Data Link Layer

Flow Control

Geoff Merrett

ELEC3222: Computer Networks

See Tanenbaum Chapter 3 (Data Link Layer)
Outline

- Flow Control
  - Introduction
  - Simplex Communication
    - Example Protocols
      - 1: Utopia
      - 2: Stop-and-Wait (Error Free)
      - 3: Stop-and-Wait (Noisy Channel)
  - Duplex Communication
    - Introduction
    - Example Protocols
      - 4: 1-bit Sliding Window
      - 5: Go-back-N
      - 6: Selective Repeat
Introduction

- We know how to:
  - Split a packet up into frames, and delimit their start and end
  - Add error detection into a frame, but...
    * what do we do when an error is detected?
  - Add error correction into a frame, but...
    * what do we do if more errors occur than can be corrected?
  - what if the frame is never received (e.g. synchronisation problem)?
  - what if the receiver wasn’t ready (e.g. transmitting, or processing the last frame)?
  - we look at the answers to these in this lecture, through a series of examples

- Flow control
  - Prevents a sender out-pacing a slow receiver, by giving feedback on data it can accept
  - Receiver gives feedback on the data it can accept
    * Rare in Link layer as NICs run at “wire speed” (take data as fast as can be sent
  - Flow control is a topic in the Link and Transport layers
Example Protocols

Simplex Communication
Example Protocols

- Commonly implemented as Network Interface Card (NIC) and Operating Systems (OS) drivers
- Network layer (IP) is often OS software
Example Protocols

• Assumptions
  – The transmitter wants to send a long data stream to the receiver, using a reliable connection-oriented service
  – The transmitter has an infinite supply of data and never has to wait for it to be produced
  – A checksum/CRC is applied in hardware at the PHY/DLL interface (hence we don’t consider it in the algorithms)
  – The machines don’t crash!

• Process
  – The DLL receives packets from the NET; encapsulated into a frame adding header and footer, and passed to the PHY.
  – The DLL receives packets from the PHY; checked, ‘de’-encapsulated and passed to the NET.
P1: Utopia

- An optimistic protocol as a starting point
  - Assumes no errors, data always available, receiver as fast as sender etc
  - Considers one-way data transfer
    - Like an unacknowledged connectionless service; assumes higher layers handle all!
    - That’s it, no error or flow control ...

Sender loops blasting frames

Receiver loops eating frames
P2: Stop-and-Wait (Error Free)

- Assumes no errors, and receiver as fast as sender
- Receiver returns a dummy frame (ACK) when ready.
- Hence, only one frame out at any one time, i.e. called stop-and-wait
- Protocol ensures sender can’t outpace receiver: we added flow control!

```
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
P3: Stop-and-Wait (Noisy Channel)

- If we have a noisy channel, three things could happen
  - The frame is successfully received at the receiver, and the ACK is successfully received at the transmitter. Everything is fine.
  - The frame is successfully received at the receiver, but the ACK is corrupted/lost and hence not received at the transmitter. The system hangs.
  - The frame is corrupted/lost and hence not received at the receiver, and hence the ACK is never transmitted (or received at the transmitter). The system hangs.

- Why not use the dummy ACK packets to determine whether or not the frame needs to be resent?
  - The sender starts a timer whenever it transmits a frame
  - If an ACK is received before the timer expires, the next frame is transmitted
  - If an ACK isn’t received when the timer expires, the same frame is retransmitted

- This is known as ARQ (Automatic Repeat reQuest)
P3: Stop-and-Wait (Noisy Channel)

- Normal operation:
P3: Stop-and-Wait (Noisy Channel)

- Lost frame:

- Looks good – so what’s the problem?
P3: Stop-and-Wait (Noisy Channel)

- Lost acknowledgement:

  transmitter
  
  timer starts
  
  frame 1
  
  receiver
  
  ack
  
  timer expiration
  
  timer starts
  
  frame 1
  
  ack
  
  timer interrupted
  
  frame 2
  
  successfully received
  
  to NET

- frame 1
  
  successfully received
  
  to NET
P3: Stop-and-Wait (Noisy Channel)

- We therefore need to number frames and acknowledgements
  - Else receiver can’t tell retransmission (due to lost ACK/early timer) from new frame
  - For stop-and-wait, 2 numbers (1 bit) are sufficient

- These numbers are *sequence numbers*

- For a frame/transmitter
  - The sequence number indicates the frame it is currently trying to transmit

- For an ACK/receiver
  - The sequence number indicates the frame that the receiver is ready to receive
P3: Stop-and-Wait (Noisy Channel)

- With sequence numbers (in square brackets):

Transmitter

- timer starts
- frame 1 [0]
- timer interrupted
- timer starts
- frame 2 [1]
- timer interrupted

Receiver

- frame 1 successfully received
- frame 2 successfully received
- ack [1]
- ack [0]

[0] [0] [1] [0]
P3: Stop-and-Wait (Noisy Channel)

- With sequence numbers (in square brackets):

  - Timer starts
  - Frame 1 [0]
  - Timer expires
  - Timer starts
  - Frame 1 [0]
  - Ack [1]
  - Frame 1 successfully received
  - Timer interrupted
  - Frame 1 never received
P3: Stop-and-Wait (Noisy Channel)

- With sequence numbers (in square brackets):

  - Frame 1 [0] transmitted
  - Timer starts
  - Frame 1 [0] successfully received
  - Ack [1]
  - Timer expires
  - Frame 1 [0] received and discarded
  - Timer interrupted
  - Ack [1] received
  - Frame 1 [0] retransmitted
P3: Stop-and-Wait (Noisy Channel)

Sender loop:

```c
void sender3(void) {
    seq_nr next_frame_to_send;
    frame s;
    packet buffer;
    event_type event;
    next_frame_to_send = 0;
    from_network_layer(&buffer);
    while (true) {
        s.info = buffer;
        s.seq = next_frame_to_send;
        to_physical_layer(&s);
        start_timer(s.seq);
        wait_for_event(&event);
        if (event == frame_arrival) {
            from_physical_layer(&s);
            if (s.ack == next_frame_to_send) {
                stop_timer(s.ack);
                from_network_layer(&buffer);
                inc(next_frame_to_send);
            }
        }
    }
}
```

- 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)
P3: Stop-and-Wait (Noisy Channel)

- Receiver loop:

```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);
    }
}
```
Design consideration: Timeout

- **Too short**: inefficient as frames are unnecessarily retransmitted
- **Too long**: inefficient when frame lost
- We need to find the Goldilocks solution!

(\textit{Neither too short nor too long})
Example Protocols

Duplex Communication
Duplex Data Transfer

• Although there’s been bi-direction communication, data transfer has only really been one way (left $\rightarrow$ right)
  – Anything going from left $\rightarrow$ right was a frame
  – Anything going from right $\rightarrow$ left was an ACK

• We can extend this by communicating frames and ACKs in both directions
  – And using a field in the frame header to indicate what type it is

• Sending an ACK is also reasonably inefficient
  – If there is a frame to send, we can piggyback an ACK into a frame header
Sliding Window Protocols

- Sender maintains window of frames it can send
  - Needs to buffer them for possible retransmission
  - Window advances with next acknowledgements

- Receiver maintains window of frames it can receive
  - Needs to keep buffer space for arrivals
  - Window advances with in-order arrivals
P4: One-Bit Sliding Window

- The same as Stop-and-Wait ARQ, but with non-binary sequence numbers
  - Here, we have a 3-bit sequence number with a sliding window of size 1

---

Diagram:

- Transmitter
  - timer starts
  - timer interrupted
- Receiver
  - frame 1 [0]
  - frame 1 successfully received
  - ack [1]
P4: One-Bit Sliding Window

- Transmitter
  - Whenever the NET passes a packet to the DLL, it is given the next highest sequence number and the upper edge of the window is incremented by 1
  - If the window grows to its maximize size (in this case, 1!), the DLL forcibly stops the NET passing it any more packets
  - Whenever an acknowledgement is received, the lower edge is advanced by 1

- Receiver
  - Any frame received that falls within the current window is put in the frame buffer
  - If a frame corresponding to the lower edge of the window is received, it is passed to the NET and the window rotated by 1
  - If a frame with a sequence number outside the window is received, it is discarded
P4: One-Bit Sliding Window

- Transfers data in both directions with stop-and-wait
  - **Piggybacks** acks on reverse data frames for efficiency
  - Handles transmission errors, flow control, early timers
- Each node is both a sender and receiver:

```c
void protocol4 (void) {
    seq_nr next_frame_to_send;
    seq_nr frame_expected;
    frame r, s;
    packet buffer;
    event_type event;
    next_frame_to_send = 0;
    frame_expected = 0;
    from_network_layer(&buffer);
    s.info = buffer;
    s.seq = next_frame_to_send;
    s.ack = 1 – frame_expected;
    to_physical_layer(&s);
    start_timer(s.seq);
}
```

1. Prepare first frame
2. Launch it, and set timer
P4: One-Bit Sliding Window

Wait for frame or timeout

If a frame with new data then deliver it

If an ACK for last send then prepare for next data frame

(Otherwise it was a timeout)

Send next data frame or retransmit old one; ACK the last data we received

```c
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);
        }
        if (r.ack == next_frame_to_send) {
            stop_timer(r.ack);
            from_network_layer(&buffer);
            inc(next_frame_to_send);
        }
    }
    s.info = buffer;
    s.seq = next_frame_to_send;
    s.ack = 1 - frame_expected;
    to_physical_layer(&s);
    start_timer(s.seq);
}
```
Sliding Window Protocols

• So if it’s the same as Stop-and-Wait ARQ, why bother with the additional complexity?

• Suppose we use this 1-bit sliding window protocol to transmit 1000-bit frames.
  – t=0ms: sender starts transmitting first frame
  – t=20ms: first frame sent
  – t=270ms: first frame received by satellite
  – t=520ms: sender received ACK for first frame (assuming a zero length acknowledgement)
  – t=520ms, sender starts transmitting the second frame

• Sender was ‘blocked’ for 500/520ms (i.e. 96%)
  – Only 4% of the available bandwidth was used
Sliding Window Protocols

- To resolve this, instead of waiting for one frame to be sent-received-acknowledged before sending the next...
- ...we want to be able to queue up lots of frames being sent and awaiting acknowledgements
  - Effectively, pipelining!

- Bandwidth-Delay = bandwidth [bps] * one-way transmit time [s]
- BD = Bandwidth-Delay [b] / frame size [b]
- An appropriate sliding-window size can be obtained by using:
  \[ w = 2BD + 1 \]
  - In the previous example, we would like the transmitter to have just finished sending frame 26 when the first acknowledgement comes back.
Sliding Window Protocols

• This leads to different ways to handling errors:
  – Go-back-N
  – Selective Repeat

• Trade bandwidth for memory/buffers
P5: Go-Back-N

- Receiver only accepts/ACKS frames that arrive in order:
  - A receive window of size $N$
  - Discards frames that follow a missing/errored frame
  - Sender times out and resends all outstanding frames
P5: Go-Back-N

- Go-back-N
  - Sender has to buffer all frame, in case it has to ‘go-back’
  - Receiver doesn’t buffer frames (just waits for the one that it is expecting)
  - Good if errors are rare. If common, wastes a lot of bandwidth through retransmission
Acknowledgments

- Acknowledgements
  - Positive acknowledgement (ACK):
    - sent by a receiver, to the transmitter, after successful receipt of a frame
  - Negative acknowledgement (NAK):
    - sent by a receiver, to the transmitter, after unsuccessful receipt of a frame
P6: Selective Repeat

- Receiver accepts frames anywhere in receive window
  - Works the same as Go-back-N, if an erroneous frame is received, it is discarded.
  - However, any good frames received after it are buffered (i.e. receiver window > 1)
  - When the sender times out, only the oldest unacknowledged frame is retransmitted
P6: Selective Repeat

- Receiver accepts frames anywhere in receive window
  - Can improve efficiency by sending a NAK if an error is detected, which causes the sender to retransmit a missing frame before a timeout
  - Cumulative ACK indicates highest in-order frame
  - Each sequence number in the window has a receive buffer associated with it, and a bit depicting whether that buffer is full or empty
P6: Selective Repeat

• Go-back-N
  – Sender has to buffer all frame, in case it has to ‘go-back’
  – Receiver doesn’t buffer frames (just waits for the one that it is expecting)
  – Good if errors are rare. If common, wastes a lot of bandwidth through retransmission

• Selective Repeat
  – More complex than Go-back-N due to receiver buffers and multiple sender timers
  – More efficient use of link bandwidth as only lost frames are resent
  – Good if errors are common. Requires a lot more memory/buffers
Questions?