NOTES

Chapter 25
TCP

Transport Level Protocols

- Transport Level Protocols provide process-to-process communication
- We will look at TCP/IP's two most popular transport level protocols:
  - UDP
  - TCP
- both UDP and TCP are layered on top of IP

Application	(e.g., Telnet)
Transport	UDP or TCP
Intenet	IP
Network Interface	(e.g., Ethernet)
Hardware	(e.g., Ethernet)

TCP

- The Transmission Control Protocol (TCP) is the most widely used transport protocol
- Provides reliable data delivery by using IP unreliable datagram delivery
- Compensates for loss, delay, duplication and similar problems in Internet components
- Reliable delivery is high-level, familiar model for construction of applications

NOTES

Features of TCP

- Connection oriented: Application requests connection to destination and then uses connection to deliver data
- Stream interface: Application delivers data to TCP as a continuous stream, with no record boundaries;
- Full duplex: The endpoints of a TCP connection can exchange data in both directions simultaneously
- Reliability: TCP guarantees data will be delivered without loss, duplication or transmission errors
- Reliable connection startup: Three-way handshake guarantees reliable, synchronized startup between endpoints
- Graceful connection shutdown: TCP guarantees delivery of all data after endpoint shutdown by application
- Flow control: keep sender from overrunning the receiver
- Congestion Avoidance: keep sender from overrunning the network

Communication Model

- Connection-oriented
- Byte-stream
  - sending process writes some number of bytes
  - TCP breaks into *segments* and sends via IP
  - receiving process reads some number of bytes
- Full duplex

NOTES

TCP Segments and Sequence Numbers

- Application delivers arbitrarily large chunks of data to TCP as a“stream”
- TCP breaks this data into segments, each of which fits into an IP datagram
- Original stream is numbered by bytes
- Segment contains sequence number of data bytes

Acknowledgments

- Receiver sends segment with sequence number of acknowledged data (not segments)
- One ACK can acknowledge many segments

NOTES

TCP and Reliable Delivery

- TCP uses many techniques described earlier to provide reliable delivery
- Recovers from

Sliding Window Revisited

- TCP uses a sliding window-based protocol to ensure reliable delivery and provide flow control
- However, TCP's SWP provides end-to-end delivery over an internet which is a very different situation than SWP at the data link level over a LAN.

NOTES

Sliding Window Revisited

- Each byte has a sequence number (as opposed to packet sequence numbers)
- ACKs are cumulative – they ACK the highest numbered byte received so far

Sliding Window Revisited (2)

- Problem : cannot assume receiving process has read (consumed) the data. If the receiver does not take "ACKed" data out of the buffer, the buffer space will fill up and there will be no more space. Even though the destination received (and ACKed) every byte sent so far, the destination's window size (i.e., the amount of room it has for new data) will go to ZERO!
- Sending side
  - bytes between `LastByteAcked` and `LastByteWritten` must be buffered
- Receiving side
  - bytes between `LastByteRead` and `LastByteRcvd` must be buffered

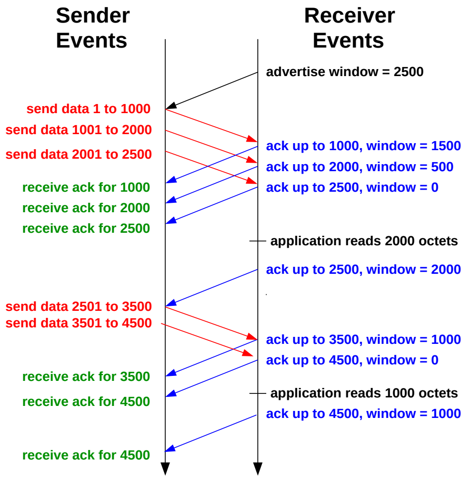
NOTES

TCP Flow Control

- Uses SWP with an Advertised Window Size
- Sender buffer size: MaxSendBuffer
- Receive buffer size: MaxRcvBuffer
- Receiving side must advertise the available space
  - AdvertisedWindow = MaxRcvBuffer - (LastByteRcvd - NextByteRead)
- Sending side must adhere to advertised window size!
  - EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)
  - block sender if (LastByteWritten - LastByteAcked) + (number bytes sender wants to write) > MaxSendBuffer
- Always send ACK in response to an arriving data segment
- How does the sender get started again?
  - Persist (send 1 byte) as long as AdvertisedWindow = 0

