides are closed, but TCP waits a time equal to twice the maximum packet lifetime to guarantee that all packets from the connection have died off, just in case the acknowledgement was lost. When the timer goes off, TCP deletes the connection record.

Now let us examine connection management from the server's viewpoint. The server does a LISTEN and settles down to see who turns up. When a *SYN* comes in, it is acknowledged and the server goes to the *SYN RCVD* state. When the server's

SYN is itself acknowledged, the three-way handshake is complete and the server goes to the *ESTABLISHED* state. Data transfer can now occur.

When the client is done transmitting its data, it does a CLOSE, which causes a *FIN* to arrive at the server (dashed box marked ''passive close''). The server is then signaled. When it, too, does a CLOSE, a *FIN* is sent to the client. When the client's acknowledgement shows up, the server releases the connection and deletes the connection record.

6.5.8 TCP Sliding Window

As mentioned earlier, window management in TCP decouples the issues of acknowledgement of the correct receipt of segments and receiver buffer allocation. For example, suppose the receiver has a 4096-byte buffer, as shown in Fig. 6-40. If the sender transmits a 2048-byte segment that is correctly received, the receiver will acknowledge the segment. However, since it now has only 2048 bytes of buffer space (until the application removes some data from the buffer), it will advertise a window of 2048 starting at the next byte expected.

Now the sender transmits another 2048 bytes, which are acknowledged, but the advertised window is of size 0. The sender must stop until the application process on the receiving host has removed some data from the buffer, at which time TCP can advertise a larger window and more data can be sent.

When the window is 0, the sender may not normally send segments, with two exceptions. First, urgent data may be sent, for example, to allow the user to kill the process running on the remote machine. Second, the sender may send a 1-byte segment to force the receiver to reannounce the next byte expected and the window size. This packet is called a **window probe**. The TCP standard explicitly provides this option to prevent deadlock if a window update ever gets lost.

Senders are not required to transmit data as soon as they come in from the application. Neither are receivers required to send acknowledgements as soon as possible. For example, in Fig. 6-40, when the first 2 KB of data came in, TCP, knowing that it had a 4-KB window, would have been completely correct in just buffering the data until another 2 KB came in, to be able to transmit a segment with a 4-KB payload. This freedom can be used to improve performance.

Consider a connection to a remote terminal, for example using SSH or telnet, that reacts on every keystroke. In the worst case, whenever a character arrives at the sending TCP entity, TCP creates a 21-byte TCP segment, which it gives to IP to send as a 41-byte IP datagram. At the receiving side, TCP immediately sends a 40-byte acknowledgement (20 bytes of TCP header and 20 bytes of IP header). Later, when the remote terminal has read the byte, TCP sends a window update, moving the window 1 byte to the right. This packet is also 40 bytes. Finally, when the remote terminal has processed the character, it echoes the character for local display using a 41-byte packet. In all, 162 bytes of bandwidth are consumed and

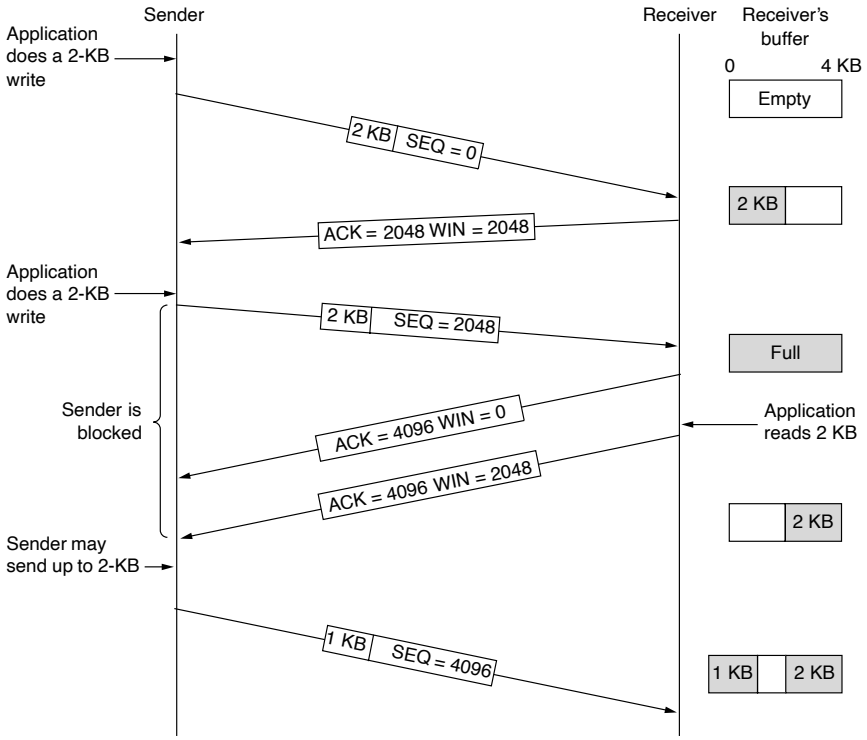


Figure 6-40. Window management in TCP.

four segments are sent for each character typed. When bandwidth is scarce, this method of doing business is not desirable.

One approach that many TCP implementations use to optimize this situation is called **delayed acknowledgements**. The idea is to delay acknowledgements and window updates for up to 500 msec in the hope of acquiring some data on which to hitch a free ride. Assuming the terminal echoes within 500 msec, only one 41-byte packet now need be sent back by the remote side, cutting the packet count and bandwidth usage in half.

Although delayed acknowledgements reduce the load placed on the network by the receiver, a sender that sends multiple short packets (e.g., 41-byte packets containing 1 byte of data) is still operating inefficiently. A way to reduce this usage is known as **Nagle's algorithm** (Nagle, 1984). What Nagle suggested is simple: when data come into the sender in small pieces, just send the first piece and buffer all the rest until the first piece is acknowledged. Then send all the buffered data in

one TCP segment and start buffering again until the next segment is acknowledged. That is, only one short packet can be outstanding at any time. If many pieces of data are sent by the application in one round-trip time, Nagle's algorithm will put the many pieces in one segment, greatly reducing the bandwidth used. The algorithm additionally says that a new segment should be sent if enough data have trickled in to fill a maximum segment.

Nagle's algorithm is widely used by TCP implementations, but there are times when it is better to disable it. In particular, in interactive games that are run over the Internet, the players typically want a rapid stream of short update packets. Gathering the updates to send them in bursts makes the game respond erratically, which makes for unhappy users. A more subtle problem is that Nagle's algorithm can sometimes interact with delayed acknowledgements to cause a temporary deadlock: the receiver waits for data on which to piggyback an acknowledgement, and the sender waits on the acknowledgement to send more data. This interaction can delay the downloads of Web pages. Because of these problems, Nagle's algorithm can be disabled (which is called the *TCP_NODELAY* option). Mogul and Minshall (2001) discuss this and other solutions.

Another problem that can degrade TCP performance is the **silly window syndrome** (Clark, 1982). This problem occurs when data are passed to the sending TCP entity in large blocks, but an interactive application on the receiving side reads data only 1 byte at a time. To see the problem, look at Fig. 6-41. Initially, the TCP buffer on the receiving side is full (i.e., it has a window of size 0) and the sender knows this. Then the interactive application reads one character from the TCP stream. This action makes the receiving TCP happy, so it sends a window update to the sender saying that it is all right to send 1 byte. The sender obliges and sends 1 byte. The buffer is now full, so the receiver acknowledges the 1-byte segment and sets the window to 0. This behavior can go on forever.

Clark's solution is to prevent the receiver from sending a window update for 1 byte. Instead, it is forced to wait until it has a decent amount of space available and advertise that instead. Specifically, the receiver should not send a window update until it can handle the maximum segment size it advertised when the connection was established or until its buffer is half empty, whichever is smaller. Furthermore, the sender can also help by not sending tiny segments. Instead, it should wait until it can send a full segment, or at least one containing half of the receiver's buffer size.

Nagle's algorithm and Clark's solution to the silly window syndrome are complementary. Nagle was trying to solve the problem caused by the sending application delivering data to TCP a byte at a time. Clark was trying to solve the problem of the receiving application sucking the data up from TCP a byte at a time. Both solutions are valid and can work together. The goal is for the sender not to send small segments and the receiver not to ask for them.

The receiving TCP can go further in improving performance than just doing window updates in large units. Like the sending TCP, it can also buffer data, so it

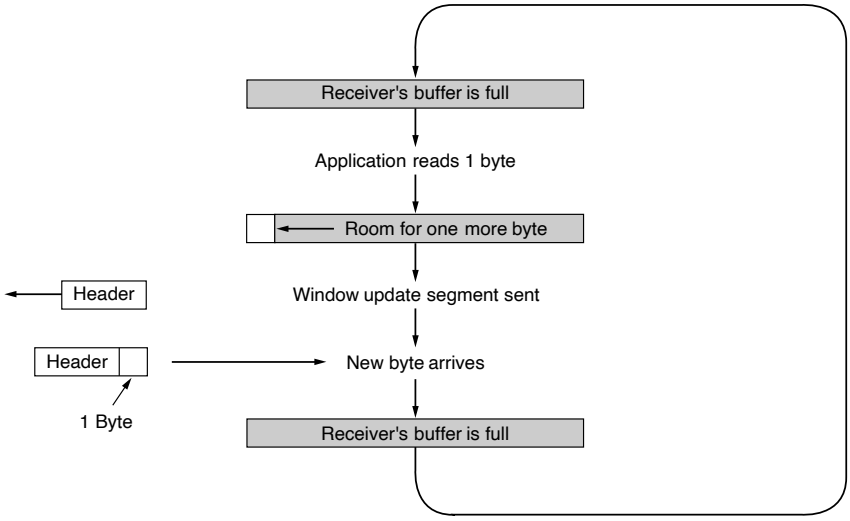


Figure 6-41. Silly window syndrome.

can block a READ request from the application until it has a large chunk of data for it. Doing so reduces the number of calls to TCP (and the overhead). It also increases the response time, but for noninteractive applications like file transfer, efficiency may be more important than response time to individual requests.

Another issue that the receiver must handle is that segments may arrive out of order. The receiver will buffer the data until it can be passed up to the application in order. Actually, nothing bad would happen if out-of-order segments were discarded, since they would eventually be retransmitted by the sender, but it would be wasteful.

Acknowledgements can be sent only when all the data up to the byte acknowledged have been received. This is a cumulative acknowledgement. If the receiver gets segments 0, 1, 2, 4, 5, 6, and 7, it can acknowledge everything up to and including the last byte in segment 2. When the sender times out, it then retransmits segment 3. As the receiver has buffered segments 4 through 7, upon receipt of segment 3 it can acknowledge all bytes up to the end of segment 7.

6.5.9 TCP Timer Management

TCP uses multiple timers (at least conceptually) to do its work. The most important of these is the **RTO** (**Retransmission TimeOut**). When a segment is sent, a retransmission timer is started. If the segment is acknowledged before the

timer expires, the timer is stopped. If, on the other hand, the timer goes off before the acknowledgement comes in, the segment is retransmitted (and the timer is started again). The question that arises is: how long should the timeout be?

This problem is much more difficult in the transport layer than in data link protocols such as 802.11. In the latter case, the expected delay is measured in microseconds and is highly predictable (i.e., has a low variance), so the timer can be set to go off just slightly after the acknowledgement is expected, as shown in Fig. 6-42(a). Since acknowledgements are rarely delayed in the data link layer (due to lack of congestion), the absence of an acknowledgement at the expected time generally means either the frame or the acknowledgement has been lost.

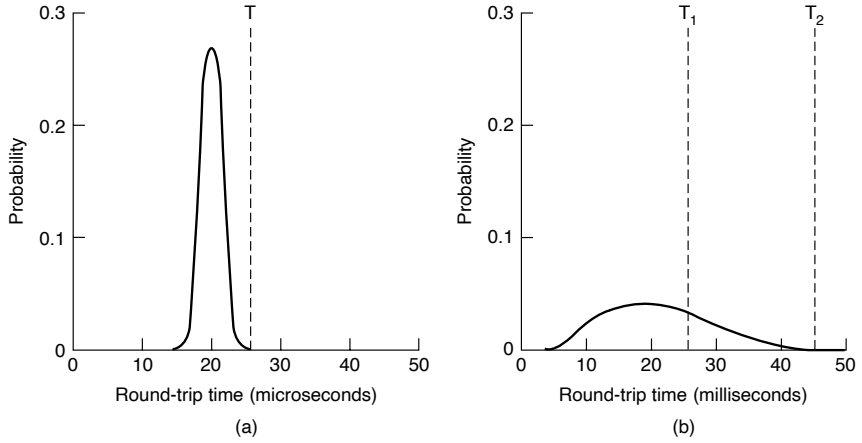


Figure 6-42. (a) Probability density of acknowledgement arrival times in the data link layer. (b) Probability density of acknowledgement arrival times for TCP.

