Computer Networking with TCP/IP

Module 4

Transport Layer

TRANSPORT-LAYER SERVICES

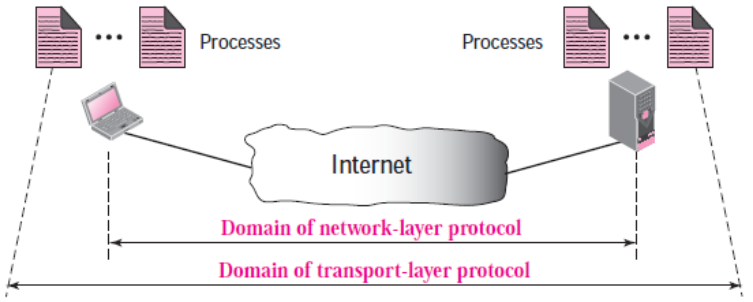
The transport layer is responsible for providing services to the application layer; it receives services from the network layer.

Process-to-Process Communication

The first duty of a transport-layer protocol is to provide process-to-process communication. A process is an application-layer entity (running program) that uses the services of the transport layer.

The network layer is responsible for communication at the computer level (host-to-host communication). A network layer protocol can deliver the message only to the destination computer. However, this is an incomplete delivery. The message still needs to be handed to the correct process. This is where a transport layer protocol takes over.

Figure 13.1 *Network layer versus transport layer*



Addressing: Port Numbers

Although there are a few ways to achieve process-to-process communication, the most common is through the client-server paradigm. A process on the local host, called a client, needs services from a process usually on the remote host, called a server.

A remote computer can run several server programs at the same time, just as several local computers can run one or more client programs at the same time. For communication, we must define the

1. Local host
2. Local process
3. Remote host
4. Remote process

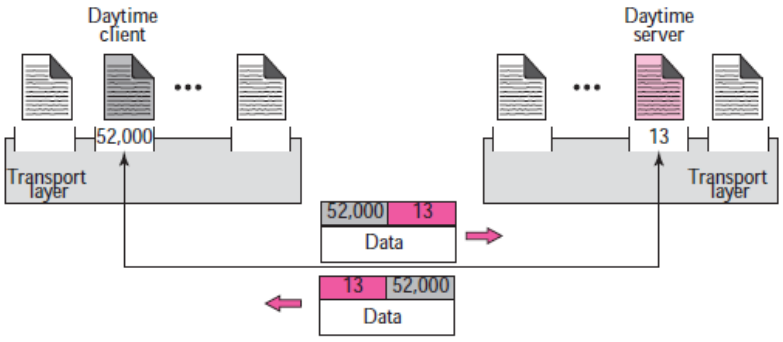
The local host and the remote host are defined using IP addresses. To define the processes, we need second identifiers called port numbers. In the TCP/IP protocol suite, the port numbers are integers between 0 and 65,535.

The client program defines itself with a port number, called the **ephemeral port number**. The word ephemeral means short lived and is used because the life of a client is normally short. An ephemeral

port number is recommended to be greater than 1,023 for some client/server programs to work properly.

The server process must also define itself with a port number. TCP/IP has decided to use universal port numbers for servers; these are called **well-known port numbers.**

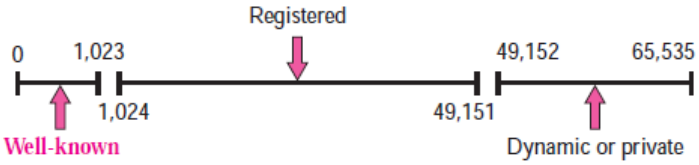
Figure 13.2 *Port numbers*



ICANN Ranges

ICANN has divided the port numbers into three ranges: well-known, registered, and dynamic (or private).

Figure 13.4 *ICANN ranges*



- **Well-known ports.** The ports ranging from 0 to 1,023 are assigned and controlled by ICANN. These are the well-known ports.
- **Registered ports**. The ports ranging from 1,024 to 49,151 are not assigned or controlled by ICANN. They can only be registered with ICANN to prevent duplication.
- **Dynamic ports.** The ports ranging from 49,152 to 65,535 are neither controlled nor registered. They can be used as temporary or private port numbers.

Socket Addresses

A transport-layer protocol in the TCP suite needs both the IP address and the port number, at each end, to make a connection. The combination of an IP address and a port number is called a socket address. The client socket address defines the client process uniquely just as the server socket address defines the server process uniquely.

Encapsulation and Decapsulation

To send a message from one process to another, the transport layer protocol encapsulates and decapsulates messages. **Encapsulation** happens at the sender site. When a process has a message to send, it passes the message to the transport layer along with a pair of socket addresses and some other pieces of information that depends on the transport layer protocol. The transport layer

receives the data and adds the transport-layer header. The packets at the transport layers in the Internet are called user datagrams, segments, or packets.

Decapsulation happens at the receiver site. When the message arrives at the destination transport layer, the header is dropped, and the transport layer delivers the message to the process running at the application layer.

Multiplexing and Demultiplexing

Whenever an entity accepts items from more than one source, it is referred to as multiplexing (many to one); whenever an entity delivers items to more than one source, it is referred to as demultiplexing (one to many). The transport layer at the source performs multiplexing; the transport layer at the destination performs demultiplexing.

Flow Control

Whenever an entity produces items and another entity consumes them, there should be a balance between production and consumption rates. If the items are produced faster than they can be consumed, the consumer can be overwhelmed and needs to discard some items. If the items are produced slower than they can be consumed, the consumer should wait; the system becomes less efficient. Flow control is related to the first issue. We need to prevent losing the data items at the consumer site.

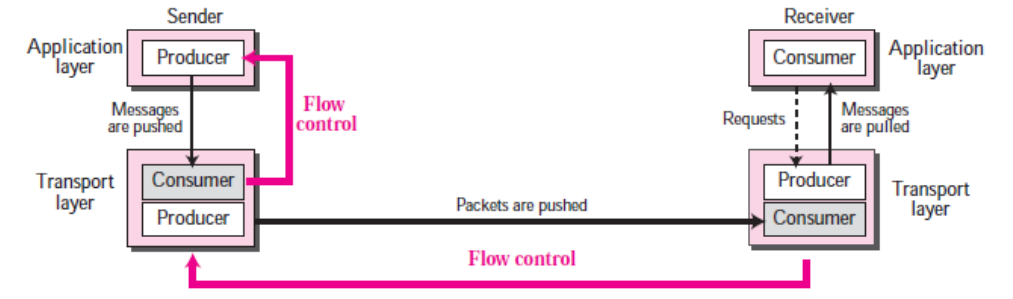
Pushing or Pulling

Delivery of items from a producer to a consumer can occur in one of the two ways: pushing or pulling. If the sender delivers items whenever they are produced without the prior request from the consumer the delivery is referred to as **pushing**. If the producer delivers the items after the consumer has requested them, the delivery is referred to as **pulling**.

Flow Control at Transport Layer

In communication at the transport layer, we are dealing with four entities: sender process, sender transport layer, receiver transport layer, and receiver process. The sending process at the application layer is only a producer. It produces message chunks and pushes them to the transport layer. The sending transport layer has a double role: it is both a consumer and the producer. It consumes the messages pushed by the producer.

Figure 13.9 *Flow control at the transport layer*



Buffers

Although flow control can be implemented in several ways, one of the solutions is normally to use two buffers. One at the sending transport layer and the other at the receiving transport layer.

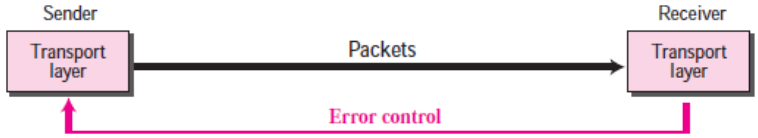
Error Control

Reliability can be achieved to add error control service to the transport layer. Error control at the transport layer is responsible to

1. Detect and discard corrupted packets.
2. Keep track of lost and discarded packets and resend them.
3. Recognize duplicate packets and discard them.
4. Buffer out-of-order packets until the missing packets arrive.

Error control, unlike the flow control, involves only the sending and receiving transport layers. We are assuming that the message chunks exchanged between the application and transport layers are error free.

Figure 13.10 *Error control at the transport layer*



Sequence Numbers

Error control requires that the sending transport layer knows which packet is to be resent and the receiving transport layer knows which packet is a duplicate, or which packet has arrived out of order. This can be done if the packets are numbered. We can add a field to the transport layer packet to hold the sequence number of the packets.

Acknowledgment

We can use both positive and negative signals as error control. The receiver side can send an acknowledgement (ACK) for each or a collection of packets that have arrived safe and sound. The receiver can simply discard the corrupted packets. The sender can detect lost packets if it uses a timer. When a packet is sent, the sender starts a timer; when the timer expires, if an ACK does not arrive before the timer expires, the sender resends the packet. Duplicate packets can be silently discarded by the receiver. Out-of-order packets can be either discarded (to be treated as lost packets by the sender) or stored until the missing ones arrives.

Combination of Flow and Error Control

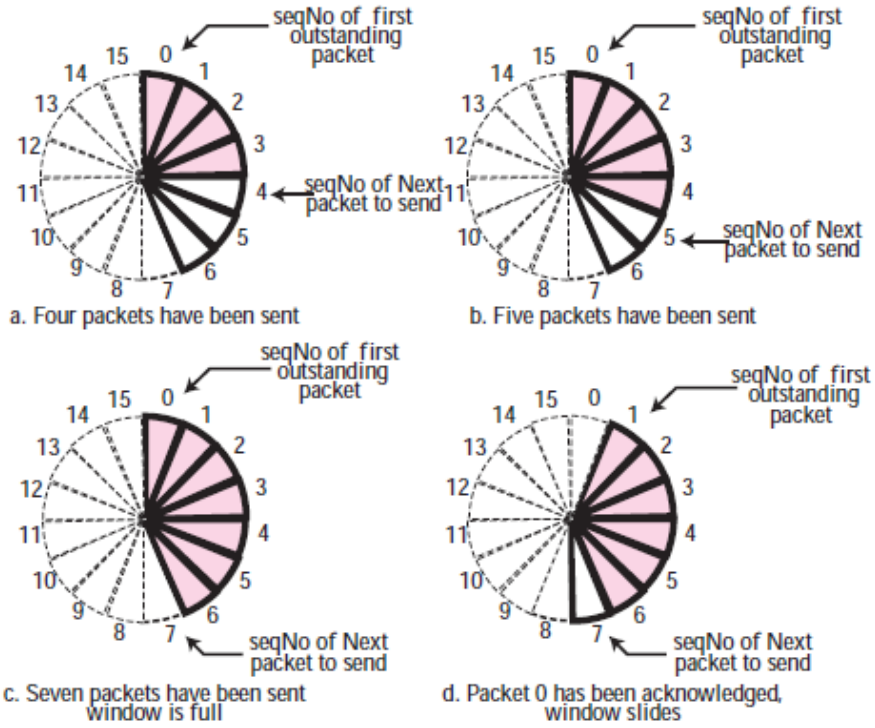
We have discussed that flow control requires the use of two buffers, one at the sender site and the other at the receiver site. We have also discussed that the error control requires the use of sequence and acknowledgment numbers by both sides. These two requirements can be combined if we use two numbered buffers, one at the sender, one at the receiver.

Sliding Window

Since the sequence numbers used modulo 2m, a circle can represent the sequence number from 0 to 2m – 1.

The buffer is represented as a set of slices, called the sliding window, that occupy part of the circle at any time. At the sender site, when a packet is sent, the corresponding slice is marked. When all the slices are marked, it means that the buffer is full and no further messages can be accepted from the application layer.

Figure 13.11 *Sliding window in circular format*



Congestion Control

An important issue in the Internet is congestion. Congestion in a network may occur if the load on the network—the number of packets sent to the network—is greater than the capacity of the network—the number of packets a network can handle. Congestion control refers to the mechanisms and techniques to control the congestion and keep the load below the capacity.

Congestion in a network or internetwork occurs because routers and switches have queues—buffers that hold the packets before and after processing.

In open-loop congestion control, policies are applied to prevent congestion before it happens. In these mechanisms, congestion control is handled by either the source or the destination.

Retransmission Policy Retransmission is sometimes unavoidable. If the sender feels that a sent packet is lost or corrupted, the packet needs to be retransmitted. Retransmission in general may increase congestion in the network. However, a good retransmission policy can prevent congestion.

Window Policy The type of window at the sender may also affect congestion.

Acknowledgment Policy The acknowledgment policy imposed by the receiver may also affect congestion. If the receiver does not acknowledge every packet it receives, it may slow down the sender and help prevent congestion.

Closed-Loop Congestion Control

Closed-loop congestion control mechanisms try to alleviate congestion after it happens. Several mechanisms have been used by different protocols. The sending transport layer can monitor the congestion in the Internet, by watching the lost packets, and use a strategy to decrease the window size if the congestion is increasing and vice versa.

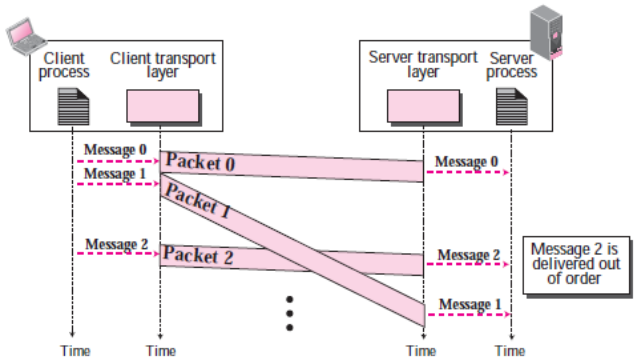
Connectionless and Connection-Oriented Services

A transport-layer protocol, like a network-layer protocol can provide two types of services: connectionless and connection-oriented. The nature of these services at the transport layer, however, is different from the ones at the network layer. At the network layer, a connectionless service may mean different paths for different datagrams belonging to the same message. At the transport layer, we are not concerned about the physical paths of packets (we assume a logical connection between two transport layers), connectionless service at the transport layer means independency between packets; connection-oriented means dependency.

Connectionless Service

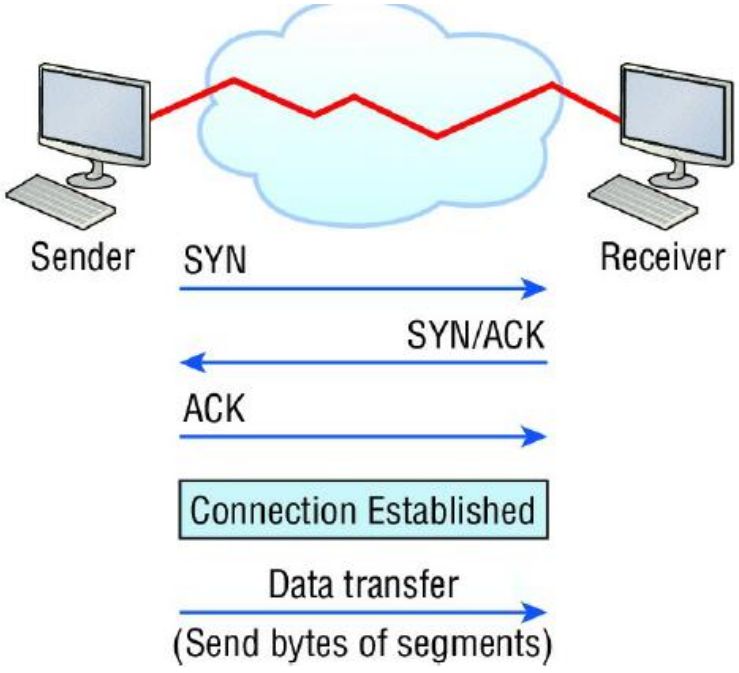
In a connectionless service, the source process (application program) needs to divide its message into chunks of data of the size acceptable by the transport layer and deliver them to the transport layer one by one. The transport layer treats each chunk as a single unit without any relation between the chunks.

Figure 13.13 *Connectionless service*



Connection-Oriented Service

In a connection-oriented service, the client and the server first need to establish a connection between themselves. The data exchange can only happen after the connection establishment.



Finite State Machine

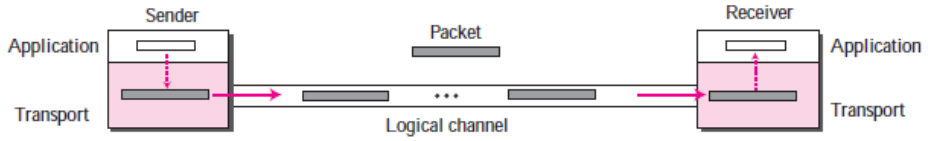
The behavior of a transport layer protocol, both when it provides a connectionless and when it provides a connection-oriented protocol, can be better shown as a finite state machine (FSM).

TRANSPORT-LAYER PROTOCOLS

Simple Protocol

Our first protocol is a simple connectionless protocol with neither flow nor error control. We assume that the receiver can immediately handle any packet it receives. In other words, the receiver can never be overwhelmed with incoming packets.

Figure 13.16 *Simple protocol*



FSMs

The sender site should not send a packet until its application layer has a message to send. The receiver site cannot deliver a message to its application layer until a packet arrives. We can show these requirements using two FSMs. Each FSM has only one state, the ready state. The sending machine remains in the ready state until a request comes from the process in the application layer.

Figure 13.17 *FSMs for the simple protocol*

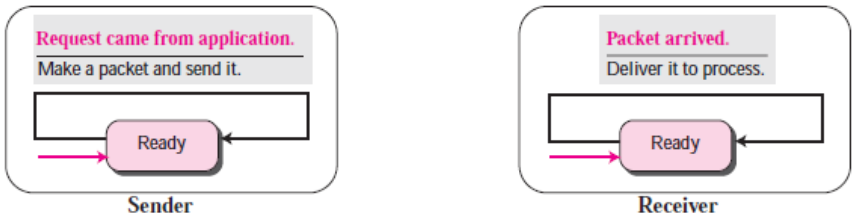
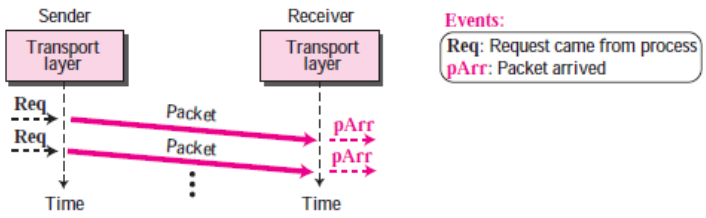


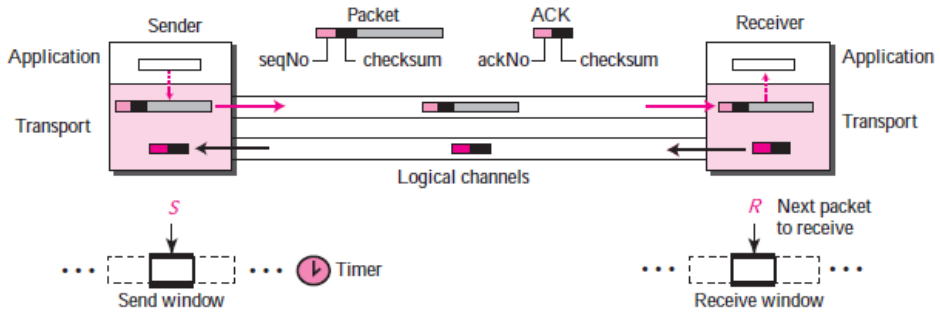
Figure 13.18 *Flow diagram for Example 13.3*



Stop-and-Wait Protocol

Our second protocol is a connection-oriented protocol called the Stop-and-Wait protocol, which uses both flow and error control. Both the sender and the receiver use a sliding window of size 1. The sender sends one packet at a time and waits for an acknowledgment before sending the next one.

Figure 13.19 *Stop-and-Wait protocol*



In Stop-and-Wait protocol, flow control is achieved by forcing the sender to wait for an acknowledgment, and error control is achieved by discarding corrupted packets and letting the sender resend unacknowledged packets when the timer expires.

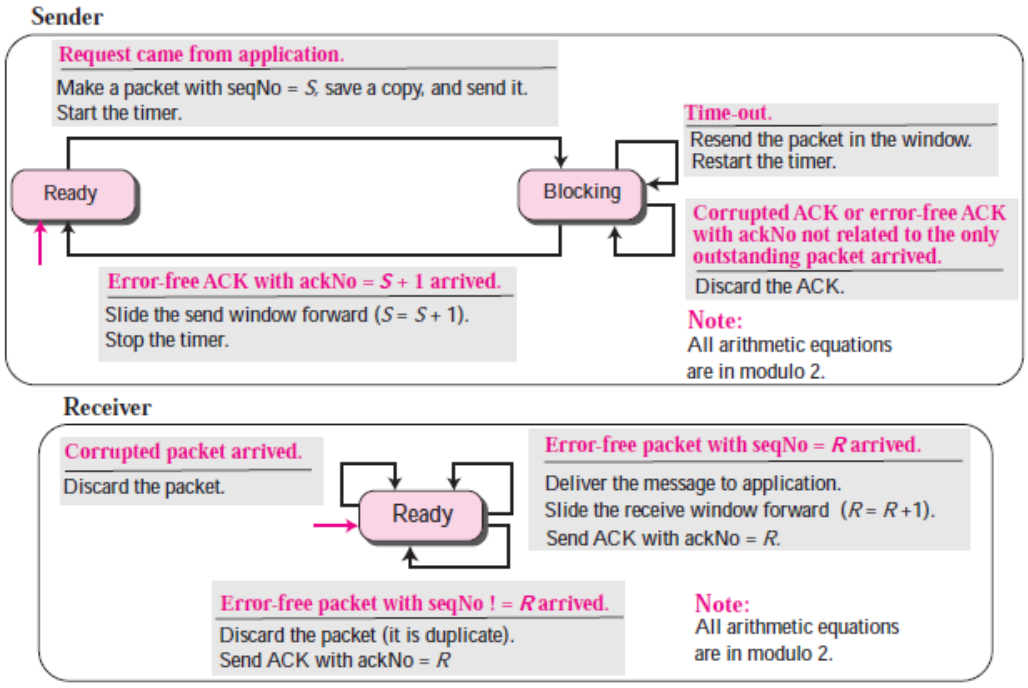
Sequence Numbers

To prevent duplicate packets, the protocol uses sequence numbers and acknowledgment numbers. A field is added to the packet header to hold the sequence number of that packet.

Acknowledgment Numbers

Since the sequence numbers must be suitable for both data packets and acknowledgments, we use this convention: The acknowledgment numbers always announce the sequence number of the next packet expected by the receiver.

Figure 13.20 *FSM for the Stop-and-Wait protocol*



Sender The sender is initially in the ready state, but it can move between the ready and blocking state. The variable S is initialized to 0.

Ready State. When the sender is in this state, it is only waiting for one event to occur. If a request comes from the application, the sender creates a packet with the sequence number set to S. A copy of the packet is stored, and the packet is sent. The sender then starts the only timer. The sender then moves to the blocking state.

Blocking State. When the sender is in this state, three events can occur:

1. If an error-free ACK arrives with ackNo related to the next packet to be sent, which means ackNo = (S + 1) modulo 2, then the timer is stopped. The window is slided, S = (S + 1) modulo 2. Finally, the sender moves to the ready state.
2. If a corrupted ACK or an error-free ACK with the ackNo ≠ (S + 1) modulo 2 arrives, the ACK is discarded.
3. If a time-out occurs, the sender resends the only outstanding packet and restarts the timer.

Receiver The receiver is always in the ready state. The variable R is initialized to 0. Three events may occur:

1. If an error-free packet with seqNo = R arrives, the message in the packet is delivered to the application layer. The window then slides, R = (R + 1) modulo 2. Finally an ACK with ackNo = R is sent.
2. If an error-free packet with seqNo ≠ R arrives. The packet is discarded, but an ACK with ackNo = R is sent.
3. If a corrupted packet arrives, the packet is discarded.

Efficiency

The Stop-and-Wait protocol is very inefficient if our channel is thick and long. By thick, we mean that our channel has a large bandwidth (high data rate); by long, we mean the round-trip delay is long. The product of these two is called the bandwidth-delay product.

