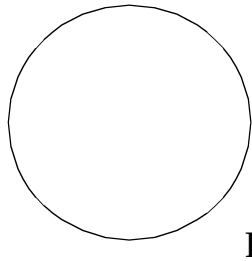
46

TRANSMISSION CONTROL PROTOCOL (TCP) FUNDAMENTALS AND GENERAL OPERATION



Many people have a difficult time understanding how the Transmission Control Protocol (TCP) works. After spending dozens of hours writing almost 100 pages on the protocol, I am quite sympathetic! I think a main reason for the difficulty is that many descriptions of the protocol leap too quickly from a brief introduction straight into the mind-boggling details of TCP's operation. The problem is that TCP works in a very particular way. Its operation is built around a few very important fundamentals that you absolutely must understand before the details of TCP operation will make much sense.

In this chapter, I describe some of the key operating fundamentals of TCP. I begin with a discussion of how TCP handles data and introduce the concepts of streams, segments, and sequences. I then describe the very important TCP sliding window system, which is used for acknowledgment, reliability, and data flow control. I discuss how TCP uses ports and how it identifies connections. I also describe the most important applications that use TCP and what ports they use for server applications.

TCP Data Handling and Processing

One of the givens in the operation of most of the protocols you'll find at upper layers in the OSI Reference Model is that the protocols are oriented around the use of messages. These messages are analogous to a written letter in an envelope that contains a specific piece of information. They are passed from higher layers down to lower ones, where they are encapsulated in the lower layers' headers (like putting them in another envelope), and then passed down further until they are actually sent out at the physical layer.

You can see a good example of this by looking at the User Datagram Protocol (UDP), TCP's transport layer peer. To use UDP, an application passes it a distinct block of data that is usually fairly short. The block is packaged into a UDP message, then sent to the Internet Protocol (IP). IP packs the message into an IP datagram and eventually passes it to a layer 2 protocol such as Ethernet. There, IP places it into a frame and sends it to layer 1 for transmission.

Increasing the Flexibility of Application Data Handling: TCP's Stream Orientation

The use of discrete messaging is pretty simple, and it obviously works well enough since most protocols make use of it. However, it is inherently limiting because it forces applications to create discrete blocks of data in order to communicate. There are many applications that need to send information continuously in a manner that doesn't lend itself well to creating "chunks" of data. Others need to send data in chunks that are so large that applications could never send them as a single message at the lower layers.

To use a protocol like UDP, many applications would be forced to artificially divide their data into messages of a size that has no inherent meaning to them. This would immediately introduce new problems that would require more work for the application. The application would have to keep track of what data is in what message, and replace any data that was lost. It would need to ensure that the messages could be reassembled in the correct order, since IP might deliver them out of order.

Of course, you could program applications to do this, but it would make little sense, because these functions are already ones that TCP is charged with handling. Instead, the TCP designers took the very smart approach of generalizing TCP so that it could accept application data of any size and structure without requiring the data to be in discrete pieces. More specifically, TCP treats data coming from an application as a *stream*—thus, the description of TCP as *stream-oriented*. Each application sends the data it wishes to transmit as a steady stream of octets (bytes). The application doesn't need to carve the data into blocks or worry about how lengthy streams will get across the internetwork. It just "pumps bytes" to TCP.

TCP Data Packaging: Segments

TCP must take the bytes it gets from an application and send them using a network layer protocol, which is IP in this case. IP is a message-oriented protocol; it is not stream-oriented. Thus, we have simply "passed the buck" to TCP, which must take the stream from the application and divide it into discrete messages for IP. These messages are called *TCP segments*.

NOTE Segment *is one of the most confusing data structure names in the world of networking. From a dictionary definition standpoint, referring to a piece of a stream as a* segment *is sensible, but most people working with networks don't think of a message as being a segment. In the industry, the term also refers to a length of cable or a part of a local area network (LAN), among other things, so watch out for that.*

IP treats TCP segments like all other discrete messages for transmission. IP places them into IP datagrams and transmits them to the destination device. The recipient unpackages the segments and passes them to TCP, which converts them back to a byte stream in order to send them to the application. This process is illustrated in Figure 46-1.

KEY CONCEPT TCP is designed to have applications send data to it as a stream of bytes, rather than requiring fixed-size messages to be used. This provides maximum flexibility for a wide variety of uses, because applications don't need to worry about data packaging and can send files or messages of any size. TCP takes care of packaging these bytes into messages called *segments*.

The TCP layer on a device accumulates data that it receives from the application process stream. On regular intervals, the TCP layer forms segments that it will transmit using IP. Two primary factors control the size of the segment. First, there is an overall limit to the size of a segment, chosen to prevent unnecessary fragmentation at the IP layer. A parameter called the *maximum segment size (MSS)* governs this size limit. The MSS is determined during connection establishment. Second, TCP is designed so that once a connection is set up, each of the devices tells the other how much data it is ready to accept at any given time. If the data is lower than the MSS value, the device must send a smaller segment. This is part of the sliding window system described a little later in this chapter.

TCP Data Identification: Sequence Numbers

The fact that TCP treats data coming from an application as a stream of octets has a couple of very significant implications for the operation of the protocol. The first is related to data identification. Since TCP is reliable, it needs to keep track of all the data it receives from an application so it can make sure that the destination receives all the data. Furthermore, TCP must make sure that the destination receives the data in the order that the application sent it, and the destination must retransmit any lost data.

If a device conveyed data to TCP in block-like messages, it would be fairly simple to keep track of the data by adding an identifier to each message. Because TCP is stream-oriented, however, that identification must be done for each byte of data! This may seem surprising, but it is actually what TCP does through the use of sequence numbers. Each byte of data is assigned a sequence number that is used to keep track of it through the process of transmission, reception, and acknowledgment (though in practice, blocks of many bytes are managed using the sequence numbers of bytes at the start and end of the block). These sequence numbers are used to ensure that the sending application transmits and reassembles the segmented data into the original stream of data. The sequence numbers are required to implement the sliding window system, which enables TCP to provide reliability and data flow control.

