Module-2

TRANSPORT LAYER

Module – 2

Transport Layer : Introduction and Transport-Layer Services: Relationship Between Transport and Network Layers, Overview of the Transport Layer in the Internet, Multiplexing and Demultiplexing: Connectionless Transport: UDP,UDP Segment Structure, UDP Checksum, Principles of Reliable Data Transfer: Building a Reliable Data Transfer Protocol, Pipelined Reliable Data Transfer Protocols, Go-Back-N, Selective repeat, Connection-Oriented Transport TCP: The TCP Connection, TCP Segment Structure, Round-Trip Time Estimation and Timeout, Reliable Data Transfer, Flow Control, TCP Connection Management, Principles of Congestion Control: The Causes and the Costs of Congestion, Approaches to Congestion Control.

Introduction

Transport layer resides in between Application layer and Network layer. It has the critical role of providing communication services directly to the application processes running on different hosts.

2.1 Introduction and Transport-Layer Services

- A transport-layer protocol provides for logical communication between application processes running on different hosts.
- **Logical communication** - from an application's perspective, it is assumed as, the hosts running the processes were directly connected; in reality, the hosts may be in remote location, connected via numerous routers and a wide range of link types.
- Application processes use the logical communication provided by the transport layer to send messages to each other, without the knowledge of physical infrastructure used to carry these messages.

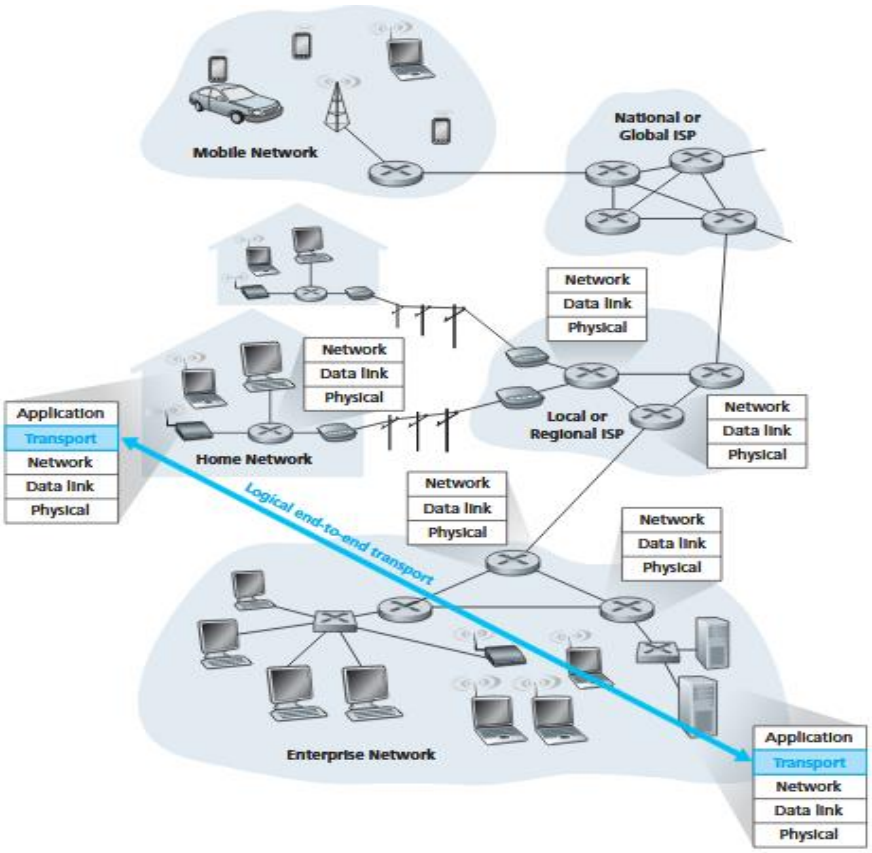


Fig 2.1: Transport layer provides logical communication between application processes

As shown in Figure 2.1, transport-layer protocols are implemented in the end systems but not in network routers.

- On the sending side, the transport layer converts the application-layer messages it receives from a sending application process into transport-layer packets, known as transport-layer segments.
- This is done by breaking the application messages into smaller chunks and adding a transport-layer header to each chunk to create the transport-layer segment.
- The transport layer then passes the segment to the network layer at the sending end system, where the segment is encapsulated within a network-layer packet (a datagram) and sent to the destination.
- Network routers act only on the network-layer fields of the datagram; that is, they do not examine the fields of the transport-layer segment encapsulated with the datagram.
- On the receiving side, the network layer extracts the transport-layer segment from the datagram and passes the segment up to the transport layer.
- The transport layer then processes the received segment, making the data in the segment available to the receiving application.

Internet has two protocols—TCP and UDP.

2.1.1 Relationship Between Transport and Network Layers

- Transport-layer protocol provides logical communication between *processes* running on different hosts, a network-layer protocol provides logical communication between *hosts*.
- Transport-layer protocols live in the end systems.
- Within an end system, a transport protocol moves messages from application processes to the network layer and vice versa.
- Intermediate routers will not recognize, any information that the transport layer may have added to the application messages.
- Transport layer provides reliable service where as network layer provides unreliable service.

2.1.2 Overview of the Transport Layer in the Internet

Two protocols at transport layer are :

UDP (User Datagram Protocol), provides an unreliable, connectionless service to the invoking application.

TCP (Transmission Control Protocol), provides a reliable, connection-oriented service to the invoking application.

Transport layer packet is referred as a *segment*. Transport-layer packet for TCP is referred as a segment and for UDP as a datagram.

Responsibility of IP at Network layer:

- The Internet's network-layer protocol has a name—IP, for Internet Protocol. IP provides logical communication between hosts.
- The IP service model is a best-effort delivery service. IP makes "best effort" to deliver segments between communicating hosts.
- IP does not guarantee segment delivery, orderly delivery of segments and the integrity of the data in the segments. Hence, IP is an unreliable service.
- Every host has at least one network-layer address, called IP address.

Responsibility of UDP and TCP at Transport layer:

UDP and TCP extends IP's delivery service between two end systems to a delivery service between two processes running on the end systems.

Extending host-to-host delivery to process-to-process delivery is called transport-layer multiplexing and demultiplexing.

UDP and TCP provides integrity checking by including error detection fields in their segments' headers.

Services provided by UDP & TCP :

UDP is an unreliable service—it does not guarantee that data sent by one process will arrive intact to the destination process. UDP traffic is unregulated.

TCP Service :

- Provides reliable data transfer. Using flow control, sequence numbers, acknowledgments, and timers.
- TCP ensures that data is delivered from sending process to receiving process, correctly and in order.
- TCP converts IP's unreliable service between end systems into a reliable data transport service between processes.
- TCP provides congestion control.
- TCP congestion control prevents any one TCP connection from swamping the links and routers between communicating hosts with an excessive amount of traffic.
- TCP strives to give each connection traversing a congested link an equal share of the link bandwidth.
- Regulates the rate at which the sending sides of TCP connections can send traffic into the network.

2.2 Multiplexing and Demultiplexing

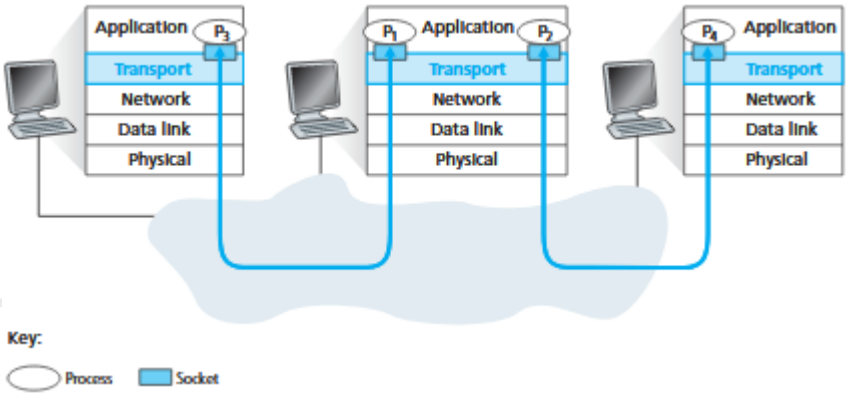


Fig 2.2.1 : Transport layer multiplexing and demultiplexing

- When the transport layer in receiving host receives data from the network layer below, it needs to direct the received data to one of the four processes P1,P2,P3,P4.

- A process can have one or more sockets, through which data passes from the network to the process and through which data passes from the process to the network. Each socket has a unique identifier.
- Thus, as shown in Figure above, the transport layer in the receiving host does not deliver data directly to a process, but instead to an intermediary socket.
- Each transport-layer segment has a set of fields in the segment to redirect data to above layer. At the receiving end, the transport layer examines these fields to identify the receiving socket and then directs the segment to that socket.
- Delivering the data in a transport-layer segment to the correct socket is called demultiplexing.
- Gathering data chunks at the source host from different sockets, encapsulating each data chunk with header information to create segments, and passing the segments to the network layer is called multiplexing.
- The transport layer in the middle host in Figure above must demultiplex segments arriving from the network layer below to either process P1 or P2 above;
- Demultiplexing is done by directing the arriving segment's data to the corresponding process's socket.
- The transport layer in the middle host must also gather outgoing data from these sockets, form transport-layer segments, and pass these segments down to the network layer

Transport-layer multiplexing requires :

(1) Unique identifiers for sockets

(2) Each segment must have special fields that indicate the socket to which the segment is to be delivered.

These special fields,are the source port number field and the destination port number field.

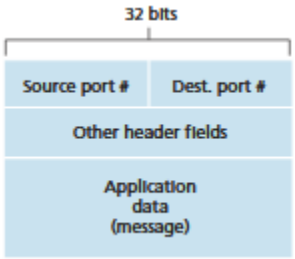


Fig 2.2.2 Source and destination port number fields in a TL segment

Each port number is a 16-bit number, ranging from 0 to 65535. The port numbers ranging from 0 to 1023 are called well-known port numbers They are reserved for use by well-known application protocols such as HTTP (80) and FTP (21).

Each socket in the host could be assigned a port number, and when a segment arrives at the host, the transport layer examines the destination port number in the segment and directs the segment to the corresponding socket. The segment's data then passes through the socket into the attached process.

Connectionless Multiplexing and Demultiplexing

clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)

When a UDP socket is created, the transport layer automatically assigns a port number to the socket.

The transport layer assigns a port number in the range 1024 to 65535 that is currently not being used by any other UDP port in the host.

Associate a specific port number (say, 19157) to this UDP socket via the socket bind() method:

clientSocket.bind((‘’, 19157))

UDP multiplexing/demultiplexing :

Suppose a process in Host A, with UDP port 19157, wants to send a chunk of application data to a process with UDP port 46428 in Host B.

The transport layer in Host A creates a transport-layer segment that includes the application data, the source port number (19157), the destination port number (46428).

The transport layer then passes the resulting segment to the network layer.

The network layer encapsulates the segment in an IP datagram and makes a best-effort attempt to deliver the segment to the receiving host.

If the segment arrives at the receiving Host B, the transport layer at the receiving host examines the destination port number in the segment (46428) and delivers the segment to its socket identified by port 46428.

As UDP segments arrive from the network, Host B directs (demultiplexes) each segment to the appropriate socket by examining the segment's destination port number.

UDP socket is fully identified by a two-tuple consisting of a destination IP address and a destination port number.

If two UDP segments have different source IP addresses and/or source port numbers, but have the same *destination* IP address and *destination* port number, then the two segments will be directed to the same destination process via the same destination socket.

Host A-to-B : In the segment , the source port number serves as part of a "return address"—when B wants to send a segment back to A, the destination port in the B-to-A segment will take its value from the source port value of the A-to-B segment.

UDPServer.py, the server uses the recvfrom() method to extract the clientside (source) port number from the segment it receives from the client; it then sends a new segment to the client, with the extracted source port number serving as the destination port number in this new segment.

Connection-Oriented Multiplexing and Demultiplexing :

Difference between a TCP socket and a UDP socket is that a TCP socket is identified by a four-tuple: (source IP address, source port number, destination IP address, destination port number).

When a TCP segment arrives from the network to a host, the host uses all four values to direct (demultiplex) the segment to the appropriate socket.

Two arriving TCP segments with different source IP addresses or source port numbers will be directed to two different sockets.

• The TCP server application has a "welcoming socket," that waits for connection establishment requests from TCP clients on port number 12000.

• The TCP client creates a socket and sends a connection establishment request segment with the lines:

clientSocket = socket(AF_INET, SOCK_STREAM)

clientSocket.connect((serverName,12000))

• A connection-establishment request is a TCP segment with destination port number 12000 and a special connection-establishment bit set in the TCP header. The segment also includes a source port number that was chosen by the client.

