Computer Networks (UE18CS301)

Unit 3

Aronya Baksy

October 2020

1 Transport Layer and Transport Layer Services

- Transport layer protocols provide **logical communication** between processes running on different*host* machines. Logical communication means that the application layer can assume that there is a direct connection between the hosts even when there is none.
- The transport layer receives data from the application layer, breaks it up into **segments**, and attaches a transport layer header to each segment.
- Segments are encapsulated into **datagrams** at the network layer and transmitted over the network.
- On the receiver side, the network layer extracts the transport layer segment from the datagram and passes it up to the transport layer. From the segment, the transport layer makes the application data available to the correct application

1.1 Transport and Network Layer relationship

- The transport layer provides communication between *processes* running on different host machines, while the network layer provides communication between *hosts*.
- Let there be 2 houses on opposite sides of a country. Each house has 12 children, and each child in one particular house sends one letter to every child in the other house (there are 12*12 = 144 letters to be sent from one house to the other).
- Let Alice in house 1, and Bob in house 2, be responsible for collecting all the letters from their respective houses, going to the post office, and mailing the letters to the other house.
- Analogous to a computer network, this can be seen as
  - Processes: Each child in each house
  - Application layer messages: Letters in each envelope
  - Hosts/End Systems: House 1 and House 2
  - Transport Layer protocol: Alice and Bob
  - Network Layer protocol: Postal Service

1.2 Transport Layer Protocols and Services

- **UDP** (User Datagram Protocol) is the most basic of Transport layer protocols. It offers the services of process-to-process *data transfer*, and *error checking*.
- UDP is an unreliable protocol, as it does not offer guarantees that the data sent by one process will reach the destination process intact (or at all). UDP Traffic is also unregulated, as there is no congestion control facility in UDP.
- **TCP** (Transmission Control Protocol) is a connection-oriented transport layer protocol. The services offered by TCP are:

1. **Reliable Data Transfer**: Using flow control, timers, sequence numbers and acknowledgements, TCP ensures that data sent by one process is received at the destination, despite the unreliability of the underlying Network Layer protocol (IP).
2. **Congestion control**: This is not strictly an application service, but rather a service provided to the entire internet by the TCP. This is done by controlling the data rate on the sender side of the TCP Connection.

2 Transport Layer Multiplexing and Demultiplexing

- The underlying network layer only provides data transfer between host machines connected on the internet.
- Multiplexing and demultiplexing are done by the Transport layer protocols to extend this host-to-host transmission to communication between *processes* running on those hosts.
- The transport layer has the responsibility of ensuring that out of all the packets arriving at a host, the packets go to the correct application that they are intended for.
- **Demultiplexing** is the process in which the transport layer examines the fields in the incoming segment and routes that segment to the appropriate socket in the destination.
- **Multiplexing** is the process of taking in chunks of application data from the sending processes, creatings segments out of them (by adding transport layer header information that is used while demultiplexing) and passing them on to the network layer.

2.1 UDP Multiplexing and Demultiplexing

- A UDP socket is identified by the 2-tuple of ⟨ Destination IP Address, Destination Port Number ⟩.
- If two incoming segments have different source IP Addresses but same destination IP address and destination port number, then they will be demultiplexed into the same process running on that socket.
- The source port number and source IP Address are used to determine the return address when the destination machine wants to send some message back to the source.

2.2 TCP Multiplexing and Demultiplexing

- A TCP socket is identified by the 4-tuple of ⟨ Source IP Address, Source port number, Destination IP Address, Destination Port Number ⟩.
- All four values are used at the destination side to demultiplex the incoming segments.
- The server has a welcoming socket on each port number that waits for incoming TCP connection requests on those port numbers. When a connection request for that port is received, then the server sets up the TCP socket on that port.

3 User Datagram Protocol (UDP)

- UDP is a connectionless transport layer protocol. It features basic capabilities of transport layer multiplexing/demultiplexing, and basic error checking using checksums.
- UDP is called connectionless as there is no handshaking between client and server before exchange of data between the two, like there is TCP.
- The advantages of UDP are:
    1. Smaller packet sizes (8 bytes for UDP header vs 20 bytes for TCP header)

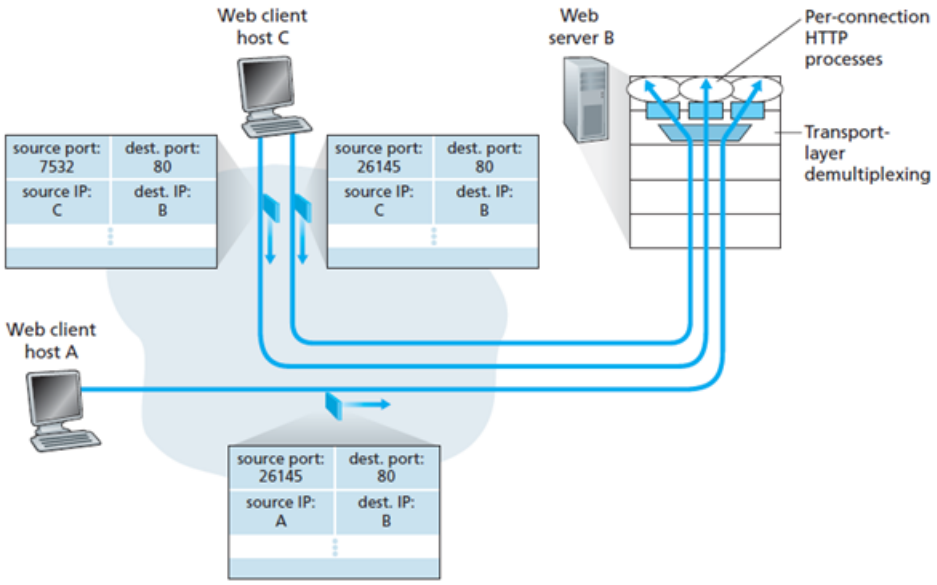


Figure 1: TCP Multiplexing and Demultiplexing

2. No need to maintain connection state unlike in TCP (receive and buffers, congestion control parameters, sequence/acknowledgement numbers are stored in the endpoints), thus less memory overheads for host machines in UDP than in TCP.
3. No delays associated with connection establishment and handshaking
4. Finer application level control on data sent and when:
   (a) TCP can throttle links as part of its congestion control, and it can also retransmit packets until an acknowledgement of receipt is received from the server.
   (b) UDP on the other hand simply packages application layer data into segments and passes them on to the network layer.
   (c) For real time applications that often require a minimum sending rate, TCP is not suited. UDP is used, and any functionality over the barebones UDP can be implemented in the application layer.