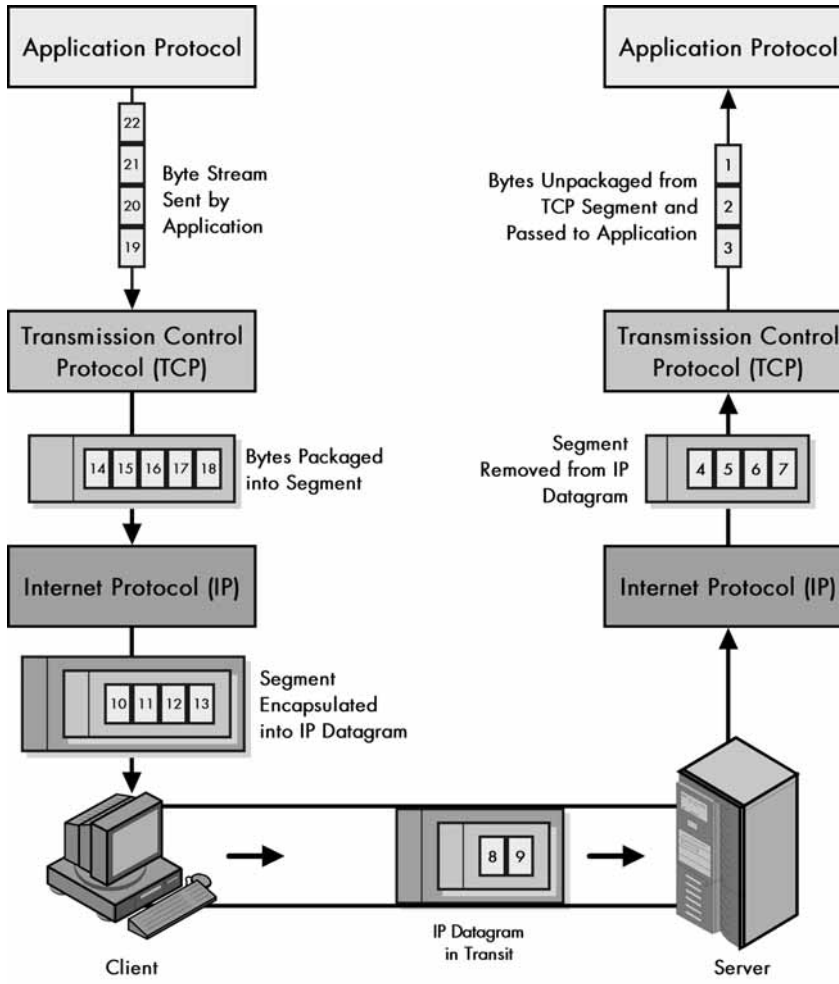


Figure 46-1: TCP data stream processing and segment packaging *TCP is different from most protocols because it does not require applications that use it to send data to it in messages. Once a TCP connection is set up, an application protocol can send TCP a steady stream of bytes that does not need to conform to any particular structure. TCP packages these bytes into segments that are sized based on a number of different parameters. These segments are passed to IP, where they are encapsulated into IP datagrams and transmitted. The receiving device reverses the process: Segments are removed from IP datagrams, and then the bytes are taken from the segments and passed up to the appropriate recipient application protocol as a byte stream.*

KEY CONCEPT Since TCP works with individual bytes of data rather than discrete messages, it must use an identification scheme that works at the byte level to implement its data transmission and tracking system. This is accomplished by assigning a sequence number to each byte that TCP processes.

The Need for Application Data Delimiting

When TCP treats incoming data as a stream, the data the application using TCP receives is called *unstructured*. For transmission, a stream of data goes into TCP on the sending device, and on reception, a stream of data goes back to the application on the receiving device. Even though TCP breaks the stream into segments for transmission, these segments are TCP-level details that remain hidden from the application. When a device wants to send multiple pieces of data, TCP provides no mechanism for indicating where the dividing line is between the pieces, since TCP doesn't examine the meaning of the data. The application must provide a means for doing this.

Consider, for example, an application that is sending database records. It needs to transmit record 579 from the Employees database table, followed by record 581 and record 611. It sends these records to TCP, which treats them all collectively as a stream of bytes. TCP will package these bytes into segments, but in a way that the application cannot predict. It is possible that each byte will end up in a different segment, but more likely that they will all be in one segment, or that part of each will end up in different segments, depending on their length. The records must have some sort of explicit markers so that the receiving device can tell where one record ends and the next starts.

KEY CONCEPT Since applications send data to TCP as a stream of bytes as opposed to prepackaged messages, each application must use its own scheme to determine where one application data element ends and the next begins.

TCP Sliding Window Acknowledgment System

What differentiates TCP from simpler transport protocols like UDP is the quality of the manner in which it sends data between devices. Rather than just sticking data in a message and saying, "off you go," TCP carefully keeps track of the data it sends. This management of data is required to facilitate the following two key requirements of the protocol:

Reliability Ensuring that data that is sent actually arrives at its destination, and if it doesn't arrive, detecting this and resending it.

Data Flow Control Managing the rate at which data is sent so that it does not overwhelm the device that is receiving it.

To accomplish these tasks, the entire operation of the protocol is oriented around something called the *sliding window acknowledgment system*. It is no exaggeration to say that comprehending how sliding windows work is critical to understanding just about everything else in TCP. It is also, unfortunately, a bit hard to follow if you try to grasp it all at once. I wanted to make sure that I explained the mechanism thoroughly without assuming that you already understood it. For this reason, I am going to start by explaining the concepts behind sliding windows, particularly how the technique works and why it is so powerful.

