Unit-3

19ECS-232

COMPUTER NETWORKS

B.SRIDEI

ASSISTANT PROFESSOR

DEPARTMENT OF CSE

**SYLLABUS**

**Transport Layer:** Introduction and Transport-Layer Services, Multiplexing and Demultiplexing, Connectionless Transport: UDP, Principles of Reliable Data Transfer, Connection-oriented Transport: TCP, Principles of Congestion Control: TCP Congestion Control

The transport layer is a central piece of the layered network architecture that resides between the application and network layers. It has the critical role of providing communication services directly to the application processes running on different hosts.

**INTRODUCTION AND TRANSPORT LAYER SERVICES**

-A transport-layer protocol provides **logical communication** between application processes running on different hosts.

-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.**

* The application messages are divided into smaller chunks and a transport-layer header is added 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.

-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.

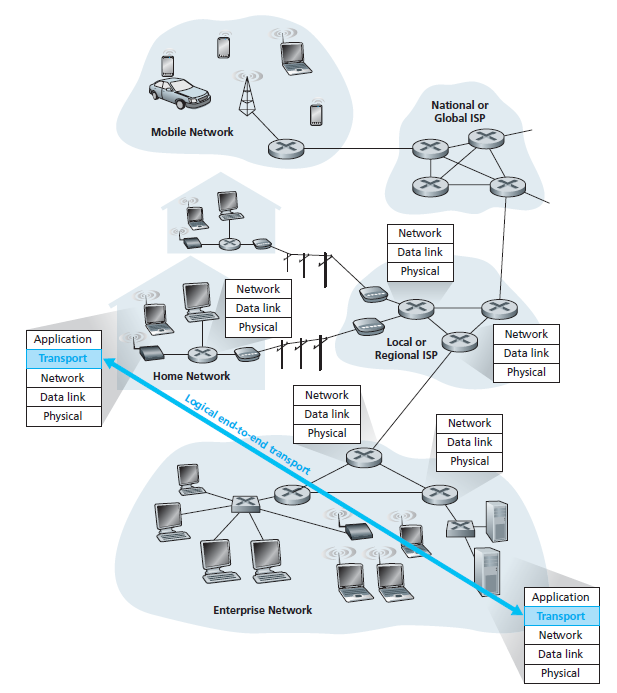
* **Relationship Between Transport and Network Layers**

-A transport-layer protocol provides logical communication between processes running on different hosts.

-A network-layer protocol provides logical communication between hosts.

-**Example:** For understanding the difference between the network layer protocol and transport layer protocol consider kids of different houses staying in different cities write letters to each other, the kids of the both the houses are cousins to each other.

* application messages = letters in envelopes
* processes = cousins
* hosts (also called end systems) = houses
* transport-layer protocol = Cousins who post letters
* network-layer protocol = postal service



**Fig:** The transport layer provides logical rather than physical

communication between application processes

-Transport-layer protocols live in the end systems.

-Within an end system, a transport protocol moves messages from application processes to the network edge (that is, the network layer) and vice versa.

-Intermediate routers neither act on, nor recognize, any information that the transport layer may have added to the application messages.

-The services that a transport protocol can provide are often constrained by the service model of the underlying network-layer protocol.

-The network-layer protocol cannot provide delay or bandwidth guarantees for transport layer

segments sent between hosts.

-In the same way transport-layer protocol cannot provide delay or bandwidth guarantees for application messages sent between processes.

-A transport protocol can use encryption to guarantee that application messages are not read by intruders, even when the network layer cannot guarantee the confidentiality of transport-layer segments.

* **Overview of the Transport Layer in the Internet**

**-**Two distinct transport-layer protocols are available to the application layer.

**-**One of the protocols is **UDP** (User Datagram Protocol), which provides an unreliable, connectionless service to the invoking application.

**-**The second protocol is **TCP** (Transmission Control Protocol), which provides a reliable, connection-oriented service to the invoking application.

**-**The Internet’s network-layer protocol has a name Internet Protocol (IP) that makes its “best effort” to deliver segments between communicating hosts, but it makes no guarantees.

**-**IP is said to be an **unreliable service,** every host has at least one network-layer address, so called IP address.

**-**The most fundamental responsibility of UDP and TCP is to extend 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**.

**-**The two minimal services that UDP protocol provides are process-to-process data delivery and error checking.

**-**TCP, on the other hand, offers several additional services to applications such as reliable data transfer, congestion control.

**-**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 thus converts IP’s unreliable service between end systems into a reliable data transport service between processes.

**MULTIPLEXING AND DEMULTIPLEXING**

-A receiving host directs an incoming transport-layer segment to the appropriate socket by considering a set of fields in the segment.

-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**.

-The job of 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**.

-Transport-layer multiplexing requires

* Sockets have unique identifiers, and
* Each segment has special fields that indicate the socket to which the segment is to be delivered.

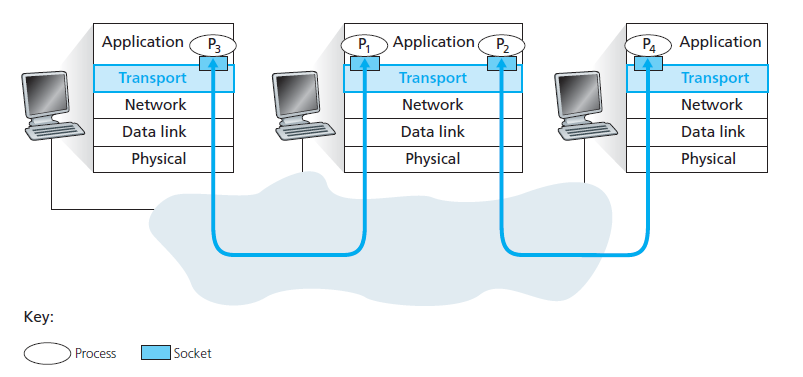
-The special fields are the **source port number** and the **destination port number field**.

* 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** and are restricted, which means that they are reserved for use by well-known application protocols such as HTTP (port number 80) and FTP (port number 21).

-In demultiplexing 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.



**Fig:** Transport-layer multiplexing and demultiplexing

* **Connectionless Multiplexing and Demultiplexing**

**-**UDP socket is been created by

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.

-The server side of the application is been assigned with a specific port number.

-A process in sender host, sends a chunk of application data with the transport layer header to a process in the receiver host.

* The transport layer passes segment to the network layer.
* The network layer encapsulates the segment in an IP datagram and makes a best-effort to deliver the segment to the receiving host.
* If the segment arrives at the receiving host, the transport layer at the receiving host examines the destination port number in the segment and delivers the segment to its socket identified by port number.

-A 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.

-The server uses the recvfrom() method to extract the client side (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**

**-**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.

-In contrast with UDP, two arriving TCP segments with different source IP addresses or source port numbers will be directed to two different sockets.

-TCP client-server programming

* The TCP server application has a “welcoming socket,” that waits for connection establishment requests from TCP clients.
* 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))

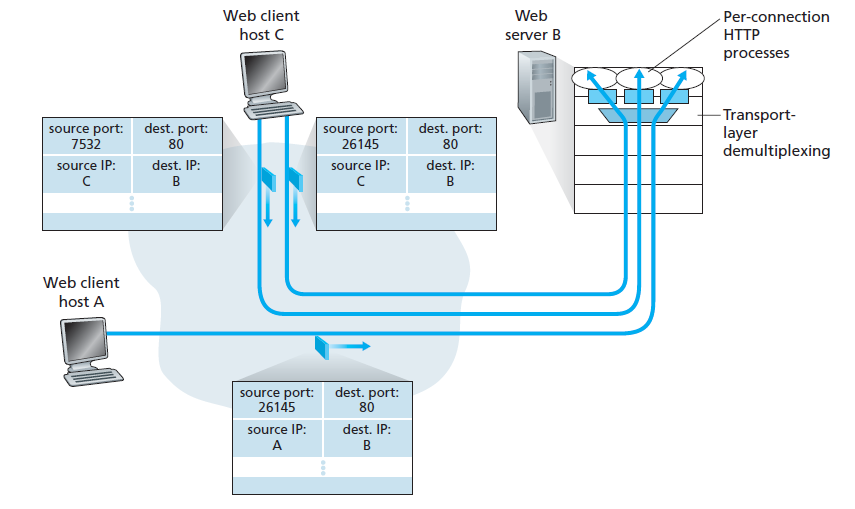
* A connection-establishment request is nothing more than a TCP segment with destination port number and a special connection-establishment bit set in the TCP header.
* The segment also includes a source port number that was chosen by the client.
* When the host operating system of the computer running the server process

receives the incoming connection-request segment it locates the server process that is waiting to accept a connection.