3.1 UDP Segment Structure

- The UDP segment header consists of 4 fields, each 2 bytes long. The fields are the source port number, destination port number, length of the application data and the checksum value
- The checksum is the error checking mechanism in UDP.
- The checksum value is calculated at the sender side by dividing the UDP segment into 16 bit words, taking the sum of all the words (overflows are wrapped around) and then taking the 1s complement of the sum.
- The computed checksum value is placed in the checksum field of the UDP segment that is sent. At the receiver side, the checksum of the received UDP segment is compared with the value that was sent. If these 2 values are the same then there were no bit-level errors.
- Despite some network layer protocols like IP providing their own error checking, UDP provides the checksum facility so that packets can be transmitted across any links (even the ones that do not provide error checking) and to protect against errors occurring inside routers while in buffer storage.
- UDP provides error checking but not error recovery mechanisms. The erroneous segment can either be discarded or passed to the application layer above with a warning, depending on the implementation.

4 Principles of Reliable Data Transfer

- In this context, reliability means that there are no corrupted bits (0 to 1 or 1 to 0), no missing bits and that all data is delivered in the order that it was transmitted.
- The reliable data transfer protocol has the responsibility of converting what is essentially an unreliable network layer below it, into a reliable channel. TCP integrates this reliable data transfer as an additional service for applications in the layer above.
- At the sender side, the RDT protocol involves a function `rdt_send()` that passes the data that the receiving application is to receive. The `udt_send()` function passes on the data to the network layer below.
- At the receiver side, the function `rdt_recv()` function is . The `rdt_recv()` catches packets that are entering the system from the unreliable channel. The receiver calls the `deliver_data()` function when these packets are to be sent to the application layer.

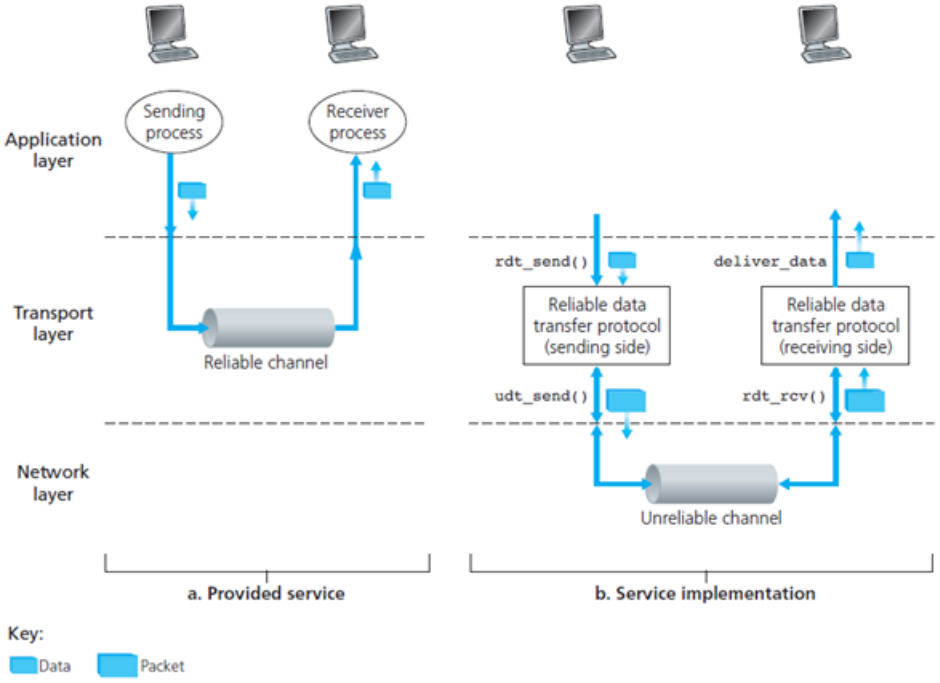


Figure 2: Reliable Data Transfer Protocol

4.1 RDT 1.0: Perfectly Reliable Channel

- The first version of RDT protocol assumes that the underlying layers are perfectly reliable and there are no errors possible.
- The notation for the FSMs shown below is as follows:
  1. The states are marked inside the circles
  2. The arrows indicate state transition
  3. Above the horizontal line is the input at the current state, and below the horizontal line is the action to be taken given the current state and the input.
- At the sender side, the sender waits for a call for `rdt_send(data)` from the application layer. It encapsulates the data in a segment using the `make_packet()` function and sends it to the channel using `udt_send()`.