TCP is faced with a radically different environment. The probability density function for the time it takes for a TCP acknowledgement to come back looks more like Fig. 6-42(b) than Fig. 6-42(a). It is larger and more variable. Determining the round-trip time to the destination is tricky. Even when it is known, deciding on the timeout interval is also difficult. If the timeout is set too short, say, T_1 in Fig. 6-42(b), unnecessary retransmissions will occur, clogging the Internet with useless packets. If it is set too long (e.g., T_2), performance will suffer due to the long retransmission delay whenever a packet is lost. Furthermore, the mean and variance of the acknowledgement arrival distribution can change rapidly within a few seconds as congestion builds up or is resolved.

The solution is to use a dynamic algorithm that constantly adapts the timeout interval, based on continuous measurements of network performance. The algorithm generally used by TCP is due to Jacobson (1988) and works as follows. For each connection, TCP maintains a variable, *SRTT* (Smoothed Round-Trip Time),

that is the best current estimate of the round-trip time to the destination in question. When a segment is sent, a timer is started, both to see how long the acknowledgement takes and also to trigger a retransmission if it takes too long. If the acknowledgement gets back before the timer expires, TCP measures how long the acknowledgement took, say, R. It then updates $SRTT$ according to the formula

$$SRTT = \alpha\ SRTT + (1 - \alpha)\ R$$

where α is a smoothing factor that determines how quickly the old values are forgotten. Typically, $\alpha = 7/8$. This kind of formula is an **EWMA** (**Exponentially Weighted Moving Average**) or low-pass filter that discards noise in the samples.

Even given a good value of $SRTT$, choosing a suitable retransmission timeout is a nontrivial matter. Initial implementations of TCP used $2xRTT$, but experience showed that a constant value was too inflexible because it failed to respond when the variance went up. In particular, queueing models of random (i.e., Poisson) traffic predict that when the load approaches capacity, the delay becomes large and highly variable. This can lead to the retransmission timer firing and a copy of the packet being retransmitted although the original packet is still transiting the network. It is all the more likely to happen under conditions of high load, which is the worst time at which to send additional packets into the network.

To fix this problem, Jacobson proposed making the timeout value sensitive to the variance in round-trip times as well as the smoothed round-trip time. This change requires keeping track of another smoothed variable, $RTTVAR$ (Round-Trip Time VARiation) that is updated using the formula

$$RTTVAR = \beta\ RTTVAR + (1 - \beta)\ |SRTT - R|$$

This is an EWMA as before, and typically $\beta = 3/4$. The retransmission timeout, RTO, is set to be

$$RTO = SRTT + 4 \times RTTVAR$$

The choice of the factor 4 is somewhat arbitrary, but multiplication by 4 can be done with a single shift, and less than 1% of all packets come in more than four standard deviations late. Note that $RTTVAR$ is not exactly the same as the standard deviation (it is really the mean deviation), but it is close enough in practice. Jacobson's paper is full of clever tricks to compute timeouts using only integer adds, subtracts, and shifts. This economy is not needed for modern hosts, but it has become part of the culture that allows TCP to run on all manner of devices, from supercomputers down to tiny devices. So far nobody has put it on an RFID chip, but someday? Who knows.

More details of how to compute this timeout, including initial settings of the variables, are given in RFC 2988. The retransmission timer is also held to a minimum of 1 second, regardless of the estimates. This is a conservative (albeit somewhat empirical) value chosen to prevent spurious retransmissions based on measurements (Allman and Paxson, 1999).

One problem that occurs with gathering the samples, R, of the round-trip time is what to do when a segment times out and is sent again. When the acknowledgement comes in, it is unclear whether the acknowledgement refers to the first transmission or a later one. Guessing wrong can seriously contaminate the retransmission timeout. Phil Karn discovered this problem the hard way. Karn is an amateur radio enthusiast interested in transmitting TCP/IP packets by ham radio, a notoriously unreliable medium. He made a simple proposal: do not update estimates on any segments that have been retransmitted. Additionally, the timeout is doubled on each successive retransmission until the segments get through the first time. This fix is called **Karn's algorithm** (Karn and Partridge, 1987). Most TCP implementations use it.

The retransmission timer is not the only timer TCP uses. A second timer is the **persistence timer**. It is designed to prevent the following deadlock. The receiver sends an acknowledgement with a window size of 0, telling the sender to wait. Later, the receiver updates the window, but the packet with the update is lost. Now the sender and the receiver are each waiting for the other to do something. When the persistence timer goes off, the sender transmits a probe to the receiver. The response to the probe gives the window size. If it is still 0, the persistence timer is set again and the cycle repeats. If it is nonzero, data can now be sent.

A third timer that some implementations use is the **keepalive timer**. When a connection has been idle for a long time, the keepalive timer may go off to cause one side to check whether the other side is still there. If it fails to respond, the connection is terminated. This feature is controversial because it adds overhead and may terminate an otherwise healthy connection due to a transient network partition.

The last timer used on each TCP connection is the one used in the *TIME WAIT* state while closing. It runs for twice the maximum packet lifetime to make sure that when a connection is closed, all packets created by it have died off.

6.5.10 TCP Congestion Control

We have saved one of the key functions of TCP for last: congestion control. When the load offered to any network is more than it can handle, congestion builds up. The Internet is no exception. The network layer detects congestion when queues grow large at routers and tries to manage it, if only by dropping packets. It is up to the transport layer to receive congestion feedback from the network layer and slow down the rate of traffic that it is sending into the network. In the Internet, TCP plays the main role in controlling congestion, as well as the main role in reliable transport. That is why it is such a special protocol.

We covered the general situation of congestion control back in Sec. 6.3. One key takeaway there was that a transport protocol using an additive increase multiplicative decrease control law in response to binary congestion signals from the

network would converge to a fair and efficient bandwidth allocation. TCP congestion control is based on implementing this approach using a window and with packet loss as the binary signal. To do so, TCP maintains a **congestion window** whose size is the number of bytes the sender may have in the network at any time. The corresponding rate is the window size divided by the round-trip time of the connection. TCP adjusts the size of the window according to the AIMD rule.

Recall that the congestion window is maintained *in addition* to the flow control window, which specifies the number of bytes that the receiver can buffer. Both windows are tracked in parallel, and the number of bytes that may be sent is the smaller of the two windows. Thus, the effective window is the smaller of what the sender thinks is all right and what the receiver thinks is all right. It takes two to tango. TCP will stop sending data if either the congestion or the flow control window is temporarily full. If the receiver says ''send 64 KB'' but the sender knows that bursts of more than 32 KB clog the network, it will send 32 KB. On the other hand, if the receiver says ''send 64 KB'' and the sender knows that bursts of up to 128 KB get through effortlessly, it will send the full 64 KB requested. The flow control window was described earlier, and in what follows we will only describe the congestion window.

Modern congestion control was added to TCP largely through the efforts of Van Jacobson (1988). It is a fascinating story. Starting in 1986, the growing popularity of the early Internet led to the first occurrence of what became known as a **congestion collapse**, a prolonged period during which goodput dropped precipitously (i.e., by more than a factor of 100) due to congestion in the network. Jacobson (and many others) set out to understand what was happening and remedy the situation.

The high-level fix that Jacobson implemented was to approximate an AIMD congestion window. The interesting part, and much of the complexity of TCP congestion control, is how he added this to an existing implementation without changing any of the message formats, which made it instantly deployable. To start, he observed that packet loss is a suitable signal of congestion. This signal comes a little late (as the network is already congested) but it is quite dependable. After all, it is difficult to build a router that does not drop packets when it is overloaded. This fact is unlikely to change. Even when terabyte memories appear to buffer vast numbers of packets, we will probably have terabit/sec networks to fill up those memories.

However, using packet loss as a congestion signal depends on transmission errors being relatively rare. This is not normally the case for wireless links such as 802.11, which is why they include their own retransmission mechanism at the link layer. Because of wireless retransmissions, network layer packet loss due to transmission errors is normally masked on wireless networks. It is also rare on other links because wires and optical fibers typically have low bit-error rates.

All the Internet TCP algorithms assume that lost packets are caused by congestion and monitor timeouts and look for signs of trouble the way miners watch their

canaries. A good retransmission timer is needed in order to detect packet loss signals accurately and in a timely manner. We have already discussed how the TCP retransmission timer includes estimates of the mean and variation in round-trip times. Fixing this timer, by including the variation factor, was an important step in Jacobson's work. Given a good retransmission timeout, the TCP sender can track the outstanding number of bytes, which are loading the network. It simply looks at the difference between the sequence numbers that are transmitted and acknowledged.

Now it seems that our task is easy. All we need to do is to track the congestion window, using sequence and acknowledgement numbers, and adjust the congestion window using an AIMD rule. As you might have expected, it is more complicated than that. A first consideration is that the way packets are sent into the network, even over short periods of time, must be matched to the network path. Otherwise the traffic will cause congestion. For example, consider a host with a congestion window of 64 KB attached to a 1-Gbps switched Ethernet. If the host sends the entire window at once, this burst of traffic may travel over a slow 1-Mbps ADSL line further along the path. The burst that took only half a millisecond on the 1-Gbps line will clog the 1-Mbps line for half a second, completely disrupting protocols such as voice over IP. This behavior might be a good idea for a protocol designed to cause congestion, but not for a protocol to control it.

However, it turns out that we can use small bursts of packets to our advantage. Fig. 6-43 shows what happens when a sender on a fast network (the 1-Gbps link) sends a small burst of four packets to a receiver on a slow network (the 1-Mbps link) that is the bottleneck or slowest part of the path. Initially the four packets travel over the link as quickly as they can be sent by the sender. At the router, they are queued while being sent because it takes longer to send a packet over the slow link than to receive the next packet over the fast link. But the queue is not large because only a small number of packets were sent at once. Note the increased length of the packets on the slow link. The same packet, of 1 KB say, is now longer because it takes more time to send it on a slow link than on a fast one.

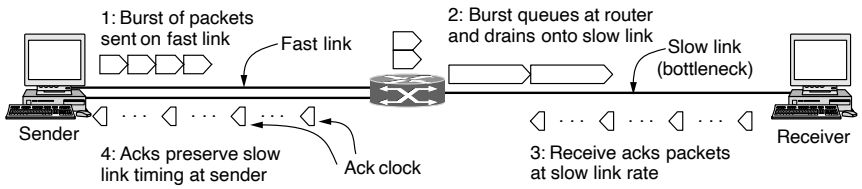


Figure 6-43. A burst of packets from a sender and the returning ack clock.

Eventually the packets get to the receiver, where they are acknowledged. The times for the acknowledgements reflect the times at which the packets arrived at

the receiver after crossing the slow link. They are spread out compared to the original packets on the fast link. As these acknowledgements travel over the network and back to the sender they preserve this timing.

The key observation is this: the acknowledgements return to the sender at about the rate that packets can be sent over the slowest link in the path. This is precisely the rate that the sender wants to use. If it injects new packets into the network at this rate, they will be sent as fast as the slow link permits, but they will not queue up and congest any router along the path. This timing is known as an **ack clock**. It is an essential part of TCP. By using an ack clock, TCP smoothes out traffic and avoids unnecessary queues at routers.

A second consideration is that the AIMD rule will take a very long time to reach a good operating point on fast networks if the congestion window is started from a small size. Consider a modest network path that can support 10 Mbps with an RTT of 100 msec. The appropriate congestion window is the bandwidth-delay product, which is 1 Mbit or 100 packets of 1250 bytes each. If the congestion window starts at 1 packet and increases by 1 packet every RTT, it will be 100 RTTs or 10 seconds before the connection is running at about the right rate. That is a long time to wait just to get to the right speed for a transfer. We could reduce this startup time by starting with a larger initial window, say of 50 packets. But this window would be far too large for slow or short links. It would cause congestion if used all at once, as we have just described.

Instead, the solution Jacobson chose to handle both of these considerations is a mix of linear and multiplicative increase. When a connection is established, the sender initializes the congestion window to a small initial value of at most four segments; the details are described in RFC 3390, and the use of four segments is an increase from an earlier initial value of one segment based on experience. The sender then sends the initial window. The packets will take a round-trip time to be acknowledged. For each segment that is acknowledged before the retransmission timer goes off, the sender adds one segment's worth of bytes to the congestion window. Plus, as that segment has been acknowledged, there is now one less segment in the network. The upshot is that every acknowledged segment allows two more segments to be sent. The congestion window is doubling every round-trip time.

This algorithm is called **slow start**, but it is not slow at all—it is exponential growth—except in comparison to the previous algorithm that let an entire flow control window be sent all at once. Slow start is shown in Fig. 6-44. In the first round-trip time, the sender injects one packet into the network (and the receiver receives one packet). Two packets are sent in the next round-trip time, then four packets in the third round-trip time.