* The server process then creates a new socket:

connectionSocket, addr = serverSocket.accept()

* The newly created connection socket is identified by four tuples; all subsequently arriving segments whose source port, source IP address, destination port, and destination IP address match the four values will be demultiplexed to this socket.
* With the TCP connection now in place, the client and server can now send data to each other.



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

**CONNECTIONLESS TRANSPORT: UDP**

-The UDP protocol has no handshaking between sending and receiving transport-layer entities before sending a segment.

* UDP is said to be connectionless.

-TCP provides a reliable data transfer service, while UDP does not but why should an application developer would ever choose to build an application over UDP rather than over TCP?

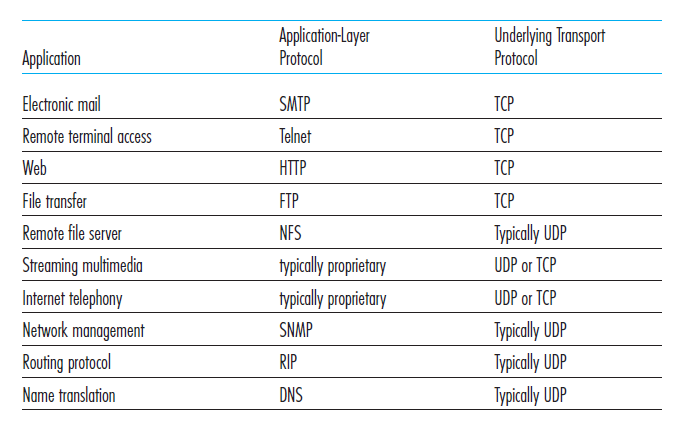
-Many applications are better suited for UDP for the following reasons:

* **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.

* The real-time applications often require a minimum sending rate, do not want to overly delay segment transmission, and can tolerate some data loss.
* So, these kinds of applications can use UDP and implement, as part of the application.
* **No connection establishment:**
* TCP uses a three-way handshake before it starts to transfer data.
* But, UDP just blasts away without any formal preliminaries and does not introduce any delay to establish a connection.
* So, DNS runs over UDP rather than TCP. DNS would be much slower if it ran over TCP.
* **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.

- Popular Internet applications and their underlying transport protocols:



* **UDP Segment Structure**

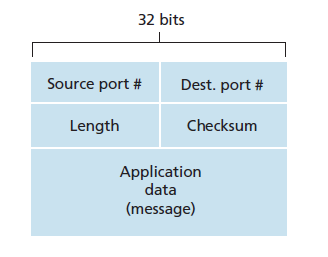
**-**The UDP header has only four fields, each consisting of two bytes.

-The port numbers allow the destination host to pass the application data to the correct process running on the destination end system.

-The length field specifies the number of bytes in the UDP segment.

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

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



* **UDP Checksum**

**-**The UDP checksum provides for error detection.

-The checksum is used to determine whether bits within the UDP segment have been altered (by noise in the links or while stored in a router) 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.

-A simple example of the checksum calculation:

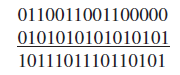
* + Three 16-bit words:

0110011001100000

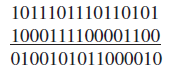
0101010101010101

1000111100001100

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



* + Adding the third word to the above sum gives



-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 clearly the sum at the receiver will be 1111111111111111.

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

**PRINCIPLES OF RELIABLE DATA TRANSFER**

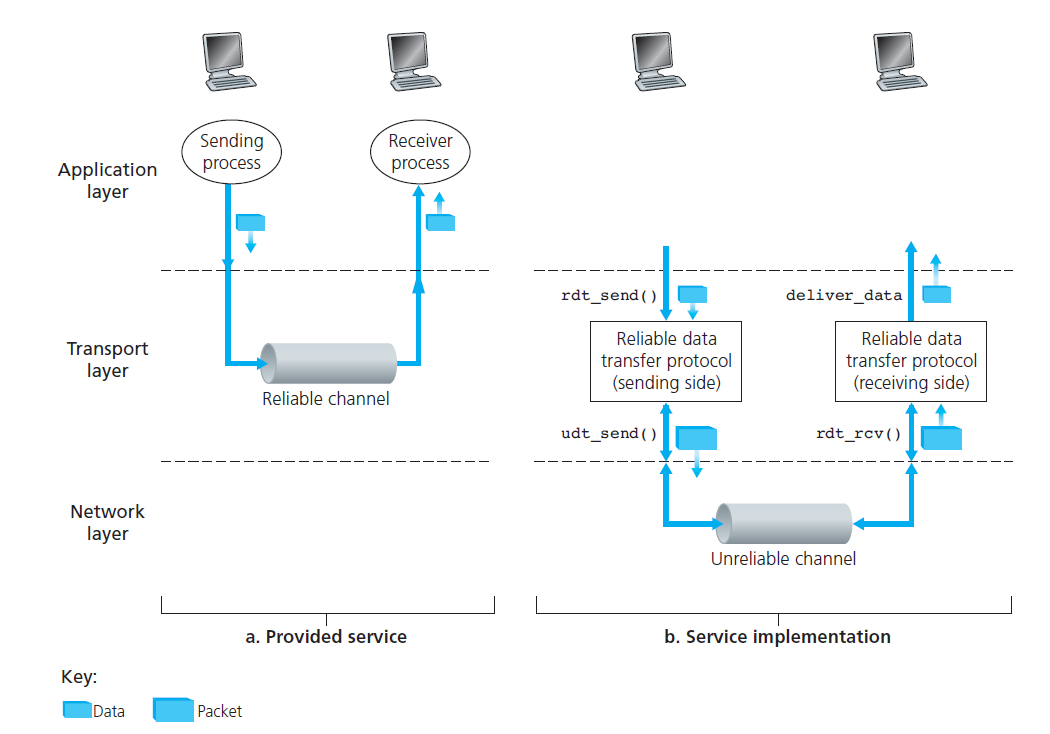
**-**The problem of implementing reliable data transfer occurs not only at the transport layer, but also at the link layer and the application layer as well.

-The service abstraction provided to the upper-layer entities is that of a reliable channel through which data can be transferred.

* + With a reliable channel, no transferred data bits are corrupted (flipped from 0 to 1, or vice versa) 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.

-But this task is made difficult by the fact that the layer below the reliable data transfer protocol may be unreliable.



**Fig:** Reliable data transfer: Service model and service implementation

-The sending side of the data transfer protocol will be invoked from above by a call to rdt\_send().

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

* Here rdt stands for reliable data transfer protocol and \_send indicates that the sending side of rdt is being called.

-On the receiving side, rdt\_rcv() will be called when a packet arrives from the receiving 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 the terminology “packet” is been used rather than transport-layer “segment.”

-The case of unidirectional data transfer is been considered.

* **Building a Reliable Data Transfer Protocol**

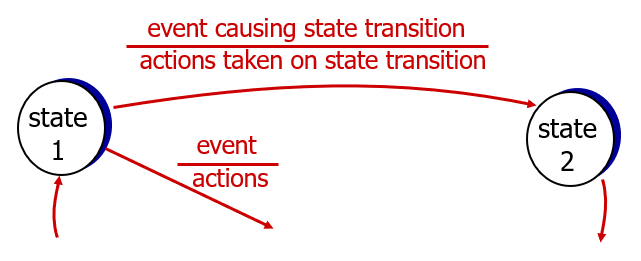
-We will incrementally develop the sender and receiver sides of a reliable data transfer protocol, considering increasingly complex models of the underlying channel.

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

-Firstly, consider the simplest case, in which the underlying channel is completely reliable.

-The protocol itself, is called rdt1.0.

-The **finite-state machine (FSM)** definitions for the rdt1.0 sender and receiver.

****

-The circle in the FSM indicates state.

-The arrows in the FSM indicates the transition of the protocol from one state to another.

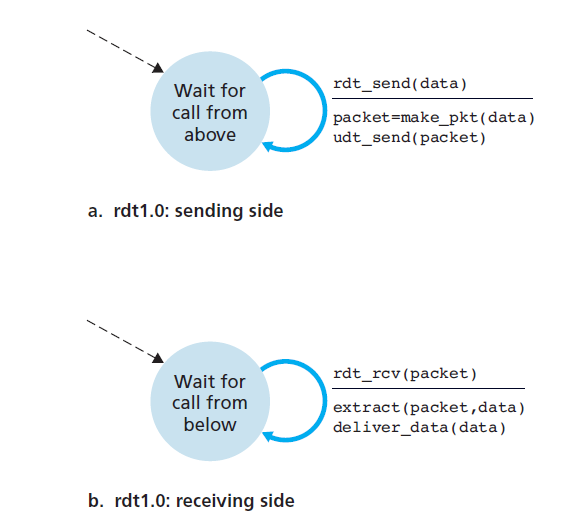
-The event causing the transition is shown above the horizontal line labelling the transition, and the actions taken when the event occurs are shown below the horizontal line.

* When no action is taken on an event, or no event occurs and an action is taken, we’ll use

the symbol ^ below or above the horizontal, respectively, to explicitly denote the

lack of an action or event.

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



**Fig:** rdt1.0 – A protocol for a completely reliable channel

-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.

-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)).

**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.

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

-A protocol for reliably communicating over such a channel uses both **positive acknowledgments** (ACK) and **negative acknowledgments** (NAK).

* These control messagesallow the receiver to let the sender know what has been received correctly, andwhat has been received in error and thus requires repeating.

-In a computer networksetting, reliable data transfer protocols based on such 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:**
* First, a mechanism is needed to allow the receiver to detect when bit errors have occurred.
* UDP uses the Internet checksum field for exactly this purpose.
* 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

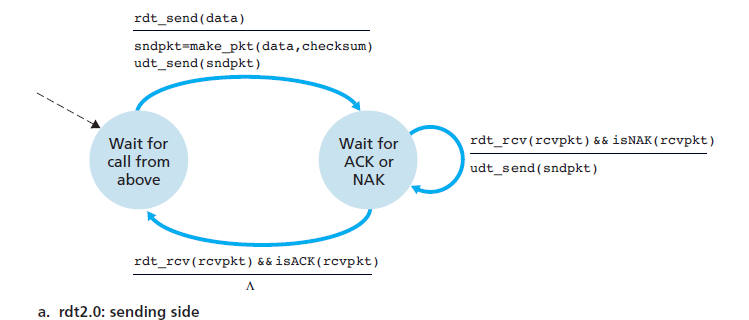
end systems, possibly separated by thousands of miles, the only way for the sender to know the receiver’s view of the world is for the receiver to provide explicit feedback to the sender.

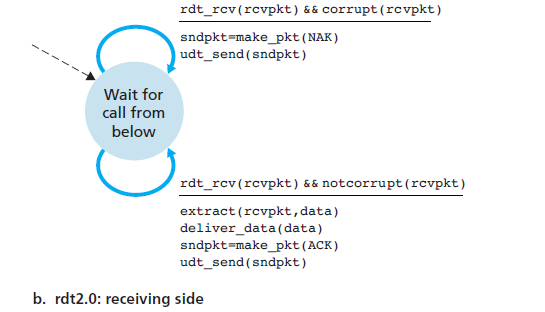
* The positive (ACK) and negative (NAK) acknowledgment replies in the

message-dictation scenario are examples of such feedback.

* rdt2.0 protocol will similarly send ACK and NAK packets back from the receiver to the sender.
* **Retransmission:**
* A packet that is received in error at the receiver will be retransmitted by the sender.

The FSM representation of rdt2.0, a data transfer protocol employing error detection, positive acknowledgments, and negative acknowledgments.





**Fig:** rdt2.0–A protocol for a channel with bit errors

-The sending side of rdt2.0 has two states.

* In the leftmost state, 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) then the sender knows that the most recently transmitted packet has been received correctly and 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.

-It is important to note that 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.

* Because of this behavior, protocols such as rdt2.0 are known as **stop-and-wait** protocols.

-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.

**Reliable Data Transfer: rdt2.1, rdt2.2**

-Protocol rdt2.0 may look as if it works but consider that the ACK or NAK packet are corrupted.

-Minimally, we will need to add checksum bits to ACK/NAK packets in order to detect such errors.

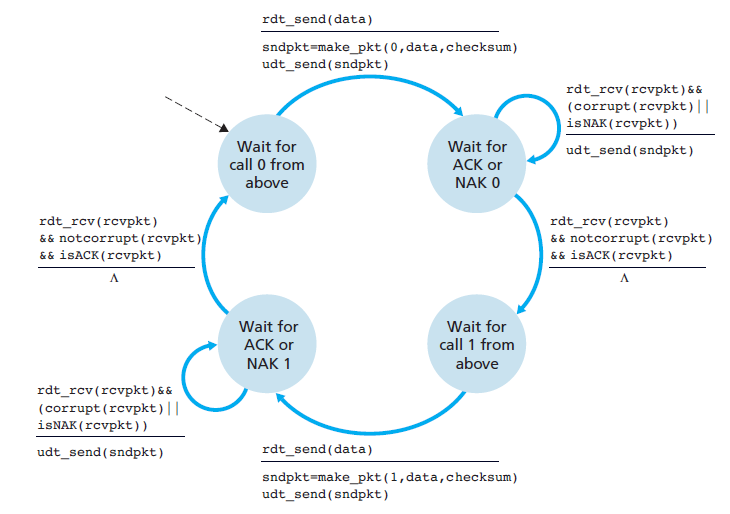
-Next the sender simply to resend the current data packet when it receives a garbled ACK or NAK packet.

-This approach, however, introduces **duplicate packets** into the sender-to-receiver channel.

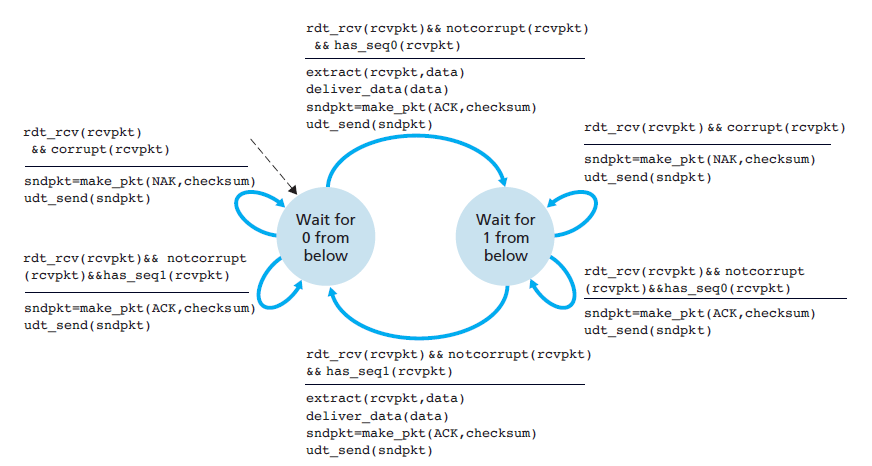
* The fundamental difficulty with duplicate packets is that the receiver doesn’t know whether the ACK or NAK it last sent was received correctly at the sender.
* A simple solution to this new problem is to add a new field to the data packet and have the sender number its data packets by putting a **sequence 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-and-wait 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 or a new packet.