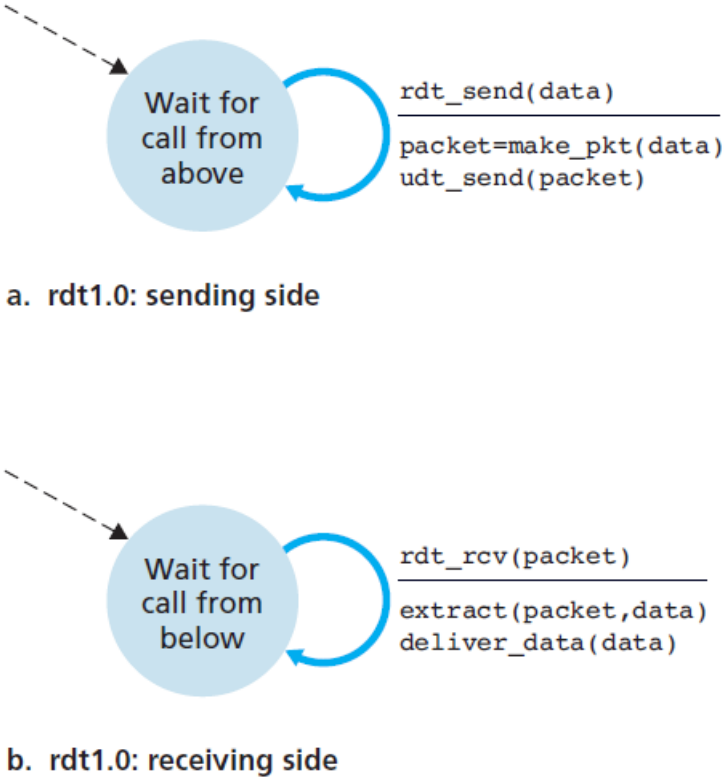


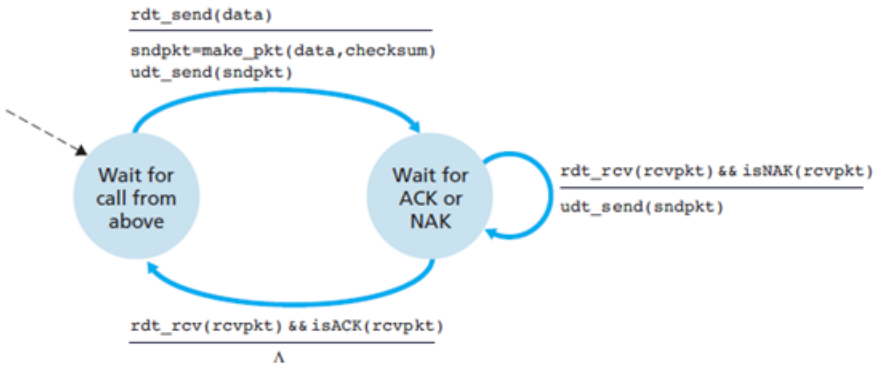
Figure 3: RDT Version 1.0

- At the receiver side, the receiver waits for a call for `rdt_recv(packet)`. The data is extracted from the packet and it is sent to the application layer above.
- It is assumed here that the sender and receiver are sending and receiving at the same rate, so the receiver does not have to ask the sender to slow down the packet rate.
- It is also assumed that the channel is perfectly reliable, with no bit-level errors or missing packets possible.

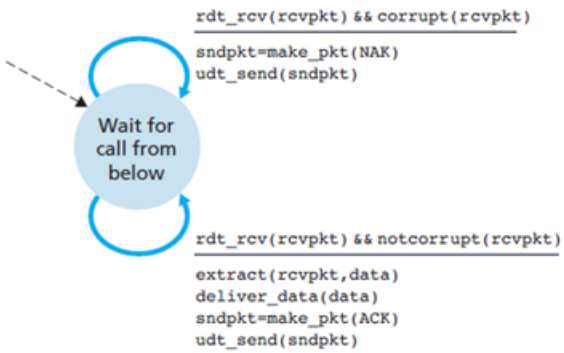
4.2 RDT 2.0: Channel with bit-level errors

- In case a packet has been received at the receiver end with bit level errors (these can be detected using error checking techniques like checksums), the receiver can ask the sender to repeat the packet (ie. send it again).
- Such protocols where this takes place are called **Automatic Repeat reQuest** or ARQ protocols.
- The extra components of rdt2.0 that make this possible are
  - **Error checking** which requires the sending of additional bits along with the data, that are aggregated in the checksum field of the rdt2.0 segment.
  - **Receiver feedback** on whether the packet was delivered properly or with bit level errors. The receiver sends a positive (**ACK**) or negative (**NAK**) acknowledgement to this end.
  - **Retransmission**, wherein a packet that is received with errors at the receiver side must be sent again by the sender.
- On the sender side, there are two states. In the leftmost state (the start state), the sender waits for the application layer to call `rdt_send(data)`. The data is made into a packet called `sendpkt` (which includes a *checksum*) and sent over the channel using `udt_send()`.

- Once the packet is sent, the sender waits for an ACK/NAK from the receiver side. The acknowledgement is received in the packet called `recvpkt`. If `recvpkt` is an ACK (+ve ack), then no action is taken and the sender once again waits for a call from the application layer.
- If `recvpkt` is a NAK (-ve ack), then the sender once again sends the `sendpkt` packet that it had sent before, and continues to wait for an ACK.
- At the receiver end, the receiver waits for a packet to arrive from the below layer.
- If the packet received (`recvpkt`) is corrupted, then a NAK packet is created and sent back to the sender.
- If the `recvpkt` is not corrupted, the data is extracted and delivered to the app layer above, and an ACK packet is sent back to the sender.
- Due to the behaviour of waiting for an ACK and allowing no other communications to take place in the meantime, RDT 2.0 is also called a **stop and wait** protocol.
- The flaw in RDT 2.0 is handling errors that might occur within the ACK and NAK packets themselves, which is handled in RDT 2.1.
- Methods of handling corrupt ACK/NAK are:
    1. Adding more checksum bits for ACK and NAK
    2. Repeating the request every time a garbled ACK/NAK is received. This can cause more delays with every garbled packet that is sent.
    3. Sender resends the packet every time it receives an ACK or NAK. This however introduces duplicate packets into the channel, and the receiver does not know the sequence in which to send ACK/NAK to the packets that it receives.



a. rdt2.0: sending side



b. rdt2.0: receiving side

Figure 4: RDT version 2.0

4.3 RDT 2.1: Handling garbled ACKs/NAKs

- RDT 2.1 adds the concept of a **sequence number** to each packet
- For a stop and wait protocol, a 1-bit sequence number is sufficient, 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 moves forward in modulo 2).
- If the received packet arrives with no errors, an ACK is sent back to the sender. If the received packet has errors, a NAK is sent to the sender.

