Message, Segment, Packet, and Frame
Error Control and Flow Control (3)

Flow control example: A’s data is limited by B’s

<table>
<thead>
<tr>
<th>A</th>
<th>Message</th>
<th>B</th>
<th>B’s Buffer</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt; request 8 buffers&gt;</td>
<td></td>
<td></td>
<td>A wants 8 buffers</td>
</tr>
<tr>
<td>2</td>
<td>&lt;ack = 15, buf = 4&gt;</td>
<td></td>
<td>0 1 2 3</td>
<td>B grants messages 0-3 only</td>
</tr>
<tr>
<td>3</td>
<td>&lt;seq = 0, data = m0&gt;</td>
<td>0 1 2 3</td>
<td></td>
<td>A has 3 buffers left now</td>
</tr>
<tr>
<td>4</td>
<td>&lt;seq = 1, data = m1&gt;</td>
<td>0 1 2 3</td>
<td></td>
<td>A has 2 buffers left now</td>
</tr>
<tr>
<td>5</td>
<td>&lt;seq = 2, data = m2&gt;</td>
<td></td>
<td>0 1 2 3</td>
<td>A has 3 buffers left now</td>
</tr>
<tr>
<td>6</td>
<td>&lt;ack = 1, buf = 3&gt;</td>
<td></td>
<td>1 2 3 4</td>
<td>Message lost but A thinks it has 1 left</td>
</tr>
<tr>
<td>7</td>
<td>&lt;seq = 3, data = m3&gt;</td>
<td>1 2 3 4</td>
<td></td>
<td>B acknowledges 0 and 1, permits 2-4</td>
</tr>
<tr>
<td>8</td>
<td>&lt;seq = 4, data = m4&gt;</td>
<td>1 2 3 4</td>
<td></td>
<td>A has 1 buffer left</td>
</tr>
<tr>
<td>9</td>
<td>&lt;seq = 5, data = m5&gt;</td>
<td>1 2 3 4</td>
<td></td>
<td>A has 0 buffers left, and must stop</td>
</tr>
<tr>
<td>10</td>
<td>&lt;ack = 4, buf = 0&gt;</td>
<td>1 2 3 4</td>
<td></td>
<td>A times out and retransmits</td>
</tr>
<tr>
<td>11</td>
<td>&lt;ack = 4, buf = 1&gt;</td>
<td>2 3 4 5</td>
<td></td>
<td>Everything acknowledged, but A still blocked</td>
</tr>
<tr>
<td>12</td>
<td>&lt;ack = 4, buf = 2&gt;</td>
<td>3 4 5 6</td>
<td></td>
<td>A may now send 5</td>
</tr>
<tr>
<td>13</td>
<td>&lt;seq = 5, data = m5&gt;</td>
<td>3 4 5 6</td>
<td></td>
<td>B found a new buffer somewhere</td>
</tr>
<tr>
<td>14</td>
<td>&lt;seq = 6, data = m6&gt;</td>
<td>3 4 5 6</td>
<td></td>
<td>A has 1 buffer left</td>
</tr>
<tr>
<td>15</td>
<td>&lt;ack = 6, buf = 0&gt;</td>
<td>3 4 5 6</td>
<td></td>
<td>A is now blocked again</td>
</tr>
<tr>
<td>16</td>
<td>&lt;ack = 6, buf = 4&gt;</td>
<td>3 4 5 6</td>
<td></td>
<td>A is still blocked</td>
</tr>
</tbody>
</table>

Potential deadlock
Acknowledgements & Timeouts

• An *acknowledgement* (ACK) is a packet sent by one host in response to a packet it has received
  • Making a packet an ACK is simply a matter of changing a field in the transport header
  • Data can be *piggybacked* in ACKs

• A *timeout* is a signal that an ACK to a packet that was sent has not yet been received within a specified timeframe
  • A timeout triggers a *retransmission* of the original packet from the sender
  • How are timers set?
Propagation Delay

• Propagation delay is defined as the delay between transmission and receipt of packets between hosts
• Propagation delay can be used to estimate timeout period
• How can propagation delay be measured?
• What else must be considered in the measurement?
Reliable Transmission

• Transfer frames without errors
  • Error Correction
  • Error Detection
  • Discard frames with error
• Acknowledgements and Timeouts
• Retransmission
• ARQ – Automatic Repeat Request
Stop and Wait with 1-bit Seq No
Stop and Wait Protocols

• Simple
• Low Throughput
  • One Frame per RTT

• Increase throughput by having more frames in flight
  • Sliding Window Protocol
Stop-and-Wait – Error-free channel
Protocol (p2) ensures sender can’t outpace receiver:

- Receiver returns a dummy frame (ack) when ready
- Only one frame out at a time – called stop-and-wait
- We added flow control!

```c
void sender2(void)
{
    frame s;
    packet buffer;
    event_type event;

    while (true) {
        from_network_layer(&buffer);
        s.info = buffer;
        to_physical_layer(&s);
        wait_for_event(&event);
    }
}

void receiver2(void)
{
    frame r, s;
    event_type event;
    while (true) {
        wait_for_event(&event);
        from_physical_layer(&r);
        to_network_layer(&r.info);
        to_physical_layer(&s);
        
    }
}
```

Sender waits to for ack after passing frame to physical layer
Receiver sends ack after passing frame to network layer
Stop-and-Wait – Noisy channel (1)

**ARQ** (Automatic Repeat reQuest) adds error control
- Receiver acks frames that are correctly delivered
- Sender sets timer and resends frame if no ack

For correctness, frames and acks must be numbered
- Else receiver can’t tell retransmission (due to lost ack or early timer) from new frame
- For stop-and-wait, 2 numbers (1 bit) are sufficient
Stop and Wait

Duplicate Frames
Stop-and-Wait – Noisy channel (2)

Sender loop (p3):

Send frame (or retransmission)
Set timer for retransmission
Wait for ack or timeout

If a good ack then set up for the next frame to send (else the old frame will be retransmitted)
Stop-and-Wait – Noisy channel (3)

Receiver loop (p3):

```c
void receiver3(void)
{
    seq_nr frame_expected;
    frame r, s;
    event_type event;
    frame_expected = 0;
    while (true) {
        wait_for_event(&event);
        if (event == frame_arrival) {
            from_physical_layer(&r);
            if (r.seq == frame_expected) {
                to_network_layer(&r.info);
                inc(frame_expected);
            }
            s.ack = 1 - frame_expected;
            to_physical_layer(&s);
        }
    }
}
```
Stop-and-Wait Process

- Sender doesn’t send next packet until he’s sure receiver has received the last packet
- The packet/Ack sequence enables reliability
- Sequence numbers help avoid problem of duplicate packets
- Problem: keeping the pipe full
- Example
  - 1.5Mbps link x 45ms RTT = 67.5Kb (8KB) delay bandwidth product
  - 1KB frames implies 1/8th link utilization
Solution: Pipelining via Sliding Window

- Allow multiple outstanding (un-ACKed) frames
- Upper bound on un-ACKed frames, called *window*
Buffering on Sender and Receiver

- Sender needs to buffer data so that if data is lost, it can be resent.
- Receiver needs to buffer data so that if data is received out of order, it can be held until all packets are received.
  - Flow control
- How can we prevent sender overflowing receiver’s buffer?
  - Receiver tells sender its buffer size during connection setup.
- How can we insure reliability in pipelined transmissions?
  - Go-Back-N
    - Send all $N$ unACKed packets when a loss is signaled.
    - Inefficient
  - Selective repeat
    - Only send specifically unACKed packets.
    - A bit trickier to implement.
Sliding Window: Sender

- Assign sequence number to each frame ($\text{SeqNum}$)
- Maintain three state variables:
  - send window size ($\text{SWS}$)
  - last acknowledgment received ($\text{LAR}$)
  - last frame sent ($\text{LFS}$)
- Maintain invariant: $\text{LFS} - \text{LAR} \leq \text{SWS}$
- Advance $\text{LAR}$ when ACK arrives
- Buffer up to $\text{SWS}$ frames
Sliding Window: Receiver

- Maintain three state variables
  - receive window size (RWS)
  - largest frame acceptable (LFA)
  - last frame received (LFR)
- Maintain invariant: \( \text{LFA} - \text{LFR} \leq \text{RWS} \)

- Frame SeqNum arrives:
  - if \( \text{LFR} < \text{SeqNum} \leq \text{LFA} \) accept
  - if \( \text{SeqNum} \leq \text{LFR} \) or \( \text{SeqNum} > \text{LFA} \) discarded
- Send cumulative ACKs – send ACK for largest frame such that all frames less than this have been received
Sequence Number Space

- **SeqNum** field is finite; sequence numbers wrap around
- Sequence number space must be larger than the number of outstanding frames
- \( SWS \leq \text{MaxSeqNum} - 1 \) is not sufficient
  - Suppose 3-bit \( \text{SeqNum} \) field (0..7)
  - \( SWS=RWS=7 \)
  - Sender transmits frames 0..6
  - Arrive successfully, but ACKs lost
  - Sender retransmits 0..6
  - Receiver expecting 7, 0..5, but receives the original incarnation of 0..5
- \( SWS < (\text{MaxSeqNum}+1)/2 \) is correct rule
- Intuitively, \( \text{SeqNum} \) “slides” between two halves of sequence number space
Sequence Number Space

For correctness, we require:

- Sequence numbers (s) at least twice the window (w)