Flow Control Example

NOTES



Silly Window Syndrome

- Under some circumstances, sliding window can result in transmission of many small segments
- If receiver window full, and receiving application consumes a few data bytes, receiver will advertise small window
- Sender will immediately send small segment to fill window
- Inefficient in processing time and network bandwidth
- Solutions:
  - Receiver delays advertising new window
  - Sender delays sending data when window is small

NOTES

TCP Packet Format

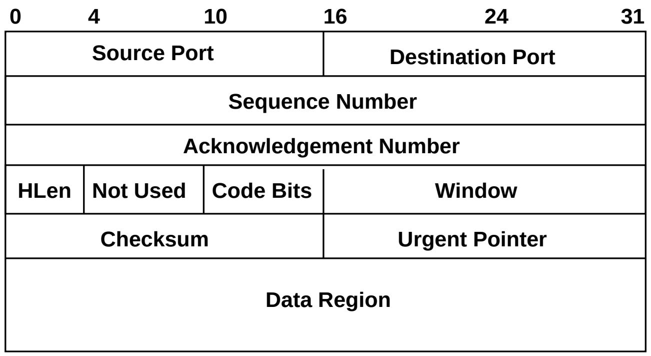


Figure 1: TCP Packet Format

Setting the Timeout

- Inappropriate timeout can cause poor performance:
  - Too long - sender waits longer than necessary before retransmitting
  - Too short - sender generates unnecessary traffic
- Timeout must be different for each connection and set dynamically
  - Host on same LAN should have shorter timeout than host 20 hops away
  - Delivery time across internet may change over time; timeout must accommodate changes

NOTES

Adaptive Retransmission

Original Algorithm

- Measure `SampleRTT` for each segment/ACK pair
- Compute weighted average of RTT
  - `EstimatedRTT` = $\alpha \times$ `EstimatedRTT` + $\beta \times$ `SampleRTT`
  - where $\alpha + \beta = 1$
  - α between 0.8 and 0.9
  - β between 0.1 and 0.2
- Set timeout based on `EstimatedRTT`
  - `TimeOut` = 2 $\times$ `EstimatedRTT`
- This OverEstimates the Timeout

Karn/Partridge Algorithm

- Do not sample RTT when retransmitting
- Double timeout after each retransmission
- This was later improved on by the Jacobson/Karels Algorithm

NOTES

Connection Establishment: Three-way handshake

- TCP uses a three-way handshake for reliable connection establishment and termination
  - Host 1 sends segment with SYN bit set and random sequence number
  - Host 2 responds with segment with SYN bit set, acknowledgment to Host 1 and random sequence number
  - Host 1 responds with acknowledgment
- TCP will retransmit lost segments
- Random sequence numbers ensure synchronization between endpoints

Connection Establishment

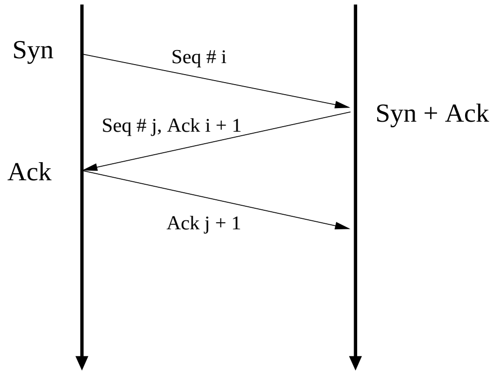


Figure 2: Connection Establishment

NOTES

Connection Management (STD)

- Use a State Transition Diagram (STD) to represent all the possible states a TCP connection can be in.
- Designed to prevent a participant from being fooled by a packet from an old/new connection.