• When the host operating system of the computer running the server process receives the incoming connection-request segment with destination port 12000, it locates the server process that is waiting to accept a connection on port number 12000. The server process then creates a new socket:

connectionSocket, addr = serverSocket.accept( )

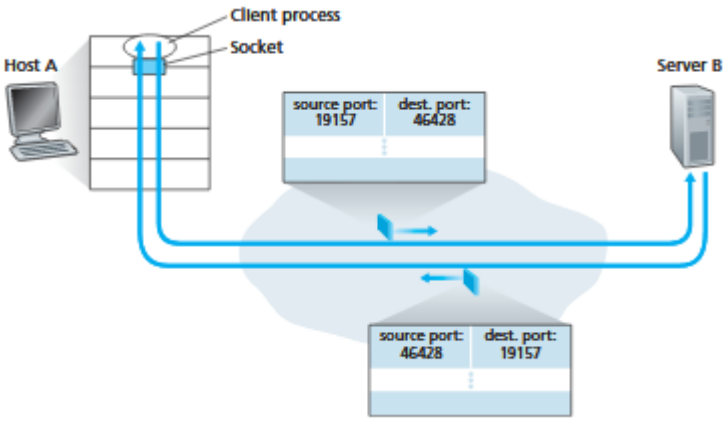


Fig 2.2.3 : The inversion of source and destination port numbers

The transport layer at the server notes the following four values in the connection-request segment:

(1) the source port number in the segment,

(2) the IP address of the source host,

(3) the destination port number in the segment, and

(4) its own IP address.

The newly created connection socket is identified by these four values; all subsequently arriving segments whose source port, source IP address, destination port, and destination IP address match these four values will be demultiplexed to this socket.

With the TCP connection, the client and server can now send data to each other.

The server host may support many simultaneous TCP connection sockets, with each socket attached to a process, and with each socket identified by its own four tuple.

When a TCP segment arrives at the host, all four fields (source IP address, source port, destination IP address, destination port) are used to direct (demultiplex) the segment to the appropriate socket.

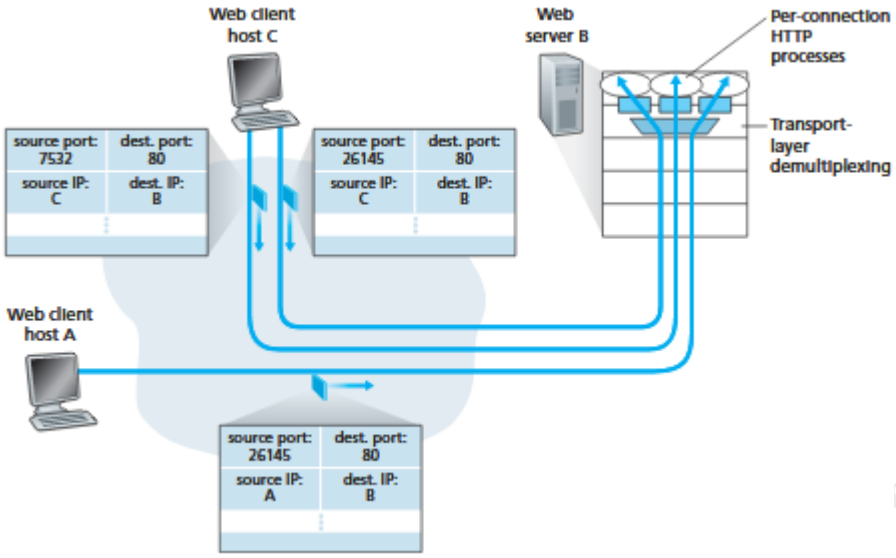


Fig 2.2.4: Two clients using same destination port number (80) to communicate with the same Web server application

Eg. Consider above Figure, in which Host C initiates two HTTP sessions to server B, and Host A initiates one HTTP session to B. Hosts A and C and server B each have their own unique IP address—A, C, and B, respectively. Host C assigns two different source port numbers (26145 and 7532) to its two HTTP connections. Because Host A is choosing source port numbers independently of C, it might also assign a source port of 26145 to its HTTP connection.

Server B will still be able to correctly demultiplex the two connections having the same source port number, since the two connections have different source IP addresses.

Web Servers and TCP

Consider a host running a Web server, such as an Apache Web server, on port 80.

When clients (for example, browsers) send segments to the server, *all* segments will have destination port 80. In particular, both the initial connection-establishment segments and the segments carrying HTTP request messages will have destination port 80.

The server distinguishes the segments from the different clients using source IP addresses and source port numbers.

As shown in Figure, each of these processes has its own connection socket through which HTTP requests arrive and HTTP responses are sent.

There is not always a one-to-one correspondence between connection sockets and processes. Web servers often use only one process and create a new thread with a new connection socket for each new client connection.

2.3 Connectionless Transport: UDP

The transport layer has to provide a multiplexing/demultiplexing service in order to pass data between the network layer and the correct application-level process.

UDP, does the multiplexing/demultiplexing function , error checking.

UDP takes messages from the application process, attaches source and destination port number fields for the multiplexing/demultiplexing service, adds two other small fields, and passes the resulting segment to the network layer.

The network layer encapsulates the transport-layer segment into an IP datagram and then makes a best-effort attempt to deliver the segment to the receiving host.

If the segment arrives at the receiving host, UDP uses the destination port number to deliver the segment's data to the correct application process.

UDP has no handshaking between sending and receiving transport-layer entities before sending a segment. Hence, UDP is said to be *connectionless.*

UDP is best suited for many applications for the following reasons:

- **Finer application-level control over what data is sent, and when.**

Under UDP, as soon as an application process passes data to UDP, UDP will package the data inside a UDP segment and immediately pass the segment to the network layer.

TCP, on the other hand, has a congestion-control mechanism that throttles the transport-layer TCP sender when one or more links between the source and destination hosts become excessively congested.

TCP will continue to resend a segment until the receipt of the segment has been acknowledged by the destination.

Since real-time applications often require a minimum sending rate, do not want to overly delay segment transmission, and can tolerate some data loss, TCP's service model is not particularly well matched to these applications' needs.

- **No connection establishment.**

TCP uses a three-way handshake before it starts to transfer data. UDP just sends data away without any formal preliminaries.

Thus UDP does not introduce any delay to establish a connection.

HTTP uses TCP rather than UDP, since reliability is critical for Web pages with text.

• No connection state.

TCP maintains connection state in the end systems. This connection state includes receive and send buffers, congestion-control parameters and sequence and acknowledgment number parameters.

UDP, does not maintain connection state and does not track any of these parameters.

• Small packet header overhead.

The TCP segment has 20 bytes of header overhead in every segment, whereas UDP has only 8 bytes of overhead.

E-mail, remote terminal access, the Web, and file transfer run over TCP—all these applications need the reliable data transfer service of TCP.

UDP and TCP are used today with multimedia applications, such as Internet phone, real-time video conferencing, and streaming of stored audio and video.

The lack of congestion control in UDP can result in high loss rates between a UDP sender and receiver.

2.3.1 UDP Segment Structure

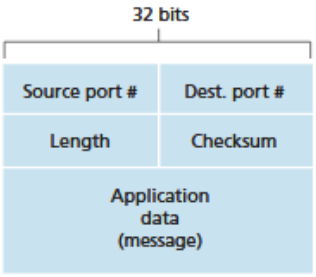


Fig 2.3.1: UDP Segment structure

The application data occupies the data field of the UDP segment. For example, For a streaming audio application, audio samples fill the data field.

The UDP header has only four fields, each consisting of two bytes. The port numbers allow the destination host to pass the application data to the correct process running on the destination end system (that is, to perform the demultiplexing function).

The length field specifies the number of bytes in the UDP segment (header plus data). An explicit length value is needed since the size of the data field may differ from one UDP segment to the next.

The checksum is used by the receiving host to check whether errors have been introduced into the segment.

The length field specifies the length of the UDP segment, including the header, in bytes.

2.3.2 UDP Checksum

The UDP checksum provides for error detection. That is, the checksum is used to determine whether bits within the UDP segment have been altered as it moved from source to destination.

UDP at the sender side performs the 1s complement of the sum of all the 16-bit words in the segment, with any overflow encountered during the sum being wrapped around. This result is put in the checksum field of the UDP segment.

Example, suppose that we have the following three 16-bit words:

0110011001100000

0101010101010101

1000111100001100

The sum of first two of these 16-bit words is

0110011001100000

0101010101010101

1011101110110101

Adding the third word to the above sum gives

```
                1011101110110101
                1000111100001100
carry → 1       0100101011000001
                                1
                0100101011000010
```

Last addition had overflow, which was wrapped around. The 1s complement is obtained by converting all the 0s to 1s and converting all the 1s to 0s.

Thus the 1s complement of the sum 0100101011000010 is 1011010100111101,

which becomes the checksum.

At the receiver, all four 16-bit words are added, including the checksum. If no errors are introduced into the packet,

then the sum at the receiver will be 1111111111111111.

If one of the bits is a 0, then errors have been introduced into the packet.

Take 1's compliment of generated sum,which is all 0's. Hence, no error has been detected.

UDP provides error checking because there is no guarantee that all the links between source and destination provide error checking; that is, one of the links may use a link-layer protocol that does not provide error checking.

Even if segments are correctly transferred across a link, it's possible that bit errors could be introduced when a segment is stored in a router's memory.

Neither link-by-link reliability nor in-memory error detection is guaranteed, UDP must provide error detection at the transport layer, *on an end-end basis*, if the end-end data transfer service is to provide error detection.

UDP has no error recovery method.

2.4 Principles of Reliable Data Transfer

With the reliable channel, no transferred data bits are corrupted or lost, and all are delivered in the order in which they were sent. This is the service model offered by TCP to the Internet applications that invoke it. It is the responsibility of a reliable data transfer protocol to implement this service abstraction.

The layer *below* the reliable data transfer protocol may be unreliable. For example, TCP is a reliable data transfer protocol that is implemented on top of an unreliable (IP) end-to-end network layer.

Figure 3.8(b) illustrates the interfaces for data transfer protocol.

The sending side of the data transfer protocol will be invoked from above by a call to rdt_send().Here rdt stands for *reliable data transfer* protocol and _send indicates that the sending side of rdt is being called.

It will pass the data to be delivered to the upper layer at the receiving side.

On the receiving side, rdt_rcv() will be called when a packet arrives from the sending side of the channel.

When the rdt protocol wants to deliver data to the upper layer, it will do so by calling deliver_data().

Here unidirectional data transfer is considered, that is, data transfer from the sending to the receiving side.

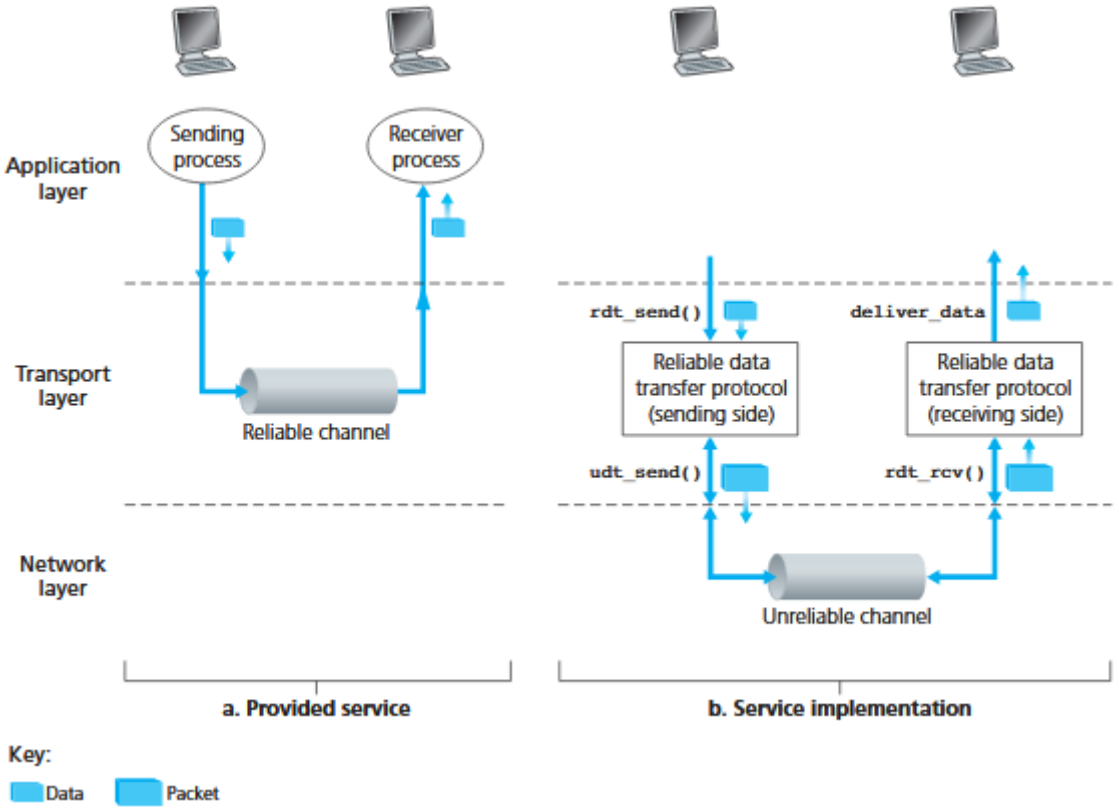


Fig 2.4 : Reliable data transfer: Service model and implementation

In addition to exchanging packets containing the data to be transferred, the sending and receiving sides of rdt will also need to exchange control packets back and forth.