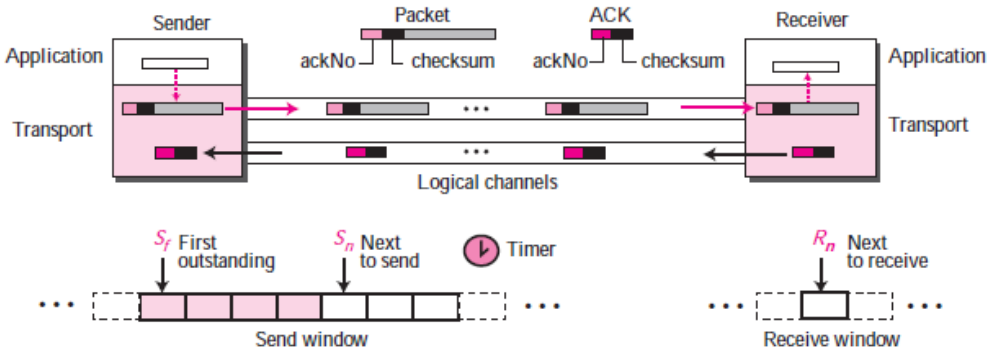
Pipelining

In networking and in other areas, a task is often begun before the previous task has ended. This is known as pipelining.

Go-Back-N Protocol

To improve the efficiency of transmission (fill the pipe), multiple packets must be in transition while the sender is waiting for acknowledgment. In other words, we need to let more than one packet be outstanding to keep the channel busy while the sender is waiting for acknowledgment.

Figure 13.22 *Go-Back-N protocol*



Sequence Numbers

As we mentioned before, the sequence numbers are used modulo 2m, where m is the size of the sequence number field in bits.

In the Go-Back-N Protocol, the sequence numbers are modulo 2m, where m is the size of the sequence number field in bits.

Acknowledgment Number

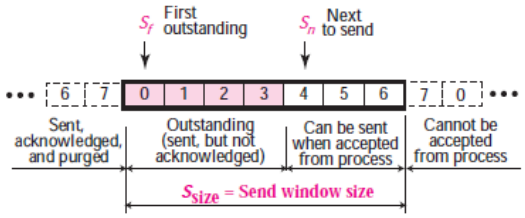
Acknowledgment number in this protocol is cumulative and defines the sequence number of the next packet expected.

In the Go-Back-N protocol, the acknowledgment number is cumulative and defines the sequence number of the next packet expected to arrive.

Send Window

The send window is an imaginary box covering the sequence numbers of the data packets that can be in transit or can be sent. In each window position, some of these sequence numbers define the packets that have been sent; others define those that can be sent. The maximum size of the window is $2^m - 1$.

Figure 13.23 *Send window for Go-Back-N*



Receive Window

The receive window makes sure that the correct data packets are received and that the correct acknowledgments are sent. In Go-back-N, the size of the receive window is always 1.

Figure 13.24 *Sliding the send window*

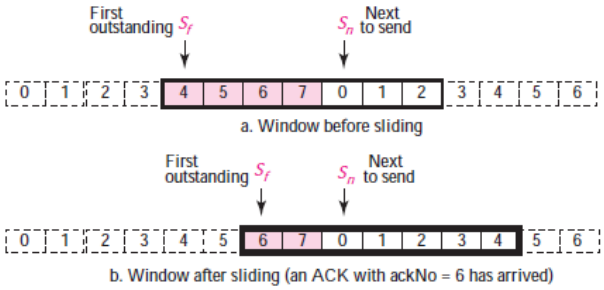
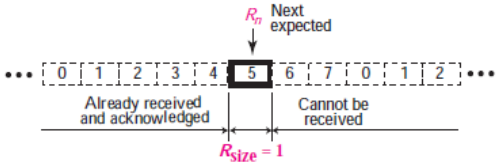


Figure 13.25 *Receive window for Go-Back-N*



Selective-Repeat Protocol

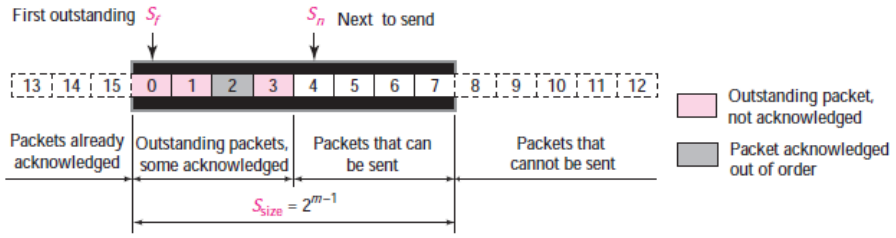
The Go-Back-N protocol simplifies the process at the receiver. The receiver keeps track of only one variable, and there is no need to buffer out-of-order packets; they are simply discarded. However, this protocol is inefficient if the underlying network protocol loses a lot of packets. Each time a single packet is lost or corrupted, the sender resends all outstanding packets although some of these packets may have been received safe and sound, but out of order. If the network layer is losing many packets because of congestion in the network, the resending of all of these outstanding packets makes the congestion worse, and eventually more packets are lost. This has an avalanche effect that may result in the total collapse of the network.

Another protocol, called the Selective-Repeat (SR) protocol, has been devised that, as the name implies, resends only selective packets, those that are actually lost.

Windows

The Selective-Repeat protocol also uses two windows: a send window and a receive window. However, there are differences between the windows in this protocol and the ones in Go-Back-N. First, the maximum size of the send window is much smaller; it is 2m–1. Second, the receive window is the same size as the send window.

Figure 13.31 *Send window for Selective-Repeat protocol*



Acknowledgments

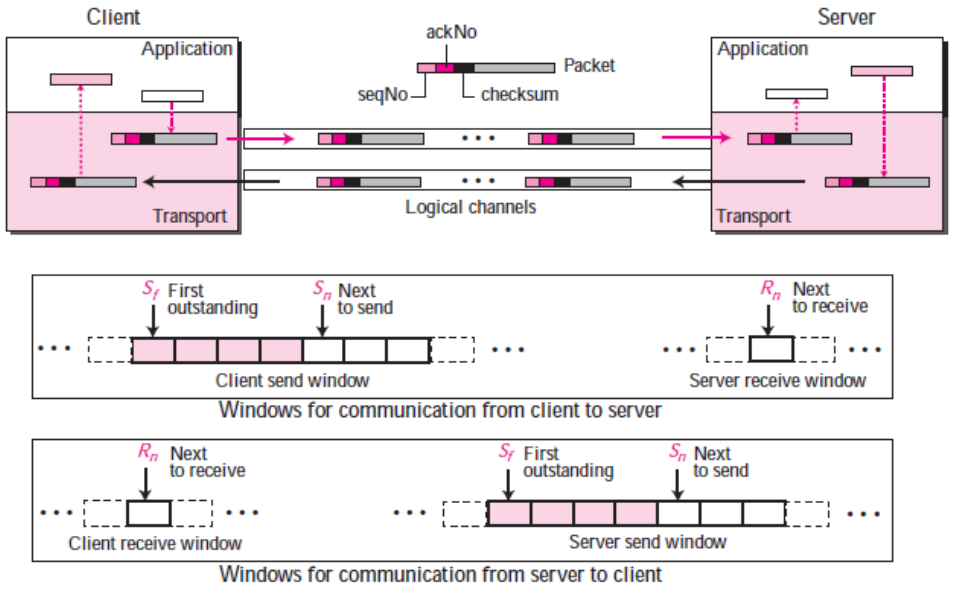
Still there is another difference between the two protocols. In GBN an ackNo is cumulative; it defines the sequence number of the next packet expected, confirming that all previous packets have been received safe and sound. The semantics of acknowledgment is different in SR. In SR, an ackNo defines the sequence number of one single packet that is received safe and sound; there is no feedback for any other.

Bidirectional Protocols: Piggybacking

Data packets flow in only one direction and acknowledgments travel in the other direction. In real life, data packets are normally flowing in both directions: from client to server and from server to client. This means that acknowledgments also need to flow in both directions.