The Problem with Unreliable Protocols: Lack of Feedback

A simple "send and forget" protocol like IP is unreliable and includes no flow control for one main reason: It is an open-loop system in which the transmitter receives no feedback from the recipient. (I am ignoring error reports using ICMP and the like for the purpose of this discussion.) A datagram is sent, and it may or may not get there, but the transmitter will never have any way of knowing because there is no mechanism for feedback. This concept is illustrated in Figure 46-2.

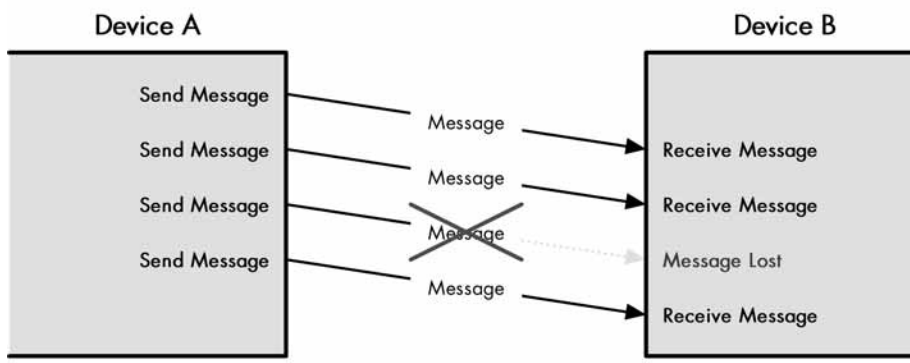


Figure 46-2: Operation of an unreliable protocol *In a system such as the one that IP uses, if a message gets to its destination, that's great; otherwise, nobody will have a clue. Some external mechanism is needed to take care of the lost message, unless the protocol doesn't really care whether a few bits and pieces are missing from its message stream.*

Providing Basic Reliability Using Positive Acknowledgment with Retransmission (PAR)

Basic reliability in a protocol running over an unreliable protocol like IP can be implemented by closing the loop so the recipient provides feedback to the sender. This is most easily done with a simple acknowledgment system. Device A sends a piece of data to Device B, which receives the data and sends back an acknowledgment saying, "Device A, I received your message." Device A then knows its transmission was successful.

But since IP is unreliable, that message may in fact never get to where it is going. Device A will sit and wait for the acknowledgment and never receive it. Conversely, it is also possible that Device B gets the message from Device A, but the acknowledgment itself vanishes somehow. In either case, we don't want Device A to sit forever waiting for an acknowledgment that is never going to arrive.

To prevent this from happening, Device A starts a timer when it first sends the message to Device B, which allows sufficient time for the message to get to Device B and for the acknowledgment to travel back, plus some reasonable time to allow for possible delays. If the timer expires before the acknowledgment is received, Device A assumes that there was a problem and retransmits its original message. Since this method involves positive acknowledgments ("Yes, I got your message") and a facility for retransmission when needed, it is commonly called *positive acknowledgment with retransmission (PAR)*, as shown in Figure 46-3.

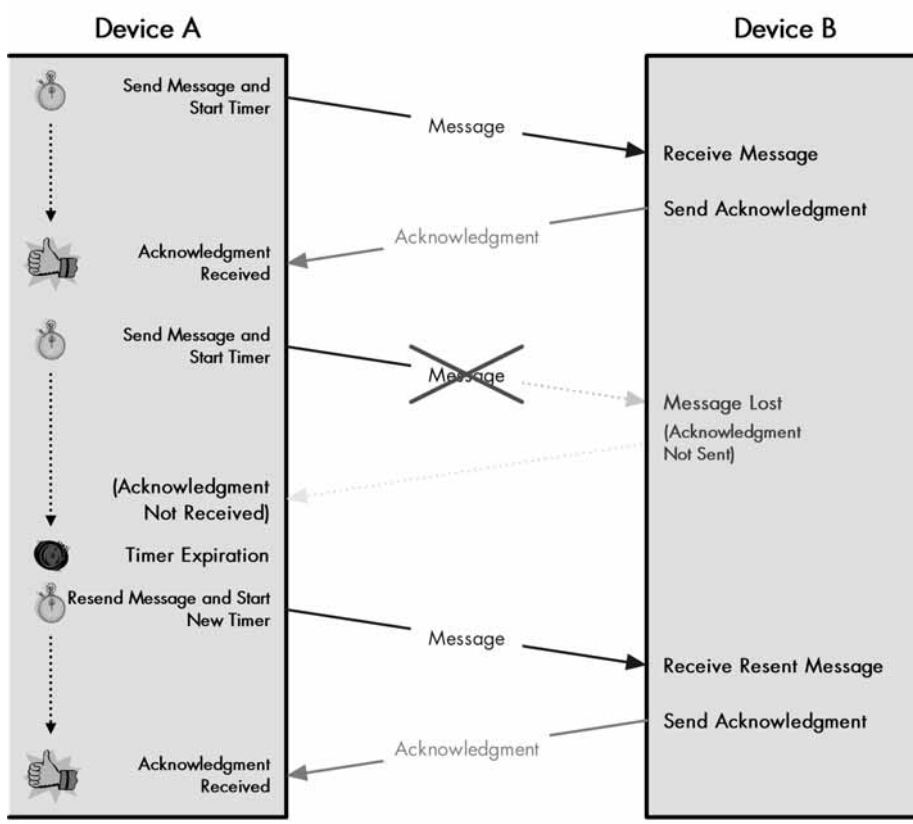


Figure 46-3: Basic reliability: positive acknowledgment with retransmission (PAR) *This diagram shows one of the most common and simple techniques for ensuring reliability. Each time Device A sends a message, it starts a timer. Device B sends an acknowledgment back to Device A when it receives a message, so that Device A knows that it successfully transmitted the message. If a message is lost, the timer goes off, and Device A retransmits the data. Note that only one message can be outstanding at any time, making this system rather slow.*

KEY CONCEPT A basic technique for ensuring reliability in communications uses a rule that requires a device to send back an acknowledgment each time it successfully receives a transmission. If a device doesn't acknowledge the transmission after a period of time, its sender retransmits the acknowledgment. This system is called *positive acknowledgment with retransmission (PAR)*. One drawback with this basic scheme is that the transmitter cannot send a second message until after the first device has acknowledged the first.