**Fig:** rdt2.1 sender

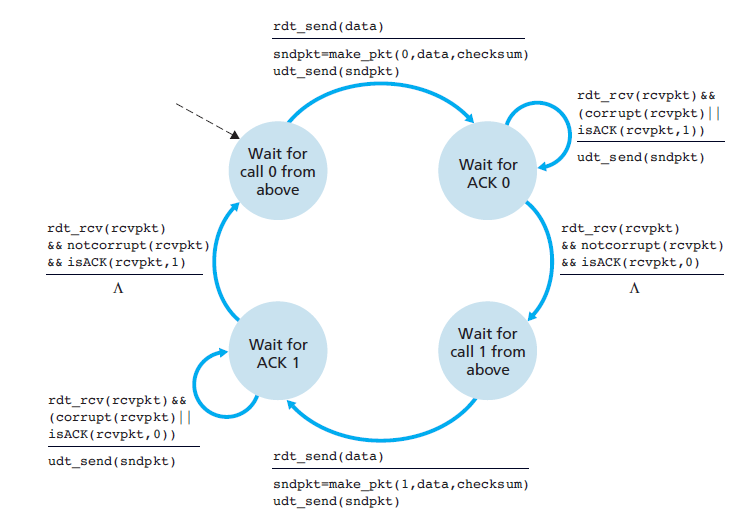


**Fig:** rdt2.1 receiver

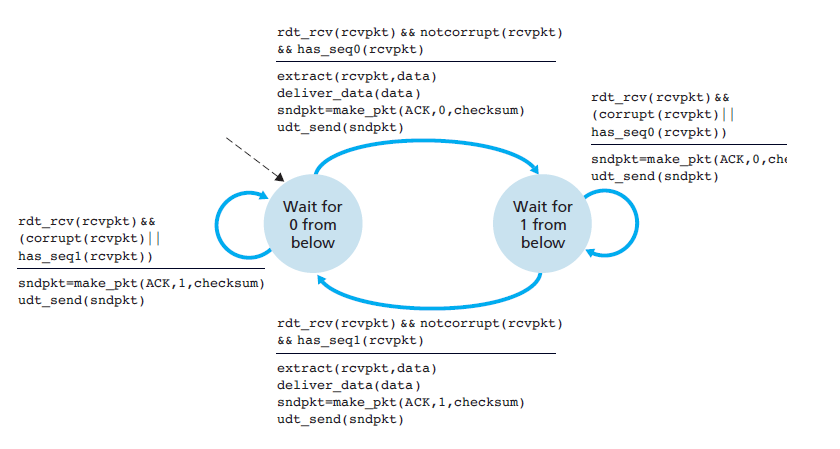
-The rdt2.1 sender and receiver FSMs each now have twice as many states as before.

* 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.

One subtle change between rtdt2.1 and rdt2.2 is that the receiver must now include the sequence number of the packet being acknowledged by an ACK message. This is done by including the ACK,0 or ACK,1 argument in make\_pkt()in the receiver FSM, and the sender must now check the sequence number of the packet being acknowledged by a received ACK message.



**Fig:** rdt2.2 sender



**Fig:** rdt2.2 receiver

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

-In addition to corrupting bits, the underlying channel can lose packets as well, a not-uncommon event in today’s computer networks.

-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.
* If the sender is willing to wait long enough so that it is certain that a packet has been lost, it can simply retransmit the data packet.
* But how long must the sender wait to be certain that something has been lost?
* The sender must clearly wait at least as long as a round-trip delay between the sender and receiver (which may include buffering at intermediate routers) plus whatever amount of time is needed to process a packet at the receiver.
* In many networks, the worst-case maximum delay is very difficult even to estimate.
* **Duplicate data packets** in the sender-to-receiver channel occurs when the packets are retransmitted.

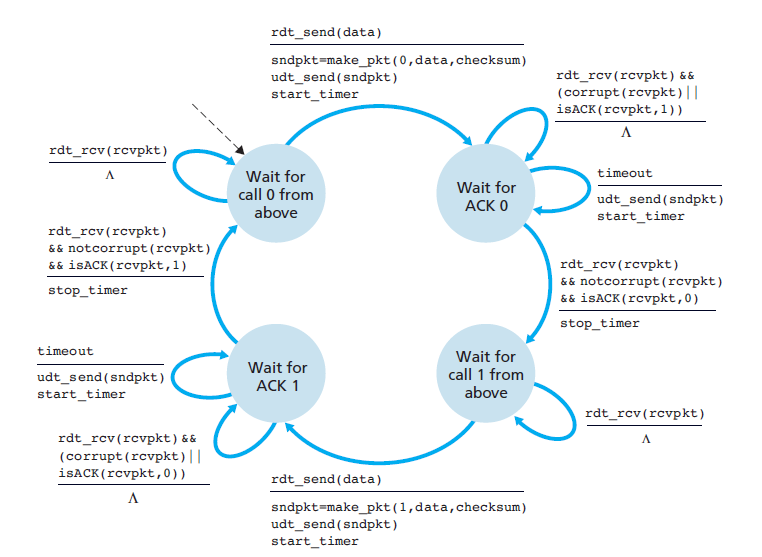
-Implementing a time-based retransmission mechanism requires a **countdown timer** that can

interrupt the sender after a given amount of time has expired.

-The sender will thus need to be able to:

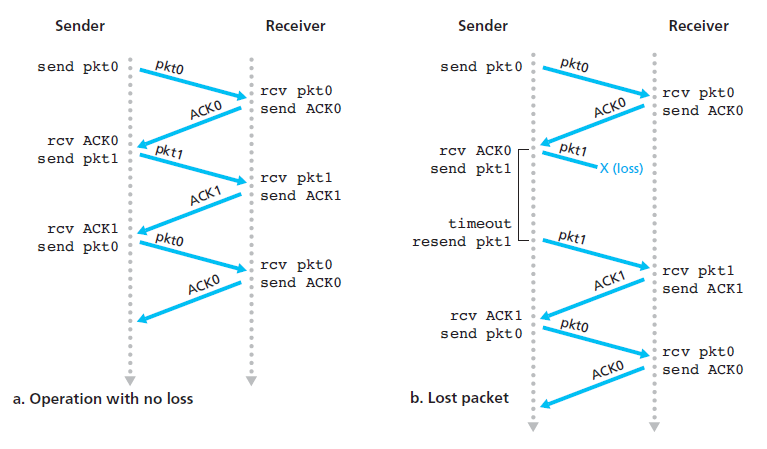
(1) start the timer each time a packet (either a first-time packet or a retransmission) is sent, (2) respond to a timer interrupt (taking appropriate actions), and

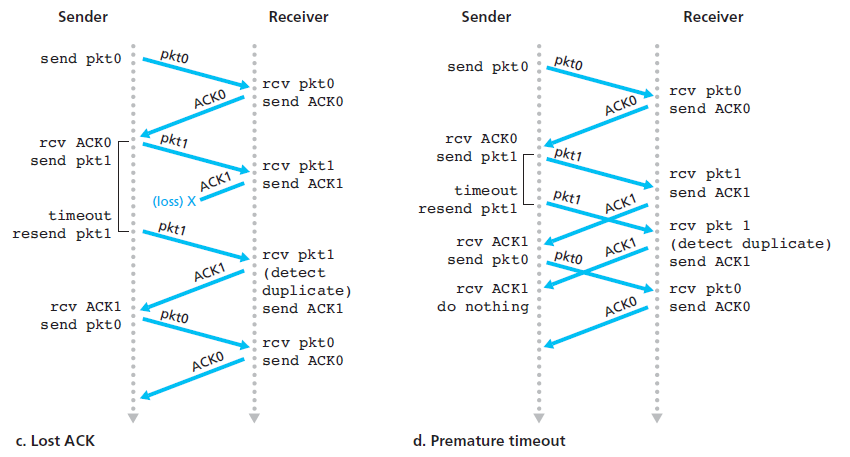
(3) stop the timer.



