CSE/EE 461 Lecture 11
Transport: Theory and Practice

Tom Anderson
tom@cs.washington.edu
Peterson, Chapter 2.5, 5.2

Transport Challenge

- IP: routers can be arbitrarily bad
  - packets can be lost, reordered, duplicated, have limited size & can be fragmented
- TCP: applications need something better
  - reliable delivery, in order delivery, no duplicates, arbitrarily long streams of data, match sender/receiver speed, process-to-process

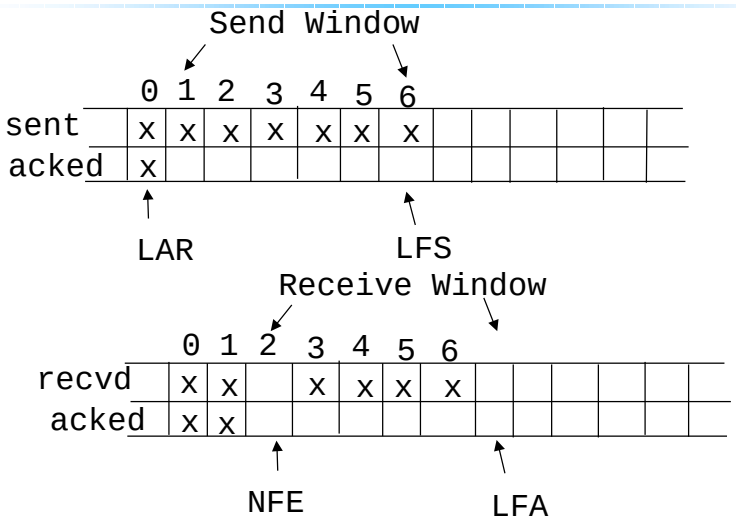
Sliding Window: Reliable, ordered delivery

- Two constraints:
  - Receiver can't deliver packet to application until all prior packets have arrived
  - Sender must prevent buffer overflow at receiver
- Solution: sliding window
  - circular buffer at sender and receiver
    - packets in transit <= buffer size
    - advance when sender and receiver agree packets at beginning have been received

Sender/Receiver State

- sender
  - packets sent and acked (LAR = last ack recvd)
  - packets sent but not yet acked
  - packets not yet sent (LFS = last frame sent)
- receiver
  - packets received and acked (NFE = next frame expected)
  - packets received out of order
  - packets not yet received (LFA = last frame ok)

Sliding Window



Sender Algorithm (Go Back N)

Send full window, set timeout

On receiving an ack:

- if it increases LAR (last ack received)
  - send next packet(s)
    - -- no more than window size outstanding at once
- else (already received this ack)
  - if receive multiple acks for LAR, next packet may have been lost; retransmit LAR + 1; called “fast retransmit”

On timeout:

- resend LAR + 1 (first packet not yet acked)

Receiver Algorithm (Go Back N)

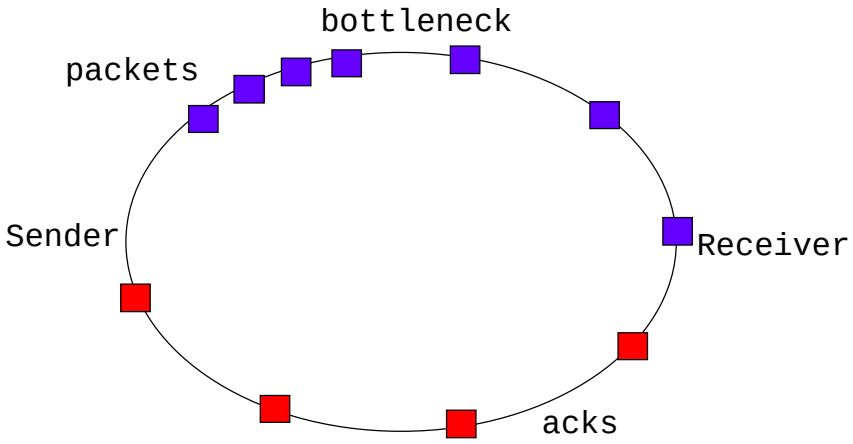
On packet arrival:

```
if packet is the NFE (next frame expected)
    send ack
    increase NFE
    hand any packet(s) below NFE to application
else if < NFE (packet already seen and acked)
    send ack and discard
else (packet is > NFE, arrived out of order)
    buffer and send ack for NFE – 1
        -- signal sender that NFE might have been lost
```

What if link is very lossy?