TCP Congestion Control

- Idea
  - assumes best-effort network (FIFO or FQ routers)
  - each source determines network capacity for itself
  - uses implicit feedback
  - ACKs pace transmission (*self-clocking*)
- Challenge
  - determining the initial available capacity
  - adjusting to changes in the available capacity

NOTES

Additive Increase/Multiplicative Decrease

- Objective: adjust to changes in the available capacity
- New state variable per connection: `CongestionWindow`
  - limits how much data source has in transit

  MaxWin = MIN(CongestionWindow, AdvertisedWindow)
  EffWin = MaxWin - (LastByteSent - LastByteAcked)
- Idea:
  - increase `CongestionWindow` when congestion goes down
  - decrease `CongestionWindow` when congestion goes up
- Question: how does the source determine whether or not the network is congested?
- Answer: a timeout occurs
  - timeout signals that a packet was lost
  - packets are seldom lost due to transmission error
  - lost packet implies congestion

Additive Increase/Multiplicative Decrease: (continued)

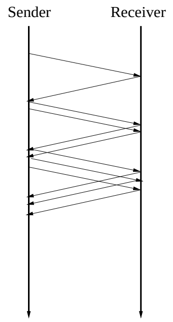
- Algorithm:
  - increment `CongestionWindow` by one packet per RTT (*linear increase*)
  - divide `CongestionWindow` by two whenever a timeout occurs (*multiplicative decrease*)
- In practice: increment a little for each ACK

  Increment = (MSS * MSS)/CongestionWindow

  CongestionWindow += Increment

NOTES

Additive Increase: (continued)



NOTES

Figure 3:

Example Trace

- Example trace: sawtooth behavior sawtooth

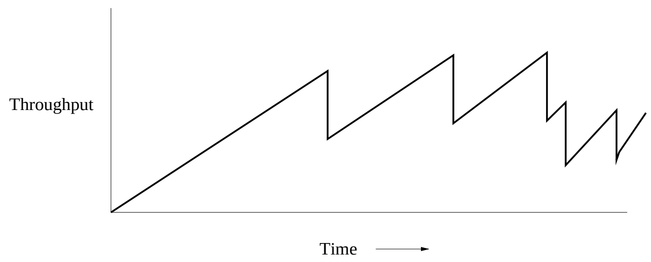


Figure 4:

NOTES

Slow Start

- Objective: determine the available capacity in the first place
- Idea:
  - begin with `CongestionWindow` = 1 packet
  - double `CongestionWindow` each RTT (increment by 1 packet for each ACK)
- Exponential growth, but slower than all in one blast
- Used...
  - when first starting connection
  - when connection goes dead waiting for a timeout

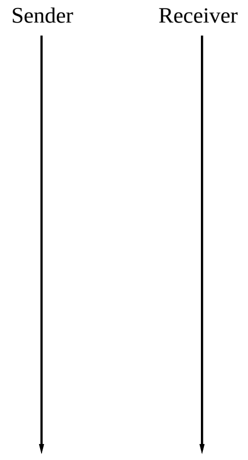
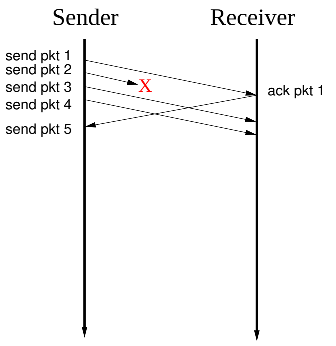


Figure 5:

NOTES

Fast Retransmit and Fast Recovery

- Problem: coarse-grain TCP timeouts lead to idle periods
- Fast retransmit: use duplicate ACKs to trigger retransmission



NOTES

Figure 6: