Transport-Layer services

- A transport-layer protocol provides logical communication between application processes running on different hosts.
- By logical communication, application processes communicate with each other by using the logical communication provided by the transport layer to send messages to each other, free from the worry of the details of the physical infrastructure used to carry these messages.
- 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.
- On the receiving side, the transport layer reassembles segments into messages, passes to application layer.
- A network-layer protocol provides logical communication betweenhosts.

Relationship between Transport and Network Layers

- The transport layer lies just above the network layer in the protocol stack.
- Whereas a transport-layer protocol provides logical communication between processes running on different hosts, a network-layer protocol provides logical communication between hosts.
- Let's examine this distinction with the aid of a household analogy.
- Consider two houses, one on the East Coast and the other on the West Coast, with each house being home to a dozen kids.
- The kids in the East Coast household are cousins of the kids in the West Coast household.
- The kids in the two households love to write to each other-each kid writes each cousin every week, with each letter delivered by the traditional postal service in a separate envelope.
- Thus, each household sends 144 letters to the other household every week.
- In each of the households there is one kid Ann in the West Coast house and Bill in the East Coast house responsible for mail collection and mail distribution.
- Each week Ann visits all her brothers and sisters, collects the mail, and gives the mail to a postal-service mail person who makes daily visits to the house.
- When letters arrive at the West Coast house, Ann also has the job of distributing the mail to her brothers and sisters. Bill has a similar job on the East Coast.
- In this example, the postal service provides logical communication between the two houses-the postal service moves mail from house to house, not from person to person.
- On the other hand, Ann and Bill provide logical communication among the cousins-Ann and Bill pick up mail from and deliver mail to their brothers and sisters.
- Note that from the cousins' perspective, Ann and Bill are the mail service, even though Ann and Bill are only a part (the end system part) of the end-to-end deliveryprocess.

- This household example serves as a nice analogy for explaining how the transport layer relates to the network layer:

  hosts (also called end systems) = houses
  processes = cousins
  application messages = letters in envelopes
  network-layer protocol = postal service (including mail persons)
  transport-layer protocol = Ann and Bill

- Continuing with this analogy, observe that Ann and Bill do all their work within their respective homes; they are not involved, for example, in sorting mail in any intermediate mail centre or in moving mail from one mail centre to another.
- Similarly, 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; but it doesn't have any say about how the messages are moved within the network core.
- In fact, intermediate routers neither act on, nor recognize, any information that the transport layer may have appended to the application messages.

Multiplexing and Demultiplexing

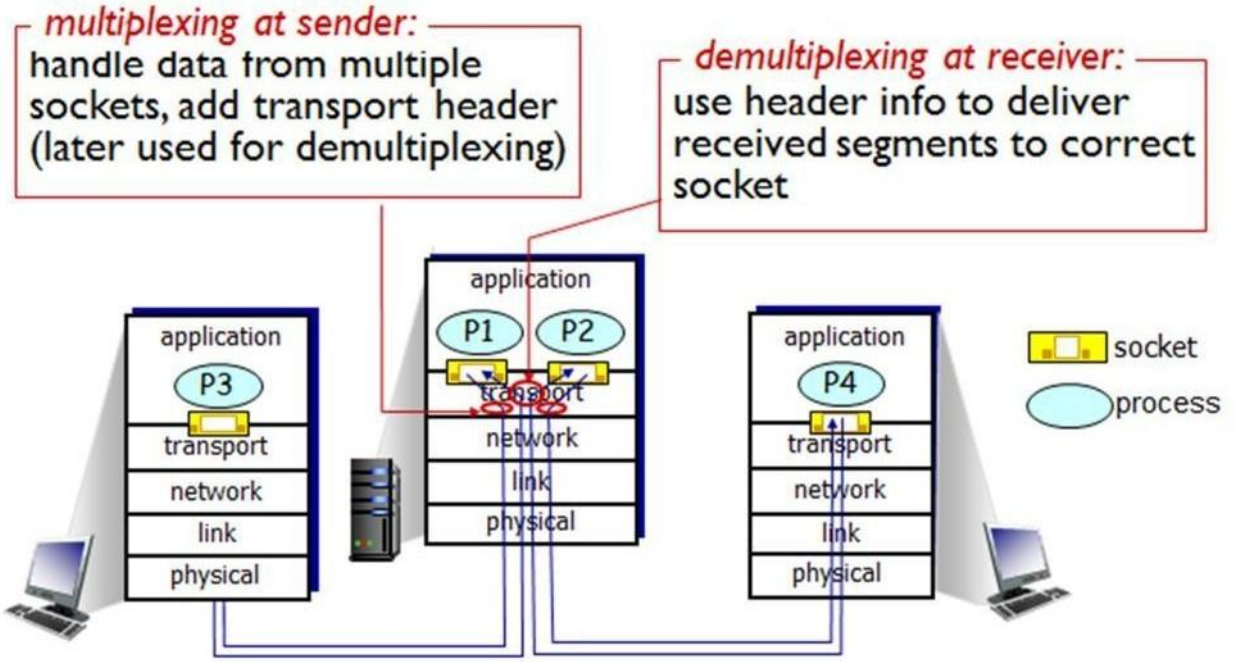


Fig. 1 Transport-layer multiplexing and demultiplexing

- The job of gathering data chunks at the source host from different sockets, encapsulating each data chunk with header information (that will later be used in demultiplexing) to create segments, and passing the segments to the network layer is called multiplexing.

- At the receiving end, the transport layer examines these fields to identify the receiving socket and then directs the segment to that socket. This job of delivering the data in a transport-layer segment to the correct socket is called demultiplexing.
- Transport layer in the middle host in Figure 1 must demultiplex segments arriving from the network layer below to either process P1 or P2 above; this 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.

Endpoint Identification

- Sockets must have unique identifiers.
- Each segment must include header fields identifying the socket, these header fields are the source port number field and the destination port number field.
- Each port number is a 16-bit number: 0 to 65535.

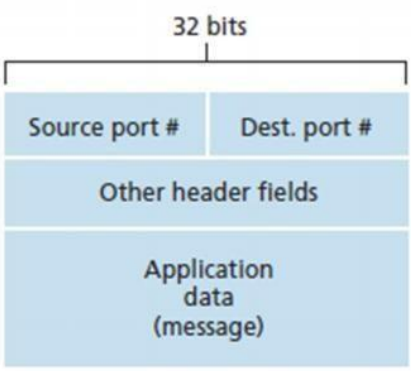


Fig. 2 Source and destination port-number fields in a transport-layer segment

Connectionless Multiplexing and Demultiplexing

- Suppose a process on Host A, with port number 19157, wants to send data to a process with UDP port 46428 on Host B.