**Error case (s=8, w=7) – too few sequence numbers**

**Correct (s=8, w=4) – enough sequence numbers**

New receive window overlaps old – retransmits ambiguous

New and old receive window don’t overlap – no ambiguity
Go-Back-N

Receiver only accepts/acks frames that arrive in order:
  • Discards frames that follow a missing/errored frame
  • Sender times out and resends all outstanding frames
Go-Back-N

Tradeoff made for Go-Back-N:
• Simple strategy for receiver; needs only 1 frame
• Wastes link bandwidth for errors with large windows; entire window is retransmitted
Selective Repeat

Receiver accepts frames anywhere in receive window

- Cumulative ack indicates highest in-order frame
- NAK (negative ack) causes sender retransmission of a missing frame before a timeout resends window
Selective Repeat

Tradeoff made for Selective Repeat:

- More complex than Go-Back-N due to buffering at receiver and multiple timers at sender
- More efficient use of link bandwidth as only lost frames are resent (with low error rates)
Throughput limits

• Buffers

• Bandwidth – subnet’s carrying capacity
  • K TPDUs per second
  • X paths then total of XK

• Flow control to manage
  • Manage window size
    • If network can handle c TPDUs/sec and Cycle time is r then the window size should be cr
Multiplexing

- Kinds of transport / network sharing that can occur:
  - Multiplexing: connections share a network address
  - Inverse multiplexing: addresses share a connection
Crash Recovery

• Network Failures
  • Transport layer handles
    • Connectionless
    • Connection oriented

• Host Crashes
  • Server crash and may reboot
    • Send broadcast asking clients to inform of prior connections (stop and wait protocol)
      • Client – one TPDU outstanding or none outstanding
Crash Recovery

Application needs to help recovering from a crash
- Transport can fail since A(ck) / W(rite) not atomic

<table>
<thead>
<tr>
<th>Strategy used by sending host</th>
<th>First ACK, then write</th>
<th>First write, then ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Never retransmit</td>
<td>LOST, OK, LOST</td>
<td>LOST, OK, OK</td>
</tr>
<tr>
<td>Retransmit in S0</td>
<td>OK, DUP, LOST</td>
<td>LOST, DUP, OK</td>
</tr>
<tr>
<td>Retransmit in S1</td>
<td>LOST, OK, OK</td>
<td>OK, OK, DUP</td>
</tr>
</tbody>
</table>

OK = Protocol functions correctly
DUP = Protocol generates a duplicate message
LOST = Protocol loses a message
Congestion Control

Two layers are responsible for congestion control:

• Transport layer, controls the offered load [here]
  • Network layer, experiences congestion [previous]
  
• Desirable bandwidth allocation »
• Regulating the sending rate »
• Wireless issues »
Desirable Bandwidth Allocation (1)

Efficient use of bandwidth gives high goodput, low delay

- Goodput rises more slowly than load when congestion sets in
- Delay begins to rise sharply when congestion sets in
Desirable Bandwidth Allocation (2)

Fair use gives bandwidth to all flows (no starvation)
  • Max-min fairness gives equal shares of bottleneck
Desirable Bandwidth Allocation (3)

We want bandwidth levels to converge quickly when traffic patterns change.

- Flow 1 slows quickly when Flow 2 starts
- Flow 1 speeds up quickly when Flow 2 stops
- Flow 2 starts
- Flow 3 starts
- Flow 2 stops

October 11, 2018
Regulating the Sending Rate (1)

Sender may need to slow down for different reasons:

- Flow control, when the receiver is not fast enough [right]
- Congestion, when the network is not fast enough [over]

A fast network feeding a low-capacity receiver $\rightarrow$ flow control is needed
Regulating the Sending Rate (2)

Our focus is dealing with this problem – congestion

A slow network feeding a high-capacity receiver → congestion control is needed

October 11, 2018
Regulating the Sending Rate (3)

Different congestion signals the network may use to tell the transport endpoint to slow down (or speed up)

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Signal</th>
<th>Explicit?</th>
<th>Precise?</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCP</td>
<td>Rate to use</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>TCP with ECN</td>
<td>Congestion warning</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>FAST TCP</td>
<td>End-to-end delay</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>CUBIC TCP</td>
<td>Packet loss</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>TCP</td>
<td>Packet loss</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Regulating the Sending Rate (3)

If two flows increase/decrease their bandwidth in the same way when the network signals free/busy they will not converge to a fair allocation.

+/- constant
Additive increase and decrease

+/- percentage
Multiplicative increase and decrease

Fairness line
Optimal point
Efficiency line
Regulating the Sending Rate (4)

The AIMD (Additive Increase Multiplicative Decrease) control law does converge to a fair and efficient point!

- TCP uses AIMD for this reason.

![Diagram showing the relationship between User 1's and User 2's bandwidth, with points indicating convergence to an optimal point on the fairness line.]
Wireless Issues

Wireless links lose packets due to transmission errors

• Do not want to confuse this loss with congestion
• Or connection will run slowly over wireless links!

