TCP Interactive Data Flow

- Problems:
  - Overhead: 40 bytes header + 1 byte data
  - To batch or not to batch: response time important
  - Batching acks:
    - Delay-ack timer: piggyback ack on echo
    - 200 ms timer (fig 19.3)
  - Batching data:
    - Nagle’s algo: Don’t send packet until next ack is received.
  - Developed because of congestion in WANs
TCP Bulk Data Flow

- Sliding window:
  - Send multiple packets while waiting for acks (fig 20.1) up to a limit (W)
  - Receiver need not ack every packet
  - Ack's are cumulative.
  - Ack # = Largest consecutive sequence number received + 1
- Two transfers of the data can have different dynamics (e.g., fig 20.1 vs fig 20.2)
- Receiver window field:
  - Reduced if TCP receiver short on buffers

TCP Bulk Data Flow (Contd)

- End-to-end flow control
- Window update acks: receiver ready
- Default buffer sizes: 4096 to 16384 bytes.
- Ideal: window and receiver buffer = bandwidth-delay product
- TCP window terminology: figs 20.4, 20.5, 20.6
  - Right edge, Left edge, usable window
  - "closes" => left edge (snd_una) advances
  - "opens" => right edge advances (receiver buffer freed => receiver window increases)
  - "shrinks" => right edge moves to left (rare)

The Congestion Problem

- Problem: demand outstrips available capacity ...
- Q: Will the "congestion" problem be solved when:
  - a) Memory becomes cheap (infinite memory)?
  - b) Links become cheap (high speed links)?

- File Transfer Time = 5 mins
- File Transfer Time = 7 hours
c) Processors become cheap (fast routers switches)?

Scenario: All links 1 Gb/s. A & B send to C.

Ans: None of the above solves congestion!

- Congestion: Demand > Capacity
  - It is a dynamic problem => Static solutions are not sufficient
  - TCP provides a dynamic solution

TCP Congestion Control

- Window flow control: avoid receiver overrun
- Dynamic window congestion control: avoid/control network overrun
  - Observation: Not a good idea to start with a large window and dump packets into network
  - Treat network like a black box and start from a window of 1 segment ("slow start")
  - Increase window size exponentially ("exponential increase") over successive RTTs => quickly grow to claim available capacity.

Technique: Every ack: increase cwnd (new window variable) by 1 segment.

- Effective window = Min(cwnd, Wrcvr)
Dynamics

- Rate of acks = rate of packets at the bottleneck: "Self-clocking" property.

Congestion Detection

- Packet loss as an indicator of congestion.
- Set slow start threshold (ssthresh) to \( \min(\text{cwnd}, \text{Wrcvr})/2 \)
- Retransmit pkt, set cwnd to 1 (reenter slow start)

Congestion avoidance

- Increment cwnd by 1 per ack until ssthresh
- Increment by \( 1/\text{cwnd} \) per ack afterwards
  ("Congestion avoidance" or "linear increase")
- Idea: ssthresh estimates the bandwidth-delay product for the connection.
- Initialization: ssthresh = Receiver window or default 65535 bytes. Larger values thru options.
- If source is idle for a long time, cwnd is reset to one.
**Timeout and RTT Estimation**
- Timeout: for robust detection of packet loss
- Problem: How long should timeout be?
  - Too long => underutilization; too short => wasteful retransmissions
- Solution: adaptive timeout: based on RTT

- RTT estimation:
  - Early method: exponential averaging:
    - $R \leftarrow \alpha R + (1 - \alpha) M$ \{ $M =$ measured RTT\}
    - $RTO = \beta R$ \{ $\beta =$ delay variance factor\}
  - Suggested values: $\alpha = 0.9$, $\beta = 2$

**RTT Estimation**
- Jacobson [1988]: this method has problems w/ large RTT fluctuations
- New method: Use mean & deviation of RTT
  - $A =$ smoothed average RTT
  - $D =$ smoothed mean deviation
  - $Err = M - A$ \{ $M =$ measured RTT\}
  - $A \leftarrow A + g*Err$ \{ $g =$ gain = 0.125\}
  - $D \leftarrow D + h*(|Err| - D)$ \{ $h =$ gain = 0.25\}
  - $RTO = A + 4D$
  - Integer arithmetic used throughout.
    - Complex initialization process ...

**Timer Backoff/Karn’s Algorithm**
- Timer backoff: If timeout, $RTO = 2*RTO$ (exponential backoff)
- Retransmission ambiguity problem:
  - During retransmission, it is unclear whether an ack refers to a packet or its retransmission. Problem for RTT estimation
  - Karn/Partridge: don’t update RTT estimators during retransmission.
    - Restart $RTO$ only after an ack received for a segment that is not retransmitted
Fast Retransmit and Recovery

- **Goals:**
  - **Timeout avoidance:** The 500 ms timer granularity can have an adverse performance impact especially for high speed n/ws
  - **Selective retransmission:** Especially when packets are dropped due to error or light congestion
  - **Fast Recovery:** Converge quickly to a state of congestion avoidance (linear increase) with half-current window -- the assumed ideal window size.
  - **Observation:** Receivers are required to send an immediate duplicate acknowledgment when they receive out-of-order data segments.

- **3 duplicate acks**: Assume loss
- **More duplicate acks**: Other packets have reached destination safely.
- **Wait for about 1/2*RTT, and resume transmitting new segments for every subsequent duplicate ack received. Stop this process once the ack for the missing segment received**

- **Fast Retransmit:** Received third duplicate ack:
  - Set `ssthresh` to 1/2 of current `cwnd`
  - Retransmit the missing segment
  - Set `cwnd` to `ssthresh`+3

- **Fast Recovery:** For each duplicate ack hence:
  - Increment `cwnd` by 1 MSS
  - New packets are transmitted once `cwnd` grows large enough.
  - [If old `cwnd` was a pipe of length 1*RTT, the network gets a relief period of 1/2*RTT]
FRR (contd)

- Upon receiving the next (non-duplicate) Ack:
  - Set cwnd to ssthresh & enter linear growth phase

\[
\text{CWND/2} \rightarrow \text{CWND} \rightarrow \text{CWND/2}
\]

New packets sent during this phase

FRR problems

- Burst loss of 3 pkts \(\Rightarrow\) Timeout + window shutdown to cwnd/8!

\[
\text{CWND/8} \rightarrow \text{CWND/4} \rightarrow \text{CWND/2} \rightarrow \text{CWND}
\]

TCP Performance Optimization

- SACK: selective acknowledgments: specifies blocks of packets received at destination.
- Random early drop (RED) scheme spreads the dropping of packets more uniformly and reduces average queue length and packet loss rate.
- Scheduling mechanisms protect well-behaved flows from rogue flows.
Congestion control summary

- Sliding window limited by receiver window.
- Dynamic windows: slow start (exponential rise), congestion avoidance (linear rise), multiplicative decrease.
- Adaptive timeout: need mean RTT & deviation
- Timer back off and Karn’s algo during retransmission
- Go-back-N or Selective retransmission
- Cumulative and Selective acknowledgements
- Timeout avoidance: FRR
- Drop policies, scheduling and ECN

Summary

- Interactive and bulk TCP flow
- TCP congestion control
- Informal exercises: Perform some of the experiments described in chaps 19-21 to see various facets of TCP in action