

# DLL Protocols

Rakesh Matam

Indian Institute of Information Technology Guwahati

*rakesh@iiitg.ac.in*

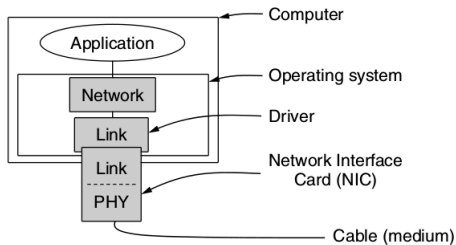
August 26, 2019

# Elementary Data Link Layer Protocols

## Assumptions:

- 1. we assume that the physical layer, data link layer, and network layer are independent processes that communicate by passing messages back and forth.
- The physical layer process and some of the data link layer process run on dedicate hardware called a NIC (Network Interface Card).
- The rest of the link layer process and the network layer process run on the main CPU as part of the operating system, with the software for the link layer process often taking the form of a **device driver**.

# Elementary Data Link Layer Protocols



# Elementary Data Link Layer Protocols

## Assumptions:

- 2. Machine **A** wants to send a long stream of data to machine **B**, using a reliable, connection-oriented service.
- Later, we will consider the case where **B** also wants to send data to **A** simultaneously.
- 3. We also assume that machines do not crash. That is, these protocols deal with communication errors, but not the problems caused by computers crashing and rebooting.

# Elementary Data Link Layer Protocols

## Assumptions:

- 2. Machine **A** wants to send a long stream of data to machine **B**, using a reliable, connection-oriented service.
- Later, we will consider the case where **B** also wants to send data to **A** simultaneously.
- 3. We also assume that machines do not crash. That is, these protocols deal with communication errors, but not the problems caused by computers crashing and rebooting.

# Preliminaries:

- 1. When the data link layer accepts a packet, it encapsulates the packet in a frame by adding a **data link header** and **trailer** to it.
- The frame is then transmitted to the data link layer on the other machine.
- There exists suitable library procedures *to\_physical\_layer* to send a frame and *from\_physical\_layer* to receive a frame.
- These procedures compute and append or check the checksum.

# Preliminaries:

- Initially, the receiver has nothing to do. It just sits around waiting for something to happen.
- This procedure (*wait\_for\_event(&event)*) only returns when something has happened (e.g., a frame has arrived).
- When a frame arrives at the receiver, the checksum is recomputed. If the checksum in the frame is incorrect (i.e., there was a transmission error), the data link layer is so informed (*event=cksum\_err*).
- If the inbound frame arrived undamaged, the data link layer is also informed (event = *frame\_arrival*) so that it can acquire the frame for inspection using *from\_physical\_layer*.
- As soon as the receiving data link layer has acquired an undamaged frame, it checks the **control information in the header**, and, if everything is all right, **passes the packet portion** to the network layer.

# Preliminaries:

```
#define MAX_PKT 1024                                /* determines packet size in bytes */

typedef enum {false, true} boolean;                  /* boolean type */
typedef unsigned int seq_nr;                          /* sequence or ack numbers */
typedef struct {unsigned char data[MAX_PKT];} packet; /* packet definition */
typedef enum {data, ack, nak} frame_kind;             /* frame_kind definition */

typedef struct {                                      /* frames are transported in this layer */
    frame_kind kind;                                /* what kind of frame is it? */
    seq_nr seq;                                     /* sequence number */
    seq_nr ack;                                     /* acknowledgement number */
    packet info;                                    /* the network layer packet */
} frame;
```



# Preliminaries:

- A *seq\_nr* is a small integer used to number the frames so that we can tell them apart.
- These sequence numbers run from 0 up to and including *MAX\_SEQ*, which is defined in each protocol needing it.
- A *packet* is the unit of information exchanged between the network layer and the data link layer on the same machine, or between network layer peers.
- In our model it always contains *MAX\_PKT* bytes, but more realistically it would be of variable length.
- A *frame* is composed of four fields: *kind*, *seq*, *ack*, and *info*.
- The first three of which contain control information and the last of which may contain actual data to be transferred. These control fields are collectively called the frame header.