PAR is a technique that is used widely in networking and communications for protocols that exchange relatively small amounts of data, or protocols that exchange data infrequently. The basic method is functional, but it is not well suited to a protocol like TCP. One main reason is that it is *inefficient*. Device A sends a message, and then waits for the acknowledgment. Device A cannot send another message to Device B until it hears that Device B received its original message, which is very wasteful and would make the protocol extremely slow.

Improving PAR

The first improvement we can make to the PAR system is to provide some means of identification to the messages that were sent, as well as the acknowledgments. For example, we could put a message ID field in the message header. The device sending the message would uniquely identify it, and the recipient would use this identifier in the acknowledgment. For example, Device A might send a piece of data in a message with the message ID 1. Device B would receive the message and then send its own message back to Device A, saying "Device A, I received your message 1." The advantage of this system is that Device A can send multiple messages at once. It must keep track of each one that it sends, and whether or not Device B sent an acknowledgment. Each device also requires a separate timer, but that's not a big problem.

Of course, we also need to consider this exchange from the standpoint of Device B. Before, Device B had to deal with only one message at a time from Device A. Now it may have several show up all at once. What if it is already busy with transmissions from another device (or ten)? We need some mechanism that lets Device B say, "I am only willing to handle the following number of messages from you at a time." We could do that by having the acknowledgment message contain a field, such as send limit, which specifies the maximum number of unacknowledged messages Device A was allowed to have in transit to Device B at one time.

Device A would use this send limit field to restrict the rate at which it sent messages to Device B. Device B could adjust this field depending on its current load and other factors to maximize performance in its discussions with Device A. This enhanced system would thus provide reliability, efficiency, and basic data flow control, as illustrated in Figure 46-4.

KEY CONCEPT The basic PAR reliability scheme can be enhanced by identifying each message to be sent, so multiple messages can be in transit at once. The use of a send limit allows the mechanism to also provide flow control capabilities, by allowing each device to control the rate at which other devices send data to it.

TCP's Stream-Oriented Sliding Window Acknowledgment System

So does TCP use this variation on PAR? Of course not! That would be too simple. Conceptually, the TCP sliding window system is very similar to this method, which is why it is important that you understand it. However, it requires some adjustment. The main reason has to do with the way TCP handles data: the matter of stream orientation compared to message orientation discussed earlier in this chapter. The technique I have outlined involves explicit acknowledgments and (if necessary) retransmissions for messages. Thus, it would work well for a protocol that exchanged reasonably large messages on a fairly infrequent basis.

TCP, on the other hand, deals with individual bytes of data as a stream. Transmitting each byte one at a time and acknowledging each one at a time would quite obviously be absurd. It would require too much work, and even with overlapped transmissions (that is, not waiting for an acknowledgment before sending the next piece of data), the result would be horribly slow.

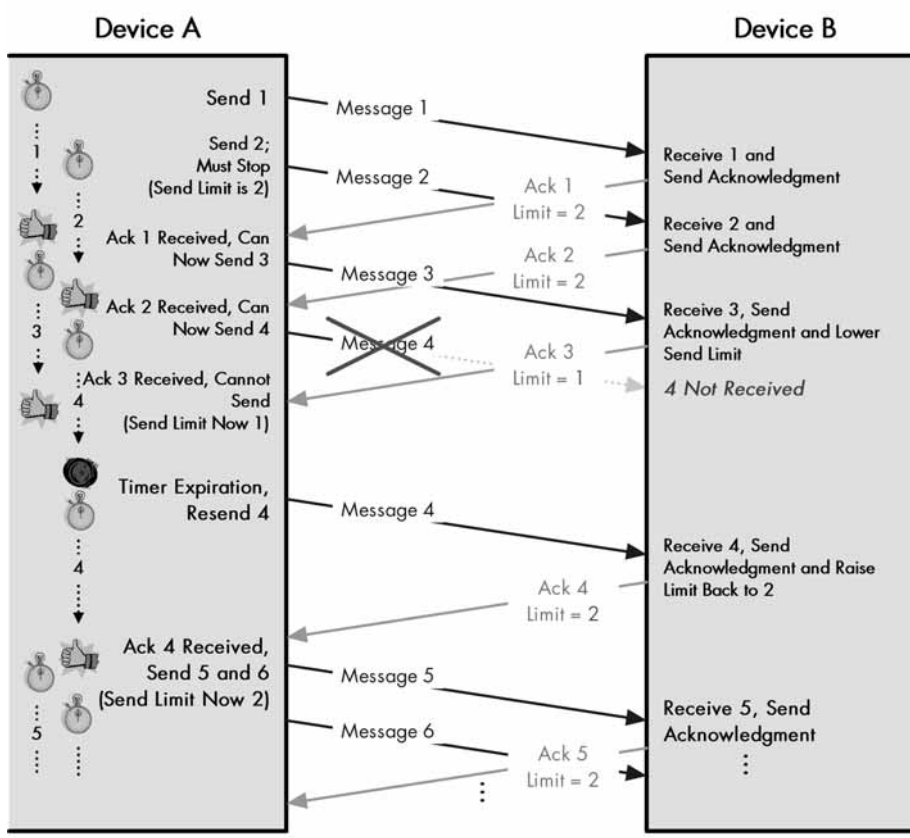


Figure 46-4: Enhanced PAR *This diagram shows two enhancements to the basic PAR scheme from Figure 46-3. First, each message now has an identification number; each can be acknowledged individually, so more than one message can be in transit at a given time. Second, Device B regularly communicates to Device A a send limit parameter, which restricts the number of messages Device A can have outstanding at once. Device B can adjust this parameter to control the flow of data from Device A.*