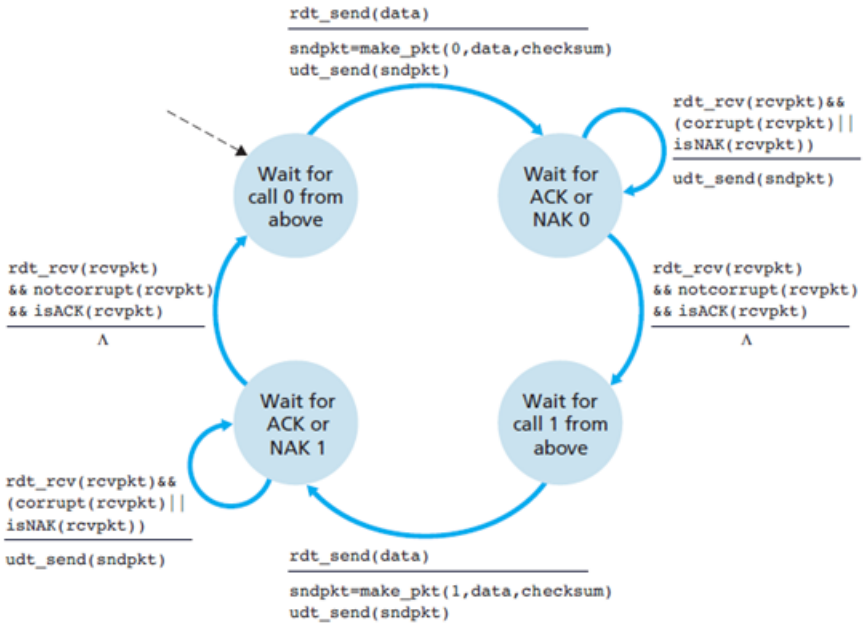


Figure 5: RDT version 2.1 Sender

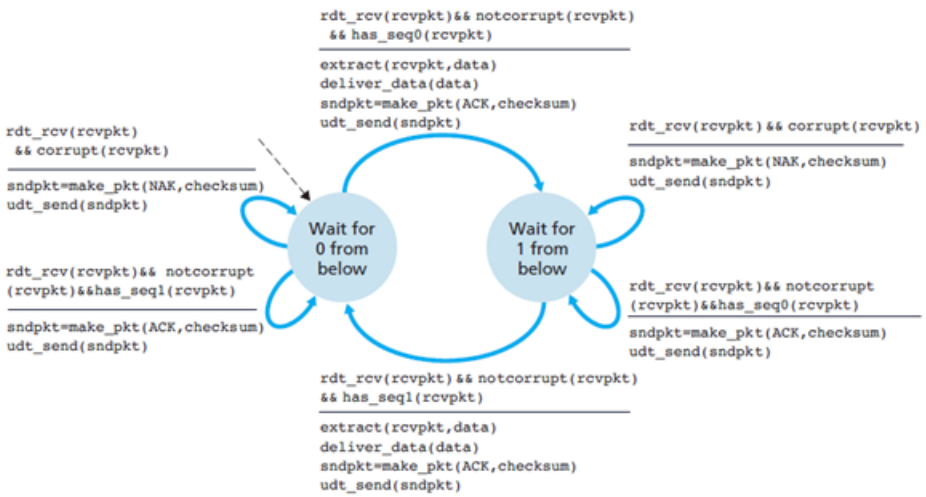


Figure 6: RDT version 2.1 Receiver

4.4 RDT 2.2: Enhanced version of 2.1

- RDT 2.2 offers the same functionality as RDT 2.1, but without the need for having any NAK packets being sent by the receiver at .
- This means that there is no need to implement the `isNAK()` functions to check packets coming in from the receiver to the sender.
- 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.
- The receiver must now include the sequence number of the packet being acknowledged by an ACK message.

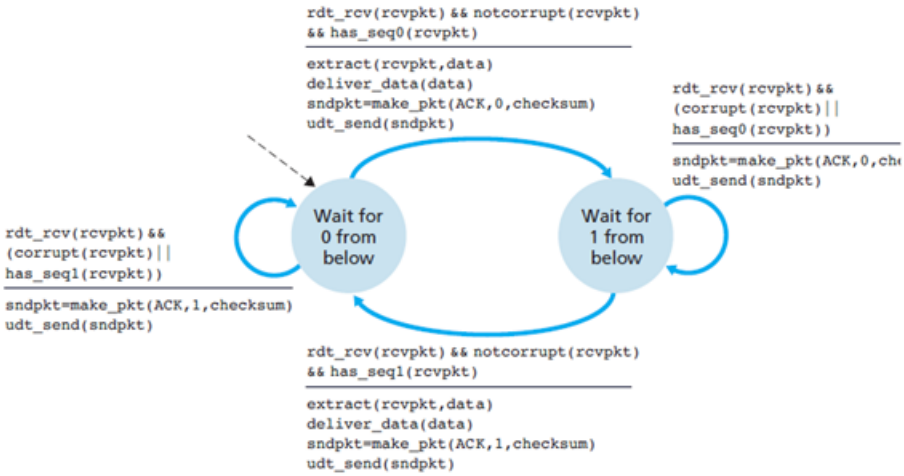


Figure 7: RDT version 2.2 Receiver

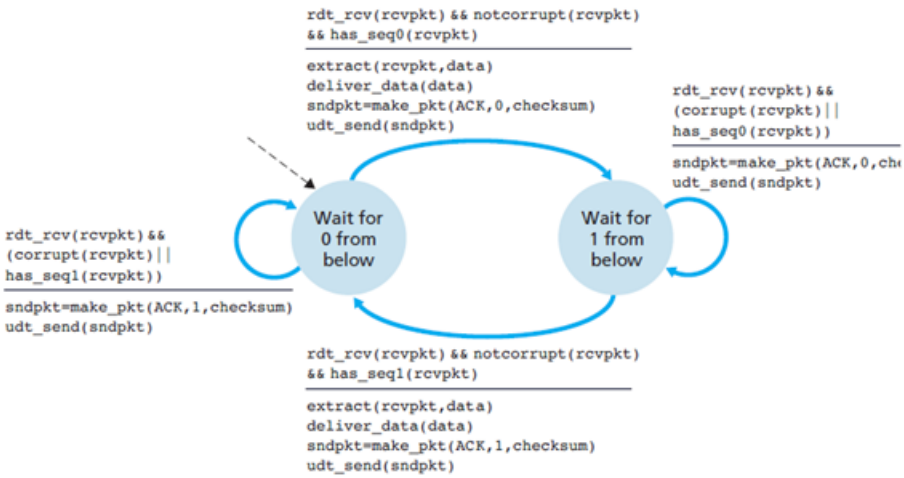


Figure 8: RDT version 2.2 Receiver

4.5 RDT 3.0: Handling missing packets

- The RDT version 3.0 handles dropped packets using the concept of a **timer**.

- The timer represents the *reasonable amount of time* that the sender waits for a positive acknowledgement from the receiver end. The sender is able to start or stop the timer as well as respond to interrupts that the timer generates.
- In RDT 3.0, retransmission happens if a packet/ACK message is garbled/lost.
- If a packet experiences a large delay (but not lost), then the sender will still retransmit as the timer will timeout on the sender side. Hence this leads to duplicate packets in the channel, which is already handled in past versions of RDT (using sequence numbers).