Both the send and receive sides of rdt send packets to the other side by a call to udt_send( ).

2.4.1 Building a Reliable Data Transfer Protocol

Reliable Data Transfer over a Perfectly Reliable Channel: rdt1.0

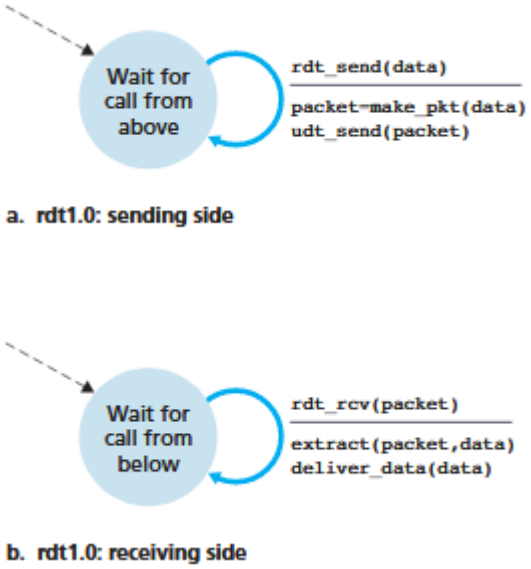


Fig 2.4.1.1 : Reliable Data Transfer over a Perfectly Reliable Channel: rdt1.0

Consider the simplest case, in which the underlying channel is completely reliable. The protocol is called **rdt1.0**. The **finite-state machine (FSM)** definitions for the rdt1.0 sender and receiver are shown in Figure above.

The FSM in Figure (a) defines the operation of the sender, while the FSM in Figure (b) defines the operation of the receiver.

There are *separate* FSMs for the sender and for the receiver. The sender and receiver FSMs in Figure each have just one state.

The arrows in the FSM description indicate the transition of the protocol from one state to another.

Since each FSM in Figure has just one state, a transition is necessarily from the one state back to itself;

The event causing the transition is shown above the horizontal line labeling the **transition**, and the **actions** taken when the event occurs are shown below the horizontal line.

When no action is taken on an event, or no event occurs and an action is taken, the symbol '_' below or above the horizontal is used . It explicitly denotes the lack of an action or event.

The initial state of the FSM is indicated by the dashed arrow.

The sending side of rdt simply accepts data from the upper layer via the rdt_send(data) event, creates a packet containing the data via the action make_pkt(data)) and sends the packet into the channel.

The rdt_send(data) event would result from a procedure call by the upper-layer application.

On the receiving side, rdt receives a packet from the underlying channel via the rdt_rcv(packet) event, removes the data from the packet (via the action extract (packet, data)) and passes the data up to the upper layer (via the action deliver_data(data)).

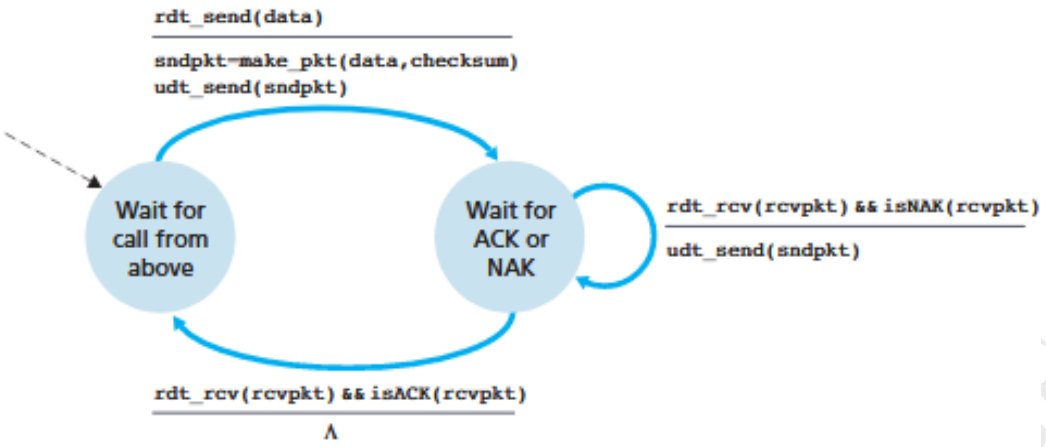
The rdt_rcv(packet) event would result from a procedure call (for example, to rdt_rcv()) from the lower layer protocol.

The packet flows from the sender to receiver; with a perfectly reliable channel there is no need for the receiver side to provide any feedback to the sender

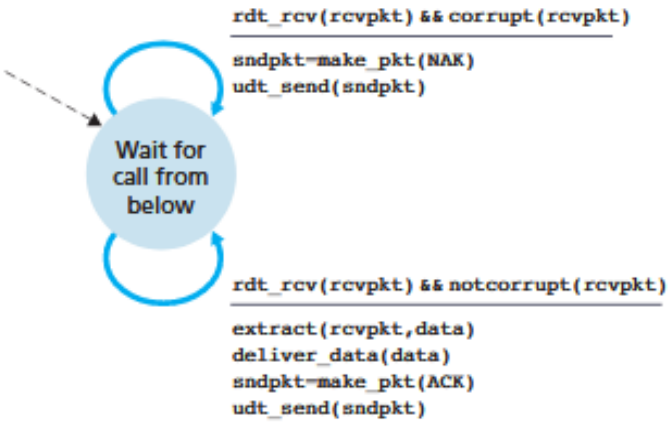
The receiver is able to receive data as fast as the sender happens to send data. Thus, there is no need for the sender to slow down its sending rate.

Reliable Data Transfer over a Channel with Bit Errors: rdt2.0

A more realistic model of the underlying channel is one in which bits in a packet may be corrupted.



a. rdt2.0: sending side



b. rdt2.0: receiving side

Such bit errors occur in the physical components of a packet is transmitted, propagates, or is buffered.

Assume that all transmitted packets are received (although their bits may be corrupted) in the order in which they were sent.

This message-dictation protocol uses both **positive acknowledgments** and **negative acknowledgments** . These control messages allow the receiver to let the sender know what has been received correctly, and what has been received in error and thus requires repeating.

Reliable data transfer protocols based on retransmission are known as **ARQ** (**Automatic Repeat reQuest) protocols**.

Three additional protocol capabilities are required in ARQ protocols to handle the presence of bit errors:

• ***Error detection***. A mechanism is needed to allow the receiver to detect bit errors if occurred. UDP uses the Internet checksum field for this purpose.

Error detection techniques allow the receiver to detect and possibly correct packet bit errors. These techniques require that extra bits(checksum) be sent from the sender to the receiver; these bits will be gathered into the packet checksum field of the rdt2.0 data packet.

• ***Receiver feedback***. Since the sender and receiver are typically executing on different end systems, it requires for the receiver to provide explicit feedback to the sender. The positive (ACK) and negative (NAK) acknowledgment replies in the message are examples of such feedback. Rdt2.0 protocol sends ACK and NAK packets back from the receiver to the sender. ACK packets is just one bit long; for example, a 0 value could indicate a NAK and a value of 1 could indicate an ACK.

• ***Retransmission***. A packet that is received in error at the receiver will be retransmitted by the sender. Figure shows the FSM representation of rdt2.0, a data transfer protocol employing error detection, positive acknowledgments, and negative acknowledgments.

The sender side of rdt2.0 has two states:

The send-side protocol is waiting for data to be passed down from the upper layer. When the **rdt_send(data**) event occurs, the sender will create a packet (sndpkt) containing the data to be sent, along with a packet checksum and then send the packet via the **udt_send(sndpkt)** operation.

In the rightmost state, the sender protocol is waiting for an ACK or a NAK packet from the receiver. If an ACK packet is received (the notation rdt_rcv(rcvpkt) && isACK (rcvpkt) in Figure corresponds to this event), the sender knows that the most recently transmitted packet has been received correctly .

Thus, the protocol returns to the state of waiting for data from the upper layer. If a NAK is received, the protocol retransmits the last packet and waits for an ACK or NAK to be returned by the receiver in response to the retransmitted data packet.

When the sender is in the wait-for-ACK-or-NAK state, it *cannot* get more data from the upper layer; that is, the rdt_send() event cannot occur; that will happen only after the sender receives an ACK and leaves this state.

Thus, the sender will not send a new piece of data until it is sure that the receiver has correctly received the current packet. Because of this behavior, protocols rdt2.0 is known as **stop-and-wait** protocol.

The **receiver-side** FSM for rdt2.0 still has a single state. On packet arrival, the receiver replies with either an ACK or a NAK, depending on whether or not the received packet is corrupted.

Notation **rdt_rcv(rcvpkt) && corrupt(rcvpkt)** corresponds to the event in which a packet is received and is found to be in error.

ACK or NAK packet could be corrupted and solution could be adding checksum bits to ACK/NAK packets in order to detect such errors.

The difficulty here is that if an ACK or NAK is corrupted, the sender has no way of knowing whether or not the receiver has correctly received the last piece of transmitted data.

Consider three possibilities for handling **corrupted ACKs or NAKs**:

• For the first possibility, if Sender receives garbled ACK/NAK , he creates a new packet – asking the receiver to resend ACK/NAK. But creation of such new packet leads to issues.

• A second alternative is to add enough checksum bits to allow the sender not only to detect, but also to recover from, bit errors. This solves the immediate problem for a channel that can corrupt packets but not lose them.

• A third approach is for the sender simply to resend the current data packet when it receives a garbled ACK or NAK packet. This approach, introduces **duplicate packets** into the sender-to-receiver channel. The fundamental difficulty with duplicate packets is that the receiver doesn't know whether the ACK or NAK it last sent was received correctly at the sender.

Solution :A simple solution to this new problem is to add a new field to the data packet and have the sender number its data packets by putting a **sequence number** into this field.

The receiver then need only check this sequence number to determine whether or not the received packet is a retransmission.

For this simple case of a stop-andwait protocol, a 1-bit sequence number will suffice, since it will allow the receiver to know whether the sender is resending the previously transmitted packet (the sequence number of the received packet has the same sequence number as the most recently received packet) or a new packet.

Since it is assumed that channel does not lose packets, ACK and NAK packets do not themselves need to indicate the sequence number of the packet they are acknowledging.

The sender knows that a received ACK or NAK packet (whether garbled or not) was generated in response to its most recently transmitted data packet.