This slowness is why TCP does not send bytes individually but divides them into segments. All of the bytes in a segment are sent together and received together, and thus acknowledged together. TCP uses a variation on the method I described earlier, in which the sequence numbers I discussed earlier identify the data sent and acknowledged. Instead of acknowledging the use of something like a message ID field, we acknowledge data using the sequence number of the last byte of data in the segment. Thus, we are dealing with a range of bytes in each case, and the range represents the sequence numbers of all the bytes in the segment.

Conceptual Division of TCP Transmission Stream into Categories

Imagine a newly established TCP connection between Device A and Device B. Device A has a long stream of bytes that it will transmit, but Device B can't accept them all at once, so it limits Device A to sending a particular number of bytes at once in segments, until the bytes in the segments already sent have been

acknowledged. Then Device A is allowed to send more bytes. Each device keeps track of which bytes have been sent and which have not, and which have been acknowledged.

At any point in time, we can take a "snapshot" of the process. If we do, we can conceptually divide the bytes that the sending TCP has in its buffer into the following four categories, and view them as a timeline (see Figure 46-5):

1. **Bytes Sent and Acknowledged** The earliest bytes in the stream will have been sent and acknowledged. These bytes are basically viewed from the standpoint of the device sending data. In the example in Figure 46-5, 31 bytes of data have already been sent and acknowledged. These would fall into category 1.
2. **Bytes Sent but Not Yet Acknowledged** These are the bytes that the device has sent but for which it has not yet received an acknowledgment. The sender cannot consider these handled until they are acknowledged. In Figure 46-5, there are 14 bytes here, in category 2.
3. **Bytes Not Yet Sent for Which Recipient Is Ready** These are bytes that the device has not sent, but which the recipient has room for based on its most recent communication to the sender regarding how many bytes it is willing to handle at once. The sender will try to send these immediately (subject to certain algorithmic restrictions that you'll explore later). In Figure 46-5, there are 6 bytes in category 3.
4. **Bytes Not Yet Sent for Which Recipient Is Not Ready** These are the bytes further down the stream, which the sender is not yet allowed to send because the receiver is not ready. In Figure 46-5, there are 44 bytes in category 4.

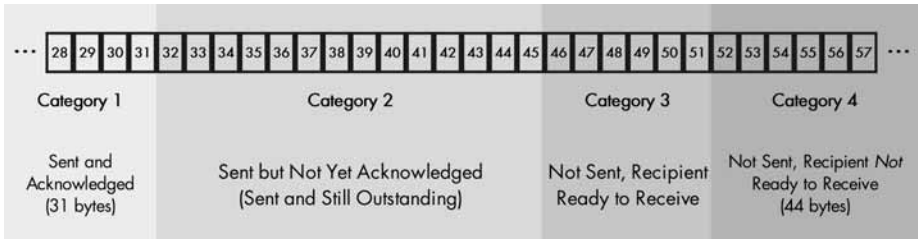


Figure 46-5: Conceptual division of TCP transmission stream into categories

NOTE *I am using very small numbers here to keep the example simple and to make the diagrams a bit easier to construct! TCP does not normally send tiny numbers of bytes around for efficiency reasons.*

The receiving device uses a similar system in order to differentiate between data received and acknowledged, data not yet received but ready to receive, and data not yet received and not yet ready to be received. In fact, both devices maintain a separate set of variables to keep track of the categories into which bytes fall in the stream they are sending, as well as the stream they are receiving. This is explored further in Chapter 48's section named "TCP Sliding Window Data Transfer and Acknowledgment Mechanics," which describes the detailed sliding window data transfer procedure.

KEY CONCEPT The TCP *sliding window system* is a variation on the enhanced PAR system, with changes made to support TCP's stream orientation. Each device keeps track of the status of the byte stream that it needs to transmit. The device keeps track by dividing the byte streams into four conceptual categories: bytes sent and acknowledged, bytes sent but not yet acknowledged, bytes not yet sent but that can be sent immediately, and bytes not yet sent that cannot be sent until the recipient signals that it is ready for them.

Sequence Number Assignment and Synchronization

The sender and receiver must agree on the sequence numbers that they will assign to the bytes in the stream. This is called *synchronization* and is done when the TCP connection is established. For simplicity, let's assume that the first byte was sent with sequence number 1 (this is not normally the case). Thus, in the example shown in Figure 46-5, the byte ranges for the four categories are as follows:

1. The bytes sent and acknowledged are bytes 1 to 31.
2. The bytes sent but not yet acknowledged are bytes 32 to 45.
3. The bytes not yet sent for which the recipient is ready are bytes 46 to 51.
4. The bytes not yet sent for which the recipient is not ready are bytes 52 to 95.

The Send Window and Usable Window

The key to the operation of the entire process is the number of bytes that the recipient is allowing the transmitter to have unacknowledged at one time. This is called the *send window*, or often, just the *window*. The window is what determines how many bytes the sender is allowed to transmit, and is equal to the sum of the number of bytes in category 2 and category 3. Thus, the dividing line between the last two categories (bytes not sent that the recipient is ready for and bytes the recipient is not ready for) is determined by adding the window to the byte number of the first unacknowledged byte in the stream. In the example shown in Figure 46-5, the first unacknowledged byte is 32. The total window size is 20.