**Fig:** rdt3.0 sender

From the below figure the protocol operates with no lost or delayed packets and how it handles lost data packets. Here time moves forward from the top of the diagram toward the bottom of the diagram; note that a receive time for a packet is necessarily later than the send time for a packet as a result of transmission and propagation delays. 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 sometimes known as the **alternating-bit protocol**.





**Fig:** Operation of rdt3.0, the alternating-bit protocol

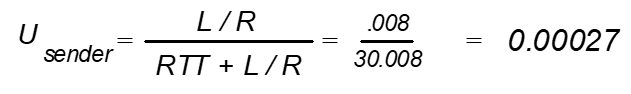
* **Pipelined Reliable Data Transfer Protocols**

-rdt3.0 is correct, but performance stinks

-Consider 1 Gbps link, 15 ms propagation delay, 8000-bit packet time needed to actually transmit the packet into the 1 Gbps link is



- U sender: utilization – fraction of time sender busy sending



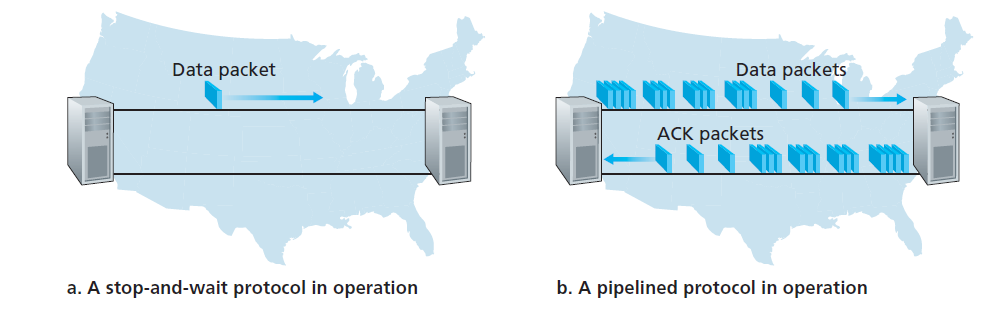
-The sender was busy only 2.7 hundredths of one percent of the time.

-Network protocol limits use of physical resources

-Also, neglected lower-layer protocol-processing times at the sender and receiver, as well as the processing and queuing delays that would occur at any intermediate routers between the sender and receiver.

-Including these effects would serve only to further increase the delay and further accentuate the poor performance.

* The solution to this particular performance problem is simple: Rather than operate in a stop-and-wait manner, the sender is allowed to send multiple packets without waiting for acknowledgments, the many in-transit sender-to-receiver packets can be visualized as filling a pipeline, this technique is known as **pipelining**.



**Fig:** Stop-and-wait versus pipelined protocol

-Pipelining has the following consequences for reliable data transfer protocols:

* The range of sequence numbers must be increased, since each in-transit packet

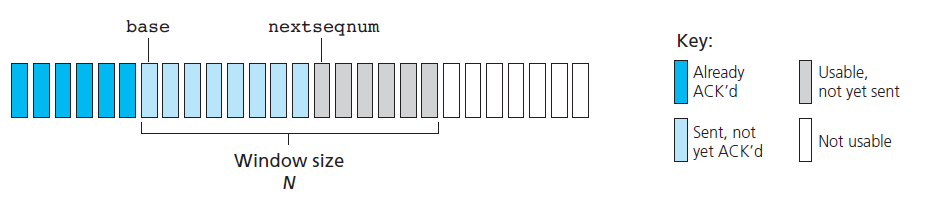
(not counting retransmissions) must have a unique sequence number and there may be multiple, in-transit, unacknowledged packets.

* The sender and receiver sides of the protocols may have to buffer more than one packet.

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

* **Go-Back-N (GBN)**

-In a **Go-Back-N (GBN) protocol**, the sender is allowed to transmit multiple packets (when available) 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:** Sender’s view of sequence numbers in Go-Back-N

**-**The sender’s view of the range of sequence numbers in a GBN protocol.

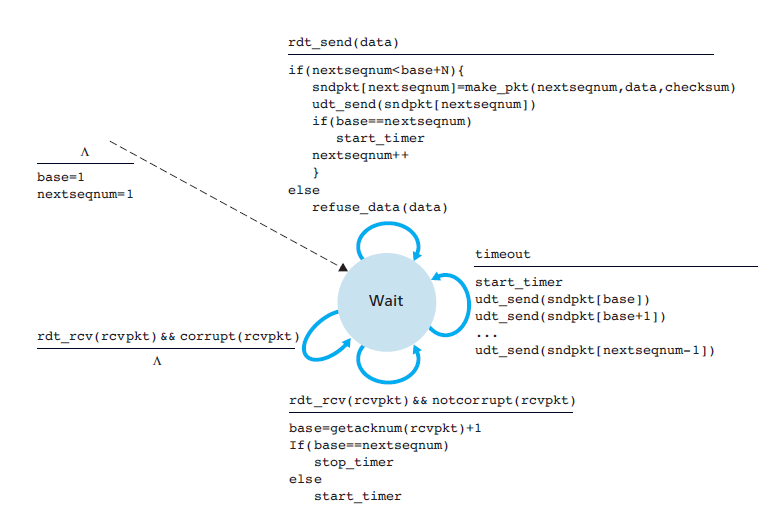
* If we define base to be the sequence number of the oldest unacknowledged packet and nextseqnum to be the smallest unused sequence number (that is, the sequence number of the next packet to be sent), then four intervals in the range of sequence numbers can be identified.
* Sequence numbers in the interval [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.

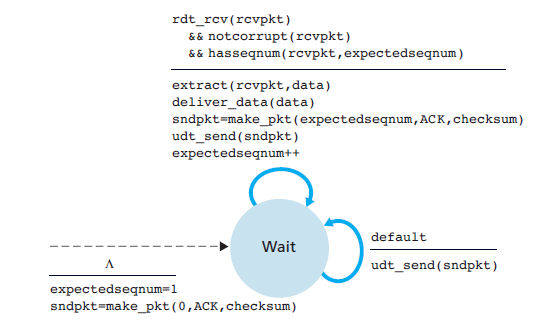
As the protocol operates, this window slides forward over the sequence number space. For this reason, *N* is often referred to as the **window size** and the GBN protocol itself as a **sliding-window protocol**.

-An extended FSM description of the sender and receiver sides of an ACK-based, NAK-free, GBN protocol.

-We refer to this FSM description as an *extended FSM* because we have added variables for base and nextseqnum, and added operations on these variables and conditional actions involving these variables.