RDT 2.1 sender(a) :

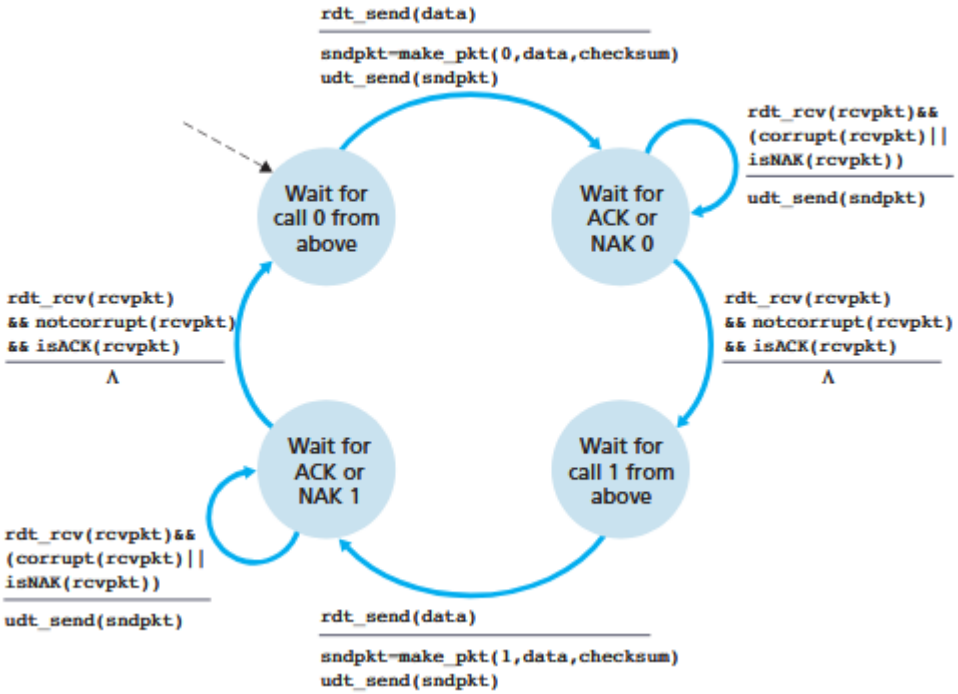


Fig 2.4.1.2: RDT 2.1 sender

RDT 2.1 receiver(b) :

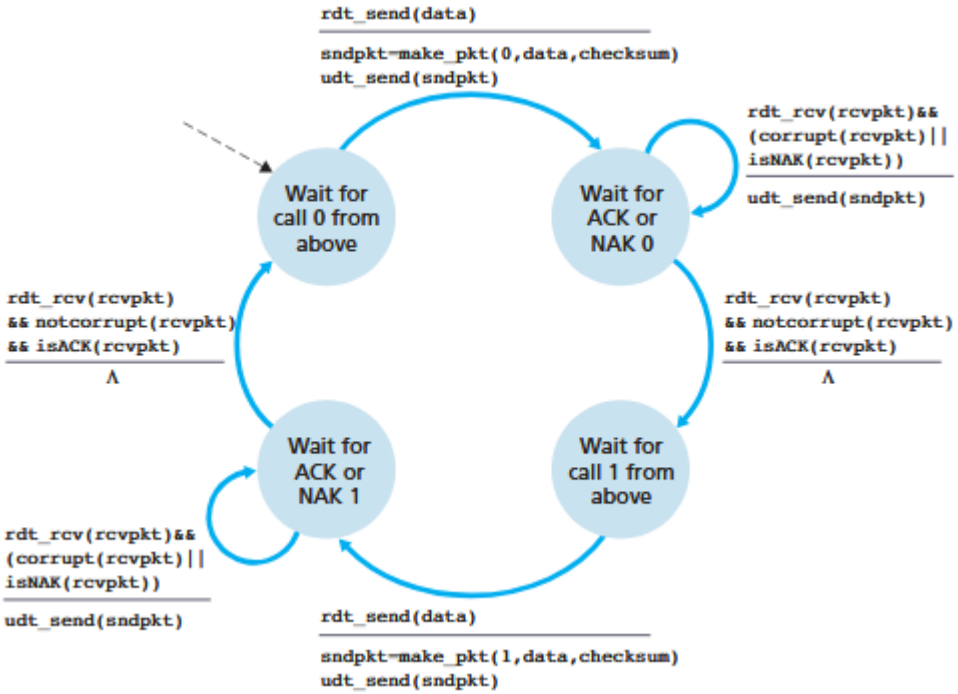


Fig. 2.4.1.3: RDT 2.1 Sender

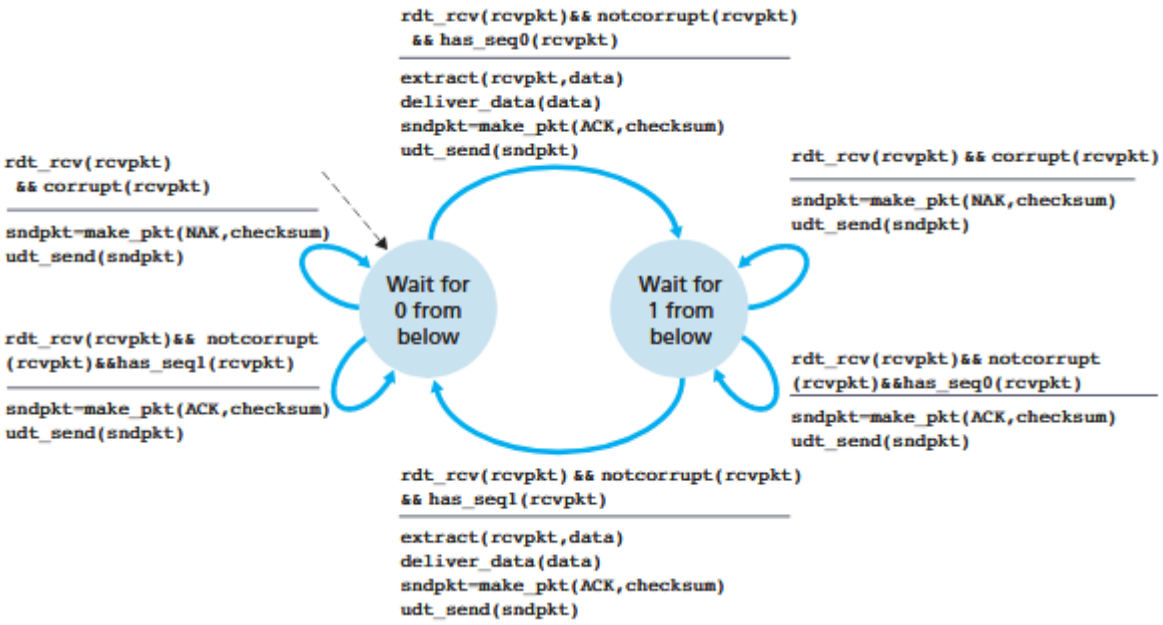


Fig. 2.4.1.4 : RDT 2.1 Receiver

Figures (a) and (b) shows the FSM description for rdt2.1.

The rdt2.1 sender and receiver FSMs each now have twice as many states as before. This is because the protocol state must now reflect whether the packet currently being sent (by the sender) or expected (at the receiver) should have a sequence number of 0 or 1.

Protocol rdt2.1 uses both positive and negative acknowledgments from the receiver to the sender.

When an out-of-order packet is received, the receiver sends a positive acknowledgment for the packet it has received.

When a corrupted packet is received, the receiver sends a negative acknowledgment.

The same effect as a NAK could be accomplished if, instead of sending a NAK, we send an ACK for the last correctly received packet.

A sender that receives two ACKs for the same packet (that is, receives duplicate ACKs) knows that the receiver did not correctly receive the packet following the packet that is being ACKed twice.

RDT 2.2 sender :

NAK-free reliable data transfer protocol for a channel with bit errors is rdt2.2.

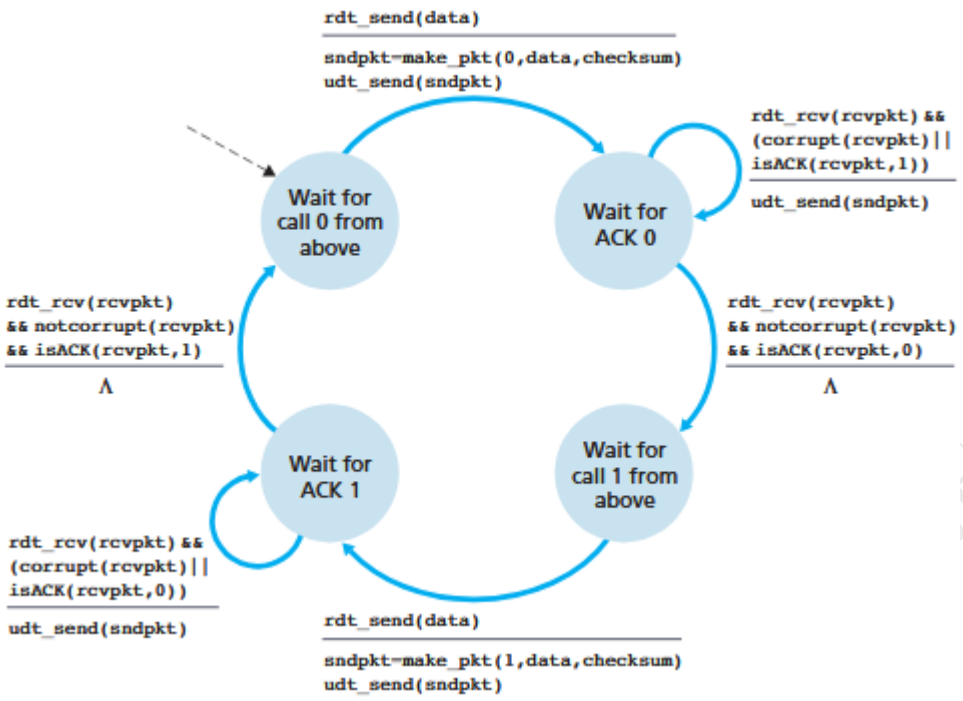


Fig 2.4.1.5 : RDT 2.2 Sender

RDT 2.2 receiver :

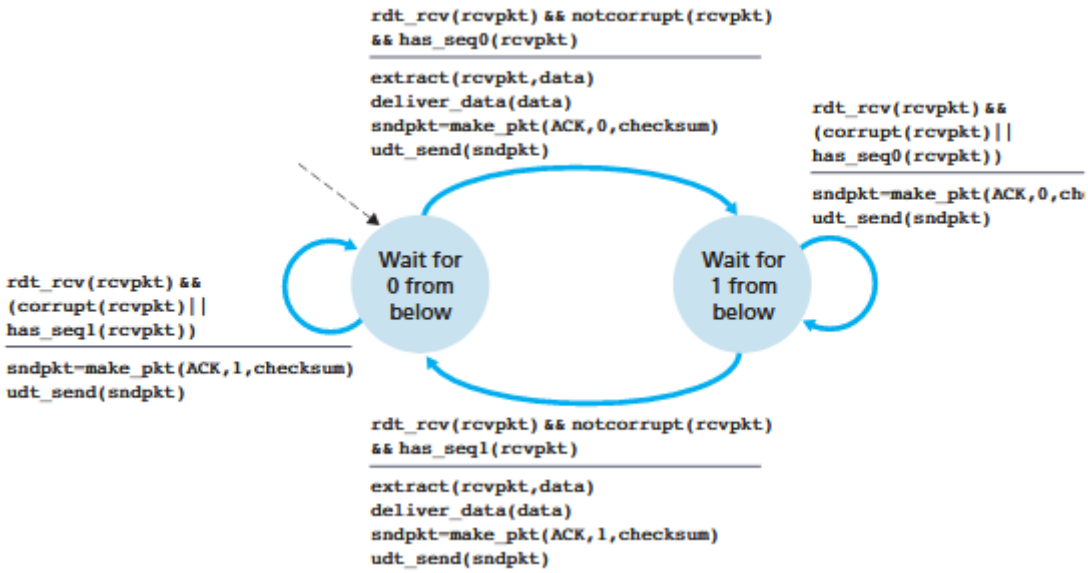


Fig 2.4.1.6 : RDT 2.2 Receiver

The sequence number is included in the packet i.e being acknowledged by an ACK message (this is done by including the ACK,0 or ACK1 argument in make_pkt() in the receiver FSM)

The sender must now check the sequence number of the packet being acknowledged by a received ACK message (this is done by including the 0 or 1 argument in **isACK()** in the sender FSM).

Reliable Data Transfer over a Lossy Channel with Bit Errors: rdt3.0

The use of checksumming, sequence numbers, ACK packets, and retransmissions—are solutions for corrupted data , out of order data , lost packet problems.

Detecting and recovering from lost packets is the responsibility of sender.

Suppose that the sender transmits a data packet and either that packet, or the receiver's ACK of that packet, gets lost. In either case, no reply is forthcoming at the sender from the receiver.

Solution is setting a timer by the sender.

The sender must clearly wait at least as long as a round-trip delay between the sender and receiver plus amount of time is needed to process a packet at the receiver.

If an ACK is not received within this timer, the packet is retransmitted. If a packet experiences a large delay, the sender may retransmit the packet even though neither the data packet nor its ACK have been lost.

This introduces the possibility of **duplicate data packets** in the sender-to-receiver channel. Sequence numbers can be used to handle the case of duplicate packets.

The sender does not know whether a data packet was lost, an ACK was lost, or if the packet or ACK was simply overly delayed. Retransmission is the solution for all these problems.

Implementing a time-based retransmission mechanism requires a **countdown timer** that can interrupt the sender after a given amount of time has expired.

The sender will thus need to be able to

(1) start the timer each time a packet (either a first-time packet or a retransmission) is sent, (2) respond to a timer interrupt (taking appropriate actions)

(3) stop the timer.