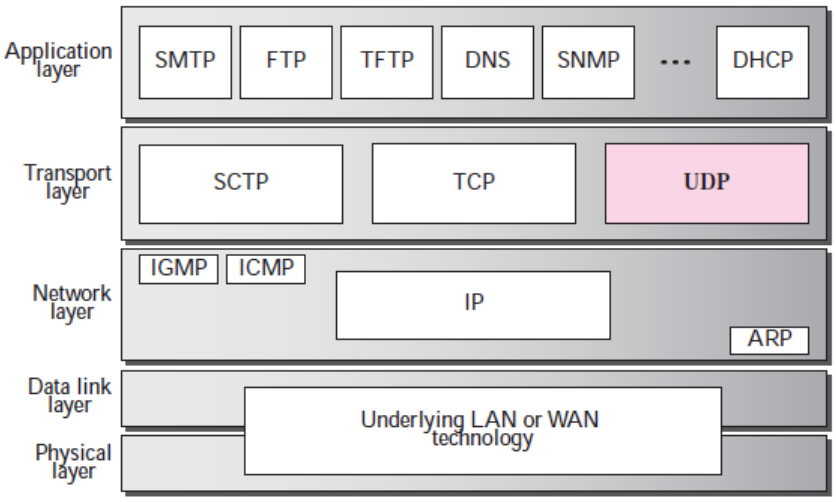
A technique called **piggybacking** is used to improve the efficiency of the bidirectional protocols. When a packet is carrying data from A to B, it can also carry acknowledgment feedback about arrived packets from B; when a packet is carrying data from B to A, it can also carry acknowledgment feedback about the arrived packets from A.

Figure 13.36 *Design of piggybacking in Go-Back-N*



User Datagram Protocol (UDP)

User Datagram Protocol (UDP) to the other protocols and layers of the TCP/IP protocol suite: UDP is located between the application layer and the IP layer, and serves as the intermediary between the application programs and the network operations.

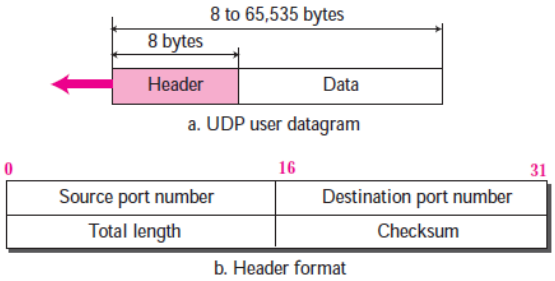


UDP is **a connectionless, unreliable transport protocol**. It does not add anything to the services of IP except for providing process-to-process communication instead of host-to-host communication.

USER DATAGRAM

UDP packets, called user datagrams, have a fixed-size header of 8 bytes.

Figure 14.2 *User datagram format*



Source port number. This is the port number used by the process running on the source host. It is 16 bits long, which means that the port number can range from 0 to 65,535.

Destination port number. This is the port number used by the process running on the destination host. It is also 16 bits long.

Length. This is a 16-bit field that defines the total length of the user datagram, header plus data.

Checksum. This field is used to detect errors over the entire user datagram (header plus data)

UDP SERVICES

Process-to-Process Communication

UDP provides process-to-process communication using sockets, a combination of IP addresses and port numbers.

Table 14.1 *Well-known Ports used with UDP*

Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
53	Domain	Domain Name Service (DNS)
67	Bootps	Server port to download bootstrap information
68	Bootpc	Client port to download bootstrap information
69	TFTP	Trivial File Transfer Protocol
111	RPC	Remote Procedure Call
123	NTP	Network Time Protocol
161	SNMP	Simple Network Management Protocol
162	SNMP	Simple Network Management Protocol (trap)

Connectionless Services

As mentioned previously, UDP provides a connectionless service. This means that each user datagram sent by UDP is an independent datagram. There is no relationship between the different user datagrams even if they are coming from the same source process and going to the same destination program. The user datagrams are not numbered. Also, there is no connection establishment and no connection termination.

Flow Control

UDP is a very simple protocol. There is no flow control, and hence no window mechanism. The receiver may overflow with incoming messages.

Error Control

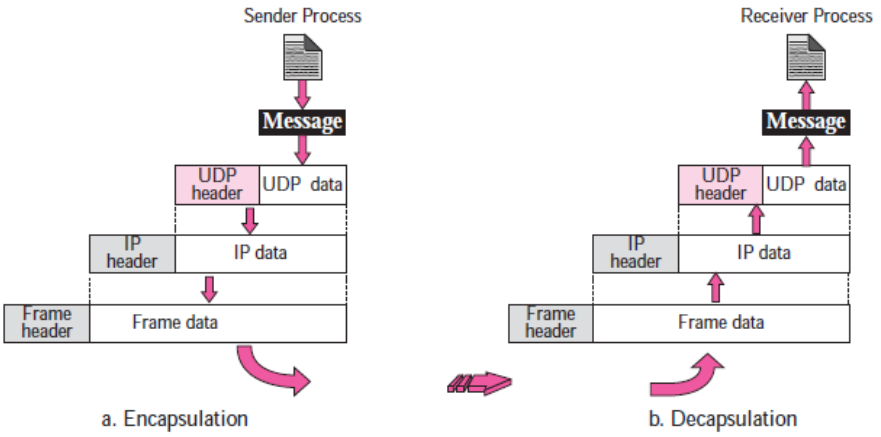
There is no error control mechanism in UDP except for the checksum. This means that the sender does not know if a message has been lost or duplicated. When the receiver detects an error through the checksum, the user datagram is silently discarded.

Congestion Control

Since UDP is a connectionless protocol, it does not provide congestion control. UDP assumes that the packets sent are small and sporadic, and cannot create congestion in the network.

Encapsulation and Decapsulation

To send a message from one process to another, the UDP protocol encapsulates and decapsulates messages.



Encapsulation

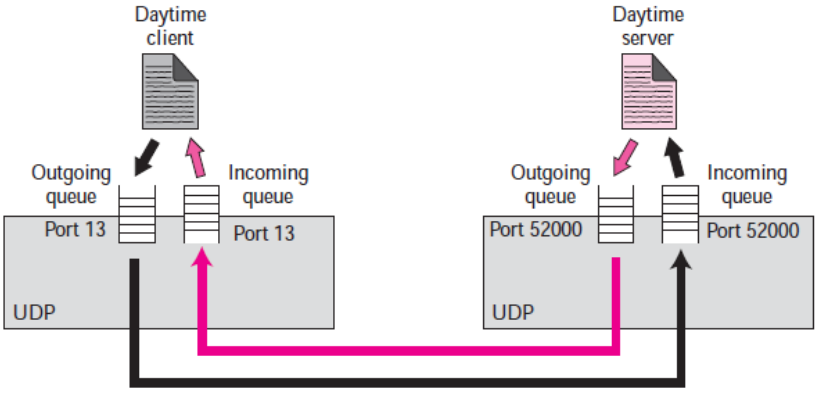
When a process has a message to send through UDP, it passes the message to UDP along with a pair of socket addresses and the length of data. UDP receives the data and adds the UDP header. UDP then passes the user datagram to IP with the socket addresses. IP adds its own header, using the value 17 in the protocol field, indicating that the data has come from the UDP protocol. The IP datagram is then passed to the data link layer. The data link layer receives the IP datagram, adds its own header (and possibly a trailer), and passes it to the physical layer. The physical layer encodes the bits into electrical or optical signals and sends it to the remote machine.

Decapsulation

When the message arrives at the destination host, the physical layer decodes the signals into bits and passes it to the data link layer. The data link layer uses the header (and the trailer) to check the data. If there is no error, the header and trailer are dropped and the datagram is passed to IP. The IP software does its own checking. If there is no error, the header is dropped and the user datagram is passed to UDP with the sender and receiver IP addresses. UDP uses the checksum to check the entire user datagram. If there is no error, the header is dropped and the application data along with the sender socket address is passed to the process. The sender socket address is passed to the process in case it needs to respond to the message received.

Queuing

At the client site, when a process starts, it requests a port number from the operating system. Some implementations create both an incoming and an outgoing queue associated with each process.

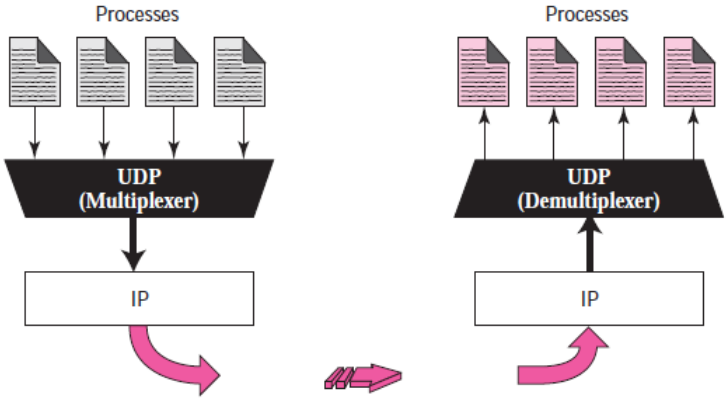


Note that even if a process wants to communicate with multiple processes, it obtains only one port number and eventually one outgoing and one incoming queue. The queues opened by the client are, in most cases, identified by ephemeral port numbers.