- Wireless packet loss rates can be 10-30%
  - end to end retransmission will still work
  - will be inefficient, especially with go back N
- Solution: hop by hop retransmission
  - performance optimization, not for correctness
- End to end principle
  - ok to do optimizations at lower layer
  - still need end to end retransmission; why?

Avoiding burstiness: ack pacing



Window size = round trip delay * bit rate

How many sequence #'s?

- Window size + 1?
  - Suppose window size = 3
  - Sequence space: 0 1 2 3 0 1 2 3
  - send 0 1 2, all arrive
    - if acks are lost, resend 0 1 2
    - if acks arrive, send new 3 0 1
- Window <= (max seq # + 1) / 2

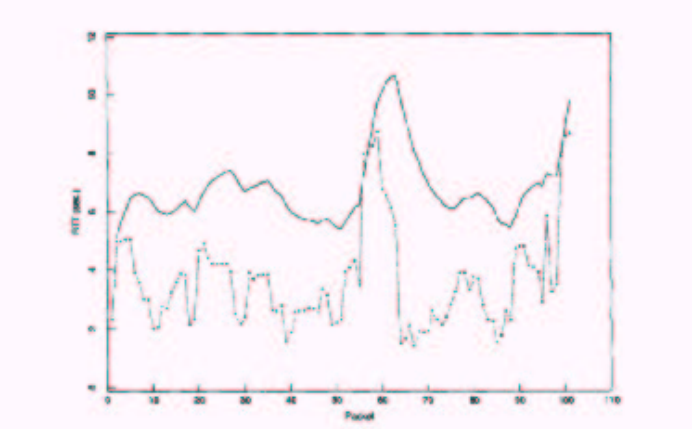
How do we determine timeouts?

- If timeout too small, useless retransmits
  - can lead to congestion collapse (and did in 86)
  - as load increases, longer delays, more timeouts, more retransmissions, more load, longer delays, more timeouts …
- If timeout too big, inefficient
  - wait too long to send missing packet
- Timeout should be based on actual round trip time (RTT)
  - varies with destination subnet, routing changes, congestion, …

Estimating RTTs

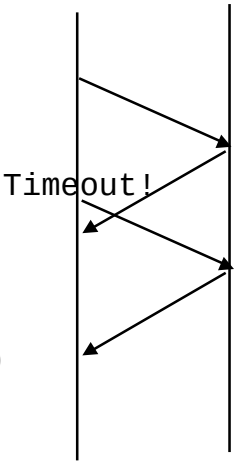
- Idea: Adapt based on recent past measurements
  - For each packet, note time sent and time ack received
  - Compute RTT samples and average recent samples for timeout
  - EstimatedRTT = α x EstimatedRTT + (1 - α) x SampleRTT
  - This is an exponentially-weighted moving average (low pass filter) that smoothes the samples. Typically, α = 0.8 to 0.9.
  - Set timeout to small multiple (2) of the estimate

Estimated Retransmit Timer



Retransmission ambiguity

- How do we distinguish first ack from retransmitted ack?
  - First send to first ack?
    - What if ack dropped?
  - Last send to last ack?
    - What if last ack dropped?
- Might never be able to fix too short a timeout!



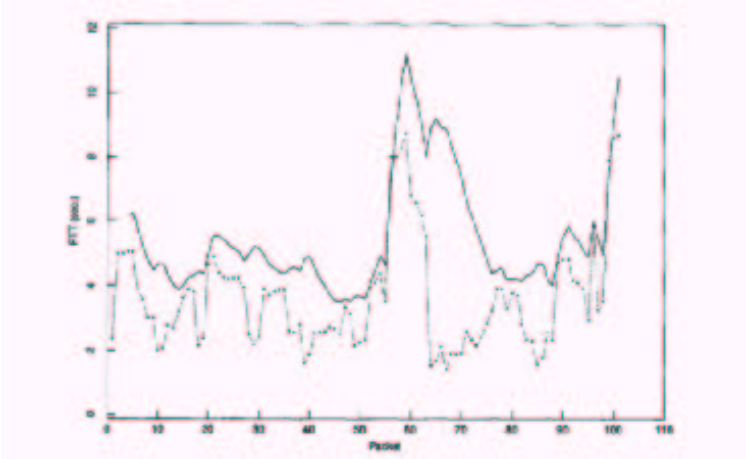
Retransmission ambiguity: Solutions?