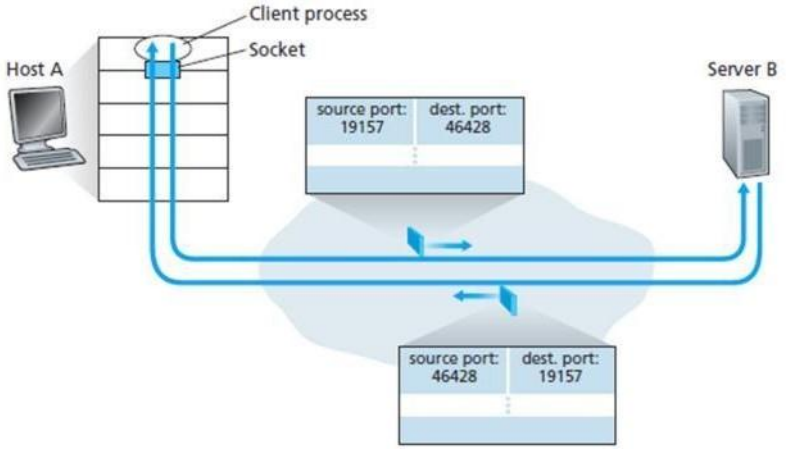


Fig. 3 The inversion of source and destination port numbers

- Transport layer in Host A creates a segment containing source port, destination port, and data and then passes it to the network layer in Host A.
- Transport layer in Host B examines destination port number and delivers segment to socket identified by port 46428.
- Note: a UDP socket is fully identified by a two-tuple consisting of
    1. a destination IP address
    2. a destination port number
- Source port number from Host A is used at Host B as "return address".

Connection-Oriented Multiplexing and Demultiplexing

- Each TCP connection has exactly two end-points.
- This means that two arriving TCP segments with different source IP addresses or source port numbers will be directed to two different sockets, even if they have the same destination port number.
- So a TCP socket is identified by four-tuple
    1. source IP address
    2. source port #
    3. destination IP address
    4. destination port #
- Whereas UDP is identified by only two-tuples
    1. destination IP address
    2. destination port #

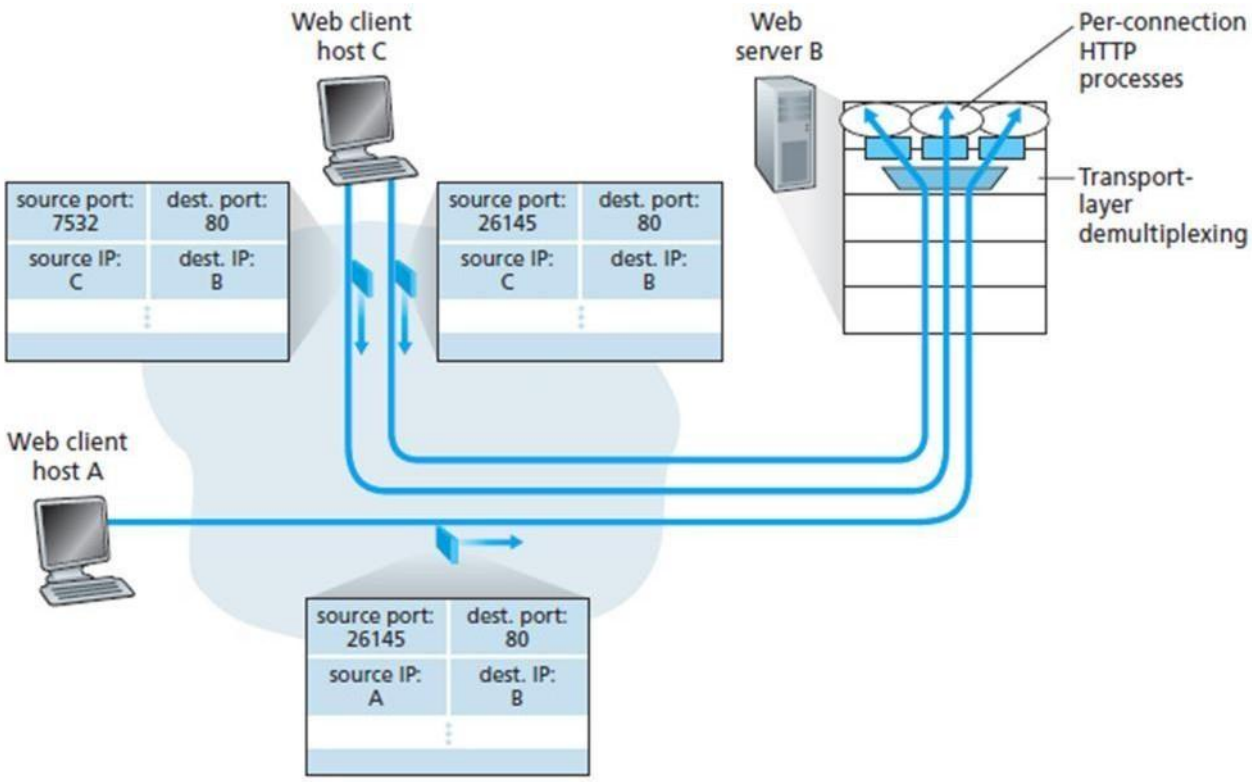


Fig. 4 Two clients, using the same destination port number (80) to communicate with the same Web server application

UDP Segment Structure

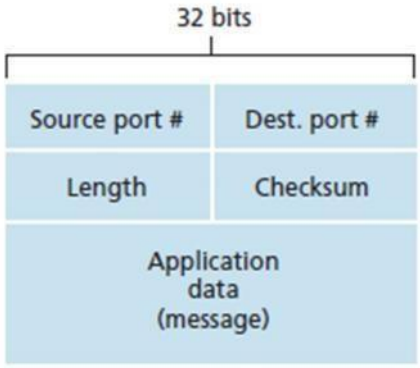


Fig. 5 UDP segment structure

- The port numbers allow the destination host to pass the application data to the correct process running on the destination end system (that is used to perform the demultiplexing function).
- The length field specifies the number of bytes in the UDP segment (header plusdata).
- The checksum is used by the receiving host to check whether errors have been introduced into the segment.

How to calculate (find) checksum:

- The UDP checksum is calculated on the sending side by summing all the 16-bit words in the segment, with any overflow being wrapped around and then the 1's complement is performed and the result is added to the checksum field inside the segment.
- At the receiver side, all words inside the packet are added and the checksum is added upon them if the result is 1111 1111 1111 1111 then the segment is valid else the segment has an error.

0110011001100110
0101010101010101
0000111100001111
} Three 16 bits words

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

0110011001100110
0101010101010101

1011101110111011 → Sum of 1st two 16 bit words.

Adding the third word to the above sum gives:

1011101110111011 → Sum of 1st two 16 bit words.
0000111100001111 → Third 16 bits word

1100101011001010 → Sum of all three 16 bit words.

Taking 1's complement for the final sum:

1100101011001010 → Sum of all three 16 bit words.
0011010100110101 → 1's complement for the final sum.

The 1's complement value is called as **Checksum**.