Slow start works well over a range of link speeds and round-trip times, and uses an ack clock to match the rate of sender transmissions to the network path. Take a look at the way acknowledgements return from the sender to the receiver in Fig. 6-44. When the sender gets an acknowledgement, it increases the congestion window by one and immediately sends two packets into the network. (One packet

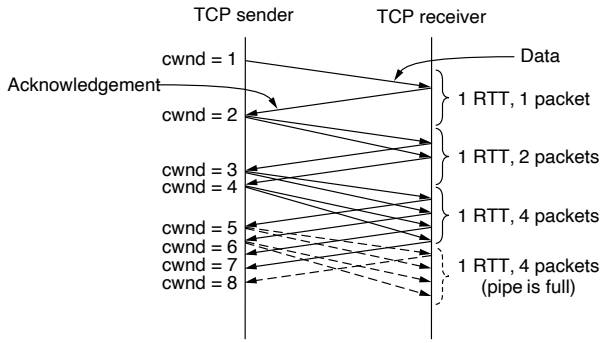


Figure 6-44. Slow start from an initial congestion window of one segment.

is the increase by one; the other packet is a replacement for the packet that has been acknowledged and left the network. At all times, the number of unacknowledged packets is given by the congestion window.) However, these two packets will not necessarily arrive at the receiver as closely spaced as when they were sent. For example, suppose the sender is on a 100-Mbps Ethernet. Each packet of 1250 bytes takes 100 μsec to send. So the delay between the packets can be as small as 100 μsec. The situation changes if these packets go across a 1-Mbps ADSL link anywhere along the path. It now takes 10 msec to send the same packet. This means that the minimum spacing between the two packets has grown by a factor of 100. Unless the packets have to wait together in a queue on a later link, the spacing will remain large.

In Fig. 6-44, this effect is shown by enforcing a minimum spacing between data packets arriving at the receiver. The same spacing is kept when the receiver sends acknowledgements, and thus when the sender receives the acknowledgements. If the network path is slow, acknowledgements will come in slowly (after a delay of an RTT). If the network path is fast, acknowledgements will come in quickly (again, after the RTT). All the sender has to do is follow the timing of the ack clock as it injects new packets, which is what slow start does.

Because slow start causes exponential growth, eventually (and sooner rather than later) it will send too many packets into the network too quickly. When this happens, queues will build up in the network. When the queues are full, one or more packets will be lost. After this happens, the TCP sender will time out when an acknowledgement fails to arrive in time. There is evidence of slow start growing too fast in Fig. 6-44. After three RTTs, four packets are in the network. These four packets take an entire RTT to arrive at the receiver. That is, a congestion window of four packets is the right size for this connection. However, as these packets are acknowledged, slow start continues to grow the congestion window, reaching eight packets in another RTT. Only four of these packets can reach the receiver in one

RTT, no matter how many are sent. That is, the network pipe is full. Additional packets placed into the network by the sender will build up in router queues, since they cannot be delivered to the receiver quickly enough. Congestion and packet loss will occur soon.

To keep slow start under control, the sender keeps a threshold for the connection called the **slow start threshold**. Initially this value is set arbitrarily high, to the size of the flow control window, so that it will not limit the connection. TCP keeps increasing the congestion window in slow start until a timeout occurs or the congestion window exceeds the threshold (or the receiver's window is filled).

Whenever a packet loss is detected, for example, by a timeout, the slow start threshold is set to half of the congestion window and the entire process is restarted. The idea is that the current window is too large because it caused congestion previously that is only now detected by a timeout. Half of the window, which was used successfully earlier, is probably a better estimate for a congestion window that is close to the path capacity without causing loss. In our example in Fig. 6-44, growing the congestion window to eight packets may cause loss, while the congestion window of four packets in the previous RTT was the right value. The congestion window is then reset to its small initial value and slow start resumes.

Whenever the slow start threshold is crossed, TCP switches from slow start to additive increase. In this mode, the congestion window is increased by one segment every round-trip time. Like slow start, this is usually implemented with an increase for every segment that is acknowledged, rather than an increase once per RTT. Call the congestion window *cwnd* and the maximum segment size *MSS*. A common approximation is to increase *cwnd* by (*MSS* × *MSS*)/*cwnd* for each of the *cwnd*/*MSS* packets that may be acknowledged. This increase does not need to be fast. The whole idea is for a TCP connection to spend a lot of time with its congestion window close to the optimum value—not so small that throughput will be low, and not so large that congestion will occur.

Additive increase is shown in Fig. 6-45 for the same situation as slow start. At the end of every RTT, the sender's congestion window has grown enough that it can inject an additional packet into the network. Compared to slow start, the linear rate of growth is much slower. It makes little difference for small congestion windows, as is the case here, but a large difference in the time taken to grow the congestion window to 100 segments, for example.

There is something else that we can do to improve performance. The defect in the scheme so far is waiting for a timeout. Timeouts are relatively long because they must be conservative. After a packet is lost, the receiver cannot acknowledge past it, so the acknowledgement number will stay fixed, and the sender will not be able to send any new packets into the network because its congestion window remains full. This condition can continue for a relatively long period until the timer fires and the lost packet is sent again. At that stage, TCP slow starts again.

There is a quick way for the sender to recognize that one of its packets has been lost. As packets beyond the lost packet arrive at the receiver, they trigger

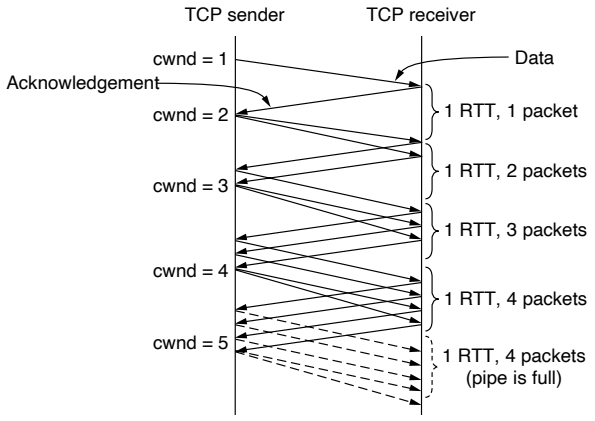


Figure 6-45. Additive increase from an initial congestion window of one segment.

acknowledgements that return to the sender. These acknowledgements bear the same acknowledgement number. They are called **duplicate acknowledgements**. Each time the sender receives a duplicate acknowledgement, it is likely that another packet has arrived at the receiver and the lost packet still has not shown up.

Because packets can take different paths through the network, they can arrive out of order. This will trigger duplicate acknowledgements even though no packets have been lost. However, this is uncommon in the Internet much of the time. When there is reordering across multiple paths, the received packets are usually not reordered too much. Thus, TCP somewhat arbitrarily assumes that three duplicate acknowledgements imply that a packet has been lost. The identity of the lost packet can be inferred from the acknowledgement number as well. It is the very next packet in sequence. This packet can then be retransmitted right away, before the retransmission timeout fires.

This heuristic is called **fast retransmission**. After it fires, the slow start threshold is still set to half the current congestion window, just as with a timeout. Slow start can be restarted by setting the congestion window to one packet. With this window size, a new packet will be sent after the one round-trip time that it takes to acknowledge the retransmitted packet along with all data that had been sent before the loss was detected.

An illustration of the congestion algorithm we have built up so far is shown in Fig. 6-46. This version of TCP is called TCP Tahoe after the 4.2BSD Tahoe release in 1988 in which it was included. The maximum segment size here is 1 KB. Initially, the congestion window was 64 KB, but a timeout occurred, so the threshold is set to 32 KB and the congestion window to 1 KB for transmission 0. The congestion window grows exponentially until it hits the threshold (32 KB).

The window is increased every time a new acknowledgement arrives rather than continuously, which leads to the discrete staircase pattern. After the threshold is passed, the window grows linearly. It is increased by one segment every RTT.

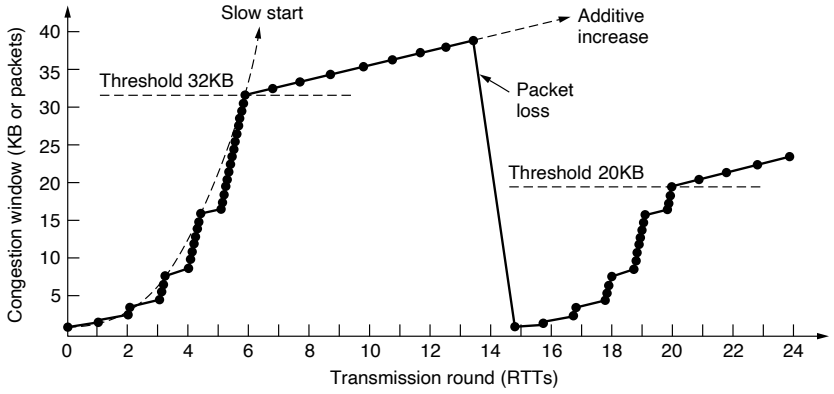


Figure 6-46. Slow start followed by additive increase in TCP Tahoe.

The transmissions in round 13 are unlucky (they should have known), and one of them is lost in the network. This is detected when three duplicate acknowledgements arrive. At that time, the lost packet is retransmitted, the threshold is set to half the current window (by now 40 KB, so half is 20 KB), and slow start is initiated all over again. Restarting with a congestion window of one packet takes one round-trip time for all of the previously transmitted data to leave the network and be acknowledged, including the retransmitted packet. The congestion window grows with slow start as it did previously, until it reaches the new threshold of 20 KB. At that time, the growth becomes linear again. It will continue in this fashion until another packet loss is detected via duplicate acknowledgements or a timeout (or the receiver's window becomes the limit).

TCP Tahoe (which included good retransmission timers) provided a working congestion control algorithm that solved the problem of congestion collapse. Jacobson realized that it is possible to do even better. At the time of the fast retransmission, the connection is running with a congestion window that is too large, but it is still running with a working ack clock. Every time another duplicate acknowledgement arrives, it is likely that another packet has left the network. Using duplicate acknowledgements to count the packets in the network, makes it possible to let some packets exit the network and continue to send a new packet for each additional duplicate acknowledgement.

Fast recovery is the heuristic that implements this behavior. It is a temporary mode that aims to maintain the ack clock running with a congestion window that is the new threshold, or half the value of the congestion window at the time of the

fast retransmission. To do this, duplicate acknowledgements are counted (including the three that triggered fast retransmission) until the number of packets in the network has fallen to the new threshold. This takes about half a round-trip time. From then on, a new packet can be sent for each duplicate acknowledgement that is received. One round-trip time after the fast retransmission, the lost packet will have been acknowledged. At that time, the stream of duplicate acknowledgements will cease and fast recovery mode will be exited. The congestion window will be set to the new slow start threshold and grows by linear increase.

The upshot of this heuristic is that TCP avoids slow start, except when the connection is first started and when a timeout occurs. The latter can still happen when more than one packet is lost and fast retransmission does not recover adequately. Instead of repeated slow starts, the congestion window of a running connection follows a **sawtooth** pattern of additive increase (by one segment every RTT) and multiplicative decrease (by half in one RTT). This is exactly the AIMD rule that we sought to implement.

This sawtooth behavior is shown in Fig. 6-47. It is produced by TCP Reno, named after the 1990 4.3BSD Reno release in which it was included. TCP Reno is essentially TCP Tahoe plus fast recovery. After an initial slow start, the congestion window climbs linearly until a packet loss is detected by duplicate acknowledgements. The lost packet is sent again and fast recovery is used to keep the ack clock running until the retransmission is acknowledged. At that time, the congestion window is resumed from the new slow start threshold, rather than from 1. This behavior continues indefinitely, and the connection spends most of the time with its congestion window near the optimum value of the bandwidth-delay product.

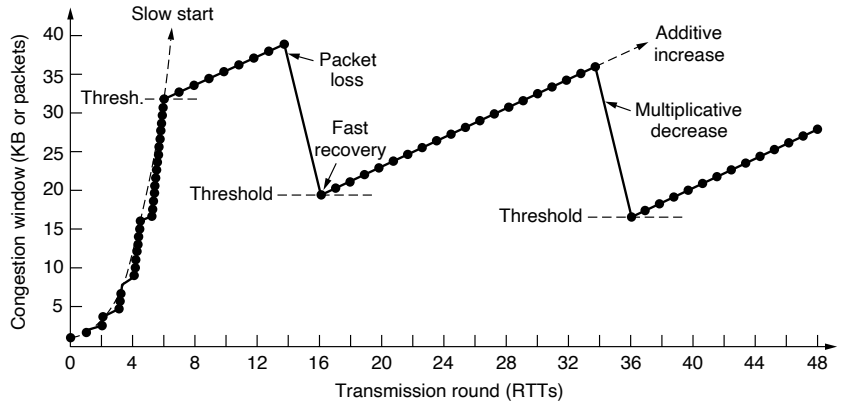


Figure 6-47. Fast recovery and the sawtooth pattern of TCP Reno.