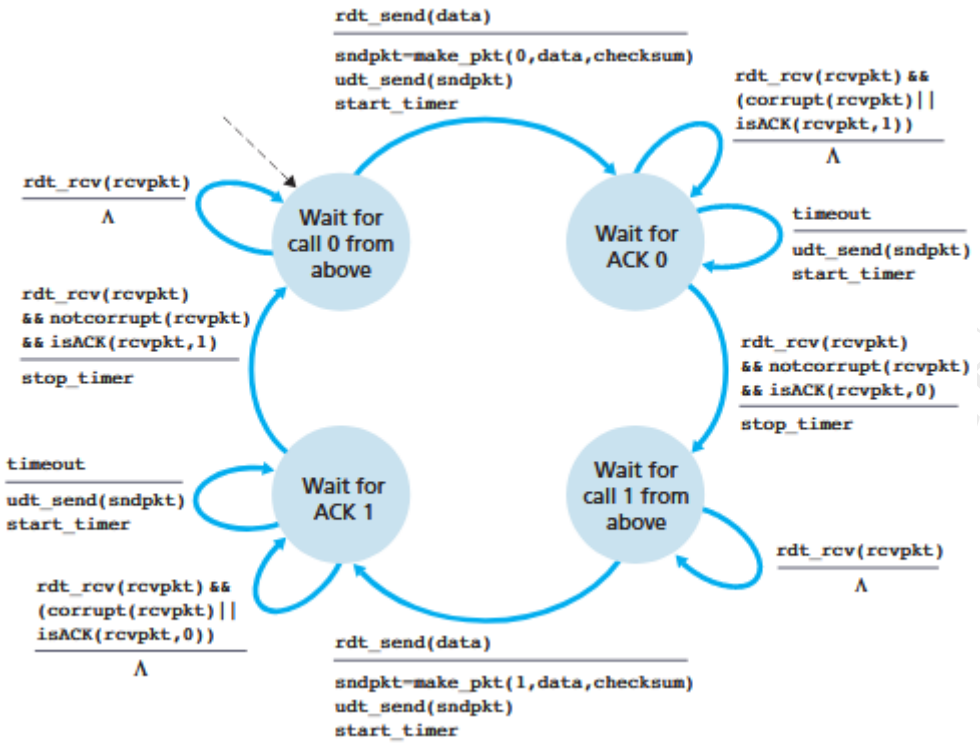


Fig 2.4.1.7 : RDT 3.0 Sender

Figure shows the sender FSM for rdt3.0, a protocol that reliably transfers data over a channel that can corrupt or lose packets.

In Figure below, time moves forward from the top of the diagram toward the bottom of the diagram; Receive time for a packet is necessarily later than the send time for a packet as a result of transmission and propagation delays.

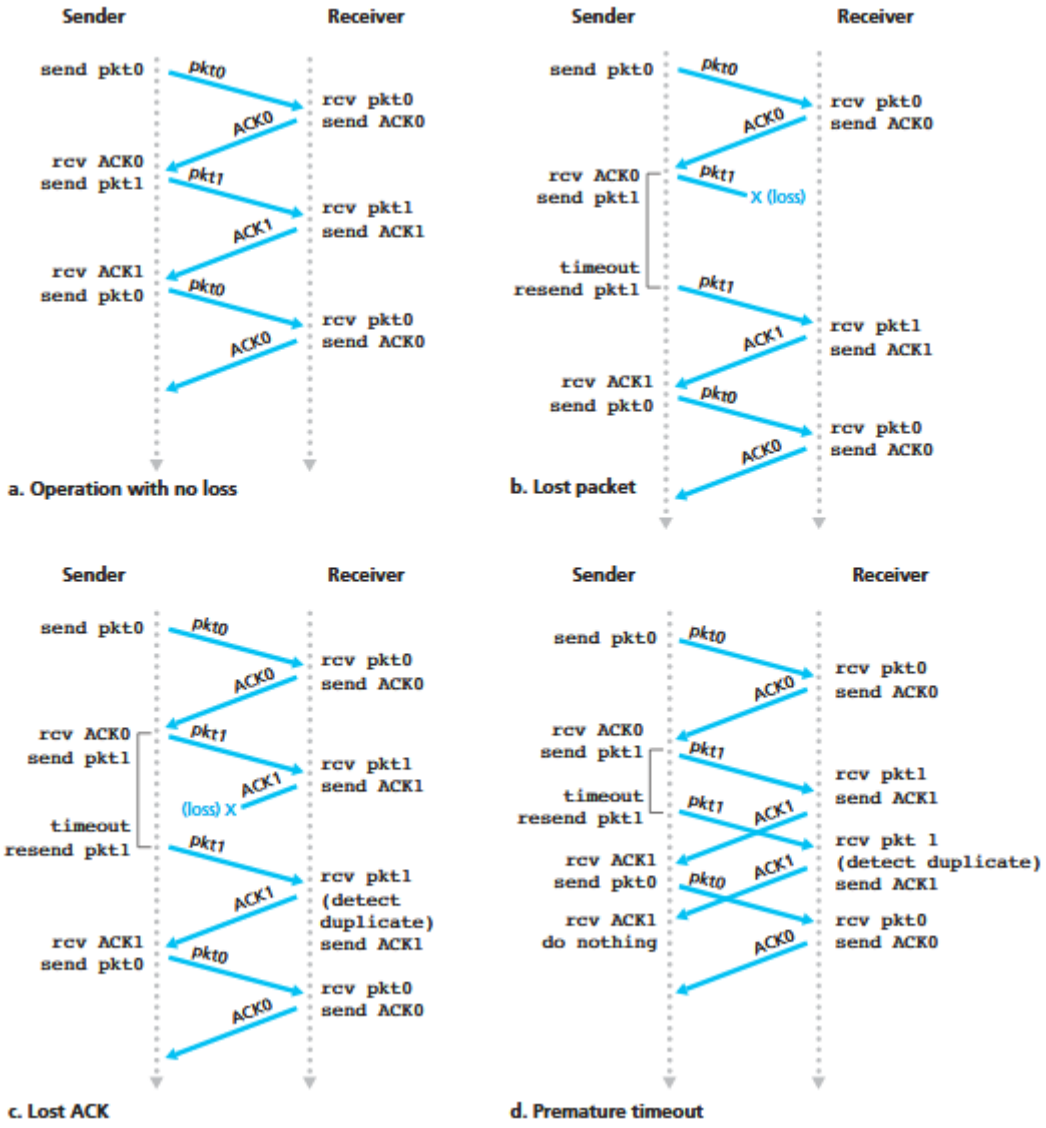


Fig 2.4.1.8 : RDT 3.0 Operation

In Figures (b)–(d), the send-side brackets indicate the times at which a timer is set and later times out.

Because packet sequence numbers alternate between 0 and 1, protocol rdt3.0 is also known as the **alternating-bit protocol**.

2.4.2 Pipelined Reliable Data Transfer Protocols

Consider an idealized case of two hosts, one located on the West Coast of the United States

and the other located on the East Coast, as shown in Figure below.

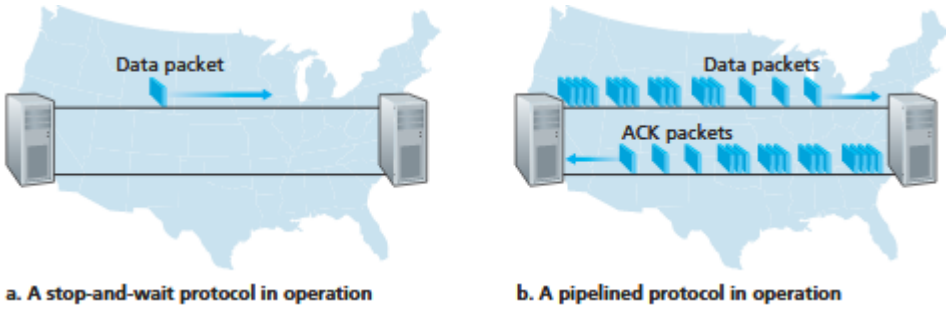


Fig 2.4.2.1 : Stop and wait v/s pipelined protocol

The round-trip propagation delay between these two end systems, RTT, is approximately 30 milliseconds. RTT=30ms

Suppose that they are connected by a channel with a transmission rate, *R*, of 1 Gbps (10^9 bits per second). R=1Gbps

With a packet size, *L*, of 1,000 bytes (8,000 bits) per packet, including both header fields and data. L=8000bitsps

The time needed to actually transmit the packet into the 1 Gbps link is *dtrans =L/R = 8000 bits/packet / 10^9 bits/sec = 8 microseconds*

Figure (a) shows that with stop-and-wait protocol, if the sender begins sending the packet at , $t = 0$, then at $t = L/R = 8$ microseconds;

The packet then makes its 15-msec journey from sender to receiver, RTT/2=15ms.

The last bit of the packet is emerging at the receiver at , t = RTT/2 + L/R =15.008 msec.

The ACK emerges back at the sender at t = RTT + L/R = 30.008 msec.(RTT= 15+15=30ms). At this point, the sender can now transmit the next message. Thus, in 30.008 msec, the sender was sending for only 0.008 msec.

If we define the **utilization** of the sender (or the channel) as the fraction of time the sender is actually busy sending bits into the channel, the analysis in below Figure (a) shows that the stop-and-wait protocol has sender utilization, *U*sender, of

*U*sender = L/R / ($RTT + L/R$) = .008 / 30.008 = 0.00027.

That is, the sender was busy only 2.7 hundredths of one percent of the time!

The solution to this performance problem is: Rather than operate in a stop-and-wait manner, the sender is allowed to send multiple packets without waiting for acknowledgments, as illustrated in below Figure (b).

Figure (b) shows that if the sender is allowed to transmit three packets before having to wait for acknowledgments, the utilization of the sender is essentially tripled.

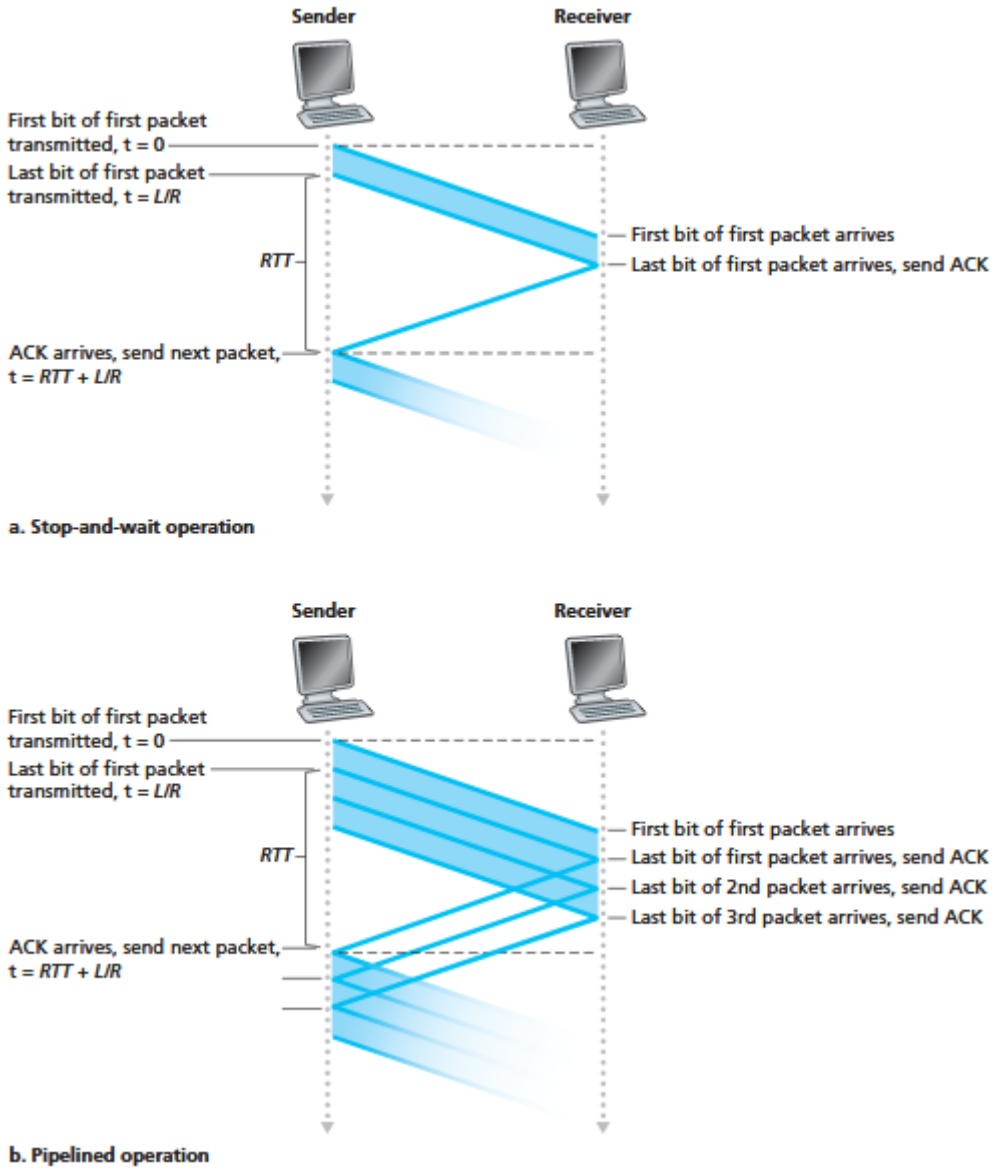


Fig 2.4.2.2 Stop & wait and Pipelined sending

Since the many in-transit sender-to-receiver packets can be visualized as filling a pipeline, this technique is known as **pipelining**.