Principles of Reliable Data Transfer

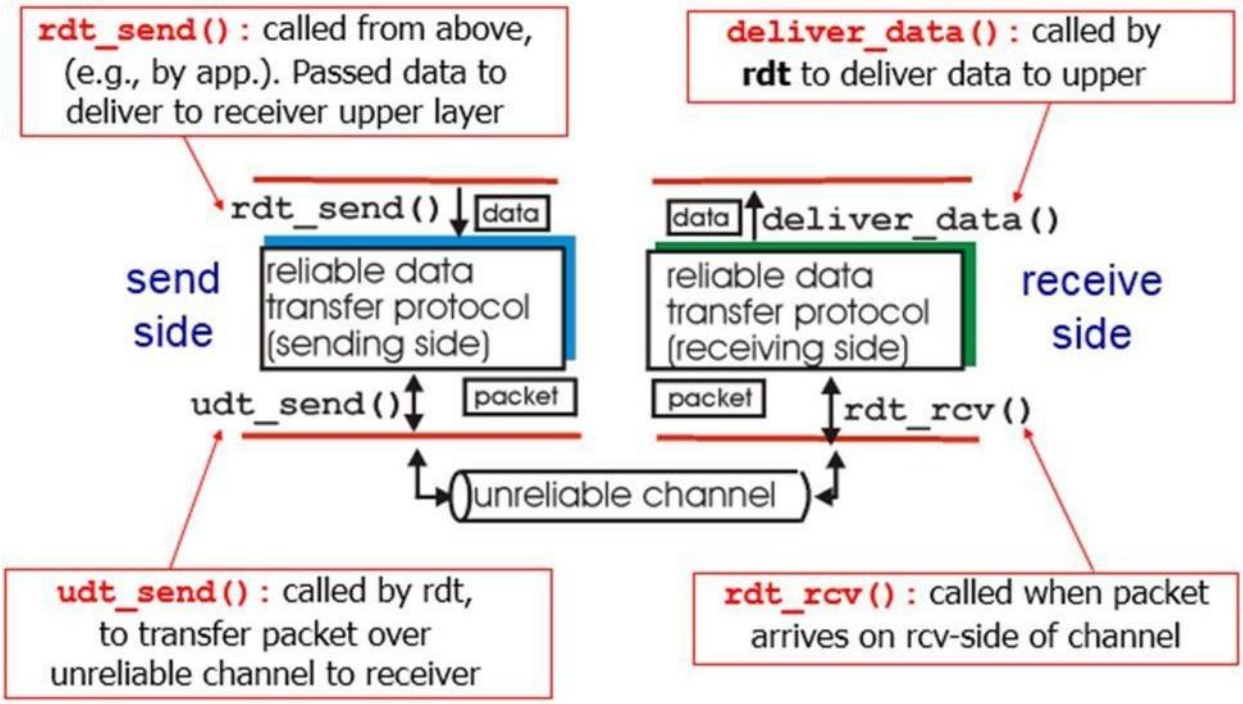


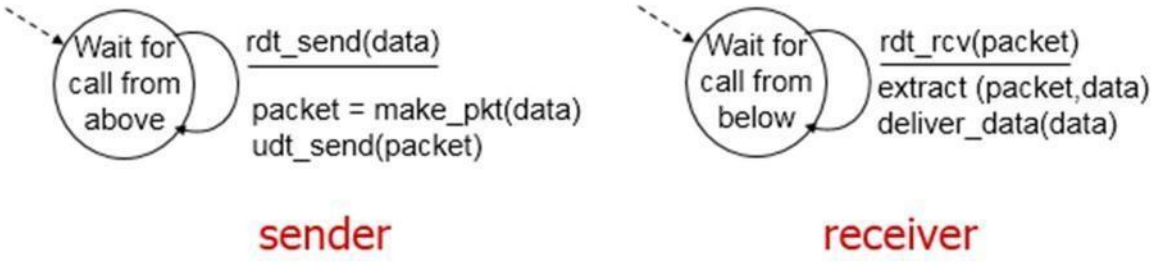
Fig. 7 Reliable data transfer commands

- The sending side of the data transfer protocol will be invoked from above by a call to rdt_send().
- 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().
- Both the send and receive sides of rdt send packets to the other side by a call to udt_send().

Building a Reliable Data Transfer Protocol

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

- We first consider the simplest case in which the underlying channel is completely reliable.
- The protocol itself, which we will call rdt1.0, is trivial.

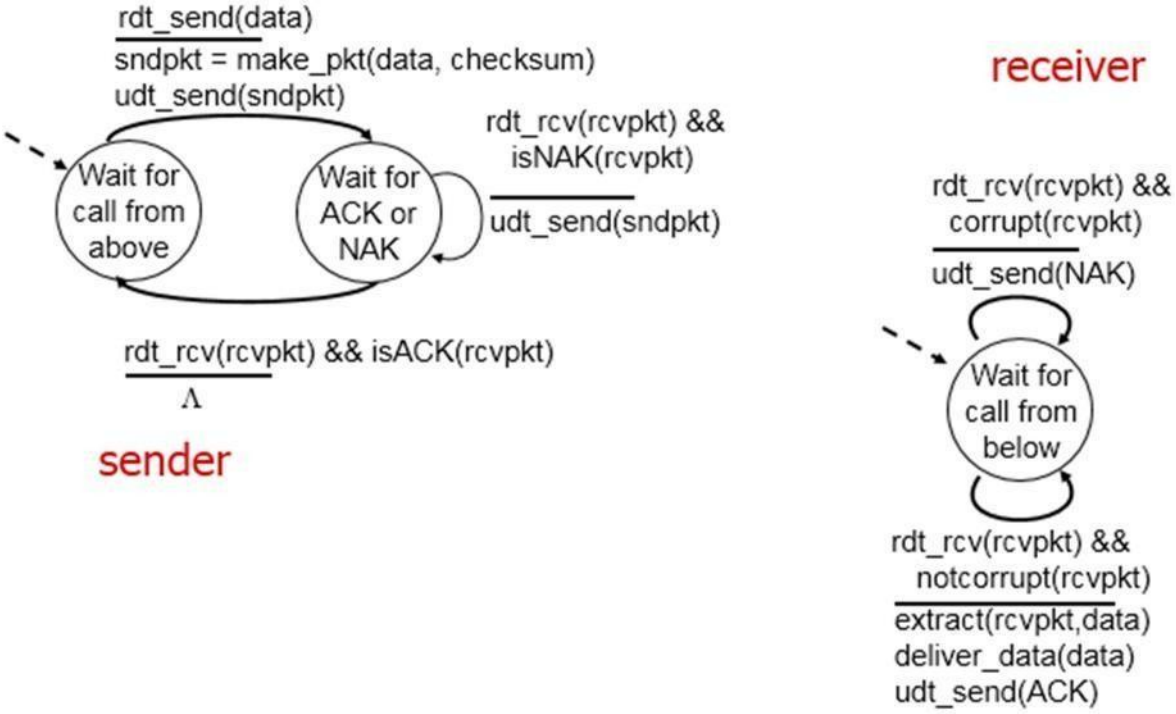


- The sender and receiver FSMs (Finite State Machines) have only one state.
- The arrows in the FSM description indicate the transition of the protocol from one state to another. (Since each FSM has just one state, a transition is necessarily from the one state back to itself).
- The event causing the transition is shown above the horizontal line labelling the transition, and the action(s) taken when the event occurs are shown below the horizontal line.
- The sending side of rdt simply accepts data from the upper-layer via the rdt_send(data) event, puts the data into a packet (via the action make_pkt(packet,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 using the action extract(packet,data) and passes the data up to the upper-layer.
- Also, all packet flow is 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 since nothing can go wrong.

