Transport layer

Multiplex and demultiplex between applications and the network. The network layer provides "host-to-host" connection, and we want to provide "process-to-process" connection at the transport layer. We may provide: connection control, flow control, congestion control, reliable transmission, in-order delivery, etc.

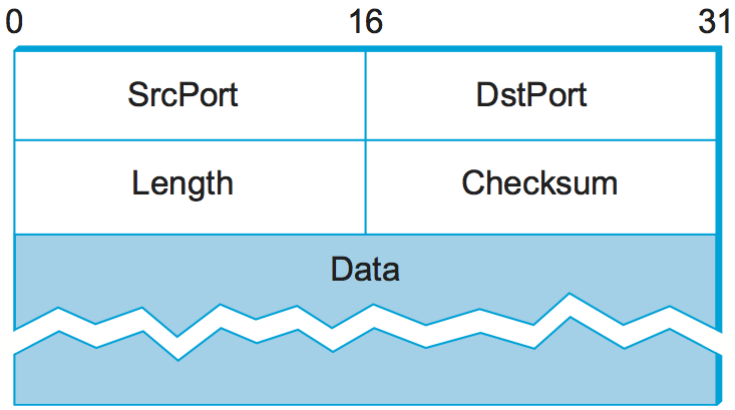
Difference between flow control and congestion control: flow control is to avoid overloading the **receiver**, while congestion control is to avoid overloading the **network**.

Port, a 16-bit identifier designed to sending and receiving applications, is an abstraction that enables multiplexing and demultiplexing.

User Datagram Protocol (UDP)

Connectionles, unreliable, unordred delivery. UDP provides simple (de)multiplexing functionality using port. Used by applications like live streaming, etc.

UDP header:



The application is unambiguously identified by `<IP address, port>` pair.

Transmission Reliability

Acknowledgement-based reliability. Timeout mechanism needed. Setting Retransmit TimeOut (RTO) is important. If RTO is too long, you lose performance; If RTO is too short, you resend data and the receiver needs to handle duplicate.

Sending one packet and waiting for ACK/timeout before sending the next, known as *stop-and-wait*, under-utilizes the bandwidth. For better performance, we send multiple packets before waiting. We add a sequence number to each packet to handle out-of-order or lost packets. The receiver uses a buffer to store when some but not all packets have arrived. When the buffer is full, the receiver flushes packets to the application.

The buffer has limited capacity, so the sender shouldn't send too many packets beyond the capacity. The *flow control* mechanism allows the receiver to tell the sender its buffer size, Receive Window Size (RWS). The sender also has a buffer, with size Send Window Size (SWS).

The mechanism of using buffers is called *sliding window*, it allows: better bandwidth utilization, reliable transmission, in-order delivery, and flow control.

Sliding Window

An acknowledgment (ACK) is a control frame, meaning that it's header without payload, although a protocol can *piggyback* an ACK on a data frame it just happens to be sending back. If the sender doesn't receive an ACK after the RTO, it *retransmits* the orignal frame.

The sender maintains 3 variables. Send Window Size, `SWS`. Sequence number of Last Acknowledgement Received, `LAR`. Sequence number of Last Frame Sent, `LFS`. Invariant: `SWS >= LFS - LAR`. When receiving an ACK, the sender possibly updates `LAR`. The sender associates a timer with **each frame** and retransmits at timeout.

