Principles of Reliable Data Transfer

Outline:
- Overview of reliable data transfer
  - a correct protocol
    - stop-and-wait
  - an efficient protocol
    - sliding window

Reliable Data Transfer

- Our goal: end-to-end solution to achieve reliable data transfer
- What is reliable data transfer?
  - guaranteed arrival
  - no error
  - in order delivery
- Why is it difficult?
  - end-to-end solution has no control of underlying communication channel, which can be error-prone and lossy
- Where is it used in computer networks?
  - reliable transport service on top of unreliable IP
  - reliable data link service on top of unreliable physical layer

Reliable Data Transfer: An Illustration

- end-to-end protocol control over unreliable channel
- consider only unidirectional data transfer for simplicity
  - protocol control data may flow in either direction

Principles of Reliable Data Transfer

- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol
  - e.g., delay in the channel is bounded in physical layer, not so for network layer
- Other services may interact with RDT protocol
  - e.g., flow control, congestion control
- Here we study the principles of reliable data transfer, so
  - we don't make assumptions about the unreliable channel
  - we don't consider interaction with other services
- Later we see what RDT is like in practice
  - in a transport layer protocol – TCP
Outline

- Overview of reliable data transfer
  - a correct protocol: stop-and-wait
    - one packet at a time
  - an efficient protocol: sliding window
    - multiple packets simultaneously

Deal with Errors

- First deal with errors, later deal with packet loss.
  - ACK-based solution: receiver check errors
    - if correct, send back positive ACK
    - otherwise, send back negative NAK
  - Solution 1: creating special acknowledgments for ACK and NAK. What if they get corrupted too??
  - Solution 2: treat a corrupted acknowledgement as NAK. Duplicated packets!!
  - To solve duplicated packets: sequence number for each packet.

What if ACK or NAK is corrupted?

- Solution 1: creating special acknowledgments for ACK and NAK. What if they get corrupted too??
- Solution 2: treat a corrupted acknowledgement as NAK. Duplicated packets!!
- To solve duplicated packets: sequence number for each packet.

Deal with Packet Loss: Timeouts

- Early timeout ⇒ duplicated packet ⇒ sequence number. (Not likely for data link protocol)
- Is this totally correct?
Deal with Duplicated ACKs

Solution: each ACK carries sequence number.
With timeout, NAK is not necessary any more.

Stop-and-Wait

Now we have a correct protocol:

- Allow one outstanding (un-ACKed) packet.
- You see why it is called stop-and-wait?
- By the way, we haven’t talked about in-order delivery.

Efficiency of Stop-and-Wait

Channel utilization efficiency = \( \frac{L}{RTT + \frac{L}{R}} \)

Efficiency of Stop-and-Wait (An Example)

- It works, but performance stinks!
- Example: 1 Gbps link, 30ms roundtrip prop. delay, 1KB (8kbits) packet.

\[
T_{\text{transmit}} = \frac{L}{R} = \frac{8 \text{ kb}}{1 \text{ gb/sec}} = 8 \text{ microsec}
\]

Throughput = \( \frac{L}{T_{\text{transmit}} + RTT} \) = \( \frac{8 \text{ kb}}{0.03008 \text{ sec}} \) = 266Kbps

- 266Kbps throughput over 1 Gbps link ⇒ 0.027% efficiency!
- network protocol limits use of physical resources!
Pipelined Protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets

A pipelined protocol: Sliding Window

- Allow multiple outstanding (un-ACKed) packets
- Upper bound on un-ACKed packets, called window

Go-Back-N: Sender

- "window" of up to N consecutive un-ACKed packets allowed; limit send buffer space
- cumulative ACK - ACK with seq #n stands for ACKs all packets up to, including seq #n
- recv ACKs in send window ⇒ sliding send window
- timer for each in-flight packet
- packet with seq #n timeouts: retransmit packet #n and all higher seq # packets in window ⇒ buffering

Increase utilization by a factor of 3!
**Go-Back-N: Receiver**

- Send ACK for correctly-received packet with highest *in-order* seq #
  - Sliding receive window
- Out-of-order packet:
  - Discard (don't buffer) -> no receiver buffering!
  - Optional: Re-ACK packet with highest in-order seq # (sort of a NACK)
    - alert sender something is wrong through duplicated ACKs
    - not critical for protocol correctness; but may improve performance

**Selective Repeat**

- receiver individually acknowledges all correctly received packets.
  - buffer out-of-order packets for eventual in-order delivery to upper layer
- sender timer for each un-ACKed packet
  - sender only resend packets whose timers expire before ACKs are received
Selective Repeat

**sender**
- data from above:
  - if next available seq # in window, send pkt
- timeout(n):
  - resend pkt n, restart timer
- ACK(n):
  - mark pkt n as received
  - if n is smallest unACKed pkt, advance window base to next unACKed seq # (sliding!)

**receiver**
- pkt n in [rcvbase, rcvbase+N-1]
  - send ACK(n)
  - out-of-order: buffer
  - in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt (sliding!)
- pkt n in [rcvbase-N, rcvbase-1]
  - ACK(n)
- otherwise:
  - ignore

**Selective Repeat in Action**

**Disclaimer**

Parts of the lecture slides contain original work of James Kurose, Larry Peterson, and Keith Ross. The slides are intended for the sole purpose of instruction of computer networks at the University of Rochester. All copyrighted materials belong to their original owner(s).