- TCP: Karn-Partridge
  - ignore RTT estimates for retransmitted pkts
  - double timeout on every retransmission
- Add sequence #'s to retransmissions (retry #1, retry #2, …)
- TCP proposal: Add timestamp into packet header; ack returns timestamp

Jacobson/Karels Algorithm

- Problem:
  - Variance in RTTs gets large as network gets loaded
  - Average RTT isn't a good predictor when we need it most
- Solution: Track variance too.
  - Difference = SampleRTT – EstimatedRTT
  - EstimatedRTT = EstimatedRTT + (δ x Difference)
  - Deviation = Deviation + δ(|Difference|- Deviation)
  - Timeout = μ x EstimatedRTT + ϕ x Deviation
  - In practice, δ = 1/8, μ = 1 and ϕ = 4

Estimate with Mean + Variance



Transport: Practice

- Protocols
  - IP -- Internet protocol
  - UDP -- user datagram protocol
  - TCP -- transmission control protocol
  - RPC -- remote procedure call
  - HTTP -- hypertext transfer protocol

How do we connect processes?

- IP provides host to host packet delivery
  - header has source, destination IP address
- For applications to communicate, need to demux packets sent to host to target app
  - Web browser (HTTP), Email servers (SMTP), hostname translation (DNS), RealAudio player (RTSP), etc.
  - Process id is OS-specific and transient

Ports

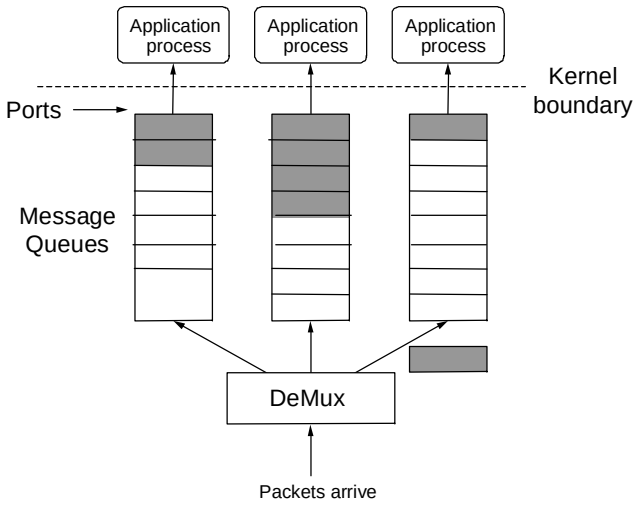
- Port is a mailbox that processes “rent”
  - Uniquely identify communication endpoint as (IP address, protocol, port)
- How do we pick port #’s?
  - Client needs to know port # to send server a request
  - Servers bind to “well-known” port numbers
    - Ex: HTTP 80, SMTP 25, DNS 53, …
    - Ports below 1024 reserved for “well-known” services
  - Clients use OS-assigned temporary (ephemeral) ports
    - Above 1024, recycled by OS when client finished

User Datagram Protocol (UDP)

- Provides application – application delivery
- Header has source & dest port #'s
  - IP header provides source, dest IP addresses
- Deliver to destination port on dest machine
- Reply returns to source port on source machine
- No retransmissions, no sequence #s

=> stateless

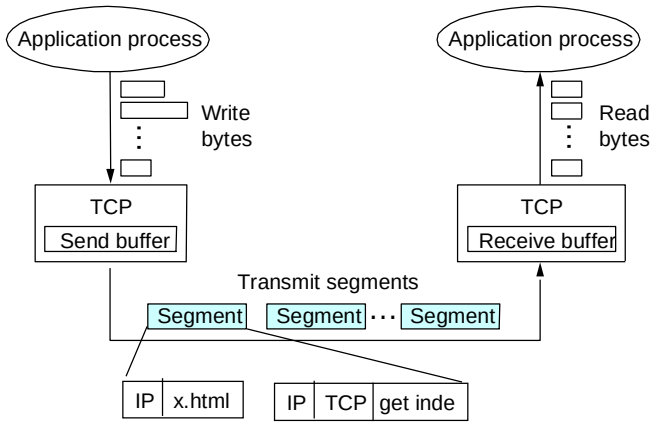
UDP Delivery



Transmission Control Protocol (TCP)

- Reliable bi-directional byte stream
  - No message boundaries
  - Uses ports to identify application endpoints
- Sliding window, go back N, RTT est, …
  - Highly tuned congestion control algorithm
- Connection setup
  - negotiate buffer sizes and initial seq #s
- Flow control
  - prevent sender from overrunning receiver buffers

TCP Delivery



TCP Sliding Window

- Per-byte, not per-packet
  - send packet says “here are bytes j-k”
  - ack says “received up to byte k”
- Send buffer >= send window
  - can buffer writes in kernel before sending
  - writer blocks if try to write past send buffer
- Receive buffer >= receive window
  - buffer acked data in kernel, wait for reads
  - reader blocks if try to read past acked data