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.
- We'll continue to assume for the moment that all transmitted packets are received (although their bits may be corrupted) in the order in which they were sent.
- Before developing a protocol for reliably communicating over such a channel, first consider how people might deal with such a situation.
- Consider how you yourself might dictate a long message over the phone. In a typical scenario, the message taker might say ``OK'' after each sentence has been heard, understood, and recorded. If the message taker hears a garbled sentence, you're asked to repeat the garbled sentence. This message dictation protocol uses both positive acknowledgements (``OK'') and negative acknowledgements (``Please repeat that''). 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. In a computer network setting, reliable data transfer protocols based on such retransmission are known ARQ (Automatic Repeat reQuest) protocols.
- Fundamentally, two 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 transport protocol uses the Internet checksum field for exactly this purpose. Error detection and correction techniques allow the receiver to detect, and possibly correct packet bit errors.

- **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 learn of the receiver's view of the world (in this case, whether or not a packet was received correctly) is for the receiver to provide explicit feedback to the sender. The positive (ACK) and negative acknowledgement (NAK) replies in the message dictation scenario are an example of such feedback. Our rdt2.0 protocol will similarly send ACK and NAK packets back from the receiver to the sender.



- The send side of rdt2.0 has two states.
- In one state, the send-side protocol is waiting for data to be passed down from the upper layer.
- In the other 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), the sender knows the most recently transmitted packet has been received correctly and thus the protocol returns to the state of waiting for data from the upperlayer.
- 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 datapacket.
- It is important to note that when the receiver is in the wait-for-ACK-or-NAK state, it cannot get more data from the upper layer; that will only happen 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 behaviour, 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.
- In above figure the notation rdt_rcv(rcvpkt) && corrupt(rcvpkt) corresponds to the event where a packet is received and is found to be in error.

rdt2.1: sender, handles garbled ACK/NAKs

- Protocol rdt2.0 may look as if it works but unfortunately has a fatal flaw.
- In particular, we haven't accounted for the possibility that the ACK or NAK packet could be corrupted.
- Minimally, we will need to add checksum bits to ACK/NAK packets in order to detect such errors.
- The more difficult question is how the protocol should recover from errors in ACK or NAK packets.
- 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.
- An approach is for the sender to simply resend the current data packet when it receives a garbled ACK or NAK packet.
- This, 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.
- Thus, it cannot know a priori whether an arriving packet contains new data or is a duplicate.
- 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 (the sequence number of the received packet has the same sequence number as the most recently received packet) or a new packet (the sequence number changes).
- Since we are currently assuming a channel that does not lose packets, ACK and NAK packets do not themselves need to indicate the sequence number of the packet they are ACKing or NAKing, since 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.