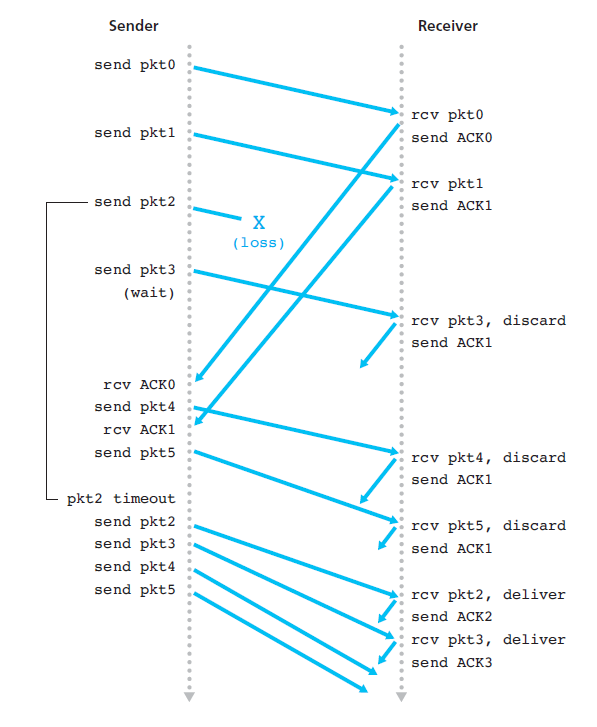
**Fig:** Extended FSM description of GBN sender



**Fig:** Extended FSM description of GBN receiver

-The receiver’s actions in GBN are also simple.

* If a packet with sequence number *n* is received correctly and is in order (that is, the data last delivered to the upper layer came from a packet with sequence number *n* – 1), the receiver sends an ACK for packet *n* and delivers the data portion of the packet to the upper layer.
* In our GBN protocol, the receiver discards out-of-order packets.
* If a timeout occurs, the sender resends all packets that have been previously sent but that have not yet been acknowledged.



**Fig:** Go-Back-N in operation

-From the above figure 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.
* Now the sender has to resend the packets 2,3,4,5 again as the receiver has discarded packets 3,4,5.
* **Selective Repeat (SR)**

-The GBN protocol allows the sender to potentially “fill the pipeline” with packets, thus avoiding the channel utilization.

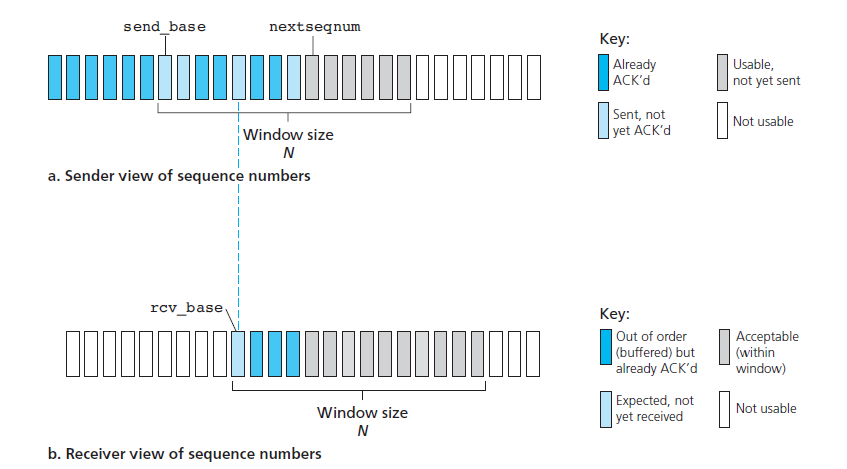
-But a single packet error can thus cause GBN to retransmit a large number of packets, many unnecessarily.

-**S**elective-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 individual, as 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.

-Sequence numbers are also considered.



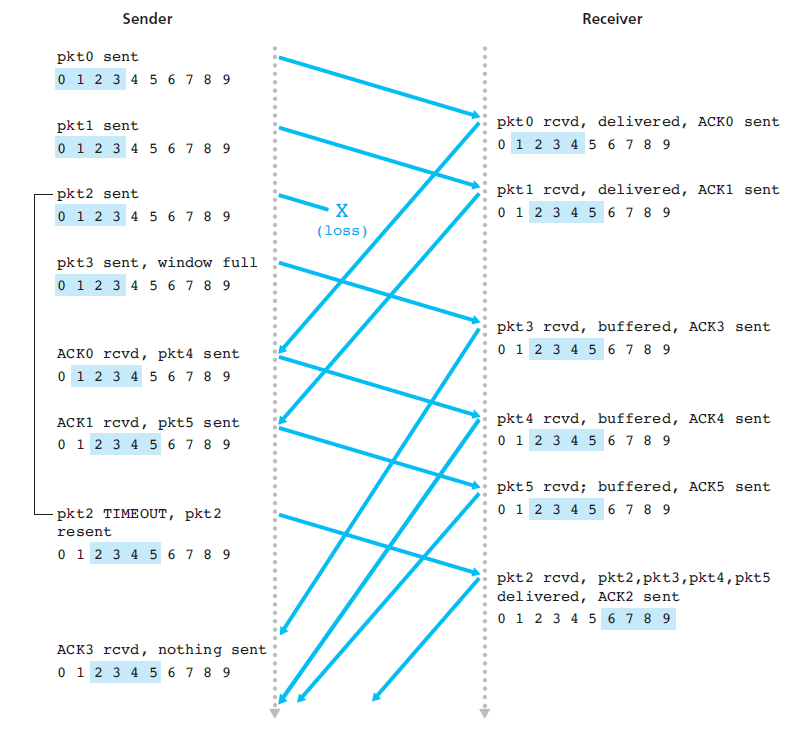
**Fig:** Selective-repeat (SR) sender and receiver views of

sequence-number space

-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.

-The SR sender will only send the packet if it is an error or when it is timeout.



**Fig:** SR operation

**CONNECTION – ORIENTED TRANSPORT: TCP**

TCP the Internet’s transport-layer, connection-oriented, reliable transport protocol.

* **The TCP Connection**

**-**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.

-The 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**.

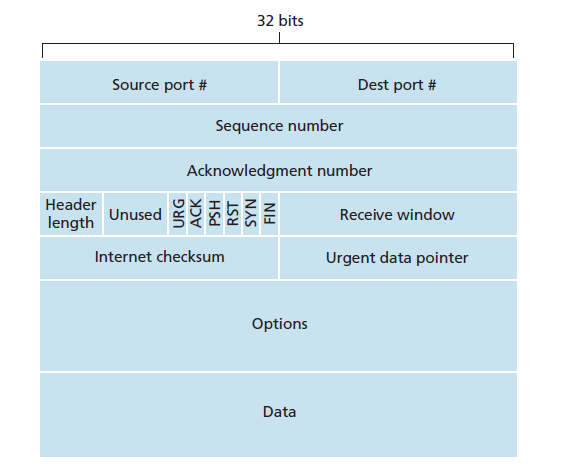
* + If there is a TCP connection between Process A on one host and Process B on another host, then application layer data can flow from Process A to Process B at the same time as application layer data flows from Process B to Process A.
  + A TCP connection is also always **point-to-point**, that is, between a single sender and a single receiver.

**-**In TCP they are two processes, the first is client process, while the other process is called the server process.

* The client application process first informs the client transport layer that it wants to establish a connection to a process in the server.
* This connection- establishment procedure is often referred to as a **three-way handshake**.

-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 typically set by first determining the length of the largest link-layer frame that can be sent by the local sending host (the so-called **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 Segment Structure**



-The TCP segment consists of header fields and a data field.

-The data field contains a chunk of application data.

-The MSS limits the maximum size of a segment’s data field.

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

-The header includes a **checksum field**.

-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 which is used to indicate the number of bytes that a receiver is willing to accept.

-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

-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.
* Finally, the **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**.

**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.

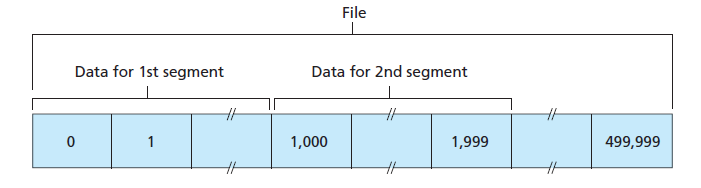
-These fields are a critical part of TCP’s reliable data transfer service.

-TCP’s use of sequence numbers reflect 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.
* 32-bit field that holds the sequence number, i.e, the byte number of the first byte that is sent in that particular segment.
* It is used to reassemble the message at the receiving end if the segments are received out of order.