TCP Reno with its mechanisms for adjusting the congestion window has formed the basis for TCP congestion control for more than two decades. Most of the changes in the intervening years have adjusted these mechanisms in minor

ways, for example, by changing the choices of the initial window and removing various ambiguities. Some improvements have been made for recovering from two or more losses in a window of packets. For example, the TCP NewReno version uses a partial advance of the acknowledgement number after a retransmission to find and repair another loss (Hoe, 1996), as described in RFC 3782. Since the mid-1990s, several variations have emerged that follow the principles we have described but use slightly different control laws. For example, Linux uses a variant called CUBIC TCP (Ha et al., 2008) and Windows includes a variant called Compound TCP (Tan et al., 2006).

Two larger changes have also affected TCP implementations. First, much of the complexity of TCP comes from inferring from a stream of duplicate acknowledgements which packets have arrived and which packets have been lost. The cumulative acknowledgement number does not provide this information. A simple fix is the use of SACK, which lists up to three ranges of bytes that have been received. With this information, the sender can more directly decide what packets to retransmit and track the packets in flight to implement the congestion window.

When the sender and receiver set up a connection, they each send the *SACK permitted* TCP option to signal that they understand selective acknowledgements. Once SACK is enabled for a connection, it works as shown in Fig. 6-48. A receiver uses the TCP *Acknowledgement number* field in the normal manner, as a cumulative acknowledgement of the highest in-order byte that has been received. When it receives packet 3 out of order (because packet 2 was lost), it sends a *SACK option* for the received data along with the (duplicate) cumulative acknowledgement for packet 1. The *SACK option* gives the byte ranges that have been received above the number given by the cumulative acknowledgement. The first range is the packet that triggered the duplicate acknowledgement. The next ranges, if present, are older blocks. Up to three ranges are commonly used. By the time packet 6 is received, two SACK byte ranges are used to indicate that packet 6 and packets 3 to 4 have been received, in addition to all packets up to packet 1. From the information in each *SACK option* that it receives, the sender can decide which packets to retransmit. In this case, retransmitting packets 2 and 5 would be a good idea.

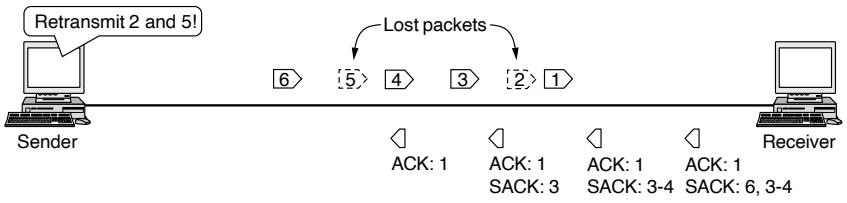


Figure 6-48. Selective acknowledgements.

SACK is strictly advisory information. The actual detection of loss using duplicate acknowledgements and adjustments to the congestion window proceed just

as before. However, with SACK, TCP can recover more easily from situations in which multiple packets are lost at roughly the same time, since the TCP sender knows which packets have not been received. SACK is now widely deployed. It is described in RFC 2883, and TCP congestion control using SACK is described in RFC 3517.

The second change is the use of ECN in addition to packet loss as a congestion signal. ECN is an IP layer mechanism to notify hosts of congestion that we described in Sec. 5.3.2. With it, the TCP receiver can receive congestion signals from IP.

The use of ECN is enabled for a TCP connection when both the sender and receiver indicate that they are capable of using ECN by setting the *ECE* and *CWR* bits during connection establishment. If ECN is used, each packet that carries a TCP segment is flagged in the IP header to show that it can carry an ECN signal. Routers that support ECN will set a congestion signal on packets that can carry ECN flags when congestion is approaching, instead of dropping those packets after congestion has occurred.

The TCP receiver is informed if any packet that arrives carries an ECN congestion signal. The receiver then uses the *ECE* (ECN Echo) flag to signal the TCP sender that its packets have experienced congestion. The sender tells the receiver that it has heard the signal by using the *CWR* (Congestion Window Reduced) flag.

The TCP sender reacts to these congestion notifications in exactly the same way as it does to packet loss that is detected via duplicate acknowledgements. However, the situation is strictly better. Congestion has been detected and no packet was harmed in any way. ECN is described in RFC 3168. It requires both host and router support, and is not yet widely used on the Internet.

For more information on the complete set of congestion control behaviors that are implemented in TCP, see RFC 5681.

6.5.11 TCP CUBIC

To cope with increasingly large bandwidth-delay products, **TCP CUBIC** was developed (Ha et al., 2008). As previously described, networks with large bandwidth-delay products take many round-trip times to reach the available capacity of the end-to-end path. The general approach behind TCP CUBIC is to increase the congestion window in such a way that is a function of the time since the last duplicate acknowledgment, rather than simply based on the arrival of ACKs.

CUBIC also adjusts its congestion window differently as a function of time. In contrast to the standard AIMD congestion control approach as we described above, the congestion window increases according to a cubic function, which initially has a growth in the congestion window, followed by a plateau period, and finally a period of faster growth. Figure 6-49 shows the evolution of TCP CUBIC's congestion window over time. Again, one of the main differences between CUBIC and other versions of TCP is that the congestion window evolves as a function of time,

since the last congestion event, increasing quickly, then plateauing to the congestion window that the sender achieved before the last congestion event, and then again increasing to probe for the optimal rate above that rate until another congestion event occurs.

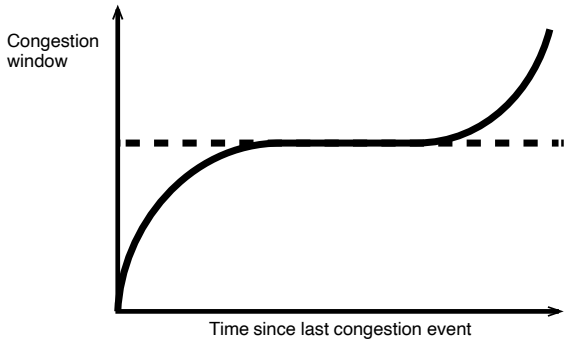


Figure 6-49. Evolution of TCP CUBIC Congestion Window.

TCP CUBIC is implemented by default in the Linux kernels 2.6.19 and above, as well as modern versions of Windows.

6.6 TRANSPORT PROTOCOLS AND CONGESTION CONTROL

As network capacity increases, some of TCP's conventional operating modes no longer achieve optimal performance. In particular, connection-oriented protocols such as TCP can suffer from high connection setup overhead, as well as performance issues on networks with large buffers. In the remainder of this section, we discuss some recent developments in transport protocols to address these issues.

6.6.1 QUIC: Quick UDP Internet Connections

QUIC, initially proposed as (**Quick UDP Internet Connections**) is a transport protocol that aims to improve some of the throughput and latency characteristics of TCP. It was used in more than half of the connections from the Chrome browser to Google's services before it was ever standardized. However, most Web browsers other than Google Chrome do not support the protocol.

As its name suggests, QUIC runs on top of UDP and its main goal has been to make application protocols such as the Web protocols (discussed in Chap. 7) faster. We will discuss how QUIC interacts with the Web's application protocols in some more detail in Chap. 7. As we will soon see, an application such as the Web relies

on establishing multiple connections in parallel to load an individual Web page. Because many of those connections are to a common server, establishing a new connection to load each individual Web object can result in significant overhead. As a result, QUIC aims to multiplex these connections over a single UDP flow, while also ensuring that if a single Web object transfer is delayed, that it does not ultimately block the transfer of other objects.

Because QUIC is based on UDP, it does not automatically achieve reliable transport. If some data is lost in one stream, the protocol can continue transferring data for other streams independently, which can ultimately improve the performance of links with high transmission error rates. QUIC also makes various other optimizations to improve performance, such as piggybacking application-level encryption information on transport-connection establishment, and encrypting each packet individually so that the loss of one packet does not prevent decryption of subsequent packets. QUIC also provides mechanisms for improving the speed of network handoff (e.g., from a cellular connection to a WiFi connection), using a connection identifier as a way to maintain state when endpoints change networks.

6.6.2 BBR: Congestion Control Based on Bottleneck Bandwidth

When bottleneck buffers are large, loss-based congestion control algorithms such as those described earlier end up filling these buffers causing a phenomenon known as **bufferbloat**. The idea behind bufferbloat is fairly straightforward: when network devices in along a network path have buffers that are too large, a TCP sender with a large congestion window can send at a rate that far exceeds the capacity of the network before it ever receives a loss signal. Buffers in the middle of the network can fill up, delaying congestion events for senders that are sending too fast (i.e., not dropping packets) and, importantly, increasing the network latency for senders whose packets are queued behind the packets in a large buffer (Gettys, 2011).

Addressing bufferbloat can be achieved in a number of ways. One possible approach is simply to reduce the size of buffers in network devices; unfortunately, this requires convincing vendors and manufacturers of network devices, from wireless access points to backbone routers, to reduce the size of the buffers in their devices. Even if that battle could be won, there are far too many legacy devices in the network to rely on this approach alone. Another approach is to develop an alternative to loss-based congestion control, which is the approach BBR takes.

The main idea behind BBR is to measure the bottleneck bandwidth and the round-trip propagation delay and use estimates of these parameters to send at exactly the appropriate operating point. BBR thus *continuously* tracks the bottleneck bandwidth and the round-trip propagation delay. TCP already tracks the round-trip time; BBR extends existing functionality by tracking the delivery rate of the transport protocol over time. BBR effectively computes the bottleneck bandwidth as

the maximum of the measured delivery rate over a given time window—typically six to ten round trips.

The general philosophy of BBR is that, up to the bandwidth-delay product of the path, the round-trip time will not increase because no additional buffering is taking place; on the other hand, the delivery rate will remain inversely proportional to the round-trip time and proportional to the amount of packets in flight (the window). Once the amount of packets in flight exceeds the bandwidth-delay product, latency begins to increase as packets are queued, and the delivery rate plateaus. It is at this point that BBR seeks to operate. Fig. 6-50 shows how the round trip time and delivery rate vary with the amount of data in flight (i.e., sent, but not acknowledged). The optimal operating point for BBR occurs when increasing the amount of traffic in flight increases the overall round-trip time but does not increase the delivery rate.

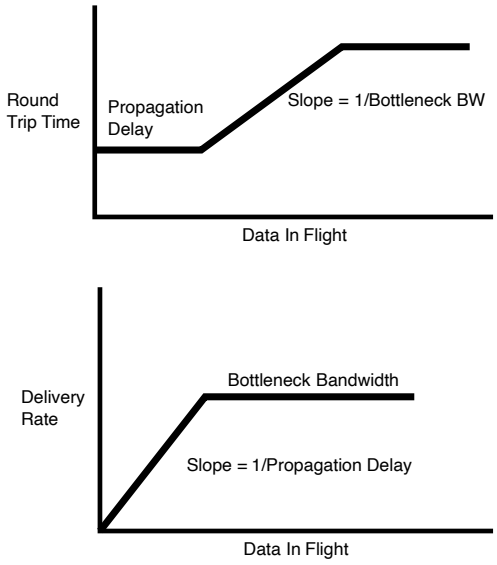


Figure 6-50. BBR Operating Point.

The key to BBR is thus to continually update estimates of the bottleneck bandwidth and round-trip latency accordingly. Each acknowledgement provides new, updated information about round-trip times and average delivery rates, with checks to make sure that the delivery rate is not application-limited (as is sometimes the case in request-response protocols). The second part of BBR is pacing the data itself to match the bottleneck bandwidth rate. The **pacing rate** is the critical parameter for BBR-based congestion control. In steady state, the rate at which BBR sends is simply a function of the bottleneck bandwidth and the round-trip time.

BBR minimizes delay by spending most of its time with exactly one bandwidth-delay product's worth of data in flight, paced at precisely the bottleneck bandwidth rate. Convergence to the bottleneck rate is quite fast.

Google has deployed BBR in a fairly widespread fashion, both on its internal backbone network, as well as in may of its applications. One open question, however, is how well BBR-based congestion control competes with conventional TCP-based congestion control. in one recent experiment, for example, researchers discovered that a BBR sender was consuming 40% of link capacity when sharing a network path with 16 other transport connections, each of which received less than 4% of the remaining bandwidth (Ware et al., 2019). It can be shown that BBR often takes a fixed share of available capacity, regardless of the number of competing TCP flows. Unfortunately, the state of the art for analyzing the fairness properties of new congestion control algorithms is simply to try them out and see what happens. In this case, it seems that there remains significant work to be done to ensure that BBR interacts well with existing TCP traffic on the Internet.

6.6.3 The Future of TCP

As the workhorse of the Internet, TCP has been used for many applications and extended over time to give good performance over a wide range of networks. Many versions are deployed with slightly different implementations than the classic algorithms we have described, especially for congestion control and robustness against attacks. It is likely that TCP will continue to evolve with the Internet. We will mention two particular issues.