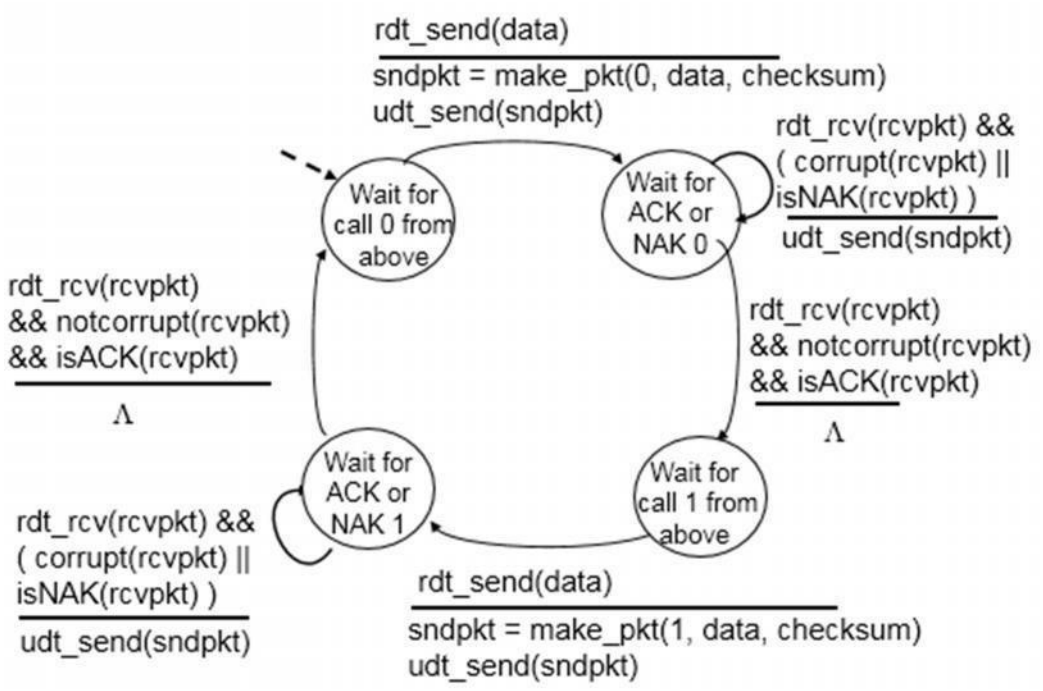


Fig 10 rdt 2.1 Sender

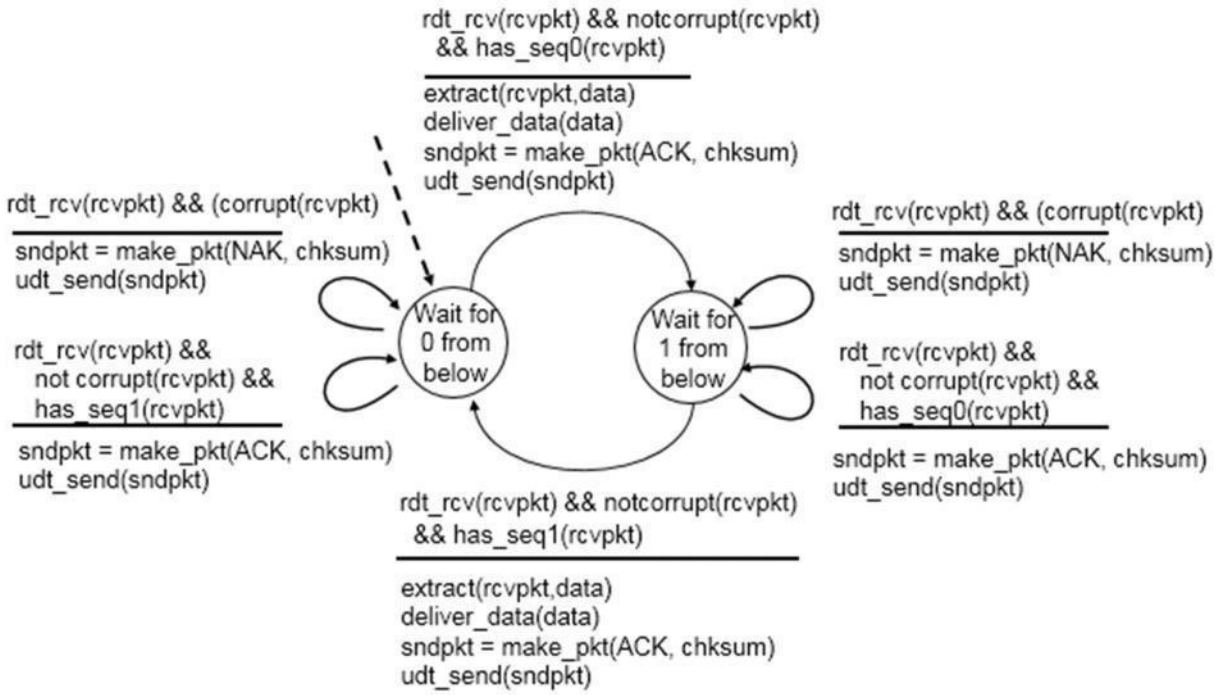


Fig 11 rdt 2.1 Receiver

- The rdt2.1 sender and receiver FSM's each now have twice as many states as rdt2.0.

- 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.
- Note that the actions in those states where a 0-numbered packet is being sent or expected are mirror images of those where a 1-numbered packet is being sent or expected; the only differences have to do with the handling of the sequence number.
- Protocol rdt2.1 uses both positive and negative acknowledgements from the receiver to the sender.
- A negative acknowledgement is sent whenever a corrupted packet, or an out of order packet, is received.
- We can accomplish the same effect as a NAK if instead of sending a NAK, we instead send an ACK for the last correctly received packet.
- A sender that receives two ACKs for the same packet (i.e., receives duplicate ACKs) knows that the receiver did not correctly receive the packet following the packet that is being ACKed twice. Our NAK-free reliable data transfer protocol for a channel with bit errors is rdt2.2.

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

- Suppose now that in addition to corrupting bits, the underlying channel can lose packets as well.
- Two additional concerns must now be addressed by the protocol: how to detect packet loss and what to do when this occurs.
- The use of checksumming, sequence numbers, ACK packets, and retransmissions - the techniques already developed in rdt 2.2 - will allow us to answer the latter concern. Handling the first concern will require adding a new protocol mechanism.
- There are many possible approaches towards dealing with packet loss.
- 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 thereceiver.
- 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? It must clearly wait at least as long as a round trip delay between the sender and receiver (which may include buffering at intermediate routers or gateways) plus whatever amount of time is needed to process a packet at thereceiver.
- If an ACK is not received within this time, the packet isretransmitted.
- Note that if a packet experiences a particularly large delay, the sender may retransmit the packet even though neither the data packet nor its ACK have beenlost.
- This introduces the possibility of duplicate data packets in the sender-to-receiver channel.
- Happily, protocol rdt2.2 already has enough functionality (i.e., sequence numbers) to handle the case of duplicate packets.

- From the sender's viewpoint, retransmission is a solution. 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.
- In all cases, the action is the same: retransmit. In order to implement a time-based retransmission mechanism, a countdown timer will be needed that can interrupt the sender after a given amount of timer has expired.
- The sender will thus need to be able to (i) start the timer each time a packet (either a first time packet, or a retransmission) is sent, (ii) respond to a timer interrupt (taking appropriate actions), and (iii) stop the timer.
- The existence of sender-generated duplicate packets and packet (data, ACK) loss also complicates the sender's processing of any ACK packet it receives.
- If an ACK is received, how is the sender to know if it was sent by the receiver in response to its (sender's) own most recently transmitted packet, or is a delayed ACK sent in response to an earlier transmission of a different data packet? The solution to this dilemma is to augment the ACK packet with an acknowledgement field. When the receiver generates an ACK, it will copy the sequence number of the data packet being ACK'ed into this acknowledgement field. By examining the contents of the acknowledgment field, the sender can determine the sequence number of the packet being positively acknowledged.