Pipelining has the following consequences for reliable data transfer protocols:

• The range of sequence numbers must be increased, since each in-transit packet (not counting retransmissions) must have a unique sequence number and there may be multiple, in-transit, unacknowledged packets.

• The sender and receiver sides of the protocols may have to buffer more than one packet. The sender will have to buffer packets that have been transmitted but not yet acknowledged. Buffering of correctly received packets may also be needed at the receiver.

• The range of sequence numbers needed and the buffering requirements will depend on the manner in which a data transfer protocol responds to lost, corrupted and overly delayed packets.

Two basic approaches toward pipelined error recovery can be: **Go-Back-N** and **selective repeat**.

2.4.3 Go-Back-N (GBN)

In a **Go-Back-N (GBN) protocol**, the sender is allowed to transmit multiple packets without waiting for an acknowledgment, but is constrained to have no more than some maximum allowable number, *N*, of unacknowledged packets in the pipeline.

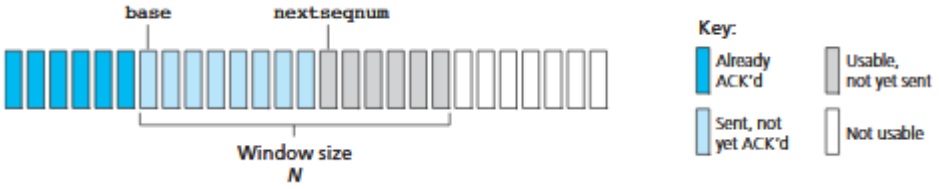


Fig 2.4.3.1 Senders view of sequence numbers in Go-Back-N

Figure above shows the sender's view of the range of sequence numbers in a GBN protocol.

Define **base** to be the sequence number of the oldest unacknowledged packet and **nextseqnum** to be the smallest unused sequence number (that is, the sequence number of the next packet to be sent).

Sequence numbers in the interval [0,base-1] correspond to packets that have already been transmitted and acknowledged. .

The interval [base,nextseqnum-1] corresponds to packets that have been sent but not yet acknowledged.

Sequence numbers in the interval [nextseqnum,base+N-1] can be used for packets that can be sent immediately, should data arrive from the upper layer.

Finally, sequence numbers greater than or equal to base+N cannot be used until an unacknowledged packet currently in the pipeline has been acknowledged.

The range of permissible sequence numbers for transmitted but not yet acknowledged packets can be viewed as a window of size *N* over the range of sequence numbers.

As the protocol operates, this window slides forward over the sequence number space. For this reason, *N* is often referred to as the **window size** and the GBN protocol as a **sliding-window protocol**.

A packet's sequence number is carried in a fixed-length field in the packet header. If *k* is the number of bits in the packet sequence number field, the range of sequence numbers is thus $[0,2^k - 1]$.

The sequence number space can be thought of as a ring of size 2^k, where sequence number $2^k - 1$ is immediately followed by sequence number 0.

The GBN sender must respond to three types of events:

• ***Invocation from above***. When rdt_send() is called from above, the sender first checks to see if the window is full, that is, whether there are *N* outstanding, unacknowledged packets.

If the window is not full, a packet is created and sent, and variables are appropriately updated.

If the window is full, the sender simply returns the data back to the upper layer, an implicit indication that the window is full. The upper layer would have to try again.

• ***Receipt of an ACK***. In our GBN protocol, an acknowledgment for a packet with sequence number *n* will be taken to be a **cumulative acknowledgment**, indicating that all packets with a sequence number up to and including *n* have been correctly received at the receiver.

• ***A timeout event***. *A* timer will be used to recover from lost data or acknowledgment packets. If a timeout occurs, the sender resends *all* packets that have been previously sent but that have not yet been acknowledged.

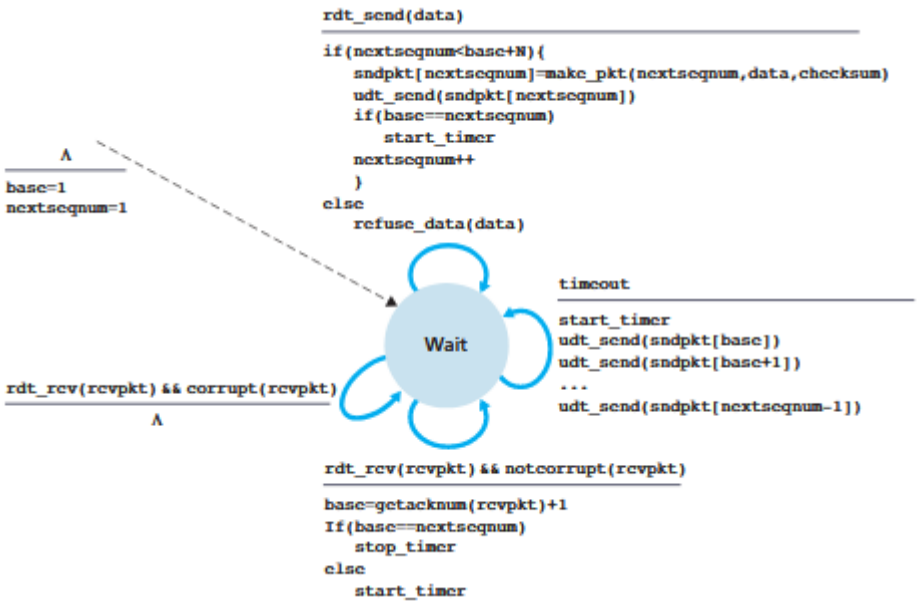


Fig 2.4.3.2 Extended FSM description of GBN Sender

Sender in above Figure uses only a single timer, which can be thought of as a timer for the oldest transmitted but not yet acknowledged packet.

If an ACK is received but there are still additional transmitted but not yet acknowledged packets, the timer is restarted.

If there are no outstanding, unacknowledged packets, the timer is stopped.

The **receiver's action:** If a packet with sequence number *n* is received correctly and is in order , the receiver sends an ACK for packet *n* and delivers the data portion of the packet to the upper layer.

In all other cases, the receiver discards the packet and resends an ACK for the most recently received in-order packet.

Since packets are delivered one at a time to the upper layer, if packet *k* has been received and delivered, then all packets with a sequence number lower than *k* have also been delivered.

Thus, cumulative acknowledgments is used for GBN.

In GBN protocol, the **receiver discards** out-of-order packets because the receiver must deliver data **in order** to the upper layer.

Suppose the packet *n* is expected, but packet *n* + 1 arrives. Because data must be delivered in order, the receiver *could* buffer (save) packet *n* + 1 and then deliver this packet to the upper layer after it had later received and delivered packet *n*.

If packet *n* is lost, both it and packet *n* + 1 will eventually be retransmitted as a result of the GBN retransmission rule at the sender.

Thus, the receiver can simply discard packet *n* + 1.

The advantage of this approach is the receiver need not buffer *any* out-of-order packets.

Thus, while the sender must maintain the upper and lower bounds of its window and the position of nextseqnum within this window.

The only piece of information the receiver need maintain is the sequence number of the next in-order packet.

This value is held in the variable expected seqnum, shown in the receiver FSM in Figure below.

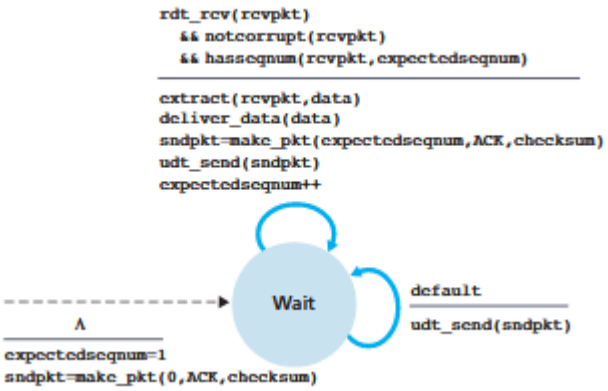


Fig 2.4.3.3 Extended FSM description of GBN Receiver

The disadvantage of discarding a correctly received packet is that the subsequent retransmission of that packet might be lost or garbled and thus even more retransmissions would be required.

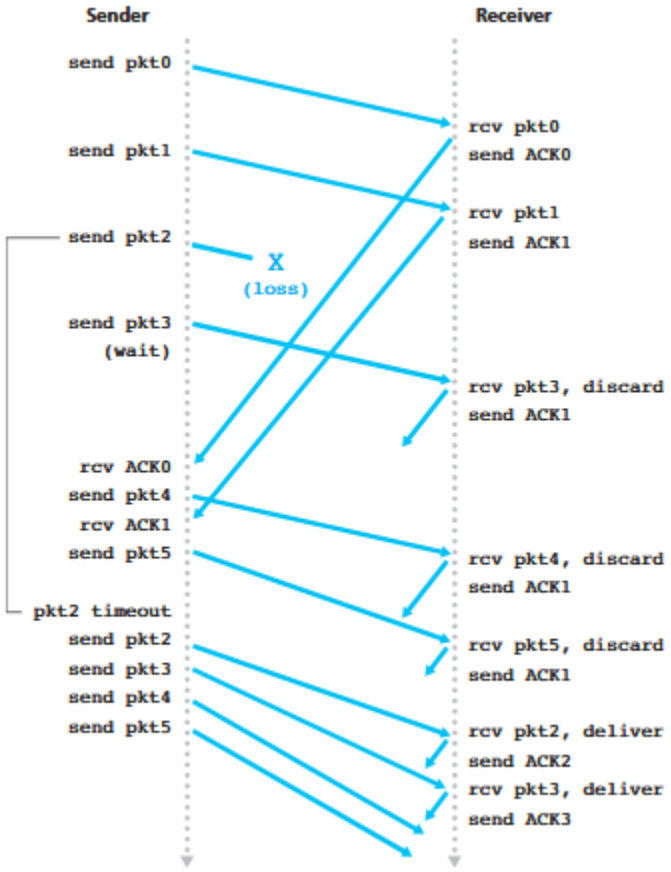


Fig 2.4.3.4 Go Back N in operation

Figure above shows the operation of the GBN protocol for the case of a window size of four packets.

Because of this window size limitation, the sender sends packets 0 through 3 but then must wait for one or more of these packets to be acknowledged before proceeding.

As each successive ACK (for example, ACK0 and ACK1) is received, the window slides forward and the sender can transmit one new packet (pkt4 and pkt5, respectively).

On the receiver side, packet 2 is lost and thus packets 3, 4, and 5 are found to be out of order and are discarded.

In the sender, the events would be :

(1) a call from the upper-layer entity to invoke rdt_send(),

(2) a timer interrupt,

(3) a call from the lower layer to invoke rdt_rcv() when a packet arrives.

2.4.4 Selective Repeat (SR)

The GBN protocol allows the sender to potentially "fill the pipeline" with packets, thus avoiding the channel utilization problems.

GBN has performance problems : When the window size and bandwidth-delay product are both large, many packets can be in the pipeline.

A single packet error can thus cause GBN to retransmit a large number of packets, many unnecessarily.

As the probability of channel errors increases, the pipeline can become filled with these unnecessary retransmissions.

Selective-repeat protocols avoid unnecessary retransmissions by having the sender retransmit only those packets that it suspects were received in error (that is, were lost or corrupted) at the receiver.

This needed retransmission will require that the receiver *individually* acknowledge correctly received packets.

A window size of *N* will again be used to limit the number of outstanding, unacknowledged packets in the pipeline.

The sender will have already received ACKs for some of the packets in the window. Figure shows the SR sender's view of the sequence number space.

Figure details the various actions taken by the SR sender.

The SR receiver will acknowledge a correctly received packet whether or not it is in order. Out-of-order packets are buffered until any missing packets (that is, packets with

lower sequence numbers) are received, at which point a batch of packets can be delivered in order to the upper layer.