# Preliminaries:

Group	Library Function	Description
Network layer	from_network_layer(&packet) to_network_layer(&packet) enable_network_layer() disable_network_layer()	Take a packet from network layer to send Deliver a received packet to network layer Let network cause “ready” events Prevent network “ready” events
Physical layer	from_physical_layer(&frame) to_physical_layer(&frame)	Get an incoming frame from physical layer Pass an outgoing frame to physical layer
Events & timers	wait_for_event(&event) start_timer(seq_nr) stop_timer(seq_nr) start_ack_timer() stop_ack_timer()	Wait for a packet / frame / timer event Start a countdown timer running Stop a countdown timer from running Start the ACK countdown timer Stop the ACK countdown timer

# Preliminaries:

- To be able to recover from the event of **frame loss**, the sending data link layer must start an internal timer or clock whenever it sends a frame.
- If no reply has been received within a certain predetermined time interval, the clock times out and the data link layer receives an interrupt signal.
- The procedures *start\_timer* and *stop\_timer* turn the timer on and off, respectively.
- A **packet** is the unit of information exchanged between the network layer and the data link layer on the same machine, or between network layer peers.
- In our model it always contains *MAX\_PKT* bytes, but more realistically it would be of variable length.
- A **frame** is composed of four fields: *kind*, *seq*, *ack*, and *info*.
- The first 3 of which contain control information and the last of which may contain actual data to be transferred.

# P1: A utopian simplex protocol

An optimistic protocol (p1) to get us started

- Assumes no errors, and receiver as fast as sender
- Considers one-way data transfer

```
void sender1(void)
{
    frame s;
    packet buffer;

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

Sender loops blasting frames

```
void receiver1(void)
{
    frame r;
    event_type event;

    while (true) {
        wait_for_event(&event);
        from_physical_layer(&r);
        to_network_layer(&r.info);
    }
}
```

Receiver loops eating frames

- That's it, no error or flow control ...

## P2: A simplex stop-and-wait protocol for an error-free channel

- One solution is to build the receiver to be powerful enough to process a continuous stream of back-to-back frames.
- A more general solution is to have the receiver provide feedback to the sender.
- After having passed a packet to its network layer, the receiver sends a little dummy frame back to the sender which, in effect, gives the sender permission to transmit the next frame.
- After having sent a frame, the sender is required by the protocol to bide its time until the little dummy (i.e., acknowledgement) frame arrives. This delay is a simple example of a **flow control protocol**.

## P2: A simplex stop-and-wait protocol for an error-free channel

```
void sender2(void)
{
    frame s;                                /* buffer for an outbound frame */
    packet buffer;                          /* buffer for an outbound packet */
    event_type event;                       /* frame_arrival is the only possibility */

    while (true) {
        from_network_layer(&buffer);        /* go get something to send */
        s.info = buffer;                   /* copy it into s for transmission */
        to_physical_layer(&s);              /* bye-bye little frame */
        wait_for_event(&event);             /* do not proceed until given the go ahead */
    }
}