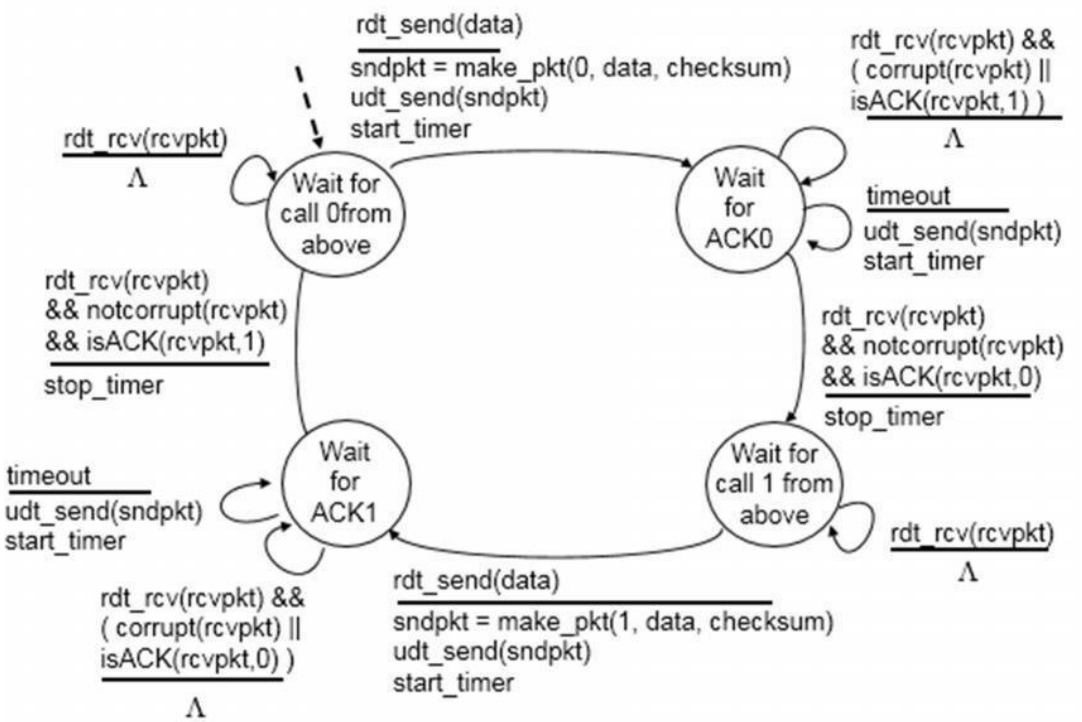


Fig 12 rdt 3.0 Sender

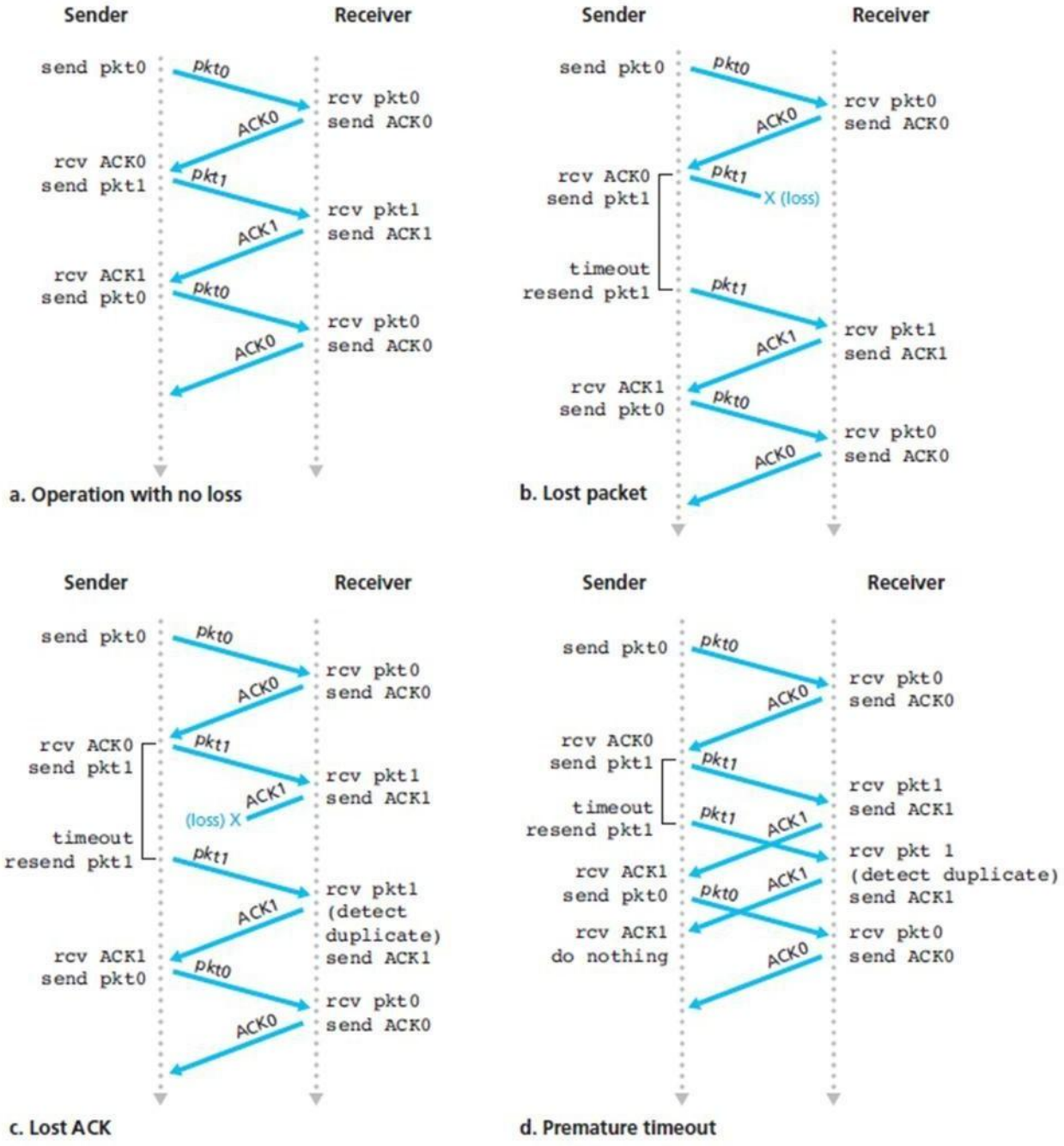
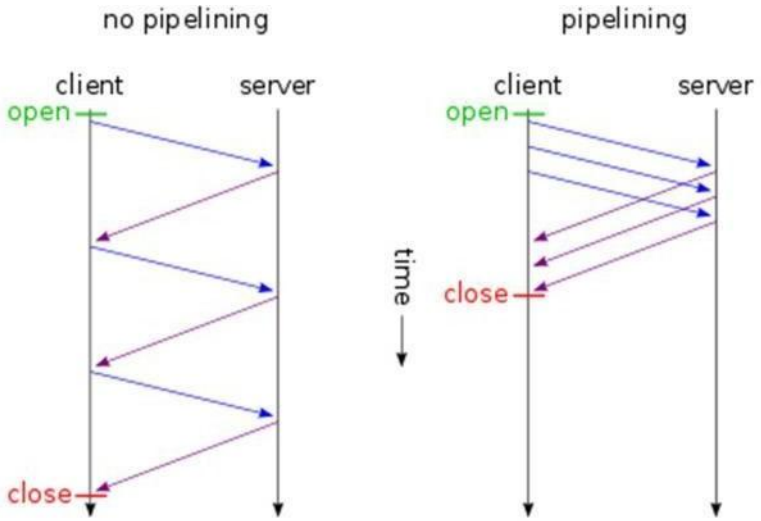


Fig. 13 Operation of rdt3.0, the alternating-bit protocol

Protocol pipelining

- Protocol pipelining is a technique in which multiple requests are written out to a single socket without waiting for the corresponding responses (acknowledged).
- Pipelining can be used in various application layer network protocols, like HTTP/1.1, SMTP and FTP.
- Range of sequence numbers must be increased.

- Data or Packet should be buffered at sender and/or receiver.



- Two generic forms of pipelined protocols are
  1. Go-Back-N
  2. Selective repeat

Go-Back-N

- Go-Back-N ARQ is a specific instance of the automatic repeat request (ARQ) protocol, in which the sending process continues to send a number of frames specified by a window size even without receiving an acknowledgement (ACK) packet from thereceiver.
- The receiver process keeps track of the sequence number of the next frame it expects to receive, and sends that number with every ACK it sends.
- The receiver will discard any frame that does not have the exact sequence number it expects (either a duplicate frame it already acknowledged or an out-of-order frame it expects to receive later) and will resend an ACK for the last correct in-orderframe.
- Once the sender has sent all of the frames in its window, it will detect that all of the frames since the first lost frame are outstanding, and will go back to the sequence number of the last ACK it received from the receiver process and fill its window starting with that frame and continue the process over again.
- Go-Back-N ARQ is a more efficient use of a connection than Stop-and-wait ARQ, since unlike waiting for an acknowledgement for each packet; the connection is still being utilized as packets are being sent.
- However, this method also results in sending frames multiple times – if any frame was lost or damaged or the ACK acknowledging them was lost or damaged then that frame and all following frames in the window (even if they were received without error) will be re-sent. To avoid this, Selective Repeat ARQ can be used.