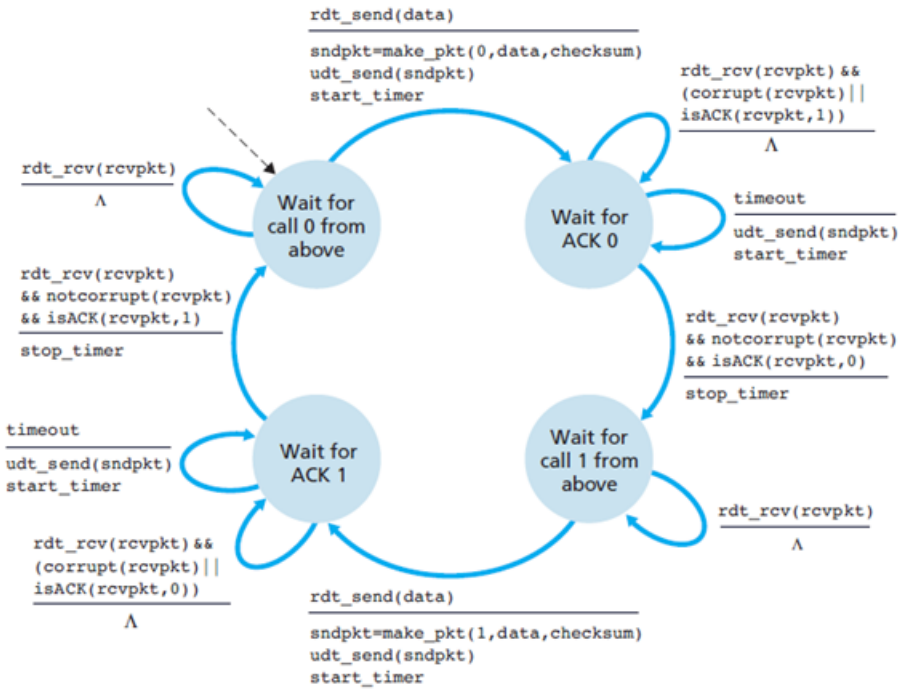


Figure 9: RDT version 3.0 Sender

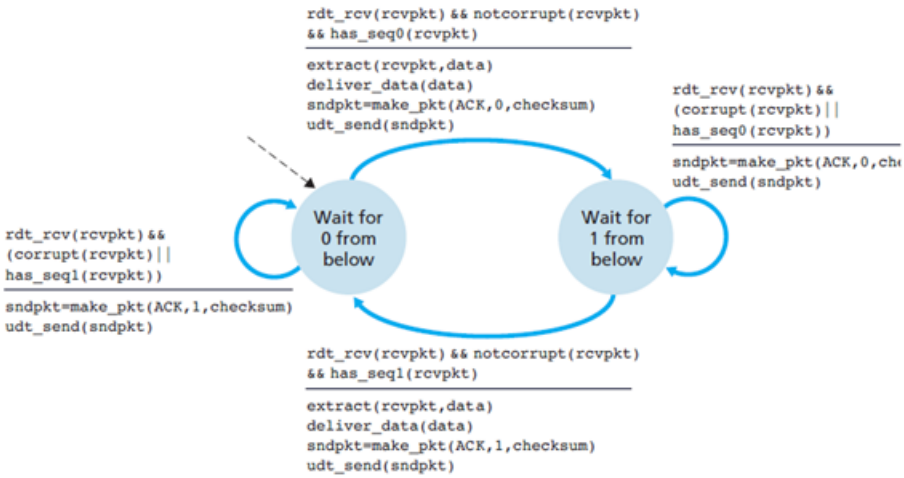


Figure 10: RDT version 3.0 Sender

5 Pipelined RDT protocols

- In a stop and wait protocol, performance is poor as the sender has to wait for the ACK of the current packet before the next packet can be transmitted. This leads to poor link utilization by the sender.
- In a pipelined protocol, the sender can send multiple packets at once without having to worry about the receipt of the ACKs for individual packet.
- In case one of the packets sent in the pipeline is garbled/lost, retransmission is handled in different methods by different protocols.
- Pipelining necessitates the following changes:
  1. Increased range for sequence numbers (instead of only 0 and 1), as there are multiple packets in transit at once.
  2. There must be buffers on both sender and receiver sides. 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 above 2 parameters vary from protocol to protocol. Two common protocols (Go Back N and Selective Repeat) are seen below

5.1 Go Back N (GBN)

- The buffer area on the sender side (also called the **sliding window**) consists of packets that have *been sent* but *not been ACKed yet*. The size of this window is upto N.
- If the acknowledgement for a packet is not received within the timeout period, then **all frames** within the current window are retransmitted.
- The sender window size is larger than the receiver window size in a GBN protocol.
- Reference: `https://youtu.be/QD3oCelHJ20`

5.2 Selective Repeat

- In a selective repeat protocol, only the packets that are dropped/garbled are retransmitted by the sender instead of the entire window at the current time.
- The sender window and reciever windows are of the same size in an SR protocol.
- SR avoids the performance problems related to GBN flooding the channel with retransmitted packets that might have reached properly but have to be retransmitted because of one dropped packet.
- Reference: `https://youtu.be/WfIhQ3o2xow`

6 Transmission Control Protocol (TCP)

- TCP is a **connection-oriented**, **full duplex**, **point-to-point** transport layer protocol that offers services such as congestion control, flow control and reliable data transfer in addition to the basic transport layer functionalities.
- TCP is called connection oriented as a connection must be established between the sender and receiver by means of a **3-way TCP handshake**.

6.1 TCP 3-Way Handshake

- The steps involved in setting up a TCP connection are:

  1. **SYN**: The client sends a segment to the server with a SYN (with an initial) that informs the server that a connection is to be started, and the starting segment number `client_isn` in the sequence number field of the TCP segment.
  2. **SYN-ACK**: The server allocates the connection variables and buffer space for the connection, and sends a connection-granted response to the client. The acknowledgement number of the header is set to `client_isn +1` and the sequence number is `server_isn`, which is the initial sequence number for all server-to-client communication.
  3. **ACK**: In the final part client acknowledges the response of server (by placing the value `server_isn+1` in the acknowledgement number field) and they both establish a reliable connection with which they will start the actual data transfer.

- The SYN and SYN-ACK steps establish the connection parameters for client to server communication (the sequence number), while the SYN-ACK and ACK steps establish conneciton parameters for the server to client direction.
- Thus, TCP establishes the full duplex communication.
- The largest size of the transport-layer segment is controlled by the **MSS** (Max Segment Size) parameter. The MSS is determined by checking the length of the largest link-layer frame that can be sent (the **MTU** or Max Transmission Unit), and then ensuring that the TCP segment + TCP/IP header size will fit in the single link-layer frame.