void receiver2(void)
{
    frame r, s;                             /* buffers for frames */
    event_type event;                       /* frame_arrival is the only possibility */
    while (true) {
        wait_for_event(&event);             /* only possibility is frame_arrival */
        from_physical_layer(&r);            /* go get the inbound frame */
        to_network_layer(&r.info);          /* pass the data to the network layer */
        to_physical_layer(&s);              /* send a dummy frame to awaken sender */
    }
}
```

## P3: Stop-and-Wait protocol for noisy channel

- Frames may be either damaged or lost completely.
- If a frame is damaged in transit, the receiver hardware will detect this while it computes checksum.
- What can we do?
- The sender could send a frame, but the receiver would only send an acknowledgement frame if the data were correctly received. If a damaged frame arrived at the receiver, it would be discarded.

After a while the sender would time out and send the frame again. This process would be repeated until the frame finally arrived intact.

## P3: Stop-and-Wait protocol for noisy channel

- The goal of the data link layer is to provide error-free, transparent communication between network layer processes.
- The network layer on machine A gives a series of packets to its data link layer, which must ensure that an identical series of packets is delivered to the network layer on machine B by its data link layer.

In particular, the network layer on B has no way of knowing that a packet has been lost or duplicated, so the data link layer must guarantee that no combination of transmission errors, however unlikely, can cause a duplicate packet to be delivered to a network layer.



## P3: Stop-and-Wait protocol for noisy channel

Consider the following scenario:

1. The network layer on *A* gives packet 1 to its data link layer. The packet is correctly received at *B* and passed to the network layer on *B*. *B* sends an acknowledgement frame back to *A*.
2. The acknowledgement frame gets lost completely. It just never arrives at all. Life would be a great deal simpler if the channel managed and lost only data frames and not control frames, but sad to say, the channel is not very discriminating.
3. The data link layer on *A* eventually times out. Not having received an acknowledgement, it (incorrectly) assumes that its data frame was lost or damaged and sends the frame containing packet 1 again.
4. The duplicate frame also arrives intact at the data link layer on *B* and is unwittingly passed to the network layer there. If *A* is sending a file to *B*, part of the file will be duplicated (i.e., the copy of the file made by *B* will be incorrect and the error will not have been detected). In other words, the protocol will fail.

## P3: Stop-and-Wait protocol for noisy channel

- Clearly, what is needed is some way for the receiver to be able to distinguish a frame that it is seeing for the first time from a retransmission.
- The obvious way to achieve this is to have the sender put a sequence number in the header of each frame it sends.

Then the receiver can check the sequence number of each arriving frame to see if it is a new frame or a duplicate to be discarded.

What should be the *MAX\_SEQ* of the sequence number?

## P3: Stop-and-Wait protocol for noisy channel

- Protocols in which the sender waits for a positive acknowledgement before advancing to the next data item are often called ARQ (Automatic Repeat reQuest) or PAR (Positive Acknowledgment with Retransmission).
- The obvious way to achieve this is to have the sender put a sequence number in the header of each frame it sends.

Then the receiver can check the sequence number of each arriving frame to see if it is a new frame or a duplicate to be discarded.

What should be the *MAX\_SEQ* of the sequence number?

# P3: Stop-and-Wait protocol for noisy channel

```
void sender3(void)
{
    seq_nr next_frame_to_send;          /* seq number of next outgoing frame */
    frame s;                            /* scratch variable */
    packet buffer;                      /* buffer for an outbound packet */
    event_type event;

    next_frame_to_send = 0;             /* initialize outbound sequence numbers */
    from_network_layer(&buffer);        /* fetch first packet */
    while (true) {
        s.info = buffer;                /* construct a frame for transmission */
        s.seq = next_frame_to_send;     /* insert sequence number in frame */
        to_physical_layer(&s);          /* send it on its way */
        start_timer(s.seq);             /* if answer takes too long, time out */
        wait_for_event(&event);         /* frame_arrival, cksum_err, timeout */
        if (event == frame_arrival) {
            from_physical_layer(&s);    /* get the acknowledgement */
            if (s.ack == next_frame_to_send) {
                stop_timer(s.ack);      /* turn the timer off */
                from_network_layer(&buffer); /* get the next one to send */
                inc(next_frame_to_send); /* invert next_frame_to_send */
            }
        }
    }
}
```

# P3: Stop-and-Wait protocol for noisy channel