-The 32-bit field that holds the acknowledgement number, is the byte number that the receiver expects to receive next.

* It is an acknowledgment for the previous bytes being received successfully.



**Fig:** Dividing file data into TCP segments

* **Round-Trip Time Estimation and Timeout**

-TCP, like our rdt protocol uses a timeout/retransmit mechanism to recover from lost segments.

-Clearly, the timeout should be larger than the connection’s round-trip time (RTT), that is, the time from when a segment is sent until it is acknowledged.

-Otherwise, unnecessary retransmissions would be sent.

**Estimating the Round-Trip Time**

-The sample RTT, denoted SampleRTT, for a segment is the amount of time between when the segment is sent (that is, passed to IP) and when an acknowledgment for the segment is received.

-At any point in time, the SampleRTT is being estimated for only one of the transmitted but currently unacknowledged segments, leading to a new value of SampleRTT approximately once every RTT.

* + TCP never computes a SampleRTT for a segment that has been retransmitted; it only measures SampleRTT for segments that have been transmitted once.

-The SampleRTT values will fluctuate from segment to segment due to congestion in the routers and to the varying load on the end systems.

* + Because of this fluctuation, any given SampleRTT value may be atypical.
  + In order to estimate a typical RTT, it is therefore natural to take some sort of average of the SampleRTT values.
  + TCP maintains an average, called EstimatedRTT, of the SampleRTT values.
  + TCP updates EstimatedRTT according to the following formula:



-The new value of EstimatedRTT is a weighted combination of the previous value of EstimatedRTT and the new value for SampleRTT.

* The recommended value of alpha is 0.125.

EstimatedRTT = 0.875 • EstimatedRTT + 0.125 • SampleRTT

-In addition to having an estimate of the RTT, it is also valuable to have a measure of the variability of the RTT.

* The RTT variation, DevRTT, as an estimate of how much SampleRTT typically deviates from EstimatedRTT:



-DevRTT is an EWMA of the difference between SampleRTT and EstimatedRTT.

-If the SampleRTT values have little fluctuation, then DevRTT will be small; on the other hand, if there is a lot of fluctuation, DevRTT will be large. The recommended value of beta is 0.25.

**Setting and Managing the Retransmission Timeout Interval**

-The interval should be greater than or equal to EstimatedRTT, or unnecessary retransmissions would be sent.

-But the timeout interval should not be too much larger than EstimatedRTT; otherwise, when a segment is lost, TCP would not quickly retransmit the segment, leading to large data transfer delays.

-It is therefore desirable to set the timeout equal to the EstimatedRTT plus some margin.

-The margin should be large when there is a lot of fluctuation in the SampleRTT values; it should be small when there is little fluctuation.

-All of these considerations are taken into account in TCP’s method for determining the retransmission timeout interval:

TimeoutInterval = EstimatedRTT + 4 • DevRTT

* **Reliable Data Transfer**

**-**TCP creates a **reliable data transfer service** on top of IP’s unreliable best effort service.

**-**TCP’s reliable data transfer service ensures that the data stream that a process reads out of its TCP receive buffer is uncorrupted, without gaps, without duplication, and in sequence; that is, the byte stream is exactly the same byte stream that was sent by the end system on the other side of the connection.

-In reliable data transfer techniques, it was conceptually easiest to assume that an individual timer is associated with each transmitted but not yet acknowledged segment.

-TCP timer management procedures use only a single retransmission timer, even if there are multiple transmitted but not yet acknowledged segments.

-We first present a highly simplified description of a TCP sender that uses only timeouts to recover from lost segments; we then present a more complete description that uses duplicate acknowledgments in addition to timeouts.

* TCP responds to the timeout event by retransmitting the segment that caused the timeout.
* TCP then restarts the timer.

-The major event that must be handled by the TCP sender is the arrival of an acknowledgment segment (ACK) from the receiver.

**Doubling the Timeout Interval**

-The first concerns the length of the timeout interval after a timer expiration.

-In this modification, whenever the timeout event occurs, TCP retransmits the not yet acknowledged segment with the smallest sequence number.

-But each time TCP retransmits, it sets the next timeout interval to twice the previous value, rather than deriving it from the last EstimatedRTT and DevRTT.

-TimeoutInterval is derived from the most recent values of EstimatedRTT and DevRTT.

-This modification provides a limited form of congestion control.

-The timer expiration is most likely caused by congestion in the network, that is, too many packets arriving at one (or more) router queues in the path between the source and destination, causing packets to be dropped and/or long queuing delays.

-In times of congestion, if the sources continue to retransmit packets persistently, the congestion

may get worse.

**Fast Retransmit**

-One of the problems with timeout-triggered retransmissions is that the timeout period can be relatively long.

-When a segment is lost, this long timeout period forces the sender to delay resending the lost packet, thereby increasing the end-to-end delay.

-The sender can often detect packet loss well before the timeout event occurs by noting so-called duplicate ACKs.

-A **duplicate ACK** is an ACK that reacknowledges a segment for which the sender has already received an earlier acknowledgment.

-When a TCP receiver receives a segment with a sequence number that is larger than the next, expected, in-order sequence number, it detects a gap in the data stream that is, a missing segment.

-This gap could be the result of lost or reordered segments within the network.

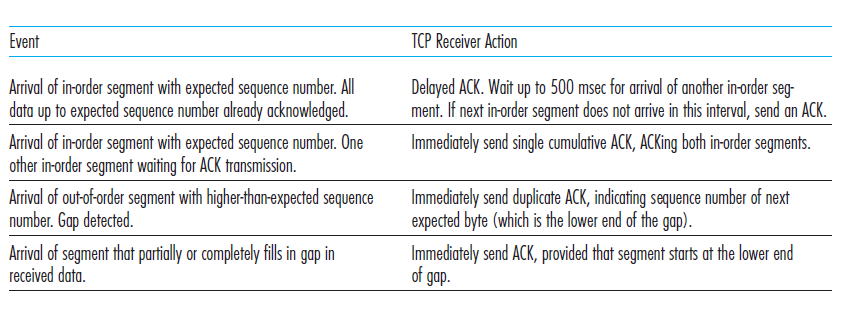
-Since TCP does not use negative acknowledgments, the receiver cannot send an explicit negative acknowledgment back to the sender.

-Instead, it simply reacknowledges the last in-order byte of data it has received.

-Because a sender often sends a large number of segments back-to-back, if one segment is lost, there will likely be many back-to-back duplicate ACKs.

-If the TCP sender receives three duplicate ACKs for the same data, it takes this as an indication that the segment following the segment that has been ACKed three times has been lost.

-In the case that three duplicate ACKs are received, the TCP sender performs a **fast retransmit**, retransmitting the missing segment before that segment’s timer expires.



**Table:** TCP ACK Generation Recommendation [RFC 5681]

* **Flow Control**

**-**When the TCP connection receives bytes that are correct and in sequence, it places the data in the receive buffer.

-The associated application process will read data from this buffer, but not necessarily at the instant the data arrives.

-Indeed, the receiving application may be busy with some other tasks and may not even attempt to read the data until long after it has arrived.

-If the application is relatively slow at reading the data, the sender can very easily overflow the connection’s receive buffer by sending too much data too quickly.

-TCP provides a **flow-control service** to its applications to eliminate the possibility of the sender overflowing the receiver’s buffer.

-Flow control is thus a speed-matching service: Matching the rate at which the sender is sending against the rate at which the receiving application is reading.

-A TCP sender can also be throttled due to congestion within the IP network; this form of sender control is referred to as **congestion control**.

-TCP provides flow control by having the sender maintain a variable called the **receive window.**

-The receive window is used to give the sender an idea of how much free buffer space is available at the receiver.