6.2 TCP Segment Structure

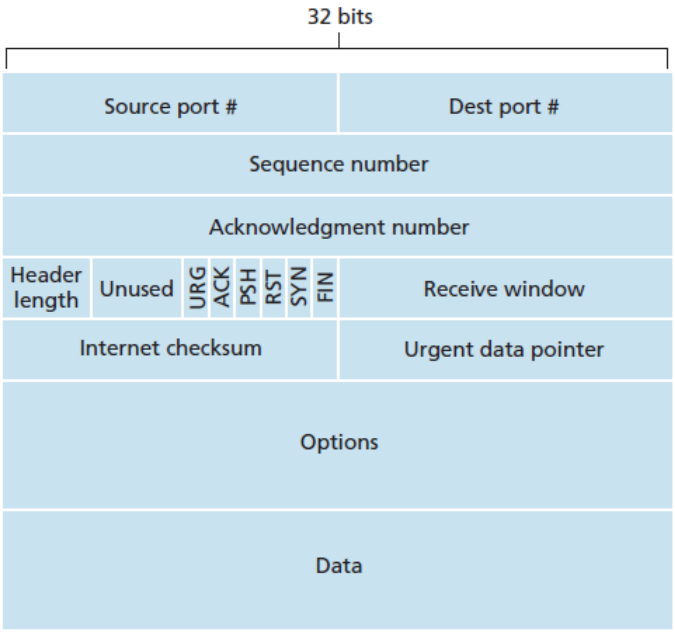


Figure 11: TCP Segment Structure

- 16-bit source and destination port numbers are used for multiplexing and demultiplexing at the end points
- 32-bit Sequence and Acknowledgement numbers are used to implement RDT.
- The 4-bit header length is the length in 32-bit words.

- 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.
- Theere are 6 flag bits: the **ACK** bit is set when the ack is sent for a successfully received segment. The **SYN**, **FIN** and **RST** are used for connection set up and teardown. Setting the **PSH** bit means that the transport layer must pass the data to the application layer immediately, and the **URG** bit is set when the app layer marks the data as urgent.
- 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 **urgent pointer** is used by TCP to inform the receiving- side upper-layer entity when urgent data exists and pass it a pointer to the end of the urgent data.

6.3 Timeout and RTT Estimation

- Deciding the timeout value when implementing RDT is a complex question. Timeout must be larger than the RTT of the network, but it cannot be too large as that introduces extra latency.
- Estimating the actual RTT value is also important. Here, the concept of an **exponentially weighted moving average** (EWMA) is used.
- The `SampleRTT` for a segment is the interval of time between the sending of the segment (ie. its tranfer to the network layer), and the receipt of the acknowledgement for that segment.
- At any point in time, the `SampleRTT` is measured for only one of the transmitted but unacknowledged segments. Also, `SampleRTT` is not measured for retransmitted segments, as this could lead to wrong RTT calculations in the case of duplicate acknowledgements.
- The estimate of the RTT is calculated as

$$EstimatedRTT_t = (1 - \alpha)EstimatedRTT_{t-1} + \alpha SampleRTT_t \tag{1}$$

  The recommmended value for α is 0.125 (RFC 6298).
- The deviation in the current `SampleRTT` from the `EstimatedRTT` is measured by the quantity `DevRTT`, which is an EWMA of $|SampleRTT - EstimatedRTT|$

$$DevRTT_t = (1 - \beta)DevRTT_{t-1} + \beta(|SampleRTT - EstimatedRTT|) \tag{2}$$

  The recommended value for β is 0.25 (RFC 6298).
- The timeout value is calculated from these quantities as

$$TimeOut = EstimatedRTT + (4 \times DevRTT) \tag{3}$$

- An initial timeout value of 1 second is recommended (RFC 6298). When a timeout occurs, the TimeOut value is doubled to ensure that premature timeout does not happen for the next segment. But as soon as the next segment is received then the TimeOut is computed using equation 3 above.

6.4 TCP Reliable Data Transfer

- In the RDT 3.0 protocol, each transmitted but unacknowledged packet has its own timer associated with it.
- To avoid timer management overheads, the TCP timer management standard (RFC 6298) specifies a single retransmission timer.
- The variable `sendBase` is the sequence number of the oldest acknowledged byte.
- Thus `sendBase`-1 is the sequence number of the last byte that is known to have been received correctly and in order at the receiver.

6.4.1 Doubling timeout interval

- Whenever a timeout event occurs, TCP retransmits the not-yet ACKed segment with the smallest sequence number. At the same time, TCP also doubles the value of the timeout interval.
- This action acts like a limited form of congestion control, as timeouts mainly happen due to network congestion, hence instead of retransmitting at a constant rate, TCP retransmits at a longer interval.

6.4.2 Fast Retransmit

- Whenever a gap occurs in the TCP stream (due to one packet being dropped while sending), the receiver sends duplicate ACKs for the last byte received for all the packets after that gap packet.
- When the sender receives 3 duplicate acknowledgements for the same data, then it takes this as an indication that the segment following the segment that has been ACKed three times has been lost.
- In this case, the sender retransmits the missing segment before that segment's timer expires. This is called a **fast retransmit**.

```
int next_seq_num = initial_seq_no;
int sendBase = initial_seq_no;
while(true){
    switch(event)
    {
        case Data_recv_from_appLayer:
        {
            Segment = create_new_segment;
            Segment.seqnumber = next_seq_num;
            if(timer not running)
                start timer;
            pass Segment to Network layer
            next_seq_num = next_seq_num + length(data);
            break;
        }
        case timer_timeout:
        {
            retransmit not-yet-ACKed segment with smallest Seq number;
            start timer;
            break;
        }
        case ACK_recv_with_ACKno_y:
        {
            if(y > sendBase)
            {
                sendBase = y;
                if(there are any unACKed segments)
                    start timer;
                break;
            }else{
                increment no of dup ACKs received for y
                if(no_dup_acks_for_y_recvd == 3)
                {
                    resend segment with seq no=y; /*TCP Fast Retransmit*/
                }
            }
        }
    } /*End of Switch*/
} /*End of while loop*/
```