How does Go-Back-N ARQ protocol works:

- Consider a scenario given in fig 14 there are four frames 0,1,2,3 respectively. Now as sending window size is 3 it will send frame 0, 1 and 2 at a time.

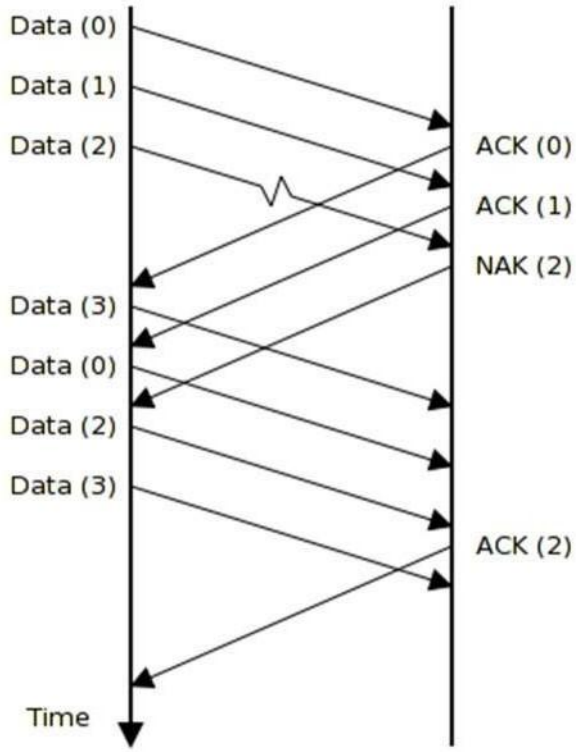


Fig. 14 Go-Back-N Protocol

- At receiver side frame 0 arrives it sends ACK for that.
- Now at sender side window shift at frame 3 and it sends frame no 3.
- At receiver side frame 1 arrives it sends ACK for that.
- Now at sender side window shift at frame 4 and it sends frame no 4.
- Now receiver detects frame 2 is missing or lost. It will send NAK for that.
- Sender receives NAK at this point sending window is pointing towards frame 2, 3 and 4 so send will resend frame 2, 3 and 4.

Selective repeat

- Selective Repeat attempts to retransmit only those packets that are actually lost due to errors.
- Receiver must be able to accept packets out of order
- Since receiver must release packets to higher layer in order, the receiver must be able to buffer some packets
- The receiver acknowledges every good packet, packets that are not ACKed before a time-out are assumed lost or in error.
- Notice that this approach must be used to be sure that every packet is eventually received.
- An explicit NAK (selective reject) can request retransmission of just one packet.
- This approach can speed up the retransmission but is not strictly needed.

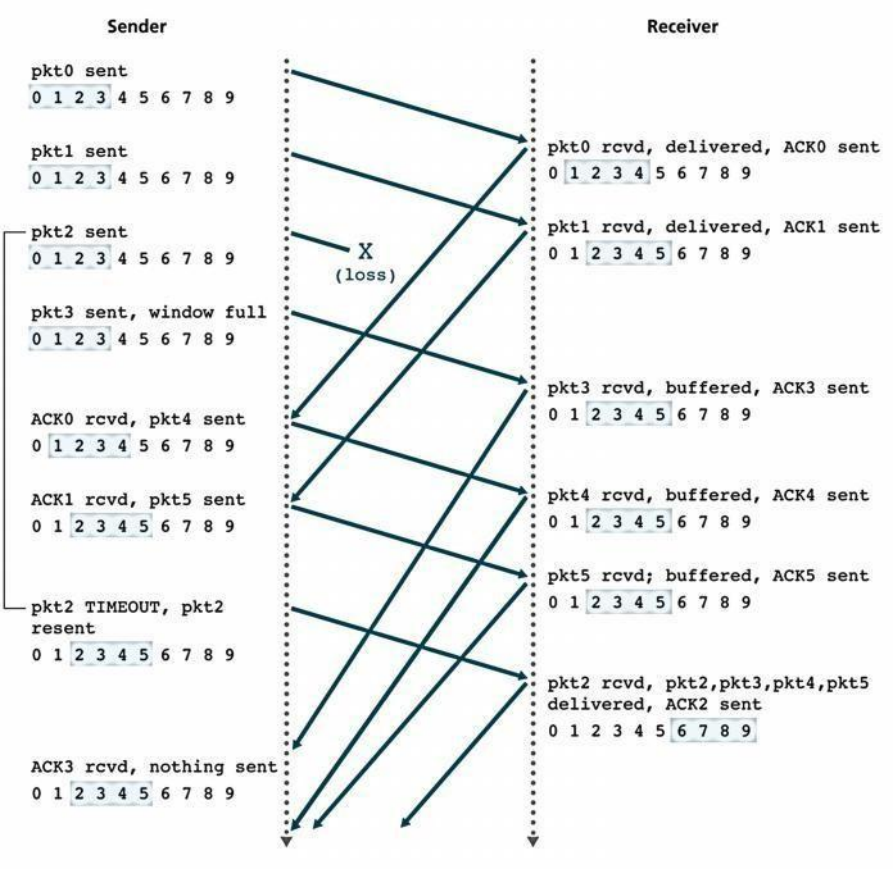


Fig. 15 Selective Repeat

TCP segment structure

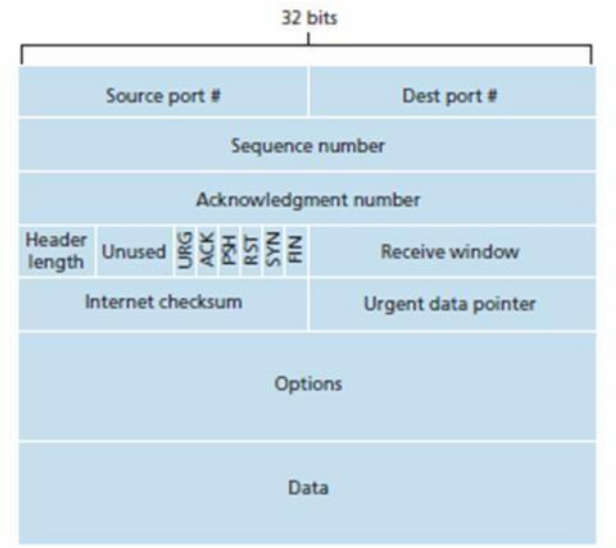


Fig. 16 TCP segment structure

- The unit of transmission in TCP is called segments.
- The header includes source and destination port numbers, which are used for multiplexing/demultiplexing data from/to upper-layer applications.
- 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 sequence number for a segment is the byte-stream number of the first byte in the segment.
- The acknowledgment number is the sequence number of the next byte a Host is expecting from another Host.
- 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 16-bit receive window field is used for flow control. It is used to indicate the number of bytes that a receiver is willing to accept.
- The 16-bit checksum field is used for error checking of the header and data.
- Unused 6 bits are reserved for future use and should be sent to zero.
- Urgent Pointer is used in combining with the URG control bit for priority data transfer. This field contains the sequence number of the last byte of urgent data.
- **Data**: The bytes of data being sent in the segment.
- **URG (1 bit)**: indicates that the Urgent pointer field is significant.
- **ACK (1 bit)**: indicates that the Acknowledgment field is significant.
- **PSH (1 bit)**: Push function. Asks to push the buffered data to the receiving application.
- **RST (1 bit)**: Reset the connection.
- **SYN (1 bit)**: Synchronize sequence numbers. Only the first packet sent from each end should have this flag set. Some other flags and fields change meaning based on this flag, and some are only valid for when it is set, and others when it is clear.
- **FIN (1 bit)**: No more data from sender.