The first one is that TCP does not provide the transport semantics that all applications want. For example, some applications want to send messages or records whose boundaries need to be preserved. Other applications work with a group of related conversations, such as a Web browser that transfers several objects from the same server. Still other applications want better control over the network paths that they use. TCP with its standard sockets interface does not meet these needs well. Essentially, the application has the burden of dealing with any problem not solved by TCP. This has led to proposals for new protocols that would provide a slightly different interface. Two examples are SCTP and SST. However, whenever someone proposes changing something that has worked so well for so long, there is always a huge battle between the ''Users are demanding more features'' and ''If it ain't broke, don't fix it'' camps.

6.7 PERFORMANCE ISSUES

Performance issues are critically important in computer networks. When hundreds or thousands of computers are interconnected, complex interactions, with unforeseen consequences, are common. Frequently, this complexity leads to poor

performance and no one knows why. In the following sections, we will examine many issues related to network performance to see what kinds of problems exist and what can be done about them.

Unfortunately, understanding network performance is more an art than a science. There is little underlying theory that is actually of any use in practice. The best we can do is give some rules of thumb gained from hard experience and present examples taken from the real world. We have delayed this discussion until we studied the transport layer because the performance that applications receive depends on the combined performance of the transport, network, and link layers, and to be able to use TCP as an example in various places.

In the next sections, we will look at eight aspects of network performance:

1. Performance problems.
2. Measuring network performance.
3. Measuring access network throughput.
4. Measuring quality of experience.
5. Host design for fast networks.
6. Fast segment processing.
7. Header compression.
8. Protocols for ''long fat'' networks.

These aspects consider network performance both at the host and across the network, and as networks are increased in speed and size.

6.7.1 Performance Problems in Computer Networks

Some performance problems, such as congestion, are caused by temporary resource overloads. If more traffic suddenly arrives at a router than the router can handle, congestion will build up and performance will suffer. We studied congestion in detail in this chapter and in Chap. 5.

Performance also degrades when there is a structural resource imbalance. For example, if a gigabit communication line is attached to a low-end PC, the poor host will not be able to process the incoming packets fast enough and some will be lost. These packets will eventually be retransmitted, adding delay, wasting bandwidth, and generally reducing performance.

Overloads can also be synchronously triggered. As an example, if a segment contains a bad parameter (e.g., the port for which it is destined), in many cases the receiver will thoughtfully send back an error notification. Now consider what could happen if a bad segment is broadcast to 1000 machines: each one might send back an error message. The resulting **broadcast storm** could cripple the network.

UDP suffered from this problem until the ICMP protocol was changed to cause hosts to refrain from responding to errors in UDP segments sent to broadcast addresses. Wireless networks must be particularly careful to avoid unchecked broadcast responses because broadcast occurs naturally and the wireless bandwidth is limited.

A second example of synchronous overload is what happens after an electrical power failure. When the power comes back on, all the machines simultaneously start rebooting. A typical reboot sequence might require first going to some (DHCP) server to learn one's true identity, and then to some file server to get a copy of the operating system. If hundreds of machines in a data center all do this at once, the server will probably collapse under the load.

Even in the absence of synchronous overloads and the presence of sufficient resources, poor performance can occur due to lack of system tuning. For example, if a machine has plenty of CPU power and memory but not enough of the memory has been allocated for buffer space, flow control will slow down segment reception and limit performance. This was a problem for many TCP connections as the Internet became faster but the default size of the flow control window stayed fixed at 64 KB.

Another tuning issue is setting timeouts. When a segment is sent, a timer is set to guard against loss of the segment. If the timeout is set too short, unnecessary retransmissions will occur, clogging the wires. If the timeout is set too long, unnecessary delays will occur after a segment is lost. Other tunable parameters include how long to wait for data on which to piggyback before sending a separate acknowledgement, and how many retransmissions to make before giving up.

Another performance problem that occurs with real-time applications like audio and video is jitter. Having enough bandwidth on average is not sufficient for good performance. Short transmission delays are also required. Consistently achieving short delays demands careful engineering of the load on the network, quality-of-service support at the link and network layers, or both.

6.7.2 Network Performance Measurement

Network operators and users alike aim to measure the performance of networks. A popular measurement to perform, for example, is access network throughput measurement (sometimes referred to simply as ''speed''). For example, many Internet users have used tools such as Speedtest (i.e., *www.speedtest.net*) to measure the performance of access networks. The conventional approach for performing these tests has long been to send as much traffic on the network as quickly as possible (essentially ''filling the pipe''). As the speed of access networks increases, however, measuring the speed of an access link has become more challenging, as filling the pipe requires more data, and as network bottlenecks between the client and the server under test move elsewhere in the network. Perhaps even more importantly, speed is becoming less relevant to network performance than

quality of experience or the performance of an application. As a result, network performance measurement is continuing to evolve, especially in the era of gigabit access networks.

6.7.3 Measuring Access Network Throughput

The conventional approach to measuring network throughput is simply to send as much data along a network path as the network will support over a given period of time, and divide the amount of data transferred by the time taken to transfer the data, thus yielding an average throughput calculation. While seemingly simple and generally appropriate, this approach encounters a number of shortcomings: most importantly, a single TCP connection often cannot exhaust the capacity of a network link, especially as the speed of access links continues to increase. Additionally, if the test captures the early part of the transfer, then the test may capture transfer rates prior to steady state (e.g., TCP slow start), which could ultimately result in a test that under-estimates the access network throughput. Finally, client-based tests (such as speedtest.net or any type of throughput test one might run from a client device) increasingly end up measuring performance limitations other than the access network (e.g., the device's radio, the wireless access network).

To account for these shortcomings, which have become increasingly acute as access networks now begin to exceed gigabit speeds, some best practices have emerged for measuring access network throughput (Feamster et al., 2020). The first is to use multiple parallel TCP connections to fill the capacity of the access link. Tests of early speed tests showed that four TCP connections was typically sufficient to fill access network capacity (Sundaresan 2011); most modern client-based tools, including Speedtest and the throughput test used by the Federal Trade Communications use at least four parallel connections to measure network capacity. Some of these tools even scale the number of network connections, so that connections that appear to have higher capacity are tested with more parallel connections.

A second best practice, which has become increasingly important as the throughput of the ISP access link exceeds that of the home network (and other parts of the end-to-end path), is to perform access network throughput tests directly from the home router. Performing tests in this fashion minimizes the likelihood that extraneous factors (e.g., a client device, the user's wireless network) constrain the throughput test.

As speeds continue to increase, it is likely that additional best practices may emerge, such as measuring to multiple Internet destinations in parallel from a single access connection. Such an approach may be necessary, particularly if the server side of these connections becomes the source of more network throughput bottlenecks. As speeds continue to increase, there is also an increased interest in developing so-called ''passive'' throughput tests, which do not inject large amounts of additional traffic into the network but rather watch traffic as it traverses the

network and attempt to estimate network throughput based on passive observations (while reliable passive access throughput measurements do not yet exist, such an approach might ultimately not be so dissimilar to BBR's approach of monitoring latency and delivery rates to estimate the bottleneck bandwidth).

6.7.4 Measuring Quality of Experience

Ultimately, as access network speeds increase, the most salient performance metrics may not be the speed of the access network in terms of throughput, but rather whether applications perform as users expect them to. For example, in the case of video, a user's experience generally does not depend on throughput, past a certain point (Ramachandran et al., 2019). Ultimately, a user's experience when streaming a video is defined by factors such as how quickly the video starts playing (startup delay), whether the video rebuffers, and the resolution of the video. Beyond about 50 Mbps, however, none of these factors particularly depend on access link throughput, but rather on other properties of the network (latency, jitter, and so forth).

Accordingly, modern network performance measurement is moving beyond simple speed tests, in an effort to estimate user quality of experience, typically based on passive observation of network traffic. These estimators are becoming fairly widespread for streaming video (Ahmed et al., 2017; Krishnamoorthy et al., 2017; Mangla et al., 2018; and Bronzino et al., 2020). The challenges lie in performing this type of optimization across a general class of video services, and ultimately for a larger class of applications (e.g., gaming, virtual reality).

Of course, a user's quality of experience is a measure of whether that person is happy with the service they are using. That metric is ultimately a human consideration and might even require human feedback (e.g., real-time surveys or feedback mechanisms from the user). Internet service providers continue to be interested in mechanisms that can infer or predict user quality of experience and engagement from things they can measure directly (e.g., application throughput, packet loss and interarrival times, etc.).

We are still a ways off from seeing automatic estimation of user quality of experience based on passive measurement of features in network traffic, but this area remains a ripe area for exploration at the intersection of machine learning and networking. Ultimately, the applications could go beyond networking, as transport protocols (and network operators) might even optimize resources for users who demand a higher quality experience. For example, the user who is streaming a video in a remote part of the house but has walked away may care much less about the quality of the application stream than the user who is deeply engrossed in a movie. Of course, distinguishing between a user who is intensely watching a video from one who went to the kitchen for a drink and did not bother to hit the pause button before departing could be tricky.

6.7.5 Host Design for Fast Networks

Measuring and tinkering can improve performance considerably, but they cannot substitute for good design in the first place. A poorly designed network can be improved only so much. Beyond that, it has to be redesigned from scratch.

In this section, we will present some rules of thumb for software implementation of network protocols on hosts. Surprisingly, experience shows that this is often a performance bottleneck on otherwise fast networks, for two reasons. First, NICs (Network Interface Cards) and routers have already been engineered (with hardware support) to run at ''wire speed.'' This means that they can process packets as quickly as the packets can possibly arrive on the link. Second, the relevant performance is that which applications obtain. It is not the link capacity, but the throughput and delay after network and transport processing.

Reducing software overheads improves performance by increasing throughput and decreasing delay. It can also reduce the energy that is spent on networking, which is an important consideration for mobile computers. Most of these ideas have been common knowledge to network designers for years. They were first stated explicitly by Mogul (1993); our treatment largely follows his. Another relevant source is Metcalfe (1993).

Host Speed Is More Important Than Network Speed

Long experience has shown that in nearly all fast networks, operating system and protocol overhead dominate actual time on the wire. For example, in theory, the minimum RPC time on a 1-Gbps Ethernet is 1 μsec, corresponding to a minimum (64-byte) request followed by a minimum (64-byte) reply. In practice, overcoming the software overhead and getting the RPC time anywhere near there is a substantial achievement. It rarely happens in practice.

Similarly, the biggest problem in running at 1 Gbps is often getting the bits from the user's buffer out onto the network fast enough and having the receiving host process them as fast as they come in. If you double the host (CPU and memory) speed, you often can come close to doubling the throughput. Doubling the network capacity has no effect if the bottleneck is in the hosts.

Reduce Packet Count to Reduce Overhead

Each segment has a certain amount of overhead (e.g., the header) as well as data (e.g., the payload). Bandwidth is required for both components. Processing is also required for both components (e.g., header processing and doing the checksum). When 1 million bytes are being sent, the data cost is the same no matter what the segment size is. However, using 128-byte segments means 32 times as much per-segment overhead as using 4-KB segments. The bandwidth and processing overheads add up fast to reduce throughput.

Per-packet overhead in the lower layers amplifies this effect. Each arriving packet causes a fresh interrupt if the host is keeping up. On a modern pipelined processor, each interrupt breaks the CPU pipeline, interferes with the cache, requires a change to the memory management context, voids the branch prediction table, and forces a substantial number of CPU registers to be saved. An *n*-fold reduction in segments sent thus reduces the interrupt and packet overhead by a factor of *n*.

You might say that both people and computers are poor at multitasking. This observation underlies the desire to send MTU packets that are as large as will pass along the network path without fragmentation. Mechanisms such as Nagle's algorithm and Clark's solution are also attempts to avoid sending small packets.

Minimize Data Touching

The most straightforward way to implement a layered protocol stack is with one module for each layer. Unfortunately, this leads to copying (or at least accessing the data on multiple passes) as each layer does its own work. For example, after a packet is received by the NIC, it is typically copied to a kernel buffer. From there, it is copied to a network layer buffer for network layer processing, then to a transport layer buffer for transport layer processing, and finally to the receiving application process. It is not unusual for an incoming packet to be copied three or four times before the segment enclosed in it is delivered.

All this copying can greatly degrade performance because memory operations are an order of magnitude slower than register–register instructions. For example, if 20% of the instructions actually go to memory (i.e., are cache misses), which is likely when touching incoming packets, the average instruction execution time is slowed down by a factor of 2.8 ($0.8 \times 1 + 0.2 \times 10$). Hardware assistance will not help here. The problem is too much copying by the operating system.

A clever operating system will minimize copying by combining the processing of multiple layers. For example, TCP and IP are usually implemented together (as ''TCP/IP'') so that it is not necessary to copy the payload of the packet as processing switches from network to transport layer. Another common trick is to perform multiple operations within a layer in a single pass over the data. For example, checksums are often computed while copying the data (when it has to be copied) and the newly computed checksum is appended to the end.

Minimize Context Switches

A related rule is that context switches (e.g., from kernel mode to user mode) are deadly. They have the bad properties of interrupts and copying combined. This cost is why transport protocols are often implemented in the kernel. Like reducing packet count, context switches can be reduced by having the library procedure that sends data do internal buffering until it has a substantial amount of

them. Similarly, on the receiving side, small incoming segments should be collected together and passed to the user in one fell swoop instead of individually, to minimize context switches.