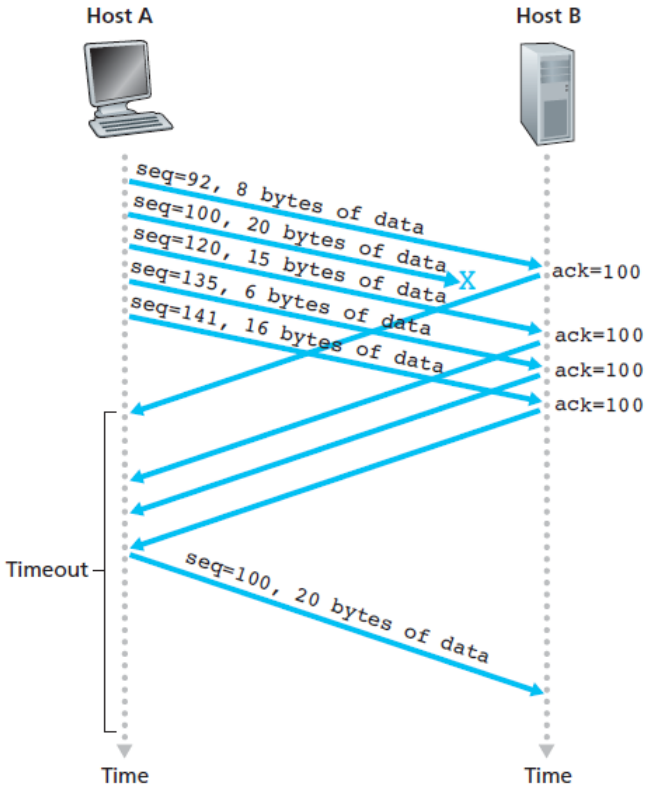


Figure 12: Fast Retransmit in TCP

6.5 TCP Flow Control

- The TCP sender maintains a state variable called the **receiver window**. Informally, this variable defines how much free buffer space is available at the receiver side.
- In the receiver, the process keeps track of the number of the last byte read, and the number of the last byte received in the buffer. If the buffer size on the receiver size is $recv\_buffer$ then we must have

$$last\_byte\_recvd - last\_byte\_read \leq recv\_buffer$$

- The receiver window size `rwnd` is calculated as

$$r\_Wnd = recv\_buffer - [last\_byte\_recvd - last\_byte\_read]$$

- The sender also keeps track of the numbers of the last byte acked and the last byte sent into the connection. To maintain flow control on the TCP connection, the sender maintains

$$last\_byte\_sent - last\_byte\_acked \leq recv\_wnd$$

- Consider the case that between hosts A and B, the recv buffer of B becomes full and advertises this to A. So A does not send any packet to B. Once free space appears in B buffer, but B does not have any packets to send to A, hence A is unaware of the newly freed space in B.
- In this case above, A is required by the TCP Specification to keep sending 1 byte data to B, and eventually an ACK packet is received by A which indicates that B has non-zero buffer space.

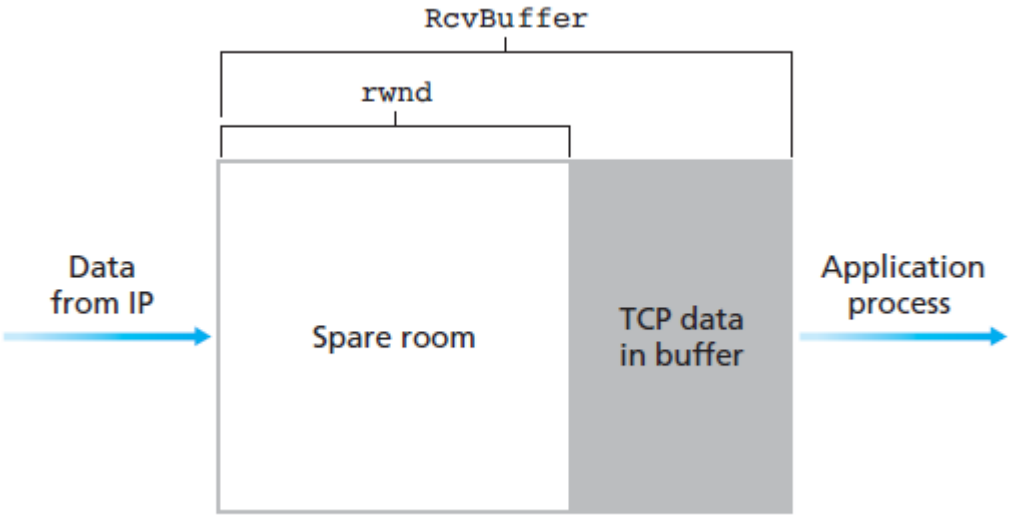


Figure 13: Receiver side Buffer

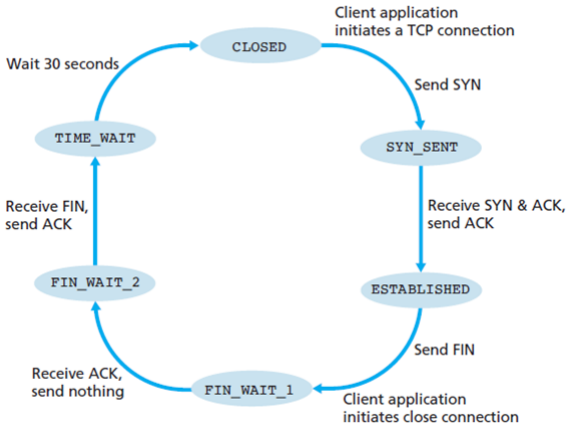


Figure 14: Client TCP States

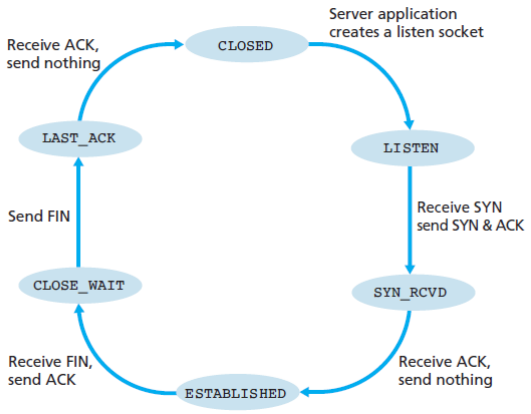


Figure 15: Server TCP States

7 Congestion Control

7.1 Costs of congestion in network

- Queueing delays become large as the arrival rate into a router approaches the link rate.
- The sender must perform retransmissions in order to compensate for dropped (lost) packets due to buffer overflow.
- Unneeded retransmissions by the sender in the face of large delays (leading to timeout) may cause a router to use its link bandwidth to forward unneeded copies of a packet.
- When a packet is dropped along a path, the transmission capacity that was used at each of the upstream links to forward that packet to the point at which it is dropped ends up having been wasted.