Flow Control

- In data communications, flow control is the process of managing the rate of data transmission between two nodes to prevent a fast sender from overwhelming a slow receiver.
- It prevent receiver from becoming overloaded.
- Receiver advertises a window rwnd with each acknowledgement
- Window
    - ⑩ closed (by sender) when data is sent and ack'd
    - opened (by receiver) when data is read
- The size of this window can be the performance limit (e.g. on a LAN).

Congestion Control

- When a connection is established, a suitable window size has to bechosen.
- The receiver can specify a window based on its buffer size.

- If the sender sticks to this window size, problems will not occur due to buffer overflow at the receiving end, but they may still occur due to internal congestion within the network.
- In Figure 17 (a), we see a thick pipe leading to a small-capacity receiver.
- As long as the sender does not send more water than the bucket can contain, no water will be lost.
- In Figure 17 (b), the limiting factor is not the bucket capacity, but the internal carrying capacity of the network.

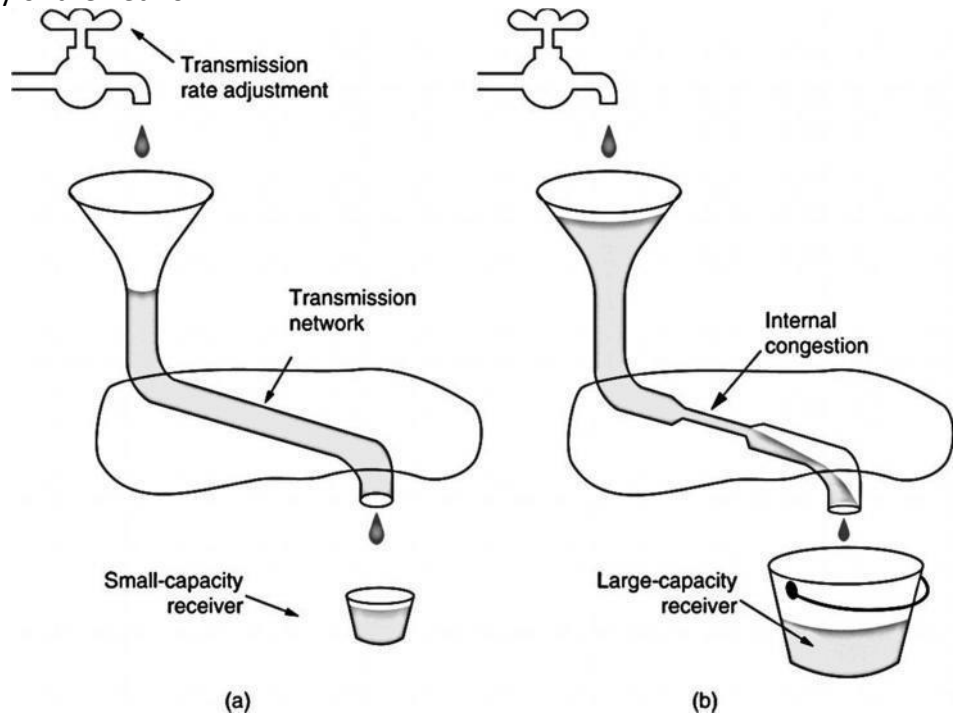


Fig. 17 TCP segment structure

- Figure 17: (a) a fast network feeding a low capacity receiver. Figure 17: (b) A slow network feeding a high-capacity receiver.
- If too much water comes in too fast, it will back up and some will be lost (in this case by overflowing the funnel).
- Each sender maintains two windows: the window the receiver has granted and a second window, the congestion window.
- Each reflects the number of bytes the sender may transmit.
- The number of bytes that may be sent is the minimum of the two windows.
- Thus, the effective window is the minimum of what the sender thinks is all right and what the receiver thinks is all right.
- When a connection is established, the sender initializes the congestion window to the size of the maximum segment in use on the connection.
- It then sends one maximum segment.

- If this segment is acknowledged before the timer goes off, it adds one segment's worth of bytes to the congestion window to make it two maximum size segments and sends two segments.
- As each of these segments is acknowledged, the congestion window is increased by one maximum segment size.
- When the congestion window is n segments, if all n are acknowledged on time, the congestion window is increased by the byte count corresponding to n segments.

TCP Slow Start

- Slow-start is part of the congestion control strategy used by TCP, the data transmission protocol used by many Internet applications.
- Slow-start is used in conjunction with other algorithms to avoid sending more data than the network is capable of transmitting to avoid causing network congestion.

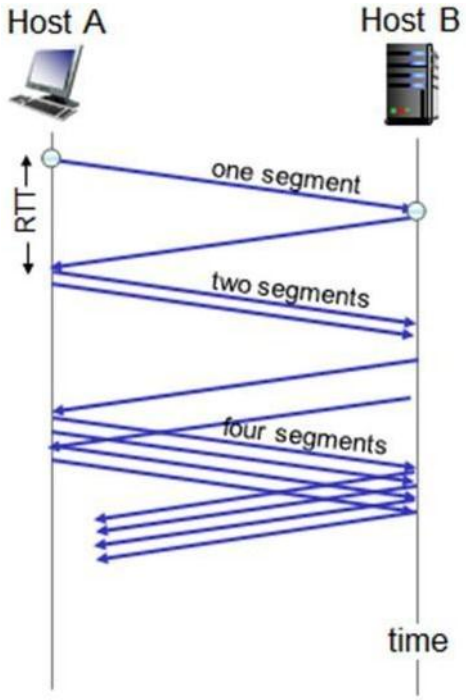


Fig. 18 TCP Slow start

- Slow-start begins initially with a congestion window Size (cwnd) of 1, 2 or 10.
- The value of the Congestion Window will be increased with each acknowledgement (ACK) received, effectively doubling the window size each round trip time ("although it is not exactly exponential because the receiver may delay its ACKs, typically sending one ACK for every two segments that it receives).
- The transmission rate will be increased with slow-start algorithm until either a loss is detected, or the receiver's advertised window (rwnd) is the limiting factor, or the slow start threshold (ssthresh) is reached.
- If a loss event occurs, TCP assumes that it is due to network congestion and takes steps to reduce the offered load on the network.

- Once ssthresh is reached, TCP changes from slow-start algorithm to the linear growth (congestion avoidance) algorithm. At this point, the window is increased by 1 segment for each RTT.