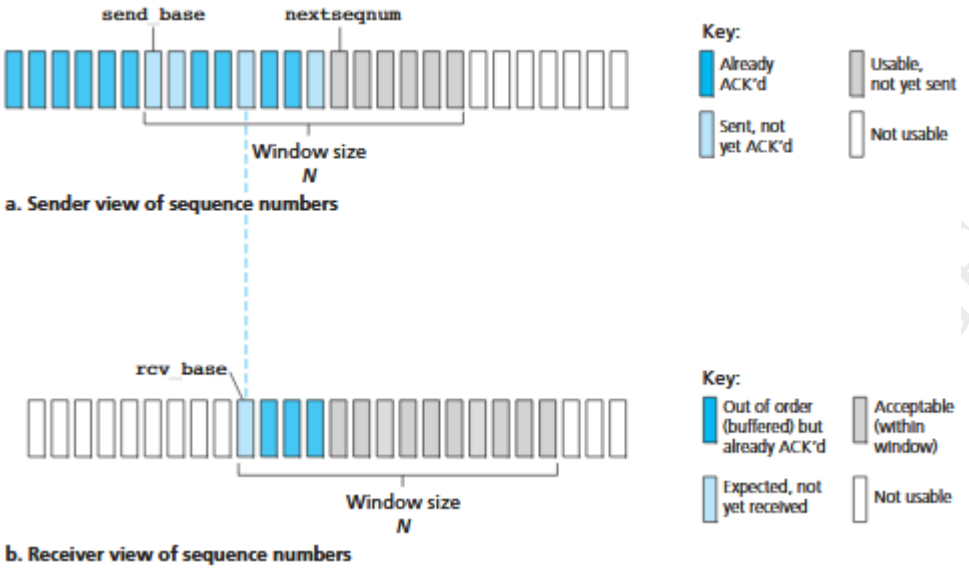


Fig 2.4.4.1 SR sender and receiver views of sequence number space

Figure above itemizes the various actions taken by the SR receiver.

In Step 2 in Figure above, the receiver reacknowledges (rather than ignores) already received packets with certain sequence numbers *below* the current window base.

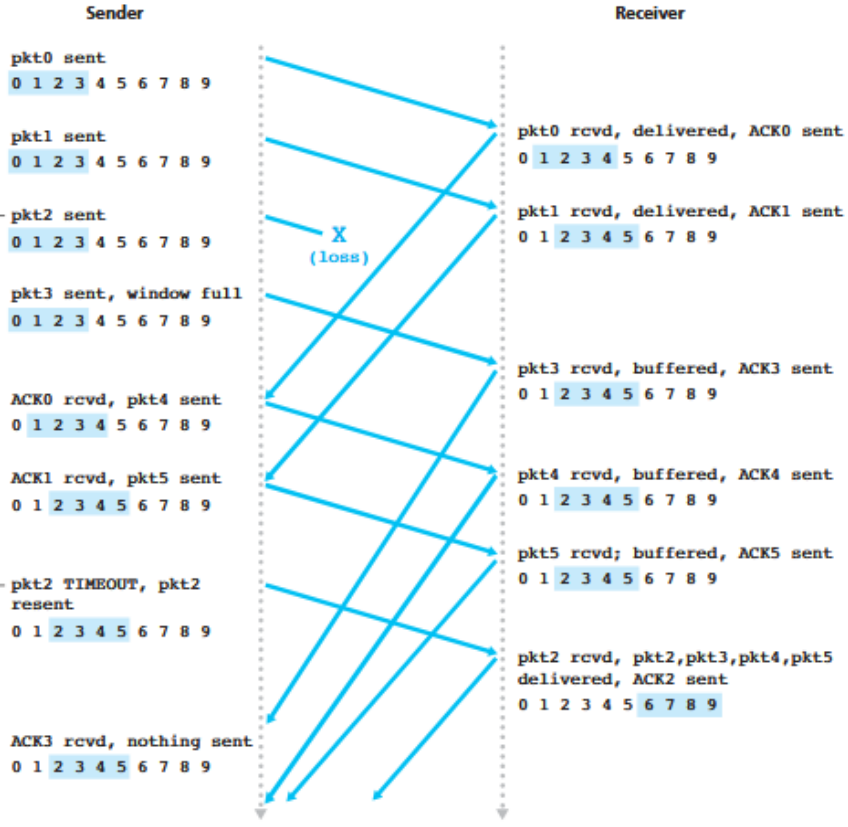


Fig 2.4.4.2 SR operation

Above Figure shows an example of SR operation in the presence of lost packets.

Here, the receiver initially buffers packets 3, 4, and 5, and delivers them together with packet 2 to the upper layer when packet 2 is finally received.

For example, if there is no ACK for packet send_base propagating from the receiver to the sender, the sender will eventually retransmit packet send_base, even though it is clear that the receiver has already received.

SR **Sender** Events and actions :

1. *Data received from above.* When data is received from above, the SR sender checks the next available sequence number for the packet. If the sequence number is within the sender's window, the data is packetized and sent; otherwise it is either buffered or returned to the upper layer for later transmission, as in GBN.

2. *Timeout*. Timers are used to protect against lost packets. However, each packet must now have its own logical timer, since only a single packet will be transmitted on timeout. A single hardware timer can be used to mimic the operation of multiple logical timers

3. *ACK received*. If an ACK is received, the SR sender marks that packet as having been received, provided it is in the window. If the packet's sequence number is equal to send_base, the window base is moved forward to the unacknowledged packet with the smallest sequence number. If the window moves and there are untransmitted packets with sequence numbers that now fall within the window, these packets are transmitted.

SR **receiver** events and actions :

1.Packet with sequence number in [rcv_base, rcv_base+N-1] *is correctly received :* The received packet falls within the receiver's window and a selective ACK packet is returned to the sender. If the packet was not previously received, it is buffered. If this packet has a sequence number equal to the base of the receive window (rcv_base) then this packet, and any previously buffered and consecutively numbered (beginning with rcv_base) packets are delivered to the upper layer.

The receive window is then moved forward by the number of packets delivered to the upper layer. As an example, consider above Figure.

When a packet with a sequence number of rcv_base=2 is received, it and packets 3, 4, and 5 can be delivered to the upper layer.

2. *Packet with sequence number in* [rcv_base-N, rcv_base-1] *is correctly received.*

In this case, an ACK must be generated, even though this is a packet that the receiver has previously acknowledged.

3. *Otherwise.* Ignore the packet. that packet. If the receiver were not to acknowledge this packet, the sender's window would never move forward.

The sender and receiver will not always have an identical view of what has been received correctly and what has not. For SR protocols, this means that the sender and receiver windows will not always coincide.

The lack of synchronization between sender and receiver windows has important consequences when we are faced with the reality of a finite range of sequence numbers.

For example, with a finite range of four packet sequence numbers, 0, 1, 2, 3, and a window size of three. Suppose packets 0 through 2 are transmitted and correctly received and acknowledged at the receiver. At this point, the receiver's window is over the fourth, fifth, and sixth packets, which have sequence numbers 3, 0, and 1, respectively.

Consider two scenarios :

In the first scenario, shown in Figure (a) below, the ACKs for the first three packets are lost and the sender retransmits these packets. The receiver thus next receives a packet with sequence number 0—a copy of the first packet sent.

In the second scenario, shown in Figure (b) below, the ACKs for the first three packets are all delivered correctly. The sender thus moves its window forward and sends the fourth, fifth, and sixth packets, with sequence numbers 3, 0, and 1, respectively.

The packet with sequence number 3 is lost, but the packet with sequence number 0 arrives—a packet containing *new* data.

Consider the receiver's viewpoint in Figure below, since the receiver cannot "see" the actions taken by the sender. All the receiver observes is the sequence of messages it receives from the channel and sends into the channel.

The two scenarios in Figure are *identical.*

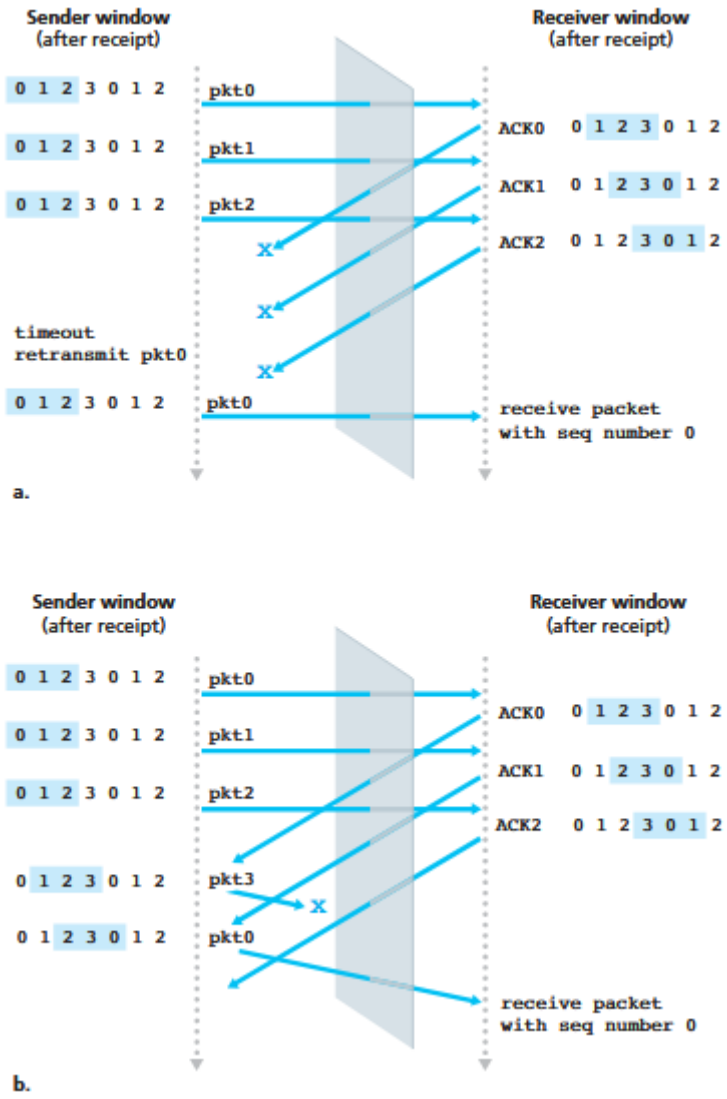


Fig 2.4.4.4: SR dilemma with too large windows

There is no way of distinguishing the retransmission of the first packet from an original transmission of the fifth packet.

Solution is the window size must be less than or equal to half the size of the sequence number space for SR protocols.

Table below summarizes these mechanisms.

Mechanism	Use Comments
Checksum	Used to detect bit errors in a transmitted packet.
Timer	Used to timeout/retransmit a packet, possibly because the packet (or its ACK) was lost within the channel. Because timeouts can occur : • When a packet is delayed but not lost (premature timeout). • When a packet has been received by the receiver • but the receiver-to-sender ACK has been lost. When Duplicate copies of a packet may be • received by a receiver.
Sequence number	• Used for sequential numbering of packets of data flowing from sender to receiver. • Gaps in the sequence numbers of received packets allow the receiver to detect a lost packet. • Packets with duplicate sequence numbers allow the receiver to detect duplicate copies of a packet.
Acknowledgment	• Used by the receiver to tell the sender that a packet or set of packets has been received correctly. • Acknowledgments will typically carry the sequence number of the packet or packets being acknowledged. • Acknowledgments may be individual or cumulative, depending on the protocol.

Negative acknowledgment	• Used by the receiver to tell the sender that a packet has not been received correctly. • Negative acknowledgments will typically carry the sequence number of the packet that was not received correctly.
Window, pipelining	• The sender may be restricted to send only packets with sequence numbers that fall within a given range. • By allowing multiple packets to be transmitted but not yet acknowledged, sender utilization can be increased over a stop-and-wait mode of operation.

Table : Summary of reliable data transfer mechanisms and their use

2.5 Connection Oriented TCP :

TCP is said to be **connection-oriented** because before one application process can begin to send data to another, the two processes must first "handshake" with each other—that is, they must send some preliminary segments to each other to establish the parameters of the ensuing data transfer.

TCP connection establishment, both sides of the connection will initialize many TCP state variables associated with the TCP connection.

TCP protocol runs only in the end systems and not in the intermediate network elements (routers and link-layer switches), the intermediate network elements do not maintain TCP connection state.

A TCP connection provides a **full-duplex service**: If there is a TCP connection between Process A on one host and Process B on another host, then application layer data can flow from Process A to Process B at the same time as application layer data flows from Process B to Process A.

A TCP connection is also always **point-to-point**, that is, between a single sender and a single receiver. So-called **"multicasting"—**the transfer of data from one sender to many receivers in a single send operation—is not possible with TCP.

Python client program command :

clientSocket.connect((serverName,serverPort))

where **serverName** is the name of the server and **serverPort** identifies the process on the server. TCP in the client then proceeds to establish a TCP connection with TCP in the server.

Client first sends a special TCP segment; the server responds with a second special TCP segment and finally the client responds again with a third special segment.

The first two segments carry no payload, that is, no application-layer data; the third of these segments may carry a payload. Because three segments are sent between the two hosts, this connection- establishment procedure is often referred to as a **three-way handshake**.