In the best case, an incoming packet causes a context switch from the current user to the kernel, and then a switch to the receiving process to give it the newly arrived data. Unfortunately, with some operating systems, additional context switches happen. For example, if the network manager runs as a special process in user space, a packet arrival is likely to cause a context switch from the current user to the kernel, then another one from the kernel to the network manager, followed by another one back to the kernel, and finally one from the kernel to the receiving process. This sequence is shown in Fig. 6-51. All these context switches on each packet are wasteful of CPU time and can have a devastating effect on network performance.

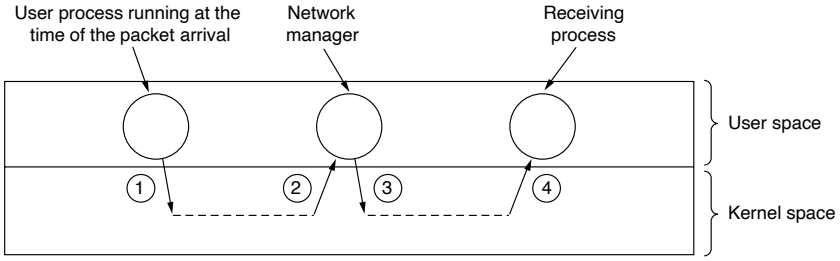


Figure 6-51. Four context switches to handle one packet with a user-space network manager.

Avoiding Congestion Is Better Than Recovering from It

The old maxim that an ounce of prevention is worth a pound of cure certainly holds for network congestion. When a network is congested, packets are lost, bandwidth is wasted, useless delays are introduced, and more. All of these costs are unnecessary, and recovering from congestion takes time and patience. Not having it occur in the first place is better. Congestion avoidance is like getting your DTP vaccination: it hurts a little at the time you get it, but it prevents something that would hurt a lot more in the future.

Avoid Timeouts

Timers are necessary in networks, but they should be used sparingly and timeouts should be minimized. When a timer goes off, some action is generally repeated. If it is truly necessary to repeat the action, so be it and do it, but repeating it unnecessarily is wasteful.

The way to avoid extra work is to be careful that timers are set a little bit on the conservative side. A timer that takes too long to expire adds a small amount of extra delay to one connection in the (unlikely) event of a segment being lost. A timer that goes off when it should not have used up host resources, wastes bandwidth, and puts extra load on perhaps dozens of routers for no good reason.

6.7.6 Fast Segment Processing

Now that we have covered general rules, we will look at some specific methods for speeding up segment processing. For more information, see Clark et al. (1989), and Chase et al. (2001).

Segment processing overhead has two components: overhead per segment and overhead per byte. Both must be attacked. The key to fast segment processing is to separate out the normal, successful case (one-way data transfer) and handle it specially. Many protocols tend to emphasize what to do when something goes wrong (e.g., a packet getting lost), but to make the protocols run fast, the designer should aim to minimize processing time when everything goes right. Minimizing processing time when an error occurs is secondary.

Although a sequence of special segments is needed to get into the *ESTABLISHED* state, once there, segment processing is straightforward until one side starts to close the connection. Let us begin by examining the sending side in the *ESTABLISHED* state when there are data to be transmitted. For the sake of clarity, we assume here that the transport entity is in the kernel, although the same ideas apply if it is a user-space process or a library inside the sending process. In Fig. 6-52, the sending process traps into the kernel to do the SEND. The first thing the transport entity does is test to see if this is the normal case: the state is *ESTABLISHED*, neither side is trying to close the connection, a regular (i.e., not an out-of-band) full segment is being sent, and enough window space is available at the receiver. If all conditions are met, no further tests are needed and the fast path through the sending transport entity can be taken. Typically, this path is taken most of the time.

In the usual case, the headers of consecutive data segments are almost the same. To take advantage of this fact, a prototype header is stored within the transport entity. At the start of the fast path, it is copied as fast as possible to a scratch buffer, word by word. Those fields that change from segment to segment are overwritten in the buffer. Frequently, these fields are easily derived from state variables, such as the next sequence number. A pointer to the full segment header plus a pointer to the user data are then passed to the network layer. Here, the same strategy can be followed (not shown in Fig. 6-52). Finally, the network layer gives the resulting packet to the data link layer for transmission.

As an example of how this principle works in practice, let us consider TCP/IP. Figure 6-53(a) shows the TCP header. The fields that are the same between consecutive segments on a one-way flow are shaded. All the sending transport entity

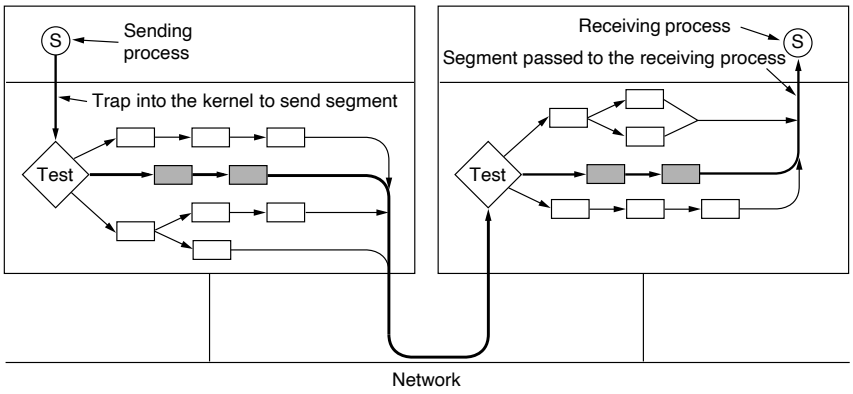


Figure 6-52. The fast path from sender to receiver is shown with a heavy line. The processing steps on this path are shaded.

has to do is copy the five words from the prototype header into the output buffer, fill in the next sequence number (by copying it from a word in memory), compute the checksum, and increment the sequence number in memory. It can then hand the header and data to a special IP procedure optimized for sending a regular, maximum segment. IP then copies its five-word prototype header [see Fig. 6-53(b)] into the buffer, fills the *Identification* field, and computes its checksum. The packet is now ready for transmission.

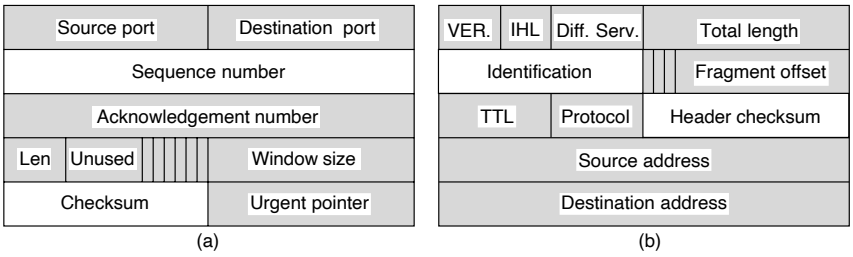


Figure 6-53. (a) TCP header. (b) IP header. In both cases, they are taken from the prototype without change.

Now let us look at fast path processing on the receiving side of Fig. 6-52. Step 1 is locating the connection record for the incoming segment. For TCP, the connection record can be stored in a hash table for which some simple function of the two IP addresses and two ports is the key. Once the connection record has been located, both addresses and both ports must be compared to verify that the correct record has been found.

An optimization that often speeds up connection record lookup even more is to maintain a pointer to the last one used and try that one first. Clark et al. (1989) tried this and observed a hit rate exceeding 90%.

The segment is checked to see if it is a normal one: the state is *ESTABLISHED*, neither side is trying to close the connection, the segment is a full one, no special flags are set, and the sequence number is the one expected. These tests take just a handful of instructions. If all conditions are met, a special fast path TCP procedure is called.

The fast path updates the connection record and copies the data to the user. While it is copying, it also computes the checksum, eliminating an extra pass over the data. If the checksum is correct, the connection record is updated and an acknowledgement is sent back. The general scheme of first making a quick check to see if the header is what is expected and then having a special procedure handle that case is called **header prediction**. Many TCP implementations use it. When this optimization and all the other ones discussed in this chapter are used together, it is possible to get TCP to run at 90% of the speed of a local memory-to-memory copy, assuming the network itself is fast enough.

Two other areas where substantial performance gains are possible are buffer management and timer management. The issue in buffer management is avoiding unnecessary copying, as mentioned above. Timer management is also important because nearly all timers set do not expire. They are set to guard against segment loss, but most segments and their acknowledgements arrive correctly. Hence, it is important to optimize timer management for the case of timers rarely expiring.

A common scheme is to use a linked list of timer events sorted by expiration time. The head entry contains a counter telling how many ticks away from expiry it is. Each successive entry contains a counter telling how many ticks after the previous entry it is. Thus, if timers expire in 3, 10, and 12 ticks, respectively, the three counters are 3, 7, and 2, respectively.

At every clock tick, the counter in the head entry is decremented. When it hits zero, its event is processed and the next item on the list becomes the head. Its counter does not have to be changed. This way, inserting and deleting timers are expensive operations, with execution times proportional to the length of the list.

A much more efficient approach can be used if the maximum timer interval is bounded and known in advance. Here, an array called a **timing wheel** can be used, as shown in Fig. 6-54. Each slot corresponds to one clock tick. The current time shown is $T = 4$. Timers are scheduled to expire at 3, 10, and 12 ticks from now. If a new timer suddenly is set to expire in seven ticks, an entry is just made in slot 11. Similarly, if the timer set for $T + 10$ has to be canceled, the list starting in slot 14 has to be searched and the required entry removed. Note that the array of Fig. 6-54 cannot accommodate timers beyond $T + 15$.

When the clock ticks, the current time pointer is advanced by one slot (circularly). If the entry now pointed to is nonzero, all of its timers are processed. Many variations on the basic idea are discussed by Varghese and Lauck (1987).

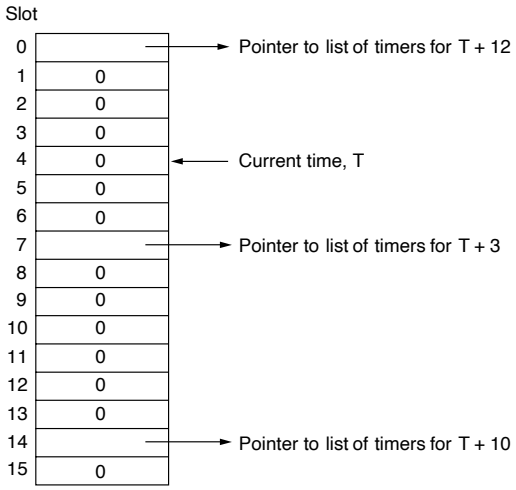


Figure 6-54. A timing wheel.

6.7.7 Header Compression

We have been looking at fast networks for too long. There is more out there. Let us now consider performance on wireless and other networks in which bandwidth is limited. Reducing software overhead can help mobile computers run more efficiently, but it does nothing to improve performance when the network links are the bottleneck.

To use bandwidth well, protocol headers and payloads should be carried with the minimum of bits. For payloads, this means using compact encodings of information, such as images that are in JPEG format rather than a bitmap, or document formats such as PDF that include compression. It also means application-level caching mechanisms, such as Web caches that reduce transfers in the first place.

What about for protocol headers? At the link layer, headers for wireless networks are typically compact because they were designed with scarce bandwidth in mind. For example, packets in connection-oriented networks have short connection identifiers instead of longer addresses. However, higher layer protocols such as IP, TCP and UDP come in one version for all link layers, and they are not designed with compact headers. In fact, streamlined processing to reduce software overhead often leads to headers that are not as compact as they could otherwise be (e.g., IPv6 has a more loosely packed headers than IPv4).

The higher-layer headers can be a significant performance hit. Consider, for example, voice-over-IP data that is being carried with the combination of IP, UDP,

and RTP. These protocols require 40 bytes of header (20 for IPv4, 8 for UDP, and 12 for RTP). With IPv6 the situation is even worse: 60 bytes, including the 40-byte IPv6 header. The headers can wind up as the majority of the transmitted data and consume more than half the bandwidth.

Header compression is used to reduce the bandwidth taken over links by higher-layer protocol headers. Specially designed schemes are used instead of general purpose methods. This is because headers are short, so they do not compress well individually, and decompression requires all prior data to be received. This will not be the case if a packet is lost.

Header compression obtains large gains by using knowledge of the protocol format. One of the first schemes was designed by Van Jacobson (1990) for compressing TCP/IP headers over slow serial links. It is able to compress a typical TCP/IP header of 40 bytes down to an average of 3 bytes. The trick to this method is hinted at in Fig. 6-53. Many of the header fields do not change from packet to packet. There is no need, for example, to send the same IP TTL or the same TCP port numbers in each and every packet. They can be omitted on the sending side of the link and filled in on the receiving side.