Note that even if a process wants to communicate with multiple processes, it obtains only one port number and eventually one outgoing and one incoming queue. The queues opened by the client are, in most cases, identified by ephemeral port numbers.

Multiplexing and Demultiplexing

In a host running a TCP/IP protocol suite, there is only one UDP but possibly several processes that may want to use the services of UDP. To handle this situation, UDP multiplexes and demultiplexes.



Multiplexing

At the sender site, there may be several processes that need to send user datagrams. However, there is only one UDP. This is a many-to-one relationship and requires multiplexing. UDP accepts messages from different processes, differentiated by their assigned port numbers. After adding the header, UDP passes the user datagram to IP.

Demultiplexing

At the receiver site, there is only one UDP. However, we may have many processes that can receive user datagrams. This is a one-to-many relationship and requires demultiplexing. UDP receives user datagrams from IP. After error checking and dropping of the header, UDP delivers each message to the appropriate process based on the port numbers.

UDP APPLICATIONS

Connectionless Service advantage and disadvantage

This feature can be considered as an advantage or disadvantage depending on the application requirement. It is an advantage if, for example, a client application needs to send a short request to a server and to receive a short response. If the request and response can each fit in one single user datagram, a connectionless service may be preferable. The overhead to establish and close a connection may be significant in this case. In the connection-oriented service, to achieve the above goal, at least 9 packets are exchanged between the client and the server; in connectionless service only two packets are exchanged. The connectionless service provides less delay; the connection-oriented service creates more delay. If delay is an important issue for the application, the connectionless service is preferred.

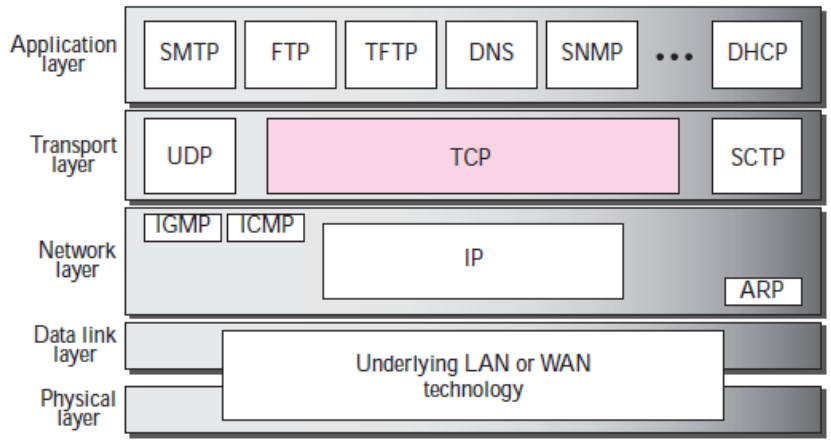
Typical Applications

The following shows some typical applications that can benefit more from the services of UDP than from those of TCP.

- UDP is suitable for a process that requires simple request-response communication with little concern for flow and error control. It is not usually used for a process such as FTP that needs to send bulk data.
- UDP is suitable for a process with internal flow and error-control mechanisms. For example, the Trivial File Transfer Protocol (TFTP) process includes flow and error control. It can easily use UDP.
- UDP is a suitable transport protocol for multicasting. Multicasting capability is embedded in the UDP software but not in the TCP software.
- UDP is used for management processes such as SNMP.
- UDP is used for some route updating protocols such as Routing Information Protocol (RIP).
- UDP is normally used for real-time applications that cannot tolerate uneven delay between sections of a received message.

Transmission Control Protocol (TCP)

TCP lies between the application layer and the network layer, and serves as the intermediary between the application programs and the network operations.



Process-to-Process Communication

As with UDP, TCP provides process-to-process communication using port numbers. Table 15.1 lists some well-known port numbers used by TCP.

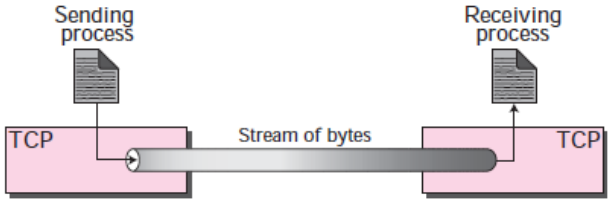
Table 15.1 *Well-known Ports used by TCP*

Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day

Port	*Protocol*	*Description*
19	Chargen	Returns a string of characters
20 and 21	FTP	File Transfer Protocol (Data and Control)
23	TELNET	Terminal Network
25	SMTP	Simple Mail Transfer Protocol
53	DNS	Domain Name Server
67	BOOTP	Bootstrap Protocol
79	Finger	Finger
80	HTTP	Hypertext Transfer Protocol

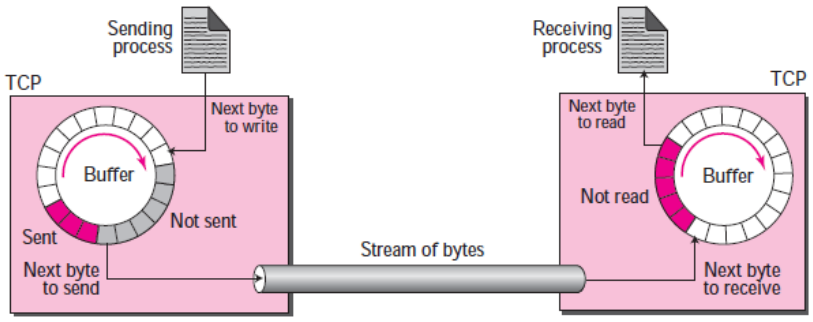
Stream Delivery Service

TCP, unlike UDP, is a stream-oriented protocol. TCP, allows the sending process to deliver data as a stream of bytes and allows the receiving process to obtain data as a stream of bytes. TCP creates an environment in which the two processes seem to be connected by an imaginary "tube" that carries their bytes across the Internet.



Sending and Receiving Buffers

Because the sending and the receiving processes may not necessarily write or read data at the same rate, TCP needs buffers for storage. There are two buffers, the sending buffer and the receiving buffer, one for each direction.



Segments

Although buffering handles the disparity between the speed of the producing and consuming processes, we need one more step before we can send data. The IP layer, as a service provider for TCP, needs to send data in packets, not as a stream of bytes. At the transport layer, TCP groups a

number of bytes together into a packet called a segment. TCP adds a header to each segment (for control purposes) and delivers the segment to the IP layer for transmission. The segments are encapsulated in an IP datagram and transmitted.

Full-Duplex Communication

TCP offers full-duplex service, where data can flow in both directions at the same time. Each TCP endpoint then has its own sending and receiving buffer, and segments move in both directions.

Multiplexing and Demultiplexing

Like UDP, TCP performs multiplexing at the sender and demultiplexing at the receiver. However, since TCP is a connection-oriented protocol, a connection needs to be established for each pair of processes.

Connection-Oriented Service

TCP, unlike UDP, is a connection-oriented protocol. When a process at site A wants to send to and receive data from another process at site B, the following three phases occur:

1. The two TCPs establish a virtual connection between them.

2. Data are exchanged in both directions.

3. The connection is terminated.

Reliable Service

TCP is a reliable transport protocol. It uses an acknowledgment mechanism to check the safe and sound arrival of data. We will discuss this feature further in the section on error control.

TCP FEATURES

Numbering System

There are two fields called the sequence number and the acknowledgment number. These two fields refer to a byte number and not a segment number.

Byte Number

TCP numbers all data bytes (octets) that are transmitted in a connection. Numbering is independent in each direction. When TCP receives bytes of data from a process, TCP stores them in the sending buffer and numbers them. The numbering does not necessarily start from 0. Instead, TCP chooses an arbitrary number between 0 and 2^{32}– 1 for the number of the first byte.