Once a TCP connection is established, the two application processes can send data to each other. Consider the sending of data from the client process to the server process.

The client process passes a stream of data through the socket.

Once the data passes through the socket, the data is in the hands of TCP running in the client. As Figure shows, TCP directs this data to the connection's **send buffer**, which is one of the buffers that is set aside during the initial three-way handshake.

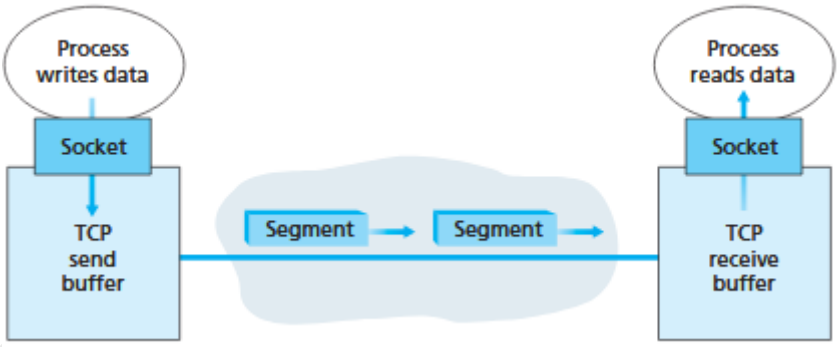


Fig 2.5.1 TCP send and receive buffers

From time to time, TCP will grab chunks of data from the send buffer and pass the data to the network layer.

The maximum amount of data that can be grabbed and placed in a segment is limited by the **maximum segment size (MSS)**.

The MSS is set by first determining the length of the largest link-layer frame that can be sent by the local sending host (**maximum transmission unit, MTU)** and then setting the MSS to ensure that a TCP segment (when encapsulated in an IP datagram) plus the TCP/IP header length (typically 40 bytes) will fit into a single link-layer frame.

TCP pairs each chunk of client data with a TCP header, thereby forming TCP segments.

The segments are passed down to the network layer, where they are separately encapsulated within network-layer IP datagrams.

The IP datagrams are then sent into the network. When TCP receives a segment at the other end, the segment's data is placed in the TCP connection's receive buffer, as shown in Figure above.

The application reads the stream of data from this buffer. Each side of the connection has its own send buffer and its own receive buffer.

Thus, TCP connection consists of buffers, variables and a socket connection to a process in one host, and another set of buffers, variables and a socket connection to a process in another host.

TCP Segment Structure :

The TCP segment consists of header fields and a data field. The data field contains a chunk of application data. MSS limits the maximum size of a segment's data field. When TCP sends a large file, such as an image as part of a Web page, it typically breaks the file into chunks of size MSS.

Interactive applications , often transmit data chunks that are smaller than the MSS;

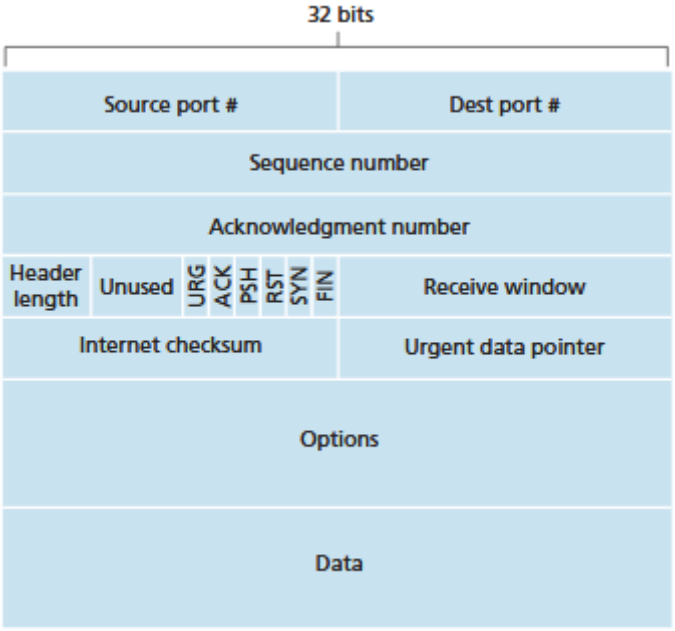


Fig 2.5.2 TCP Segment structure

Figure above shows the structure of the TCP segment. The header includes **source and destination port numbers**, which are used for multiplexing / demultiplexing data from/to upper-layer applications.

Header includes a **checksum field** for error detection.

A TCP segment header also contains the following fields:

• The 32-bit **sequence number field** and the 32-bit **acknowledgment number field** are used by the TCP sender and receiver in implementing a reliable data transfer service.

• The 16-bit **receive window** field is used for flow control.

• The 4-bit **header length field** specifies the length of the TCP header in 32-bit words. The TCP header can be of variable length due to the TCP options field.

• The optional and variable-length **options field** is used when a sender and receiver negotiate the maximum segment size (MSS) or as a window scaling factor for use in high-speed networks. A time-stamping option is also defined.

• The **flag field** contains 6 bits. The **ACK bit** is used to indicate that the value carried in the acknowledgment field is valid; that is, the segment contains an acknowledgment for a segment that has been successfully received.

The **RST**, **SYN**, and **FIN** bits are used for connection setup and teardown .

Setting the **PSH** bit indicates that the receiver should pass the data to the upper layer immediately.

URG bit is used to indicate that there is data in this segment that the sending-side upper-layer entity has marked as "urgent."

The location of the last byte of this urgent data is indicated by the 16-bit **urgent data pointer field**.

TCP must inform the receiving- side upper-layer entity when urgent data exists and pass it a pointer to the end of the urgent data.

Sequence Numbers and Acknowledgment Numbers

Two of the most important fields in the TCP segment header are the sequence number field and the acknowledgment number field. These fields are a critical part of TCP's reliable data transfer service.

TCP views data as an unstructured, but ordered, stream of bytes. TCP's use of sequence numbers reflects this view in that sequence numbers are over the stream of transmitted bytes and *not* over the series of transmitted segments.

The **sequence number for a segment** is therefore the byte-stream number of the first byte in the segment.

Example. Suppose a process in Host A wants to send a stream of data to a process in Host B over a TCP connection. The TCP in Host A will implicitly number each byte in the data stream. Suppose that the data stream consists of a file consisting of 500,000 bytes, that the MSS is 1,000 bytes, and that the first byte of the data stream is numbered 0.

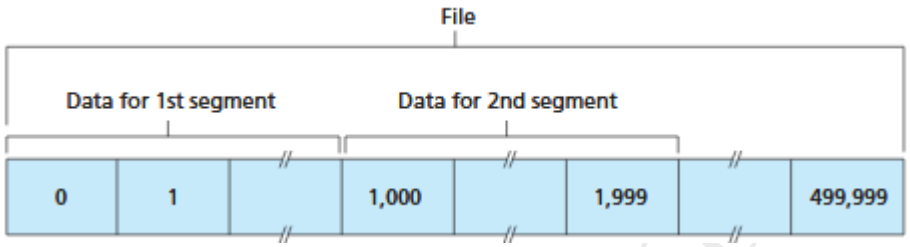


Fig 2.5.3 Dividing file data into TCP Segments

As shown in Figure above, TCP constructs 500 segments out of the data stream. The first segment gets assigned sequence number 0, the second segment gets assigned sequence number 1,000, the third segment gets assigned sequence number 2,000, and so on.

Each sequence number is inserted in the sequence number field in the header of the appropriate TCP segment.

Consider acknowledgment numbers.

TCP is full-duplex, so that Host A may be receiving data from Host B while it sends data to Host B (as part of the same TCP connection).

The acknowledgment number that Host A puts in its segment is the sequence number of the next byte Host A is expecting from Host B.

Suppose that Host A has received all bytes numbered 0 through 535 from B and suppose that it is about to send a segment to Host B. Host A is waiting for byte 536 and all the subsequent bytes in Host B's data stream. So Host A puts 536 in the acknowledgment number field of the segment it sends to B.

Suppose Host A has received one segment from Host B containing bytes 0 through 535 and another segment containing bytes 900 through 1,000. For some reason Host A has not yet received bytes 536 through 899. In this example, Host A is still waiting for byte 536 (and beyond) in order to re-create B's data stream. Thus, A's next segment to B

will contain 536 in the acknowledgment number field. Because TCP only acknowledges bytes up to the first missing byte in the stream, TCP is said to provide **cumulative acknowledgments**.

Host A received the third segment (bytes 900 through 1,000) before receiving the second segment (bytes 536 through 899). Thus, the third segment arrived out of order. The issue is: Host A receives out-of-order segments in a TCP connection.

There are two choices:

(1) the receiver immediately discards out-of-order segments

(2) the receiver keeps the out-of-order bytes and waits for the missing bytes to fill in the gaps.

In Figure above, we assumed that the initial sequence number was zero. Both sides of a TCP connection randomly choose an initial sequence number.

Telnet: A Case Study for Sequence and Acknowledgment Numbers

Suppose Host A initiates a Telnet session with Host B. Because Host A initiates the session, it is labeled the client, and Host B is labeled the server.

Each character typed by the user (at the client) will be sent to the remote host; the remote host will send back a copy of each character, which will be displayed on the Telnet user's screen.

"Echo back" is used to ensure that characters seen by the Telnet user have already been received and processed at the remote site.

Each character thus traverses the network twice between the time the user hits the key and the time the character is displayed on the user's monitor.

Suppose the user types a single letter, 'C'.

As shown in Figure below, the starting sequence numbers are 42 and 79 for the client and server, respectively.

The sequence number of a segment is the sequence number of the first byte in the data field. Thus, the first segment sent from the client will have sequence number 42; the first segment sent from the server will have sequence number 79.

The acknowledgment number is the sequence number of the next byte of data that the host is waiting for.

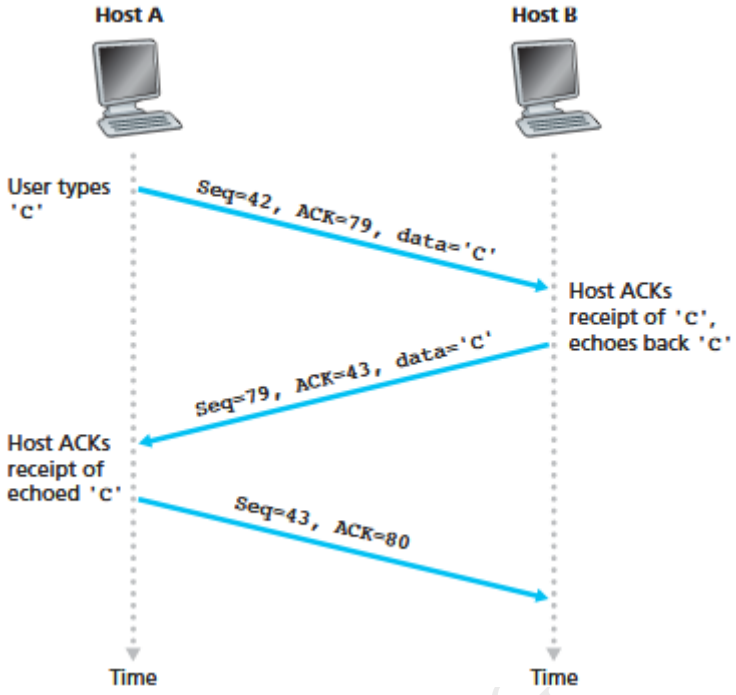


Fig 2.5.4 Sequence and Ack number for a Telnet Application

After the TCP connection is established but before any data is sent, the client is waiting for byte 79 and the server is waiting for byte 42.

As shown in Figure above, three segments are sent.

- The first segment is sent from the client to the server, containing the 1-byte ASCII representation of the letter 'C' in its data field. This first segment also has 42 in its sequence number field.

Because the client has not yet received any data from the server, this first segment will have 79 in its acknowledgment number field.

- The second segment is sent from the server to the client. It serves a dual purpose.

First it provides an acknowledgment of the data the server has received. By putting 43 in the acknowledgment field, the server is telling the client that it has successfully received everything up through byte 42 and is now waiting for bytes 43 onward.

The second purpose of this segment is to echo back the letter 'C.' Thus, the second segment has the ASCII representation of 'C' in its data field. This second segment has the sequence number 79, the initial sequence number of the server-to client data flow of this TCP connection, as this is the very first byte of data that the server is sending. This acknowledgment is said to be **piggybacked** on the server-to-client data segment.

- The third segment is sent from the client to the server. Its purpose is to acknowledge the data it has received from the server. This segment has an empty data field .

The segment has 80 in the acknowledgment number field because the client has received the stream of bytes up through byte sequence number 79 and it is now waiting for bytes 80 .