Similarly, other fields change in a predictable manner. For example, barring loss, the TCP sequence number advances with the data. In these cases, the receiver can predict the likely value. The actual number only needs to be carried when it differs from what is expected. Even then, it may be carried as a small change from the previous value, as when the acknowledgement number increases when new data is received in the reverse direction.

With header compression, it is possible to have simple headers in higher-layer protocols and compact encodings over low bandwidth links. **ROHC** (**RObust Header Compression**) is a modern version of header compression that is defined as a framework in RFC 5795. It is designed to tolerate the loss that can occur on wireless links. There is a profile for each set of protocols to be compressed, such as IP/UDP/RTP. Compressed headers are carried by referring to a context, which is essentially a connection; header fields may easily be predicted for packets of the same connection, but not for packets of different connections. In typical operation, ROHC reduces IP/UDP/RTP headers from 40 bytes to 1 to 3 bytes.

While header compression is mainly targeted at reducing bandwidth needs, it can also be useful for reducing delay. Delay is comprised of propagation delay, which is fixed given a network path, and transmission delay, which depends on the bandwidth and amount of data to be sent. For example, a 1-Mbps link sends 1 bit in 1 μsec. In the case of media over wireless networks, the network is relatively slow so transmission delay may be an important factor in overall delay and consistently low delay is important for quality of service.

Header compression can help by reducing the amount of data that is sent, and hence reducing transmission delay. The same effect can be achieved by sending smaller packets. This will trade increased software overhead for decreased transmission delay. Note that another potential source of delay is queueing delay to

access the wireless link. This can also be significant because wireless links are often heavily used as the limited resource in a network. In this case, the wireless link must have quality-of-service mechanisms that give low delay to real-time packets. Header compression alone is not sufficient.

6.7.8 Protocols for Long Fat Networks

Since the 1990s, there have been gigabit networks that transmit data over large distances. Because of the combination of a fast network, or ''fat pipe,'' and long delay, these networks are called **long fat networks**. When these networks arose, people's first reaction was to use the existing protocols on them, but various problems quickly arose. In this section, we will discuss some of the problems with scaling up the speed and delay of network protocols.

The first problem is that many protocols use 32-bit sequence numbers. When the Internet began, the lines between routers were mostly 56-kbps leased lines, so a host blasting away at full speed took over 1 week to cycle through the sequence numbers. To the TCP designers, 2^{32} was a pretty decent approximation of infinity because there was little danger of old packets still being around a week after they were transmitted. With 10-Mbps Ethernet, the wrap time became 57 minutes, much shorter, but still manageable. With a 1-Gbps Ethernet pouring data out onto the Internet, the wrap time is about 34 sec., well under the 120-sec maximum packet lifetime on the Internet. All of a sudden, 2^{32} is not nearly as good an approximation to infinity since a fast sender can cycle through the sequence space while old packets still exist.

The problem is that many protocol designers simply assumed, without stating it, that the time required to use up the entire sequence space would greatly exceed the maximum packet lifetime. Consequently, there was no need to even worry about the problem of old duplicates still existing when the sequence numbers wrapped around. At gigabit speeds, that unstated assumption fails. Fortunately, it proved possible to extend the effective sequence number by treating the timestamp that can be carried as an option in the TCP header of each packet as the high-order bits. This mechanism is called PAWS, as described earlier.

A second problem is that the size of the flow control window must be greatly increased. Consider, for example, sending a 64-KB burst of data from San Diego to Boston in order to fill the receiver's 64-KB buffer. Suppose that the link is 1 Gbps and the one-way speed-of-light-in-fiber delay is 20 msec. Initially, at $t = 0$, the pipe is empty, as illustrated in Fig. 6-55(a). Only 500 μsec later, in Fig. 6-55(b), all the segments are out on the fiber. The lead segment will now be somewhere in the vicinity of Brawley, still deep in Southern California. However, the transmitter must stop until it gets a window update.

After 20 msec, the lead segment hits Boston, as shown in Fig. 6-55(c), and is acknowledged. Finally, 40 msec after starting, the first acknowledgement gets back to the sender and the second burst can be transmitted. Since the transmission

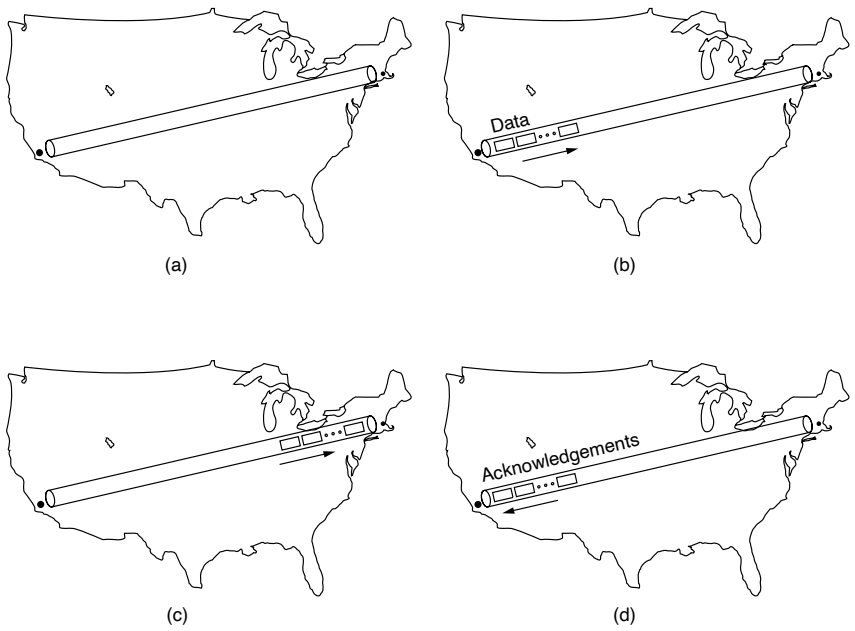


Figure 6-55. The state of transmitting 1 Mbit from San Diego to Boston. (a) At $t = 0$. (b) After 500 μsec. (c) After 20 msec. (d) After 40 msec.

line was used for 1.25 msec out of 100, the efficiency is about 1.25%. This situation is typical of an older protocols running over gigabit lines.

A useful quantity to keep in mind when analyzing network performance is the **bandwidth-delay product**. It is obtained by multiplying the bandwidth (in bits/sec) by the round-trip delay time (in sec). The product is the capacity of the pipe from the sender to the receiver and back (in bits).

For the example of Fig. 6-55, the bandwidth-delay product is 40 million bits. In other words, the sender would have to transmit a burst of 40 million bits to be able to keep going full speed until the first acknowledgement came back. It takes this many bits to fill the pipe (in both directions). This is why a burst of half a million bits only achieves a 1.25% efficiency: it is only 1.25% of the pipe's capacity.

The conclusion that can be drawn here is that for good performance, the receiver's window must be at least as large as the bandwidth-delay product, and preferably somewhat larger since the receiver may not respond instantly. For a transcontinental gigabit line, at least 5 MB are required.

A third and related problem is that simple retransmission schemes, such as the go-back-n protocol, perform poorly on lines with a large bandwidth-delay product.

Consider, the 1-Gbps transcontinental link with a round-trip transmission time of 40 msec. A sender can transmit 5 MB in one round trip. If an error is detected, it will be 40 msec before the sender is told about it. If go-back-n is used, the sender will have to retransmit not just the bad packet, but also the 5 MB worth of packets that came afterward. Clearly, this is a massive waste of resources. More complex protocols such as selective-repeat are needed.

A fourth problem is that gigabit lines are fundamentally different from megabit lines in that long gigabit lines are delay limited rather than bandwidth limited. In Fig. 6-56 we show the time it takes to transfer a 1-Mbit file 4000 km at various transmission speeds. At speeds up to 1 Mbps, the transmission time is dominated by the rate at which the bits can be sent. By 1 Gbps, the 40-msec round-trip delay dominates the 1 msec it takes to put the bits on the fiber. Further increases in bandwidth have hardly any effect at all.

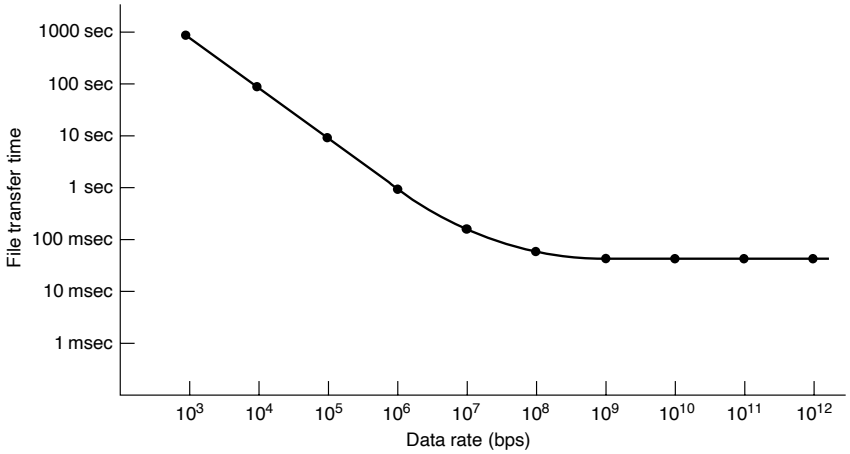


Figure 6-56. Time to transfer and acknowledge a 1-Mbit file over a 4000-km line.

Figure 6-56 has unfortunate implications for network protocols. It says that stop-and-wait protocols, such as RPC, have an inherent upper bound on their performance. This limit is dictated by the speed of light. No amount of technological progress in optics will ever improve matters (new laws of physics would help, though). Unless some other use can be found for a gigabit line while a host is waiting for a reply, the gigabit line is no better than a megabit line, just more expensive.

A fifth problem is that communication speeds have improved faster than computing speeds. (Note to computer engineers: go out and beat those communication engineers! We are counting on you.) In the 1970s, the ARPANET ran at 56 kbps

and had computers that ran at something like 1 MIPS. Compare these numbers to 1000-MIPS computers exchanging packets over a 1-Gbps line. The number of instructions per byte has decreased by more than a factor of 10. The exact numbers are debatable depending on dates and scenarios, but the conclusion is this: there is less time available for protocol processing than there used to be, so protocols must become simpler.

Let us now turn from the problems to ways of dealing with them. The basic principle that all high-speed network designers should learn by heart is:

Design for speed, not for bandwidth optimization.

Old protocols were often designed to minimize the number of bits on the wire, frequently by using small fields and packing them together into bytes and words. This concern is still valid for wireless networks, but not for gigabit networks. Protocol processing is the problem, so protocols should be designed to minimize it. The IPv6 designers clearly understood this principle.

A tempting way to go fast is to build fast network interfaces in hardware. The difficulty with this strategy is that unless the protocol is exceedingly simple, hardware just means a plug-in board with a second CPU and its own program. To make sure the network coprocessor is cheaper than the main CPU, it is often a slower chip. The consequence of this design is that much of the time the main (fast) CPU is idle waiting for the second (slow) CPU to do the critical work. It is a myth to think that the main CPU has other work to do while waiting. Furthermore, when two general-purpose CPUs communicate, race conditions can occur, so elaborate protocols are needed between the two processors to synchronize them correctly and avoid races. Usually, the best approach is to make the protocols simple and have the main CPU do the work.

Packet layout is an important consideration in gigabit networks. The header should contain as few fields as possible, to reduce processing time, and these fields should be big enough to do the job and be word-aligned for fast processing. In this context, ''big enough'' means that problems such as sequence numbers wrapping around while old packets still exist, receivers being unable to advertise enough window space because the window field is too small, etc. do not occur.

The maximum data size should be large, to reduce software overhead and permit efficient operation. For high-speed networks, 1500 bytes is too small, which is why gigabit Ethernet supports jumbo frames of up to 9 KB and IPv6 supports jumbogram packets in excess of 64 KB.

Let us now look at the issue of feedback in high-speed protocols. Due to the (relatively) long delay loop, feedback should be avoided if at all possible: it takes too long for the receiver to signal the sender. One example of feedback is governing the transmission rate by using a sliding window protocol. Future protocols may switch to rate-based protocols to avoid the (long) delays inherent in the receiver sending window updates to the sender. In such a protocol, the sender can

send all it wants to, provided it does not send faster than some rate the sender and receiver have agreed upon in advance.

A second example of feedback is Jacobson's slow start algorithm. This algorithm makes multiple probes to see how much the network can handle. With high-speed networks, making half a dozen or so small probes to see how the network responds wastes a huge amount of bandwidth. A more efficient scheme is to have the sender, receiver, and network all reserve the necessary resources at connection setup time. Reserving resources in advance also has the advantage of making it easier to reduce jitter. In short, going to high speeds inexorably pushes the design toward connection-oriented operation, or something fairly close to it.

Another valuable feature is the ability to send a normal amount of data along with the connection request. In this way, one round-trip time can be saved.