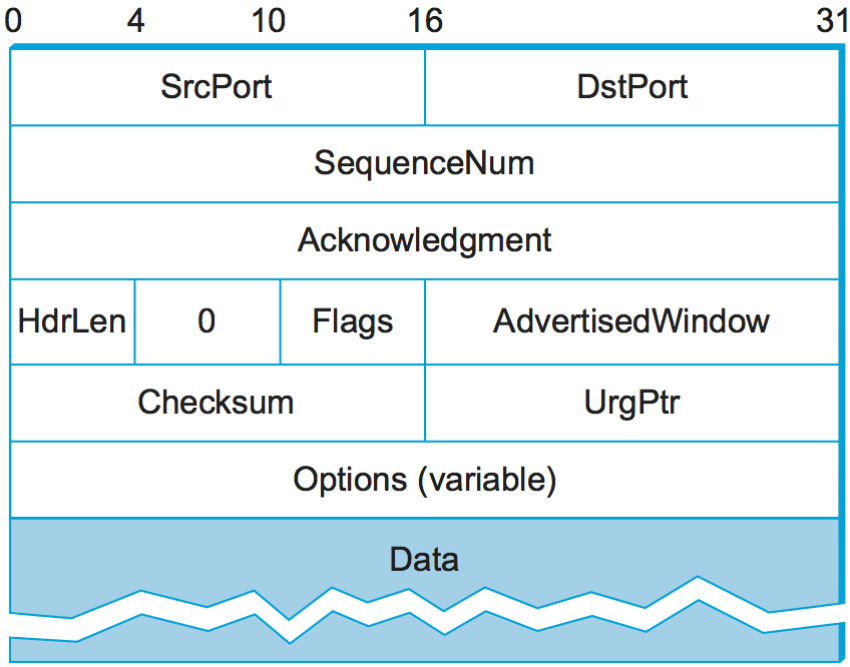
The receiver maintains 3 variables. Receive Window Size, `RWS`. Sequence number of Last Acceptable Frame, `LAF`. Sequence number of Last Frame Received, `LFR`. Invariant: `RWS >= LAF - LFR`. When receiving a frame with `SeqNum`, do the following:

```
if SeqNum <= LFR OR SeqNum > LAF:
    discard // outside the receiver's window
else:
    // LFR < SeqNum <= LAF, within receiver's window
    // Let SeqNumToAck be the max sequence number not yet ACKed,
    // such that all frames <= SeqNumToAck has been received.
    Send ACK of SeqNumToAck
    LFR = SeqNumToAck
    LAF = LFR + RWS
```

For example, suppose `LFR = 5` (i.e., the last ACK the receiver sent was for sequence number 5), and `RWS = 4`. This implies that `LAF = 9`. Suppose `SeqNumToAck = 5`. Should frames 7 and 8 arrive, they will be buffered because they are within the receiver's window. However, the receiver still sends ACK on **5**, since Frame 6 hasn't arrived. Frames 7 and 8 are said to have arrived out of order. Should frame 6 then arrives, `SeqNumToAck` is now `8`. the receiver acknowledges frame 8, bumps `LFR` to 8, and sets `LAF` to 12.

Transmission Control Protocol (TCP)

Connection-oriented, reliable-transmission, full duplex (bi-directional conversation), flow control, congestion control, byte-oriented. Header format:



- Byte oriented

  `SequenceNum`: sequence number of the 1st byte of data carried in the segment. `Acknowledgement`: sequence number of the **next** byte of data expected. `AdvertisedWindow`: sent in **every ACK** packet. The sender can have no more than this amount of unACKed data at any time. `Flags` canbe SYN, FIN, ACK, etc. A packet can be of multiple types. This is the flow control mechanism.