```
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);
        }
    }
}
```

/\* possibilities: frame\_arrival, cksum\_err \*/  
/\* a valid frame has arrived \*/  
/\* go get the newly arrived frame \*/  
/\* this is what we have been waiting for \*/  
/\* pass the data to the network layer \*/  
/\* next time expect the other sequence nr \*/  
/\* tell which frame is being acked \*/  
/\* send acknowledgement \*/

# Sliding Window Protocols

- Using same link for data in both directions.
- Using something like a **kind** field in the header of an incoming frame, the receiver can tell whether the frame is data or an acknowledgement.
- Further improvement can be achieved by **piggybacking**.

# Sliding Window Protocols

- Each outbound frame contains a sequence number, ranging from 0 to  $2^n-1$ . This is so that the sequence number fits exactly in an n-bit field.
- The essence of all sliding window protocols is that at any instant of time, the sender maintains a set of sequence numbers corresponding to frames it is permitted to send.
- These frames are said to fall within the **sending window**.

# Sliding Window Protocols

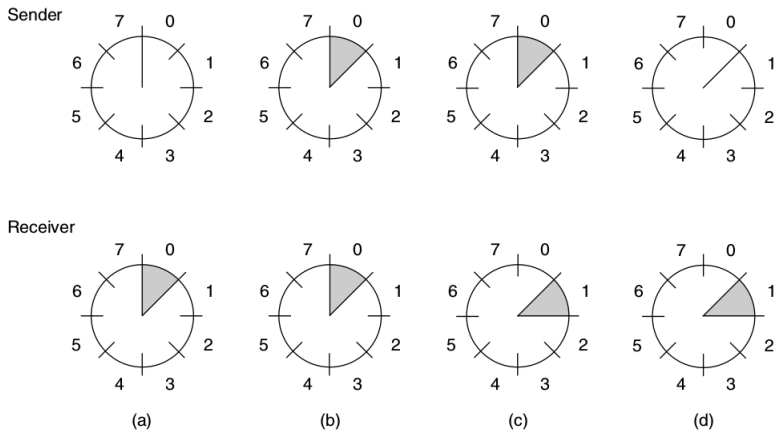
- The receiver also maintains a **receiving window** corresponding to the set of frames it is permitted to accept.
- The sender's window and the receiver's window need not have the same lower and upper limits or even have the same size.



# Sliding Window Protocols

- The sequence numbers within the sender's window represent frames that have been sent or can be sent but are as yet not acknowledged.
- Whenever a new packet arrives from the network layer, it is given the next highest sequence number, and the upper edge of the window is advanced by one.
- When an acknowledgement comes in, the lower edge is advanced by one. In this way the window continuously maintains a list of unacknowledged frames.

# Sliding Window Protocols



**Figure 3-15.** A sliding window of size 1, with a 3-bit sequence number. (a) Initially. (b) After the first frame has been sent. (c) After the first frame has been received. (d) After the first acknowledgement has been received.

# Sliding Window Protocols

- Since frames currently within the sender's window may ultimately be lost or damaged in transit, the sender must keep all of these frames in its memory for possible retransmission.
- Thus, if the maximum window size is  $n$ , the sender needs  $n$  buffers to hold the unacknowledged frames.
- If the window ever grows to its maximum size, the sending data link layer must forcibly shut off the network layer until another buffer becomes free.

# Sliding Window Protocols

- The receiving data link layer's window corresponds to the frames it may accept. Any frame falling within the window is put in the receiver's buffer.