Sequence Number

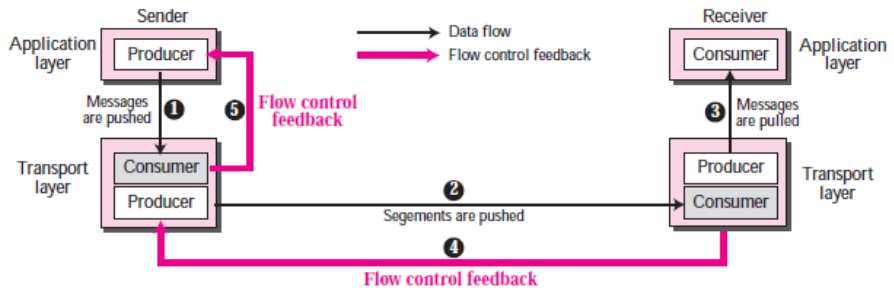
After the bytes have been numbered, TCP assigns a sequence number to each segment that is being sent.

Acknowledgment Number

The value of the acknowledgment field in a segment defines the number of the next byte a party expects to receive. The acknowledgment number is cumulative.

Flow Control

TCP, unlike UDP, provides flow control. The sending TCP controls how much data can be accepted from the sending process; the receiving TCP controls how much data can to be sent by the sending TCP. This is done to prevent the receiver from being overwhelmed with data.



Opening and Closing Windows

To achieve flow control, TCP forces the sender and the receiver to adjust their window sizes.

Shrinking of Windows

As we said before, the receive window cannot shrink. But the send window can shrink if the receiver defines a value for rwnd that results in shrinking the window.

Window Shutdown

The receiver can temporarily shut down the window by sending a rwnd of 0. This can happen if for some reason the receiver does not want to receive any data from the sender for a while.

Error Control

To provide reliable service, TCP implements an error control mechanism. Although error control considers a segment as the unit of data for error detection (loss or corrupted segments), error control is byte-oriented.

Checksum

Each segment includes a checksum field, which is used to check for a corrupted segment. If a segment is corrupted as deleted by an invalid checksum, the segment is discarded by the destination TCP and is considered as lost. TCP uses a 16-bit checksum that is mandatory in every segment.

Acknowledgment

TCP uses acknowledgments to confirm the receipt of data segments. Control segments that carry no data, but consume a sequence number, are also acknowledged. ACK segments are never acknowledged.

Acknowledgment Type

In the past, TCP used only one type of acknowledgment: cumulative acknowledgment.

Retransmission

The heart of the error control mechanism is the retransmission of segments. When a segment is sent, it is stored in a queue until it is acknowledged. When the retransmission timer expires or when the sender receives three duplicate ACKs for the first segment in the queue, that segment is retransmitted.

Retransmission after RTO

The sending TCP maintains one retransmission time-out (RTO) for each connection.

Out-of-Order Segments

TCP implementations today do not discard out-of-order segments. They store them temporarily and flag them as out-of-order segments until the missing segments arrive.

Congestion Control

TCP, unlike UDP, takes into account congestion in the network. The amount of data sent by a sender is not only controlled by the receiver (flow control), but is also determined by the level of congestion, if any, in the network.

Congestion Window

The sender has two pieces of information: the receiver-advertised window size and the congestion window size. The actual size of the window is the minimum (rwnd, cwnd).

Congestion Policy

TCP's general policy for handling congestion is based on three phases: slow start, congestion avoidance, and congestion detection.

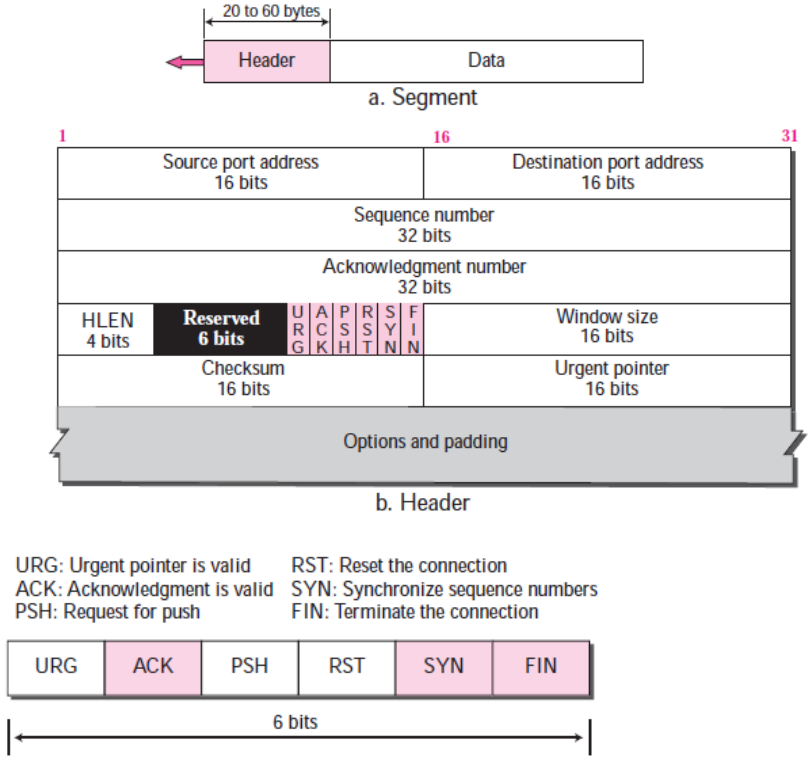
SEGMENT

A packet in TCP is called a segment.

Format

The segment consists of a header of 20 to 60 bytes, followed by data from the application program. The header is 20 bytes if there are no options and up to 60 bytes if it contains options.

Figure 15.5 *TCP segment format*



URG: Urgent pointer is valid
ACK: Acknowledgment is valid
PSH: Request for push
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: Terminate the connection

Encapsulation

A TCP segment encapsulates the data received from the application layer. The TCP segment is encapsulated in an IP datagram, which in turn is encapsulated in a frame at the data-link layer.

A TCP CONNECTION

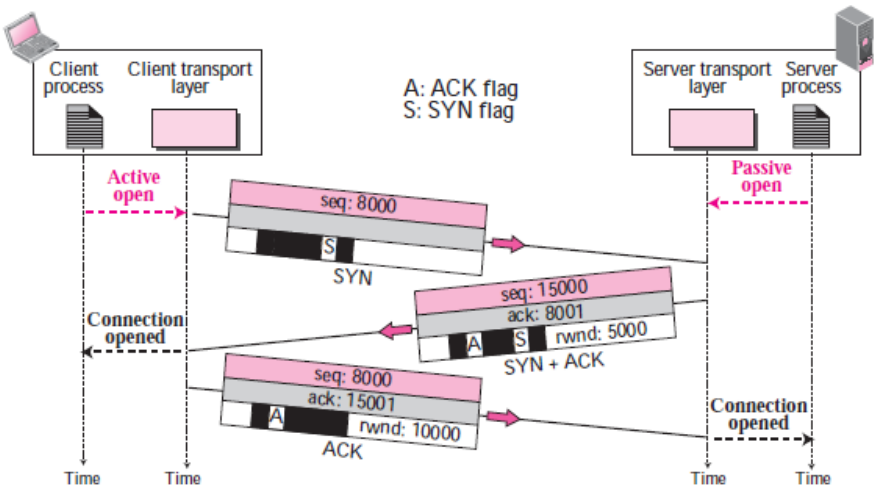
In TCP, connection-oriented transmission requires three phases: connection establishment, data transfer, and connection termination.

Connection Establishment

TCP transmits data in full-duplex mode. When two TCPs in two machines are connected, they are able to send segments to each other simultaneously. This implies that each party must initialize communication and get approval from the other party before any data are transferred.

Three-Way Handshaking

The connection establishment in TCP is called three-way handshaking. In our example, an application program, called the client, wants to make a connection with another application program, called the server, using TCP as the transport layer protocol.



1. The client sends the first segment, a SYN segment, in which only the SYN flag is set. This segment is for synchronization of sequence numbers.
2. The server sends the second segment, a SYN + ACK segment with two flag bits set: SYN and ACK. This segment has a dual purpose. First, it is a SYN segment for communication in the other direction. The server uses this segment to initialize a sequence number for numbering the bytes sent from the server to the client. The server also acknowledges the receipt of the SYN segment from the client by setting the ACK flag and displaying the next sequence number it expects to receive from the client.
3. The client sends the third segment. This is just an ACK segment. It acknowledges the receipt of the second segment with the ACK flag and acknowledgment number field.