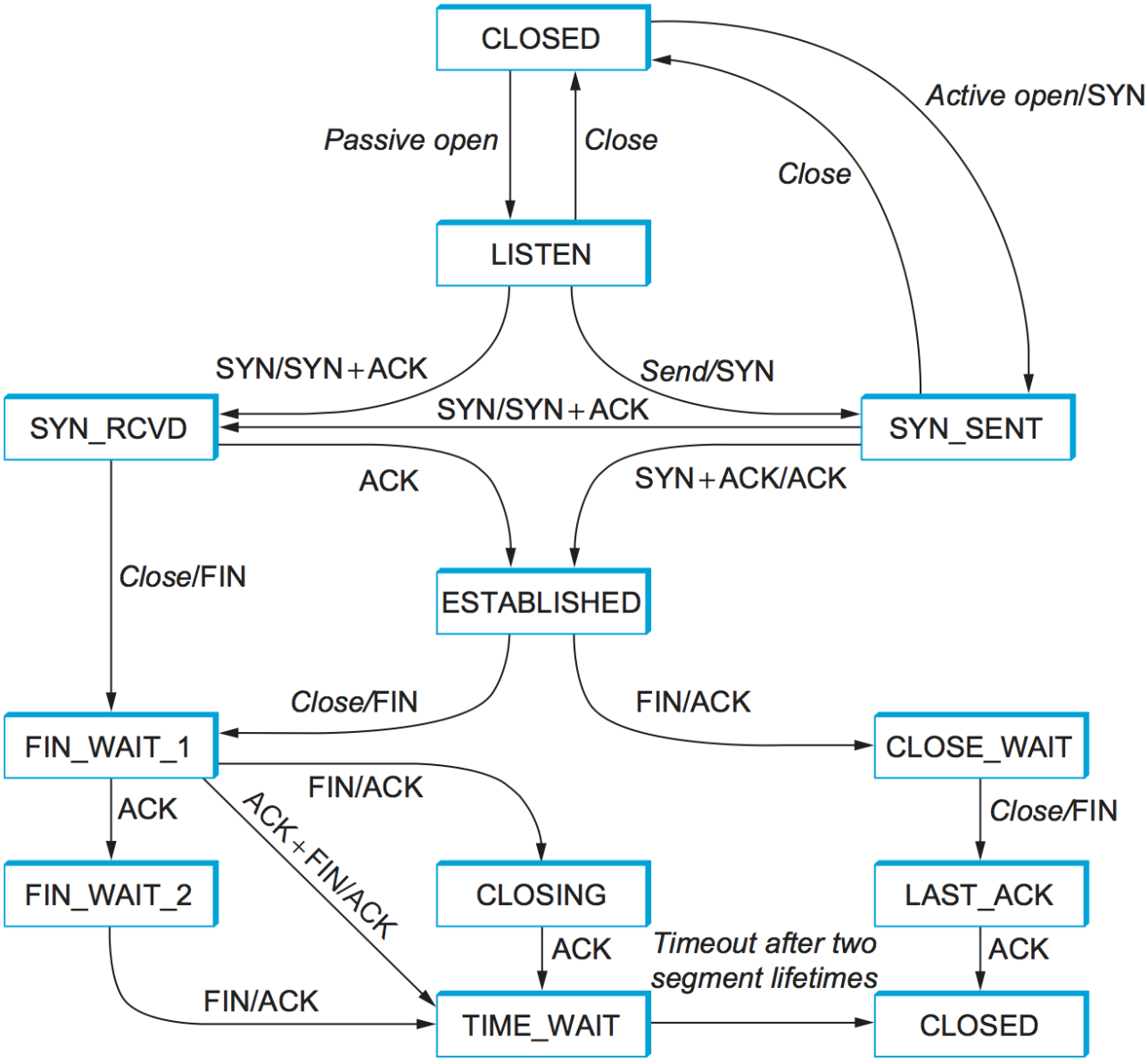
  Accumulative ACK: The sender sends 3 packets with bytes 0-535, 536-800, 801-999, and 536-800 gets lost. The 1st ACK is 536, and the 2nd ACK is also 536! Suppose after RTO the sender resends the packet 536-800 and is received, the 3rd ACK would be 1000.

- Sliding window with Control Flow

  TCP sliding window works just like mentioned above, with flow control.

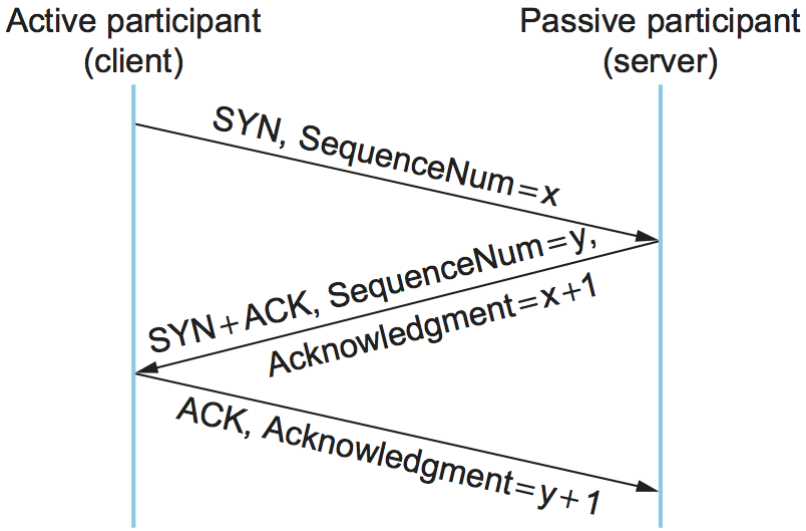
- Connection management

  Connection establishment is asymmetric (client does an active open and the server does a passive open), connection teardown is symmetric (each side has to close the connection independently). After one side closes the connection, it cannot send data to the other side, but it can still receive data from the other side.