Strategy:

• Wireless links use ARQ, which masks errors
A Simple Transport Protocol

• The Example Service Primitives
• The Example Transport Entity
• The Example as a Finite State Machine

Similar to TCP but simpler
Service Primitives

- Connect
  - Parameters – local and remote TSAPs
  - Caller is blocked
    - If connection succeeds the caller is unblocked and transmission starts
- Listen – specifies a TSAP to listen to
- Disconnect
- Send
- Receive
- ** Library procedures
Service Primitives

- Connum=LISTEN(local)
- Connum=Connect(local,remote)
- Status = Send(Connum, buffer, bytes)
  - No Connection, illegal buffer address, negative count
- Status = Receive(Connum, buffer, bytes)
- Status = Disconnect(Connum)
The Transport Entity

• Use connection oriented, reliable network service
• Transport Entity is part of the user process
• Network Layer interface
  • To_net and from_net
  • Parameters –
    • Connection Identifier
    • Q bit – control message
    • M bit – more data from this message to follow
    • Packet Type
    • Pointer to data
The Example Transport Entity

The network layer packets used in our example.

<table>
<thead>
<tr>
<th>Network packet</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL REQUEST</td>
<td>Sent to establish a connection</td>
</tr>
<tr>
<td>CALL ACCEPTED</td>
<td>Response to CALL REQUEST</td>
</tr>
<tr>
<td>CLEAR REQUEST</td>
<td>Sent to release a connection</td>
</tr>
<tr>
<td>CLEAR CONFIRMATION</td>
<td>Response to CLEAR REQUEST</td>
</tr>
<tr>
<td>DATA</td>
<td>Used to transport data</td>
</tr>
<tr>
<td>CREDIT</td>
<td>Control packet for managing the window</td>
</tr>
</tbody>
</table>
The Example Transport Entity (2)

Each connection is in one of seven states:
1. Idle – Connection not established yet.
2. Waiting – CONNECT has been executed, CALL REQUEST sent.
3. Queued – A CALL REQUEST has arrived; no LISTEN yet.
4. Established – The connection has been established.
5. Sending – The user is waiting for permission to send a packet.
6. Receiving – A RECEIVE has been done.
7. DISCONNECTING – a DISCONNECT has been done locally.
State Transitions

• A primitive is executed
• A packet arrives
• A timer expires
Internet Protocols – UDP

• Introduction to UDP
• Remote Procedure Call
• Real-Time Transport
User Datagram Protocol

- Connectionless
- Does not do
  - Flow control
  - Error control
  - Retransmissions
- Useful in client-server situations
- Sends segments consisting of an 8-byte header followed by the payload
Introduction to UDP (1)

UDP (User Datagram Protocol) is a shim over IP

- Header has ports (TSAPs), length and checksum.
Introduction to UDP (2)

Checksum covers UDP segment and IP pseudoheader

- Fields that change in the network are zeroed out
- Provides an end-to-end delivery check
RPC (Remote Procedure Call)
- RPC connects applications over the network with the familiar abstraction of procedure calls
  - Stubs package parameters/results into a message
  - UDP with retransmissions is a low-latency transport
Limitations of RPC

• Pointers
• Weakly Typed languages – variable length arrays
• Not possible always to deduce parameter types
• Global variables
Real-Time Transport (1)

RTP (Real-time Transport Protocol) provides support for sending real-time media over UDP

- Often implemented as part of the application
Real-Time Transport (2)

RTP header contains fields to describe the type of media and synchronize it across multiple streams

- RTCP sister protocol helps with management tasks
RTP Header Fields

- Ver – 2
- P – Packet padded to multiple of 4 bytes
- X – extension header present
- CC – number of contributing sources
- M bit – Application specific marker
- Payload Type – encoding used
- Sequence Number
- Time stamp – produced by the source
- Synchronizations Source Identifier – which stream the packet belongs to
RTP Profiles

• RTP payloads may contain multiple samples coded in any way the application wants

• Profiles – to support interworking
  • Single Audio Stream

• Multiple encoding formats may be supported
  • 8-bit pcm samples at 8KHz
  • Delta encoding
  • Predictive encoding
  • MP3
  • …
RTCP – Real-time Transport Control Protocol

• Control Protocol for RTP
• Does not transport any data
• Handles:
  • Feedback
    • Delay
    • Jitter
    • Bandwidth
    • Congestion, etc.
• Synchronization
  • Interstream Synchronization – Different clocks, drifts, etc.
• User Interface
Real-Time Transport (3)

Buffer at receiver is used to delay packets and absorb jitter so that streaming media is played out smoothly.
Real-Time Transport (3)

High jitter, or more variation in delay, requires a larger playout buffer to avoid playout misses

- Propagation delay does not affect buffer size