Simultaneous Open

A rare situation may occur when both processes issue an active open. In this case, both TCPs transmit a SYN + ACK segment to each other and one single connection is established between them.

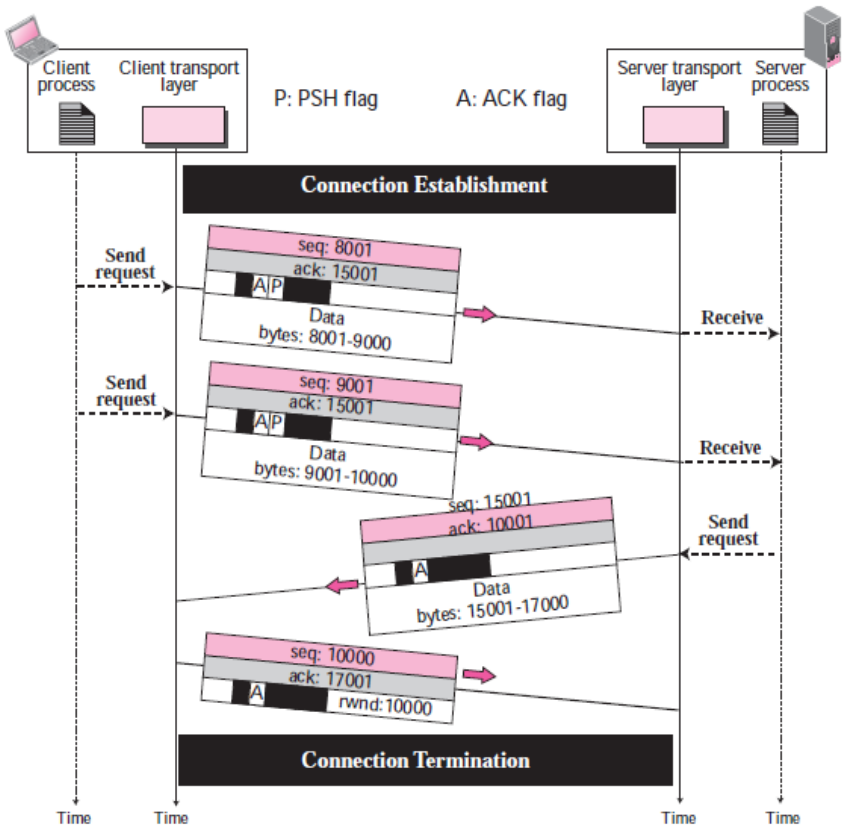
SYN Flooding Attack

The connection establishment procedure in TCP is susceptible to a serious security problem called SYN flooding attack. This happens when one or more malicious attackers send a large number of SYN segments to a server pretending that each of them is coming from a different client by faking the

source IP addresses in the datagrams. The server, assuming that the clients are issuing an active open, allocates the necessary resources. This SYN flooding attack belongs to a group of security attacks known as a **denial of service attack,** in which an attacker monopolizes a system with so many service requests that the system overloads and denies service to valid requests

Data Transfer

After connection is established, bidirectional data transfer can take place. The client and server can send data and acknowledgments in both directions.



Pushing Data

The application program at the sender can request a push operation. This means that the sending TCP must not wait for the window to be filled. It must create a segment and send it immediately. The sending TCP must also set the push bit (PSH) to let the receiving TCP know that the segment includes data that must be delivered to the receiving application program as soon as possible and not to wait for more data to come.

Urgent Data

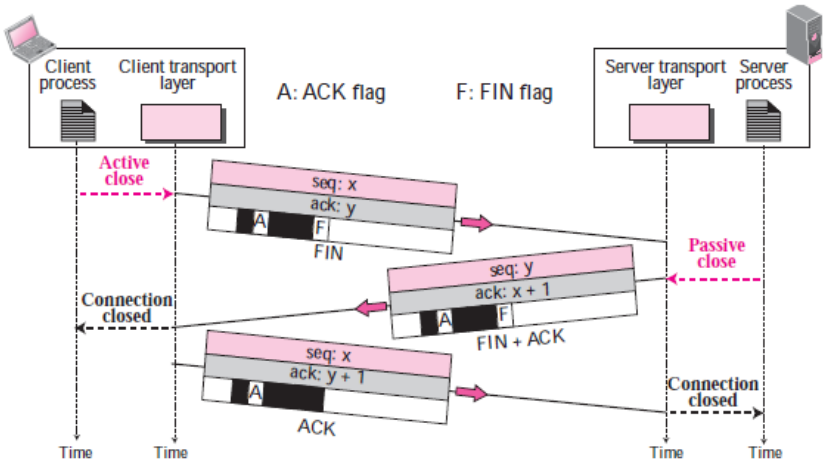
The sending application program tells the sending TCP that the piece of data is urgent. The sending TCP creates a segment and inserts the urgent data at the beginning of the segment. The rest of the segment can contain normal data from the buffer. The urgent pointer field in the header defines the end of the urgent data (the last byte of urgent data).

Connection Termination

Any of the two parties involved in exchanging data (client or server) can close the connection, although it is usually initiated by the client. Most implementations today allow two options for connection termination: three-way handshaking and four-way handshaking with a half-close option.

Three-Way Handshaking

1. In a common situation, the client TCP, after receiving a close command from the client process, sends the first segment, a FIN segment in which the FIN flag is set. Note that a FIN segment can include the last chunk of data sent by the client or it can be just a control segment.



2. The server TCP, after receiving the FIN segment, informs its process of the situation and sends the second segment, a FIN+ACK segment, to confirm the receipt of the FIN segment from the client and at the same time to announce the closing of the connection in the other direction.
3. The client TCP sends the last segment, an ACK segment, to confirm the receipt of the FIN segment from the TCP server.

Half-Close

In TCP, one end can stop sending data while still receiving data. This is called a half close.

Connection Reset

TCP at one end may deny a connection request, may abort an existing connection, or may terminate an idle connection. All of these are done with the RST (reset) flag.

Denying a Connection

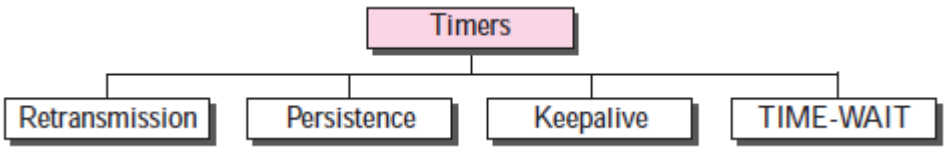
Suppose the TCP on one side has requested a connection to a nonexistent port. The TCP on the other side may send a segment with its RST bit set to deny the request.

Aborting a Connection

One TCP may want to abort an existing connection due to an abnormal situation. It can send an RST segment to close the connection.

TCP TIMERS

To perform its operation smoothly, most TCP implementations use at least four timers.



Retransmission Timer

To retransmit lost segments, TCP employs one retransmission timer (for the whole connection period) that handles the retransmission time-out (RTO), the waiting time for an acknowledgment of a segment.

Round-Trip Time (RTT)

To calculate the retransmission time-out (RTO), we first need to calculate the roundtrip time (RTT).

Measured RTT

We need to find how long it takes to send a segment and receive an acknowledgment for it. This is the measured RTT.

Smoothed RTT

The measured RTT, RTT_M, is likely to change for each round trip. The fluctuation is so high in today's Internet that a single measurement alone cannot be used for retransmission time-out purposes. Most implementations use a smoothed RTT, called RTT_S.

Retransmission Time-out (RTO)

The value of RTO is based on the smoothed round-trip time and its deviation.

Persistence Timer

To deal with a zero-window-size advertisement, TCP needs another timer. If the receiving TCP announces a window size of zero, the sending TCP stops transmitting segments until the receiving TCP sends an ACK segment announcing a nonzero window size. This ACK segment can be lost.

Keepalive Timer

A keepalive timer is used in some implementations to prevent a long idle connection between two TCPs. Suppose that a client opens a TCP connection to a server, transfers some data, and becomes silent. Perhaps the client has crashed. In this case, the connection remains open forever.

TIME-WAIT Timer

The TIME-WAIT (2MSL) timer is used during connection termination.