Outline

- Development of reliable protocol
- Sliding window protocols
  - Go-Back-N, Selective Repeat
- Protocol performance
- Sockets, UDP, TCP, and IP
- UDP operation
- TCP operation
  - connection management
  - flow control
  - congestion control

Parameters

- System properties
  - $C =$ channel capacity in bps
  - $I =$ interrupt/service time + propagation delay
- Frame format
  - $D =$ number of data bits per frame
  - $H =$ number of bits in the frame header
  - $F = D + H =$ (total frame length)
  - $A =$ number of bits in an ACK frame
Parameters (cont)

- Error probabilities
  - $E = P($bit being in error$)$
  - $L = P($frame or its ACK is lost or damaged$)$
  - $P_1 = P($data frame is lost or damaged$)$
  - $P_2 = P($ACK frame is lost or damaged$)$

- Protocol parameters
  - $W =$ window size
  - $T =$ timeout interval

Stop-and-Wait with No Errors

- At time $(F/C + A/C + 2I)$, the sender has processed the ACK
- Bandwidth occupied by one frame $= C(F/C + A/C + 2I) = F + A + 2CI$
- $D$ bits of data are actually sent
- So, utilization is?
  $$\frac{F/C}{F/C + A/C + 2I} \quad \text{or} \quad \frac{D}{F + A + 2CI}$$
Example:

- $C = 10\text{Mbs link capacity}$
- $I = 5\text{ms (1000Km link) propagation delay}$
- $D = 8000\text{ bits per frame}$
- $H = 106\text{ bits in frame header}$
- $F = 8000 + 106 = 106$
- $A = 106\text{ bits}$
- $F/C + A/C + 2I = 0.0008106 + 0.0000106 + 0.01 = 0.0108212 \text{ sec or 10.8ms}$
- Sender Utilization = $.0008106/.0108212 = 0.07491 = 7.5\%$
- Bandwidth Utilization = $8000/108212 = 7.39\%$

Stop-and-Wait with Errors

- Lost frame uses $F + CT\text{ bits of transmission capacity}$
- $R$: mean number of retransmissions per frame
- So, total capacity used by a frame is $R(F + CT) + (F + A + 2CI)$
- Probability that the frame and ACK arrive intact is $(1 - P_1)(1 - P_2)$
- Therefore, $L = 1 - (1 - P_1)(1 - P_2)$
Stop-and-Wait with Errors

- Probability that exactly k attempts are needed is \((1 - L)L^{k-1}\)
- Expected number of transmissions per frame is:
  \[ R = \sum_{k=1}^{\infty} k \times (1-L) \times L^{(k-1)} = \frac{1}{1-L} \]
- So, utilization is?

\[
\frac{D}{R(F + CT) + (F + A + 2CI)}
\]

Example:

- \(P1 = 0.2, P2 = 0.2\)
- \(L = 1 - (1 - 0.2)(1-0.2) = 1 - 0.64 = 0.32\)
- Expected number of retransmissions
  \[ R = \frac{1}{1-L} = \frac{1}{1-0.32} = 1.5625 \]

\[
\frac{D}{R(F + CT) + (F + A + 2CI)} = \frac{8000}{1.5625 \times (8106 + 10^7 \times 0.005) + (8106 + 106 + 2 \times 10^7 \times 0.005)}
\]

With Retransmissions Bandwidth Utilization = 4.02%
**Sliding Window with No Errors**

- In order to simplify analysis, assume:
  - Acks are piggybacked and can be ignored.
  - Interrupt processing time is negligible, so \( I = \tau \), the one-way propagation delay.
- Sender can send for \( W \frac{F}{C} \) seconds before it must stop and wait.
- Ack of first frame arrives at time \( \frac{F}{C} + 2I \).

**Case 1: Large Window**

- Sender may transmit continuously
  \( W \frac{F}{C} \geq \frac{F}{C} + 2I \) (fast short link)
  or
  \( W \geq 1 + 2CI/F \)

- Hence, \( U = \)?
  \[
  \frac{W \frac{F}{C}}{\frac{F}{C} + 2I} \quad \text{or} \quad \frac{W}{1 + 2CI/F} > 1 \quad \text{so 100%}
  \]
**Case 2: Small Window**

- Sender must stop and wait
  \[ W \frac{F}{C} \leq F/C + 2I \text{ (fast long link)} \]
  or
  \[ W \leq 1 + 2CI/F \]

  Sender can transmit \( W \) frames in time 
  \( F/C + 2I \).