- Connection establishment

  The idea is for sender and receiver to agree on the sequence numberes.



  x is the starting SequenceNum that client would send to server.

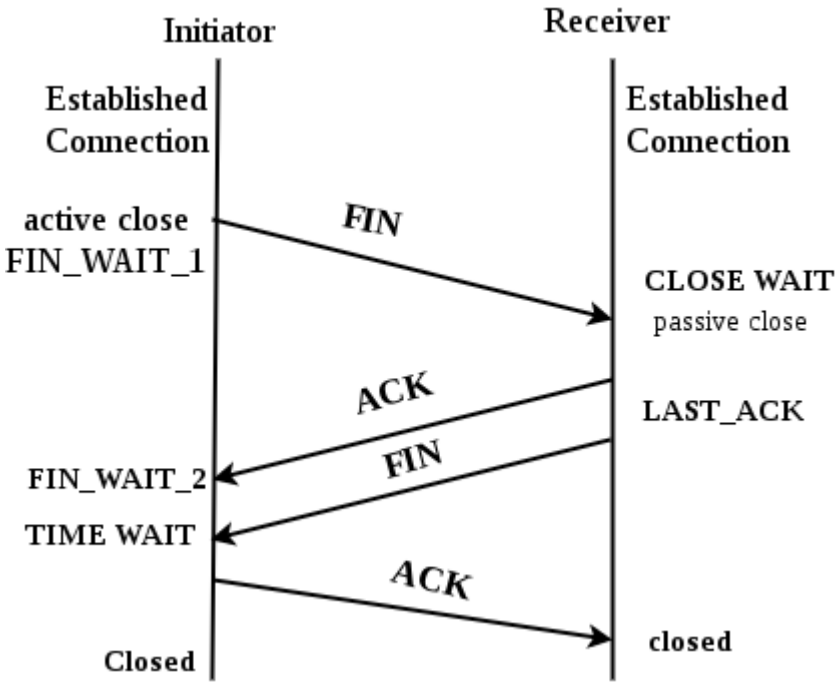
  y is the starting SequenceNum that server would send to client.

The client could send data on the ACK it sends back to the server (called *piggyback*), because its connection to the server is open when SYN+ACK is received. But only when the server receives ACK, the server side connection is open.

During three-way handshake to establish the connection, if the client's ACK to the server is lost, the connection still functions correctly. The reason is that once receiving ACK from server, the client is in ESTABLISHED state and the local application can send data to the server. Each of the **data segments** would have `ACK` flag set and the correct value in `Acknowledgement` field, indicating the next sequence number it's expecting. Thus, when the server receives the first data segment, it also moves to ESTABLISHED state. (RFC specifies that once a connection is established, the `ACK` flag is alwasy set. Reference.)

- Connection teardown

  Connection is closed independently on each side.



A slightly different situation is possible. When the server receives the FIN from client, it also chooses to close the server-side connection. So it can send back a FIN+ACK packet, and in this way only 3 packets are exchanged.

A connection in the TIME_WAIT state cannot move to the CLOSED state until it has waited for two times the maximum amount of time an IP datagram might live in the Internet (i.e., 120 seconds). The reason for this is that, while the local side of the connection has sent an ACK in response to the other side's FIN segment, it does not know that the ACK was successfully delivered. As a consequence, the other side might retransmit its FIN segment, and this second FIN segment might be delayed in the network. If the connection were allowed to move directly to the CLOSED state, then another pair of application processes might come along and open the same connection (i.e., use the same pair of port numbers), and the delayed FIN segment would immediately initiate the termination of the later connection.

More efficient TCP

- Delayed ACKs.

  Instead of sending an ACK for every data packet received, send one ACK for all packets received within the time period.

- Timeout calculation

  The timeout should adapt to network conditions, like RTTs.

  - Exponentially Weighted Moving Average (EWMA)

    Simple means for estimating network/RTT conditions. Assumes that we measure RTT for every data/ACK pair (sample RTT). Whenever a data/ack pair transmission completes without exceeding RTO, we get a sample. The formula is: $$ EstimatedRTT = \alpha EstimatedRTT + (1 - \alpha) SampleRTT $$ a typical value for α is 0.9. And $$ RTO = 2 \times EstimatedRTT $$

  - Karn-Patridge Algorithm

    The problem is that ACK acknowledges *receipt*, instead of *transmission*. Whenever a packet is retransmitted and an ACK arrives at the sender, we can't tell whether it should be associated with which transmission.