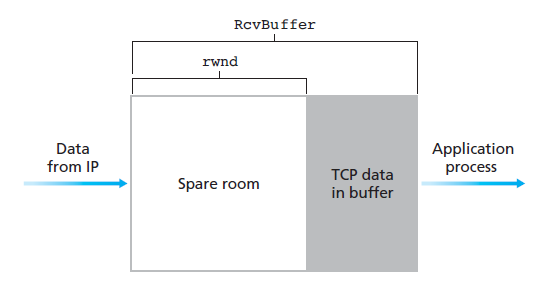
-Because TCP is full-duplex, the sender at each side of the connection maintains a distinct receive window.

* Suppose that Host Ais sending a large file to Host B over a TCP connection.
* Host B allocates a receive buffer to this connection; denote its size by RcvBuffer.
* From time to time, the application process in Host B reads from the buffer.
* **LastByteRead:** the number of the last byte in the data stream read from the

buffer by the application process in B

* **LastByteRcvd:** the number of the last byte in the data stream that has arrived

from the network and has been placed in the receive buffer at B



**Fig:** The receive window (rwnd) and the receive buffer(RcvBuffer)

-Because TCP is not permitted to overflow the allocated buffer, we must have

LastByteRcvd – LastByteRead <=RcvBuffer

-The receive window, denoted rwnd is set to the amount of spare room in the buffer:

rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]

-Because the spare room changes with time, rwnd is dynamic.

-Host B tells Host A how much spare room it has in the connection buffer by placing its current value of rwnd in the receive window field of every segment it sends to A.

-Initially, Host B sets rwnd = RcvBuffer.

-Host A makes sure throughout the connection’s life that

LastByteSent – LastByteAcked <=rwnd

* **TCP Connection Management**

**-**In this subsection we take a closer look at how a TCP connection is established and torn down.

**-**Suppose a process running in one host (client) wants to initiate a connection with another process in another host (server).

* The client application process first informs the client TCP that it wants to establish a connection to a process in the server.
* The TCP in the client then proceeds to establish a TCP connection with the TCP in the server in the following manner:

**Step 1:**

-The client-side TCP first sends a special TCP segment to the server-side TCP.

-This special segment contains no application-layer data.

-But one of the flag bits in the segment’s header the SYN bit, is set to 1.

-For this reason, this special segment is referred to as a **SYN segment**.

-In addition, the client randomly chooses an initial sequence number and puts this number in the sequence number field of the initial TCP SYN segment.

-This segment is encapsulated within an IP datagram and sent to the server.

**Step 2:**

-Once the IP datagram containing the TCP SYN segment arrives at the server host the server extracts the TCP SYN segment from the datagram, allocates the TCP buffers and variables to the connection, and sends a connection-granted segment to the client TCP.

-It does contain three important pieces of information in the segment header.

* First, the SYN bit is set to 1.
* Second, the acknowledgment field of the TCP segment header is set.
* The connection granted segment is referred to as a **SYNACK segment**.

**Step 3:**

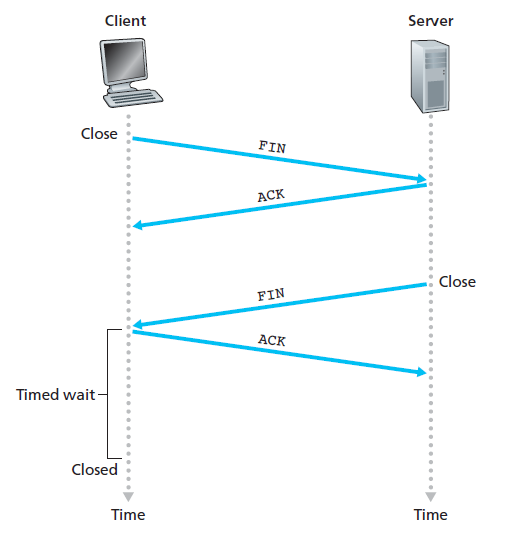
-Upon receiving the SYNACK segment, the client also allocates buffers and variables to the connection.

-The client host then sends the server yet another segment; this last segment acknowledges the server’s connection-granted segment.

-The SYN bit is set to zero, since the connection is established.

-This third stage of the three-way handshake may carry client-to-server data in the segment payload.

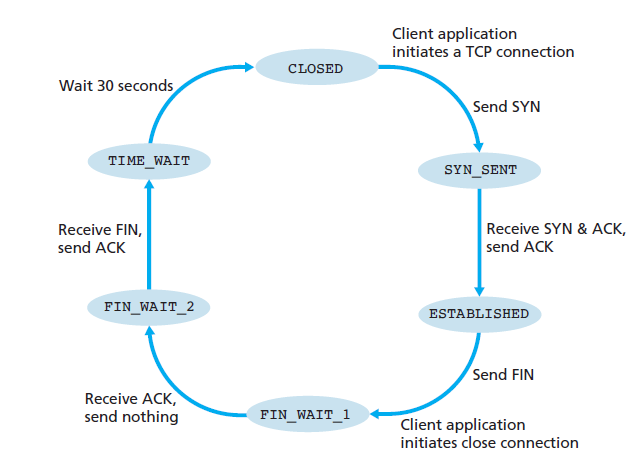
Once these three steps have been completed, the client and server hosts can send segments containing data to each other. In each of these future segments, the SYN bit will be set to zero. Note that in order to establish the connection, three packets are sent between the two hosts. This connection establishment procedure is often referred to as a **three-way handshake**.



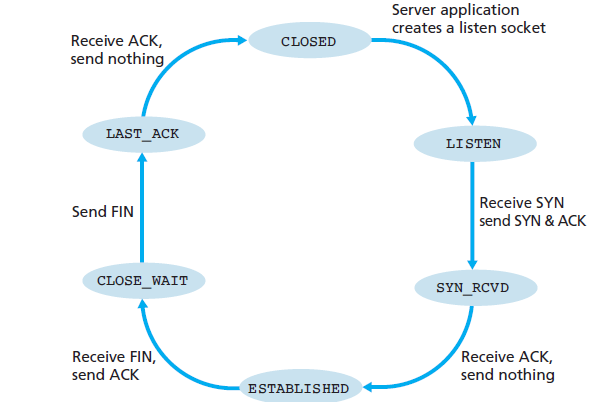
**Fig:** Closing a TCP connection

Suppose that the client application decides it wants to **close the connection**.

* This causes the client TCP to send a TCP segment with the FIN bit set to 1 and to enter the FIN\_WAIT\_1 state.
* While in the FIN\_WAIT\_1 state, the client TCP waits for a TCP segment from the server with an acknowledgment.
* When it receives this segment, the client TCP enters the FIN\_WAIT\_2 state.
* While in the FIN\_WAIT\_2 state, the client waits for another segment from the server with the FIN bit set to 1; after receiving this segment, the client TCP acknowledges the server’s segment and enters the TIME\_WAIT state.
* The TIME\_WAIT state lets the TCP client resend the final acknowledgment in case the ACK is lost.
* The time spent in the TIME\_WAIT state is implementation-dependent, but typical values are 30 seconds, 1 minute, and 2 minutes.
* After the wait, the connection formally closes and all resources on the client side (including port numbers) are released.



**Fig:** A typical sequence of TCP states visited by a client TCP



**Fig:** A typical sequence of TCP states visited by a server-side TCP

-The life of a TCP connection, the TCP protocol running in each host makes transitions through various **TCP states**.

-The client TCP begins in the CLOSED state.

-The application on the client side initiates a new TCP connection.

-This causes TCP in the client to send a SYN segment to TCP in the server.

-After having sent the SYN segment, the client TCP enters the SYN\_SENT state.

-While in the SYN\_SENT state, the client TCP waits for a segment from the server TCP that includes an acknowledgment for the client’s previous segment and has the SYN bit set to 1.

-Having received such a segment, the client TCP enters the ESTABLISHED state.

-While in the ESTABLISHED state, the TCP client can send and receive TCP segments containing payload (that is, application-generated) data.