7.2 Approaches to Congestion Control

- **End-to-end**:
  - The transport layer takes full responsibility of congestion control, and the network layer provides no feedback regarding the same.
  - The transport layer infers from events (like dropping of packets) that congestion is present in the network.
  - This is the approach primarily taken by TCP. TCP uses timeout or triple duplicate ACKs as a signal of congestion.
- **Network-assisted**:
  - Network-layer components (routers) provide explicit feedback to the sender regarding the congestion state in the network.
  - Simplest approaches indicate one single hardware bit indicating N/w congestion or not, or the sending of a **choke packet** from the router to the sender indicating congestion.
  - More complex mechanisms for congestion control like ATM ABR allow a router to inform the sender explicitly of the transmission rate it (the router) can support on an outgoing link.

7.3 TCP Congestion Control

- The sender maintains an additional TCP variable to enforce congestion control, called the **congestion window** (`cwnd`). The cwnd value is a constraint on the aount of data that can be sent. We must always have:

$$last\_byte\_sent - last\_byte\_acked \leq min(recv\_wnd, cwnd)$$

- The sender rate can be roughly calculated as $\frac{cwnd}{RTT}$, as at the beginning of an RTT, the sender can send cwnd bytes of information. Increasing `cwnd` increases sender rate.
- A *loss event* is defined as either a **timeout** or the receipt of **triple duplicate ACKs** (the fast retransmit case).
- The value of `cwnd` is decided by the following general guidelines:
  - Lost segment implies connection, hence `cwnd` should reduce.
  - ACK received by sender implies no congestion, hence hence `cwnd` can be increased.
  - *Bandwidth probing*: The practice of slowly increasing the sender rate until the point that congestion occurs, then slowly reducing from that point. After backing off, the sender once again probes to see if the network's congestion has changed or not.

7.3.1 Slow Start

- The inital value of `cwnd` is set to $1 \times MSS$. Every time a transmitted segment is acknowledged, the `cwnd` increases by 1 MSS.
- First one MSS is sent, once it is ACKed, the cwnd becomes 2MSS and 2 MSS packets are sent out. For each positive ACK, the cwnd increases by one MSS, hence from 2MSS the cwnd becomes 4MSS.
- Hence, in every RTT, the `cwnd` becomes double (exponential growth).
- If a timeout occurs, the cwnd is set to 1 MSS and slow start process is repeated again. At the first timeout, the `ssthresh` (slow start threshold) is set to `cwnd/2` (half of the value of cwnd at which the loss event occurred).
- Subsequently, if `cwnd` exceeds `ssthresh` then TCP transitions to the *congestion avoidance* state.
- If 3 duplicate ACKs are received, then TCP performs a fast retransmit and enters the *fast recovery* state.

7.3.2 Congestion Avoidance

- In this state, the cwnd value increases linearly rather than exponentially over a single RTT.
- When a timeout occurs in congestion avoidance mode, the `ssthresh` is set to `cwnd/2`, and the `cwnd` is updated to 1 MSS.
- When 3 duplicate ACKs are received in congestion avoidance mode, TCP enters the fast recovery mode. The `ssthresh` is set to half the cwnd value when 3 dup ACKs were received, and then the cwnd is updated to half that value with 3 MSS added ($cwnd = cwnd/2 + 3MSS$.

7.3.3 Fast Recovery

- This is an optional component of TCP congestion control mechanism.
- In fast recovery state, the cwnd is increased by 1 MSS for every duplicate ACK received. When the missing ACK arrives, the cwnd is set back to ssthresh, and it enters congestion avoidance mode.
- In the fast recovery state, if a timeout event occurs, the ssthresh is set to cwnd/2 cwnd size is set to 1, and TCP enters Slow Start.
- In the recent version of TCP (called **TCP Reno**) fast recovery is used.
- In the older version **TCP Tahoe**, the ssthresh is set to cwnd/2 and cwnd is cut to 1 MSS irrespective of what loss event happened.

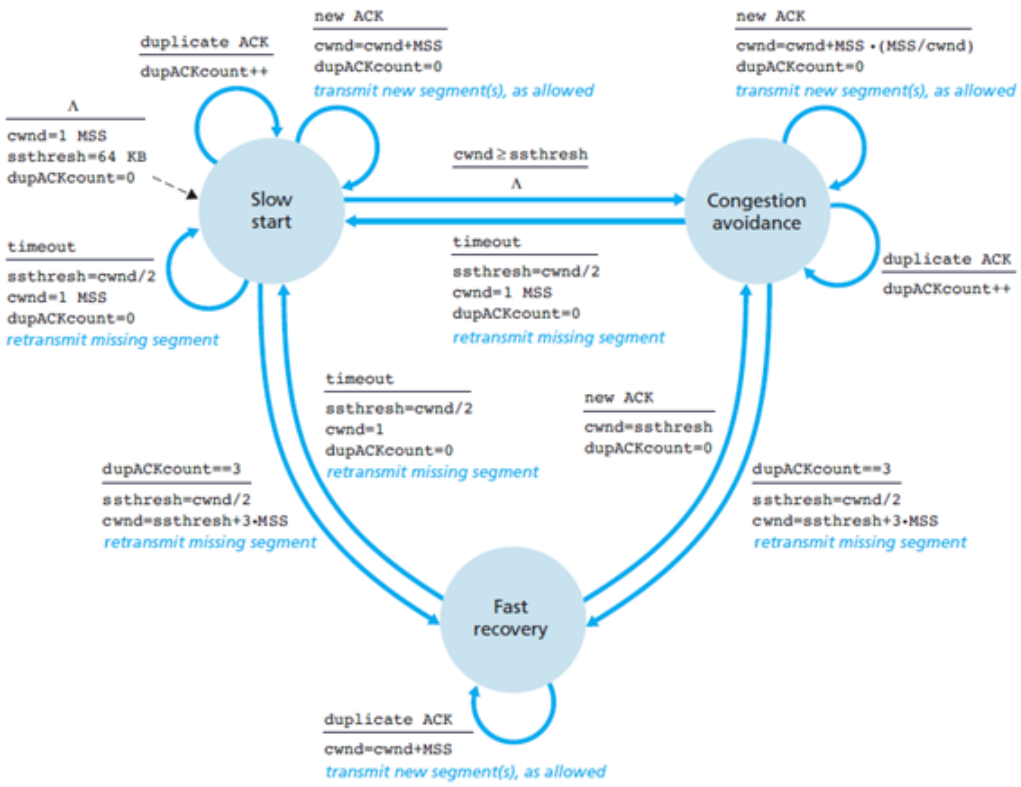


Figure 16: TCP Congestion Control