    The solution is to meansure `SampleRTT` only for segments that're sent exactly once. Also, if TCP retransmits, set `RTO` to be twice as original `RTO`, instead of twice of `EstRTT`, **for that packet**. In short:

    - Do not sample RTT for data packets that have been resent due to time out.
    - Exponentially increase RTO value for **resent** packet.

  - Jacobson's algorithm

    The problem with the original computation is that it does not consider **variance**. If the variation among samples is small, then the `EstimatedRTT` can be better trusted and there is no reason for `RTO = 2 * EstimatedRTT`. On the other hand, a large variance in the samples suggests that the timeout value should differ much from `EstimatedRTT`.

    In the new approach, the sender measures a new `SampleRTT` as before, but folds this new sample into `RTO` differently:

    ```
    Difference = SampleRTT - EstimatedRTT
    EstimatedRTT = EstimatedRTT + (delta x Difference), where 0 < delta < 1
    Deviation = Deviation + delta (|Difference| - Deviation)
    Timeout = mu * EstRTT + phi * Deviation, where mu = 1 and phi = 4
    ```

    With this, when variance is small, `Timeout` is close to `EstimiatedRTT`; when variance is large, `Deviation` dominates the computation of `Timeout`.

TCP Congestion Control

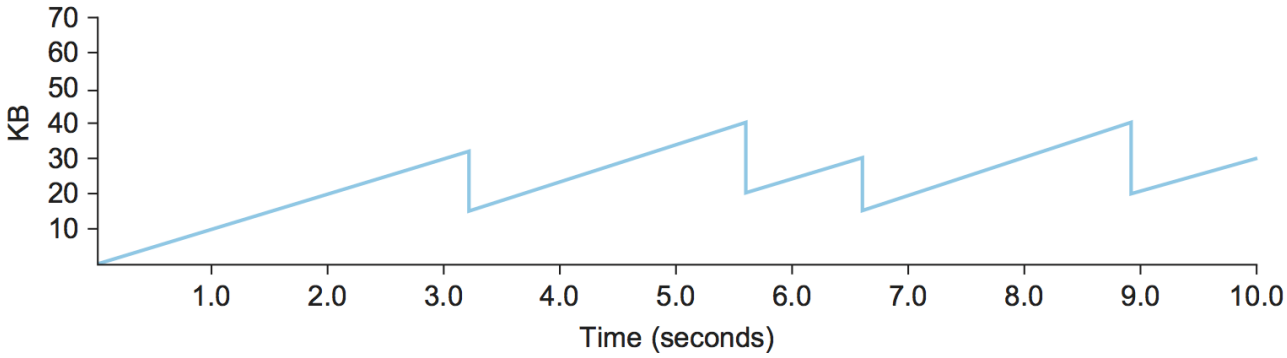
- Additive Increase/Multiplicative Decrease (AIMD) - **Congestion Avoidance**

TCP maintains a new variable for each connection, `CongestionWindow`, limiting how much data is allowed to be in transit at a given time. It's the congestion control's couterpart of flow controls' `AdvertisedWindow`. Now, TCP's max unacknowledged data = `min(CongestionWindow, AdvertisedWindow)`. The sender needs to learn this by itself.

The size of a single packet is called Maximum Segment Size (MSS).

Multiplicative Decrease: The main reason packets aren't acknowledged before timeout is that a packet is dropped due to congestion. TCP interprets timeouts as a sign of congestion and halves `CongestionWindow` for each timeout.