- Therefore, \( U = 37.5\% \)

---

**Example:**

\( C = 10 \text{Mbs link capacity} \)
\( I = 5 \text{ms (1000Km link) propagation delay} \)
\( D = 8000 \text{ bits per frame} \)
\( H = 106 \text{ bits in frame header} \)
\( F = D + H = 8000 + 106 = 8106 \)
\( A = 106 \text{ bits} \)

**Case 1 - Large Window**
\( W = 50 \)
\( W \frac{F}{C} = 0.0405 \)
\( F/C + 2I = 0.0108 \)
\( U = 100\% \)

**Case 2 - Small Window**
\( W = 5 \)
\( W \frac{F}{C} = 0.00405 \)
\( F/C + 2I = 0.0108 \)
\( U = 0.00405/0.0108 = 37.5\% \)
Sliding Window with Errors

- Only consider selective repeat here.
- From before, \( R = 1/(1-L) \)
- So, to receive \( W \) frames, have to send how many frames? \( W \times R \)
- Case 1: \( W \geq 1 + 2CI/F \)
  - transmission is still continuous, but extra frames must be sent to correct damages ones
  - \( U =? \)
- Case 2: \( W \leq 1 + 2CI/F \)
  - For small \( =? \)

Outline

- Development of reliable protocol
- Sliding window protocols
  - Go-Back-N, Selective Repeat
- Protocol performance
- Sockets, UDP, TCP, and IP
- UDP operation
- TCP operation
  - connection management
  - flow control
  - congestion control
Multiplexing/demultiplexing

Demultiplexing at rcv host: delivering received segments to correct socket

Multiplexing at send host: gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

= socket = process

How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number
- host uses IP addresses & port numbers to direct segment to appropriate socket

TCP/UDP segment format

- 32 bits
- source port #
- dest port #
- other header fields
- application data (message)
**Connectionless Demultiplexing**

- **Receiver**: Create socket and bind with port numbers
- **UDP socket** identified by two-tuple: \((\text{dest IP address, dest port number})\)
- **When host OS receives UDP segment**:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- **IP datagrams** with different source IP addresses and/or source port numbers can be directed to same socket

---

**Connectionless demux (cont)**

Assume server (IP address C) binds port # 6428 to its socket

![Diagram showing IP addresses and port numbers](image)

SP (Source Port) provides “return address”
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address, source port number
  - dest IP address, dest port number
- receiving host uses all four values to direct segment to appropriate socket (typically, hash function)
- Server host may support many simultaneous TCP sockets with the same port number:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - E.g., non-persistent HTTP will allocate a different socket for each arriving HTTP request

Connection-oriented demux (cont)

Web server waiting for connections on port 80.
### Connection-oriented demux: Threaded Web Server

![Diagram](image)

---

**Outline**

- Development of reliable protocol
- Sliding window protocols
  - Go-Back-N, Selective Repeat
- Protocol performance
- Sockets, UDP, TCP, and IP
- UDP operation
- TCP operation
  - connection management
  - flow control
  - congestion control
UDP: User Datagram Protocol [RFC 768]

- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
  - lost
  - delivered out of order to app
- connectionless:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why, again, is there UDP?
- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small packet header
- no congestion control: UDP can blast away as fast as desired

UDP Uses and Packet Structure

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS (!) Why?
  - SNMP
- reliable transfer over UDP?
  Why would anyone implement reliability at application layer??
**UDP Checksum**

- **Goal:** detect errors (e.g., flipped bits) in transmitted segment (header and data)
- **Wait!** I thought UDP was unreliable?!
- **We differentiate between:**
  - Error detection and
  - Error recovery (doing something about it)
- **Arriving packets with errors detected are dropped by UDP (referred to as erasures)**
- **UDP checksum can be turned off (setsockopt system call). Why would anyone do that?**

**UDP checksum calculation**

**Sender:**
- treat segment contents as sequence of 16-bit integers
- checksum: addition (1’s complement sum) of segment contents
- sender puts checksum value into UDP checksum field

**Receiver:**
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected.

*(of course, no error detection code is perfect)*
Internet Checksum Example

- **Note**
  - When adding numbers, a carryout from the most significant bit needs to be added to the result.

- **Example:** add two 16-bit integers
  - \[\begin{array}{c}
  11100110100110 \\
  11010101010101 \\
  \end{array}\]
  - **wraparound sum:** \[1101110110111011\]
  - **sum:** \[1011110111101100\]
  - **checksum:** \[0100010010000111\]

Internet Checksum

- **Same checksum used on TCP segments and IP headers**
- **Frankly, this checksum is not very good...**
- **What are some errors it will miss?**

- **So, why is it used??**
**UDP Pseudo-Header**

- Contains the source IP address, destination IP address, protocol type, length of UDP datagram, padding
- The pseudo-header does not get transmitted with the UDP datagram

---

**UDP Pseudo-Header**

- When we compute the checksum for the UDP datagram, we compute it over the datagram (data and header) and the pseudo-header.

- Why?