The term *usable window* is defined as the amount of data the transmitter is still allowed to send given the amount of data that is outstanding. It is thus exactly equal to the size of category 3. You may also commonly hear the *edges* of the window mentioned. The left edge marks the first byte in the window (byte 32). The right edge marks the last byte in the window (byte 51). See Figure 46-6 for an illustration of these concepts.

KEY CONCEPT The *send window* is the key to the entire TCP sliding window system. It represents the maximum number of unacknowledged bytes that a device is allowed to have outstanding at one time. The *usable window* is the amount of the send window that the sender is still allowed to send at any point in time; it is equal to the size of the send window less the number of unacknowledged bytes already transmitted.

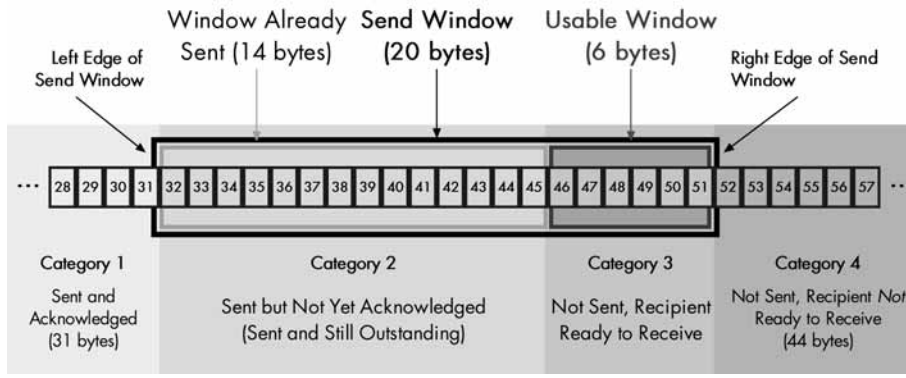


Figure 46-6: TCP transmission stream categories and send window terminology *This diagram shows the same categories as the ones in Figure 46-5, except that it shows the send window as well. The black box is the overall send window (categories 2 and 3 combined); the light gray box represents the bytes already sent (category 2), and the dark gray box is the usable window (category 3).*

Changes to TCP Categories and Window Sizes After Sending Bytes in the Usable Window

Now let's suppose that in the example shown in Figure 46-6 there is nothing stopping the sender from immediately transmitting the 6 bytes in category 3 (the usable window). When the sender transmits them, the 6 bytes will shift from category 3 to category 2. The byte ranges will now be as follows (see Figure 46-7):

1. The bytes sent and acknowledged are bytes 1 to 31.
2. The bytes sent but not yet acknowledged are bytes 32 to 51.
3. The bytes not yet sent for which the recipient is ready are none.
4. The bytes not yet sent for which the recipient is not ready are bytes 52 to 95.

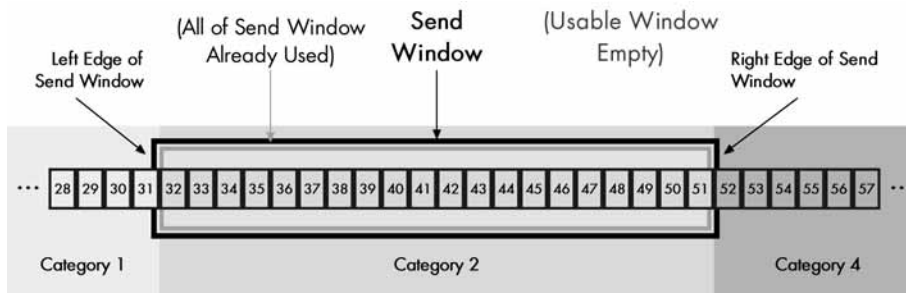


Figure 46-7: TCP stream categories and window after sending usable window bytes *This diagram shows the result of the device sending all the bytes that it is allowed to transmit in its usable window. It is the same as Figure 46-6, except that all the bytes in category 3 have moved to category 2. The usable window is now zero and will remain so until it receives an acknowledgment for bytes in category 2.*

Processing Acknowledgments and Sliding the Send Window

Some time later, the destination device sends back a message to the sender and provides an acknowledgment. The destination device will not specifically list out the bytes that it has acknowledged, because as I said earlier, listing the bytes would

be inefficient. Instead, the destination device will acknowledge a range of bytes that represents the longest contiguous sequence of bytes it has received since the ones it had previously acknowledged.

For example, let's suppose that the bytes already sent but not yet acknowledged at the start of the example (bytes 32 to 45) were transmitted in four different segments. These segments carried bytes 32 to 34, 35 to 36, 37 to 41, and 42 to 45, respectively. The first, second, and fourth segments arrived, but the third did not. The receiver will send back an acknowledgment only for bytes 32 to 36 (32 to 34 and 35 to 36). The receiver will hold bytes 42 to 45 but won't acknowledge them, because this would imply that the receiver has received bytes 37 to 41, which have not shown up yet. This is necessary because TCP is a cumulative acknowledgment system that can use only a single number to acknowledge data. That number is the number of the last contiguous byte in the stream that was successfully received. Let's also say that the destination keeps the window size the same at 20 bytes.

NOTE *An optional feature called* selective acknowledgments *does allow noncontiguous blocks of data to be acknowledged. This is explained in Chapter 49's section named "TCP Noncontiguous Acknowledgment Handling and Selective Acknowledgment (SACK)"; we'll ignore this complication for now.*

When the sending device receives this acknowledgment, it will be able to transfer some of the bytes from category 2 to category 1, because they have now been acknowledged. When it does so, something interesting will happen. Since 5 bytes have been acknowledged, and the window size didn't change, the sender is allowed to send 5 more bytes. In effect, the window shifts or slides over to the right in the timeline. At the same time 5 bytes move from category 2 to category 1, 5 bytes move from category 4 to category 3, creating a new usable window for subsequent transmission. So, after the groups receive the acknowledgment, they will look like what you see in Figure 46-8. The byte ranges will be as follows:

1. The bytes sent and acknowledged are bytes 1 to 36.
2. The bytes sent but not yet acknowledged are bytes 37 to 51.
3. The bytes not yet sent for which the recipient is ready are bytes 52 to 56.
4. The bytes not yet sent for which the recipient is not ready are bytes 57 to 95.