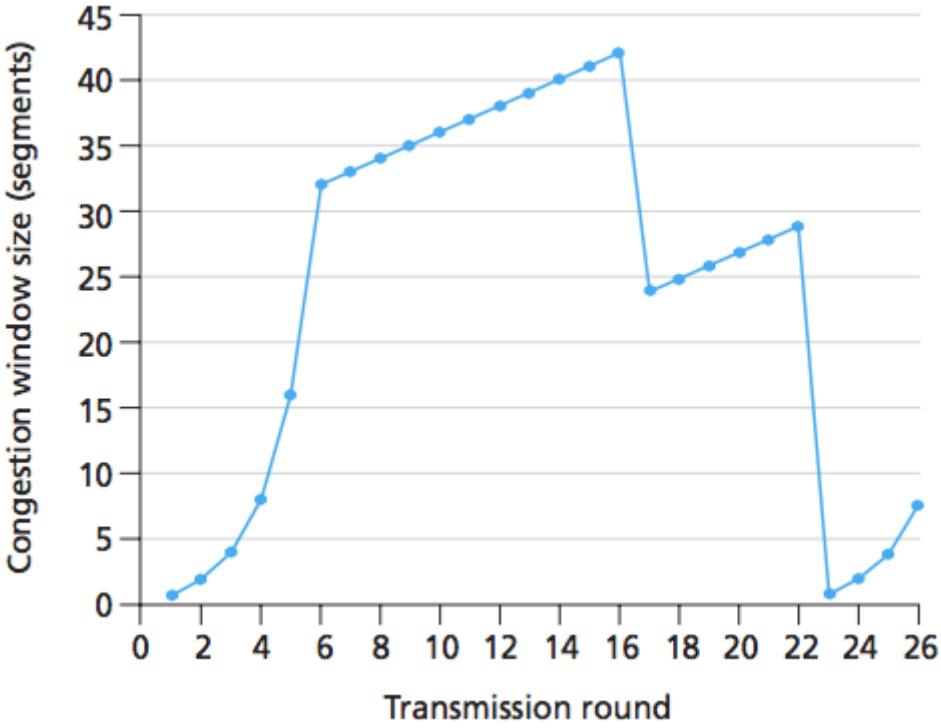
Additive Increase: Every time the sender successfully sends a `CongestionWindow`'s worth of packets, it increments `CongestionWindow` by one MSS.



- **Slow Start**

AIMD increases `CongestionWindow` linearly (too slow), while slow start increases it exponentially. When the `CongestionWindow` is relatively large, slow start is too aggressive. So we have `SlowStartThreshold`, SST. When `CongestionWindow` exceeds SST, we use AI; when `CongestionWindow` is below SST, we use slow start.

With slow start, congestion control acts differently. The sender starts out with `CongestionWindow` = 1, SST = infinity, and does slow start. (1) When a timeout fires, set SST = `0.5 * CongestionWindowSize`, `CongestionWindow` = 1, and restart slow start. (2) When `CongestionWindow` exceeds SST, converts from slow start to AI.



Note that in the picture above, the first decrease in `CongestionWindow` is due to duplicate ACKs, while the second decrease is due to timeout.