6.8 SUMMARY

The transport layer is the key to understanding layered protocols. It provides various services, the most important of which is an end-to-end, reliable, connection-oriented byte stream from sender to receiver. It is accessed through service primitives that permit the establishment, use, and release of connections. A common transport layer interface is the one provided by Berkeley sockets.

Transport protocols must be able to do connection management over unreliable networks. Connection establishment is complicated by the existence of delayed duplicate packets that can reappear at inopportune moments. To deal with them, three-way handshakes are needed to establish connections. Releasing a connection is easier than establishing one but is still far from trivial due to the two-army problem.

Even when the network layer is completely reliable, the transport layer has plenty of work to do. It must handle all the service primitives, manage connections and timers, allocate bandwidth with congestion control, and run a variable-sized sliding window for flow control.

Congestion control should allocate all of the available bandwidth between competing flows fairly, and it should track changes in the usage of the network. The AIMD control law converges to a fair and efficient allocation.

The Internet has two main transport protocols: UDP and TCP. UDP is a connectionless protocol that is mainly a wrapper for IP packets with the additional feature of multiplexing and demultiplexing multiple processes using a single IP address. UDP can be used for client-server interactions, for example, using RPC. It can also be used for building real-time protocols such as RTP.

The main Internet transport protocol is TCP. It provides a reliable, bidirectional, congestion-controlled byte stream with a 20-byte header on all segments. A great deal of work has gone into optimizing TCP performance, using algorithms from Nagle, Clark, Jacobson, Karn, and others.

UDP and TCP have survived over the years very well, but there is still room for improvement to enhance performance and solve problems caused by modern high-speed networks. TCP CUBIC, QUIC, and BBR are a few of the modern improvements.

Network performance is typically dominated by protocol and segment processing overhead, and this situation gets worse at higher speeds. Protocols should be designed to minimize the number of segments and work for large bandwidth-delay paths. For gigabit networks, simple protocols and streamlined processing work best.

PROBLEMS

1. In our example transport primitives of Fig. 6-2, LISTEN is a blocking call. Is this strictly necessary? If not, explain how a nonblocking primitive could be used. What advantage would this have over the scheme described in the text?

2. Primitives of the transport service assume asymmetry between the two end points during connection establishment: one end (server) executes LISTEN while the other end (client) executes CONNECT. However, in peer-to-peer applications such file sharing systems, e.g. BitTorrent, all end points are peers. There is no server or client functionality. How can transport service primitives be used to build such peer-to-peer applications?

3. A chat application using TCP repeatedly calls receive( ), and prints the received data as a new message. Can you think of a problem with this approach?

4. In the underlying model of Fig. 6-4, it is assumed that packets may be lost by the network layer and thus must be individually acknowledged. Suppose that the network layer is 100 percent reliable and never loses packets. What changes, if any, are needed to Fig. 6-4?

5. In both parts of Fig. 6-6, there is a comment that the value of *SERVER_PORT* must be the same in both client and server. Why is this so important?

6. In the Internet File Server example (Fig. 6-6), can the connect( ) system call on the client fail for any reason other than listen queue being full on the server? Assume that the network is perfect.

7. One criteria for deciding whether to have a server active all the time or have it start on demand using a process server is how frequently the service provided is used. Can you think of any other criteria for making this decision?

8. Suppose that the clock-driven scheme for generating initial sequence numbers is used with a 15-bit wide clock counter. The clock ticks once every 100 msec, and the maximum packet lifetime is 60 sec. How often need resynchronization take place

(a) in the worst case?
(b) when the data consumes 240 sequence numbers/min?

9. How would the following scenarios affect Fig. 6-10(b)?
(a) The number of bits used for the clock/sequence number increases.
(b) Maximum packet lifetime increases.
(c) Clock tick-rate increases.
Sketch a new figure for each scenario. Explain what happens.

10. Why does the maximum packet lifetime, *T*, have to be large enough to ensure that not only the packet but also its acknowledgements have vanished?

11. Consider a connection-oriented transport layer protocol that uses a time-of-day clock to determine packet sequence numbers. The clock uses an 9-bit counter, and ticks once every 250 msec. The maximum packet lifetime is 32 seconds. If the sender sends 3 packets per second, how long could the connection last without entering the forbidden region?

12. Imagine that a two-way handshake rather than a three-way handshake were used to set up connections. In other words, the third message was not required. Are deadlocks now possible? Give an example or show that none exist.

13. Imagine a generalized *n*-army problem, in which the agreement of any two of the blue armies is sufficient for victory. Does a protocol exist that allows blue to win?

14. Consider the problem of recovering from host crashes (i.e., Fig. 6-18). If the interval between writing and sending an acknowledgement, or vice versa, can be made relatively small, what are the two best sender-receiver strategies for minimizing the chance of a protocol failure?

15. In Figure 6-20, suppose that a new flow *E* is added that takes a path from *R1* to *R2* to *R6*. How does the max-min bandwidth allocation change for the five flows?

16. In Fig. 6-20, suppose the flows are rearranged such that *A* goes through *R*1, *R*2, and *R*3, *B* goes through *R*1, *R*5, and *R*6, *C* goes through *R*4, *R*5, and *R*6, and *D* goes through *R*4, *R*5, and *R*6. What is the max-min bandwidth allocation?

17. Discuss the advantages and disadvantages of credits versus sliding window protocols.

18. Some other policies for fairness in congestion control are Additive Increase Additive Decrease (AIAD), Multiplicative Increase Additive Decrease (MIAD), and Multiplicative Increase Multiplicative Decrease (MIMD). Discuss these three policies in terms of convergence and stability.

19. Consider a transport-layer protocol that uses Additive Increase Square Root Decrease (AISRD). Does this version converge to fair bandwidth sharing?

20. Two hosts simultaneously send data through a network with a capacity of 1 Mbps. Host A uses UDP and transmits a 100 bytes packet every 1 msec. Host B generates data with a rate of 600 kbps and uses TCP. Which host will obtain higher throughput?

21. Why does UDP exist? Would it not have been enough to just let user processes send raw IP packets?

22. Consider a simple application-level protocol built on top of UDP that allows a client to retrieve a file from a remote server residing at a well-known address. The client first

sends a request with a file name, and the server responds with a sequence of data packets containing different parts of the requested file. To ensure reliability and sequenced delivery, client and server use a stop-and-wait protocol. Ignoring the obvious performance issue, do you see a problem with this protocol? Think carefully about the possibility of processes crashing.

23. Both UDP and TCP use port numbers to identify the destination entity when delivering a message. Give two reasons why these protocols invented a new abstract ID (port numbers), instead of using process IDs, which already existed when these protocols were designed.

24. Several RPC implementations provide an option to the client to use RPC implemented over UDP or RPC implemented over TCP. Under what conditions will a client prefer to use RPC over UDP and under what conditions will he prefer to use RPC over TCP?

25. Consider two networks, $N1$ and $N2$, that have the same average delay between a source A and a destination D. In $N1$, the delay experienced by different packets is uniformly distributed with maximum delay being 10 seconds, while in $N2$, 99% of the packets experience less than one second delay with no limit on maximum delay. Discuss how RTP may be used in these two cases to transmit live audio/video stream.

26. What is the total size of the minimum TCP MTU, including TCP and IP overhead but not including data link layer overhead?

27. Datagram fragmentation and reassembly are handled by IP and are invisible to TCP. Does this mean that TCP does not have to worry about data arriving in the wrong order?

28. RTP is used to transmit CD-quality audio, which makes a pair of 16-bit samples 44,100 times/sec, one sample for each of the stereo channels. How many packets per second must RTP transmit?

29. Would it be possible to place the RTP code in the operating system kernel, along with the UDP code? Explain your answer.

30. A process on host 1 has been assigned port p, and a process on host 2 has been assigned port q. Is it possible for there to be two or more TCP connections between these two ports at the same time?

31. In Fig. 6-36, we saw that in addition to the 32-bit *acknowledgement* field, there is an *ACK* bit in the fourth word. Does this really add anything? Why or why not?

32. The maximum payload of a TCP segment is 65,495 bytes. Why was such a strange number chosen?

33. Consider a TCP connection that is sending data at such a high rate that it starts reusing sequence numbers within the maximum segment lifetime. Can this be prevented by increasing the segment size? Why (not)?

34. Describe two ways to get into the *SYN RCVD* state of Fig. 6-39.

35. You are playing an online game over a high-latency network. The game requires you

to quickly tap objects on the screen. However, the game only shows the result of your actions in bursts. Could this behavior be caused by a TCP option? Can you think of another (network-related) cause?

36. Consider the effect of using slow start on a line with a 10-msec round-trip time and no congestion. The receive window is 24 KB and the maximum segment size is 2 KB. How long does it take before the first full window can be sent?

37. Suppose that the TCP congestion window is set to 18 KB and a timeout occurs. How big will the window be if the next four transmission bursts are all successful? Assume that the maximum segment size is 1 KB.

38. Consider a connection that uses TCP Reno. The connection has an initial congestion window size of 1 KB, and an initial threshold of 64. Assume that additive increase uses a step-size of 1 KB. What is the size of the congestion window in transmission round 8, if the first transmission round is number 0?

39. If the TCP round-trip time, *RTT*, is currently 30 msec and the following acknowledgements come in after 26, 32, and 24 msec, respectively, what is the new *RTT* estimate using the Jacobson algorithm? Use $\alpha = 0.9$.

40. A TCP machine is sending full windows of 65,535 bytes over a 1-Gbps channel that has a 10-msec one-way delay. What is the maximum throughput achievable? What is the line efficiency?

41. To address the limitations of IP version 4, a major effort had to be undertaken via IETF that resulted in the design of IP version 6 and there are still is significant reluctance in the adoption of this new version. However, no such major effort is needed to address the limitations of TCP. Explain why this is the case.

42. In a network whose max segment is 128 bytes, max segment lifetime is 30 sec, and has 8-bit sequence numbers, what is the maximum data rate per connection?

43. Consider a TCP connection that uses a maximum segment lifetime of 128 seconds. Assume that the connection uses the timestamp option, with the timestamp increasing once per second. What can you say about the maximum data rate?

44. Suppose that you are measuring the time to receive a segment. When an interrupt occurs, you read out the system clock in milliseconds. When the segment is fully processed, you read out the clock again. You measure 0 msec 270,000 times and 1 msec 730,000 times. How long does it take to receive a segment?

45. A CPU executes instructions at the rate of 1000 MIPS. Data can be copied 64 bits at a time, with each word copied costing 10 instructions. If an coming packet has to be copied four times, can this system handle a 1-Gbps line? For simplicity, assume that all instructions, even those instructions that read or write memory, run at the full 1000-MIPS rate.

46. To get around the problem of sequence numbers wrapping around while old packets still exist, one could use 64-bit sequence numbers. However, theoretically, an optical fiber can run at 75 Tbps. What maximum packet lifetime is required to make sure that

future 75-Tbps networks do not have wraparound problems even with 64-bit sequence numbers? Assume that each byte has its own sequence number, as TCP does.

47. Consider a 1000 MIPS computer than can execute one instruction per nanosecond. Suppose that it takes 50 instructions to process a packet header, independent of the payload size and 10 instructions for each 8 bytes of payload. How many packets per second can it process if the packets are (a) 128 bytes and (b) 1024 bytes? What is the goodput in bytes/sec in both cases?

48. For a 1-Gbps network operating over 4000 km, the delay is the limiting factor, not the bandwidth. Consider a MAN with the average source and destination 20 km apart. At what data rate does the round-trip delay due to the speed of light equal the transmission delay for a 1-KB packet?

49. What is the bandwidth-delay product for a 50-Mbps channel on a geostationary satellite? If the packets are all 1500 bytes (including overhead), how big should the window be in packets?

50. Name some of the possible causes that a client-based speed test of an access network might not measure the true speed of the access link

51. Consider the TCP header in Fig. 6-36. Every time a TCP segment is sent, it includes 4 unused bits. How does removing these bits, and shifting all subsequent fields four bits to the left, affect performance?

52. The file server of Fig. 6-6 is far from perfect and could use a few improvements. Make the following modifications.

(a) Give the client a third argument that specifies a byte range.
(b) Add a client flag `-w` that allows the file to be written to the server.

53. One common function that all network protocols need is to manipulate messages. Recall that protocols manipulate messages by adding/striping headers. Some protocols may break a single message into multiple fragments, and later join these multiple fragments back into a single message. To this end, design and implement a message management library that provides support for creating a new message, attaching a header to a message, stripping a header from a message, breaking a message into two messages, combining two messages into a single message, and saving a copy of a message. Your implementation must minimize data copying from one buffer to another as much as possible. It is critical that the operations that manipulate messages do not touch the data in a message, but rather, only manipulate pointers.

54. Design and implement a chat system that allows multiple groups of users to chat. A chat coordinator resides at a well-known network address, uses UDP for communication with chat clients, sets up chat servers for each chat session, and maintains a chat session directory. There is one chat server per chat session. A chat server uses TCP for communication with clients. A chat client allows users to start, join, and leave a chat session. Design and implement the coordinator, server, and client code.