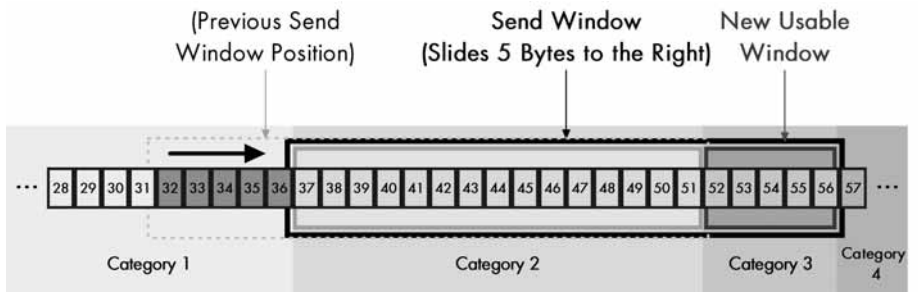


Figure 46-8: Sliding the TCP send window *After receiving acknowledgment for bytes 32 to 36, the bytes move from category 2 to 1 (shown in dark shading). The send window shown in Figure 46-7 slides right by 5 bytes; shifting 5 bytes from category 4 to 3, and opening a new usable window.*

This process will occur each time an acknowledgment is received, thereby causing the window to slide across the entire stream in order to be transmitted. And thus, ladies and gentlemen, you have the TCP sliding window acknowledgment system!

It is a very powerful technique that allows TCP to easily acknowledge an arbitrary number of bytes using a single acknowledgment number. It provides reliability to the byte-oriented protocol without spending time on an excessive number of acknowledgments. For simplicity, the example I've used here leaves the window size constant, but in reality, it can be adjusted to allow a recipient to control the rate at which data is sent, thereby enabling flow control and congestion handling.

KEY CONCEPT When a device gets an acknowledgment for a range of bytes, it knows the destination has successfully received them. It moves them from the "sent but unacknowledged" to the "sent and acknowledged" category. This causes the send window to slide to the right, allowing the device to send more data.

Dealing with Missing Acknowledgments

But what about bytes 42 through 45 in the example shown in Figure 46-8? Until segment 3 (containing bytes 37 to 41) shows up, the receiving device will not send an acknowledgment for those bytes, and it won't send any others that show up after it. The sending device will be able to send the new bytes that were added to category 3, namely, bytes 52 to 56. The sending device will then stop, and the window will be stuck on bytes 37 to 41.

KEY CONCEPT TCP acknowledgments are cumulative and tell a transmitter that the receiving device successfully received all the bytes up to the sequence number indicated in the acknowledgment. Thus, if the receiving device receives bytes out of order, the device cannot acknowledge them until all the preceding bytes are received.

Like the PAR system, TCP includes a system for timing transmissions and retransmitting. Eventually, the TCP device will resend the lost segment. Unfortunately, one drawback of TCP is that since it does not separately acknowledge segments, it may have to retransmit other segments that the recipient actually received (such as the segment with bytes 42 to 45). This starts to get very complex, as I discussed in the topic on TCP retransmissions in Chapter 49.

More Information on TCP Sliding Windows

Despite the length of this explanation, the preceding is just a summary description of the overall operation of sliding windows. This chapter does not include all of the modifications used in modern TCP! As you can see, the sliding window mechanism is at the heart of the operation of TCP as a whole. In the chapter that describes segments and discusses data transfer, you will see in more detail how TCP transmitters decide how and when to create segments for transmission. Chapter 49 provides much more information about how sliding windows enable a device to manage the flow of data to it on a TCP connection. It also discusses special problems that can

arise if window size is not carefully managed and how you can avoid problems such as congestion in TCP implementations through key changes to the basic sliding window mechanism described in this section.

TCP Ports, Connections, and Connection Identification

The two TCP/IP transport layer protocols, TCP and UDP, play the same architectural role in the protocol suite, but do it in very different ways. In fact, one of the few functions that the two have in common is that they both provide a method of transport layer addressing and multiplexing. Through the use of *ports*, both protocols allow the data from many different application processes to be aggregated and sent through the IP layer, and then returned up the stack to the proper application process on the destination device. I explain TCP ports in detail in Chapter 43.

Despite this commonality, TCP and UDP diverge somewhat even in how they deal with processes. UDP is a connectionless protocol, which means that devices do not set up a formal connection before sending data. UDP does not have to use sliding windows or keep track of how long it has been since UDP sent a transmission and so forth. When the UDP layer on a device receives data, it just sends it to the process that the destination port indicates, and that's that. UDP can seamlessly handle any number of processes that are sending it messages because UDP handles them all identically.

In contrast, since TCP is connection-oriented, it has many more responsibilities. Each TCP software layer needs to be able to support connections to several other TCPs simultaneously. The operation of each connection is separate from of each other connection, and the TCP software must manage each operation independently. TCP must be sure that it not only routes data to the right process, but that it also manages transmitted data on each connection without any overlap or confusion.

The first consequence of this is that TCP must uniquely identify each connection. It does this by using the pair of socket identifiers that correspond to the two endpoints of the connection, where a socket is simply the combination of the IP address and the port number of each process. This means a socket pair contains four pieces of information: source address, source port, destination address, and destination port. Thus, TCP connections are sometimes said to be described by this addressing quadruple.