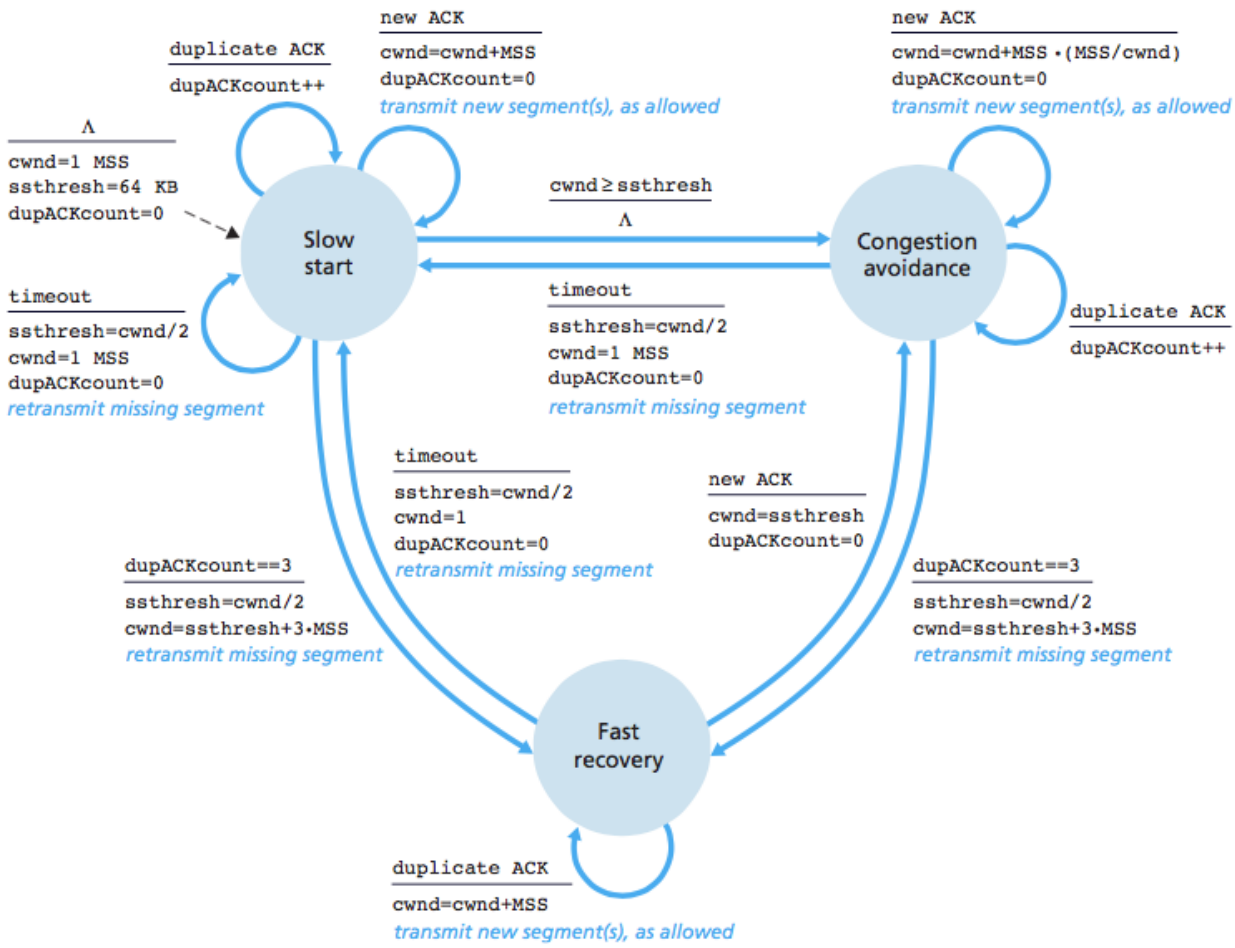
- Fast Retransmit and Fast Recovery

  In practice, it's hard to keep a timer on each packet. So a timer is set for a group of packets. However, waiting for timeout to fire before retransmitting can be slow, fast retransmit triggers the retransmission of a lost packet sooner than regular timeout.

  Other than timeout, duplicate ACKs are also a sign of congestion. Every time a data packet arrives at the receiving side, the receiver responds with an ACK, even if this sequence number has already been ACKed. Thus, when a packet arrives out of order, the receiver sends back an ACK for a packet that has been ACKed before, called *duplicate ACK*. When the sender sees a duplicate ACK, it knows that the other side must have received a packet out of order. This can be because an earlier packet was lost, **or** the earlier packet has been delayed. As we have two possibilities, sender waits until it sees some number of duplicate ACKs and then retransmits the missing packet. In practice, TCP waits for 3 duplicate ACKs before retransmitting. For example, suppose three duplicate ACKs for packet P is received, the sender immediately retransmits packet P+1 only. This is called ***Fast Retransmit***.

  Additionally, compared with timeout, duplicate ACKs are a mild sign of congestion, so we take mild actions. Originally when doing slow start, if encoutering timeout, we set SST = `0.5 * CongestionWindow` and set `CongestionWindow` = 1. When encountering 3 duplicate ACKs, we set SST = `0.5 * CongestionWindow`, but setting `CongestionWindow = SST + 3 * MSS`. This behavior is called ***Fast Recovery***.

- Summary



Notes about the state diagram:

- When timeout happens, convert from Fast Recovery/Congestion Avoidance to slow start, setting SST = `CongestionWindow/2`, `CongestionWindow` = 1;
- When receiving 3 duplicate ACKs, convert to Fast Recovery. Fast Recovery is a short state. If receiving new ACK, then convert to Congesion Avoidance.

TCP Tahoe: slow start + congestion avoidance.

TCP Reno: slow start + congestion avoidance + fast retransmit & fast recovery.

Advanced Congestion Control

Skipped for now. Doesn't seem like a poplular interview topic.