I introduced this concept in Chapter 43, where I gave the example of a Hypertext Transfer Protocol (HTTP) request that a client sends at 177.41.72.6 to a website at 41.199.222.3. The server for that website will use well-known port number 80, so the server's socket is 41.199.222.3:80. If the server assigns a client ephemeral port number 3022 for the web browser, the client socket is 177.41.72.6:3022. The overall connection between these devices can be described using this socket pair: (41.199.222.3:80, 177.41.72.6:3022).

This identification of connections using both client and server sockets is what provides the flexibility in allowing multiple connections between devices that we probably take for granted on the Internet. For example, busy application server processes (such as web servers) must be able to handle connections from more than one client; otherwise, the Web would be pretty much unusable. Since the client and server's socket identify the connection, this is no problem. At the same

time that the web server maintains the connection, it can easily have another connection to say, port 2199 at IP address 219.31.0.44. The connection identifier that represents this as follows: (41.199.222.3:80, 219.31.0.44:2199).

In fact, you can have multiple connections from the same client to the same server. Each client process will be assigned a different ephemeral port number, so even if they all try to access the same server process (such as the web server process at 41.199.222.3:80), they will all have a different client socket and represent unique connections. This difference is what lets you make several simultaneous requests to the same website from your computer.

Again, TCP keeps track of each of these connections independently, so each connection is unaware of the others. TCP can handle hundreds or even thousands of simultaneous connections. The only limit is the capacity of the computer running TCP, and the bandwidth of the physical connections to it—the more connections running at once, the more each one has to share limited resources.

KEY CONCEPT Each device can handle simultaneous TCP connections to many different processes on one or more devices. The socket numbers of the devices in the connection, called the connection's *endpoints*, identify each connection. Each endpoint consists of the device's IP address and port number, so the four-way communication between client IP address and port number, and server IP address and port number identifies each connection.

TCP Common Applications and Server Port Assignments

In the overview of TCP in Chapter 45, you saw that the protocol originally included the functions of both modern TCP and IP. TCP was split into TCP and IP in order to allow applications that didn't need TCP's complexity to bypass it, using the much simpler UDP as a transport layer protocol instead. This bypass was an important step in the development of the TCP/IP protocol suite, since there are several important protocols for which UDP is ideally suited, and even some for which TCP is more of a nuisance than a benefit.

Most commonly, however, UDP is used only in special cases. I describe the two types of applications that may be better suited to UDP than TCP in Chapter 44: applications where speed is more important than reliability, and applications that send only short messages infrequently. The majority of TCP/IP applications do not fall into these categories. Thus, even though the layering of TCP and IP means that most protocols aren't required to use TCP, most of them do anyway. The majority of the protocols that use TCP employ all, or at least most, of the features that it provides. The establishment of a persistent connection is necessary for many interactive protocols, such as Telnet, as well as for ones that send commands and status replies, like HTTP. Reliability and flow control are essential for protocols like the File Transfer Protocol (FTP) or the email protocols, which send large files.

Table 46-1 shows some of the more significant application protocols that run on TCP. For each protocol, I have shown the well-known or registered port number that's reserved for that protocol's server process (clients use ephemeral ports, not the port numbers in the table). I have also shown the special keyword shortcut for each port assignment and provided brief comments on why the protocol is well matched to TCP.

Table 46-1: Common TCP Applications and Server Port Assignments

Port #	Keyword	Protocol	Comments
20 and 21	ftp-data/ftp	File Transfer Protocol (FTP, data and control)	Used to send large files, so it is ideally suited for TCP.
23	telnet	Telnet Protocol	Interactive session-based protocol. Requires the connection-based nature of TCP.
25	smtp	Simple Mail Transfer Protocol (SMTP)	Uses an exchange of commands, and sends possibly large files between devices.
53	domain	Domain Name Server (DNS)	An example of a protocol that uses both UDP and TCP. For simple requests and replies, DNS uses UDP. For larger messages, especially zone transfers, DNS uses TCP.
70	gopher	Gopher Protocol	A messaging protocol that has been largely replaced by the WWW.
80	http	Hypertext Transfer Protocol (HTTP/World Wide Web)	The classic example of a TCP-based messaging protocol.
110	pop3	Post Office Protocol (POP version 3)	Email message retrieval protocols that use TCP to exchange commands and data.
119	nntp	Network News Transfer Protocol (NNTP)	Used for transferring NetNews (Usenet) messages, which can be lengthy.
139	netbios-ssn	NetBIOS Session Service	A session protocol, clearly better suited to TCP than UDP.
143	imap	Internet Message Access Protocol (IMAP)	Another email message retrieval protocol.
179	bgp	Border Gateway Protocol (BGP)	While interior routing protocols like RIP and OSPF use either UDP or IP directly, BGP runs over TCP. This allows BGP to assume reliable communication even as it sends data over potentially long distances.
194	irc	Internet Relay Chat (IRC)	IRC is like Telnet in that it is an interactive protocol that is strongly based on the notion of a persistent connection between a client and server.
2049	nfs	Network File System (NFS)	NFS was originally implemented using UDP for performance reasons. Given that it is responsible for large transfers of files and given UDP's unreliability, NFS was probably not the best idea, so developers created TCP versions. The latest version of NFS uses TCP exclusively.
6000–6063	TCP	x11	Used for the X Window graphical system. Multiple ports are dedicated to allow many sessions.

A couple of the protocols in Table 46-1 use both TCP and UDP in order to get the best of both worlds. UDP can send short, simple messages, while TCP moves larger files. Many of the protocols that use both TCP and UDP are actually utility/diagnostic protocols (such as Echo, Discard, and the Time Protocol). These are special cases, because they developers designed them to use both UDP and TCP specifically to allow their use for diagnostics on both protocols.

I have not included an exhaustive list of TCP applications in Table 46-1. See Chapter 42 for common TCP/IP applications and port numbers, and also a reference to the full (massive) list of well-known and registered TCP server ports.