- When a frame whose sequence number is equal to the lower edge of the window is received, it is passed to the network layer and the window is rotated by one.
- Any frame falling outside the window is discarded.
- In all of these cases, a subsequent acknowledgement is generated so that the sender may work out how to proceed.

# One bit Sliding Window Protocol

- The sender transmits a frame and waits for its acknowledgement before sending the next one.
- *Next\_frame\_to\_send* tells which frame the sender is trying to send and *frame\_expected* tells which frame the receiver is expecting. In both cases 0 and 1 are the only possibilities.
- Under normal circumstances, one of the two data link layers goes first and transmits the first frame.
- When this (or any) frame arrives, the receiving data link layer checks to see if it is a duplicate.

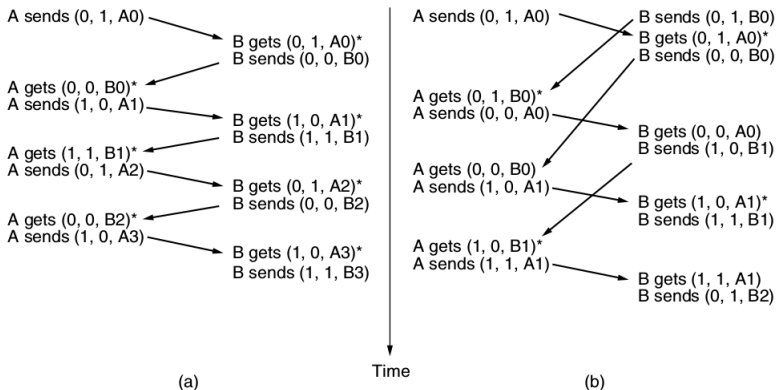
# One bit Sliding Window Protocol

- If the frame is the one expected, it is passed to the network layer and the receiver's window is slid up.
- The acknowledgement field contains the number of the last frame received without error.

# One bit Sliding Window Protocol:Resiliency

- Assume that computer **A** is trying to send its frame 0 to computer **B** and that **B** is trying to send its frame 0 to **A**.

# One bit Sliding Window Protocol



**Figure 3-17.** Two scenarios for protocol 4. (a) Normal case. (b) Abnormal case. The notation is (seq, ack, packet number). An asterisk indicates where a network layer accepts a packet.



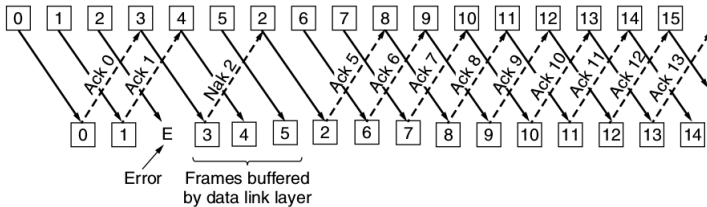
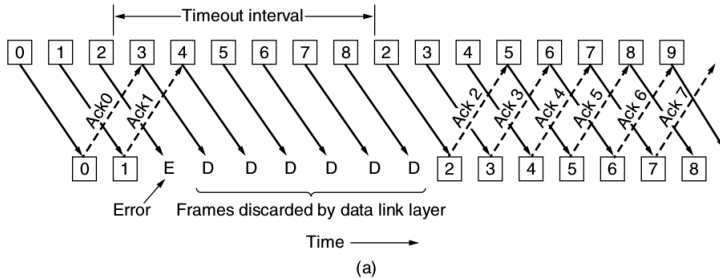
- Till now the assumption was that the transmission time required for a frame to arrive at the receiver plus the transmission time for the acknowledgement to come back is negligible. Which is false.
- In these situations the long round-trip time can have important implications for the efficiency of the bandwidth utilization.
- Using Protocol 4: Evaluate the efficiency of a 50-kbps satellite channel with 500msec round-trip propagation delay? Consider a 1000bit frame.

- The problem described here can be viewed as a consequence of the rule requiring a sender to wait for an acknowledgement before sending another frame.
- The solution lies in allowing the sender to transmit up to  $w$  frames before blocking, instead of just 1.
- With a large enough choice of  $w$  the sender will be able to continuously transmit frames since the acknowledgements will arrive for previous frames before the window becomes full, preventing the sender from blocking.

- To find an appropriate value for  $w$  we need to know how many frames can fit inside the channel as they propagate from sender to receiver.
- This capacity is determined by the bandwidth in bits/sec multiplied by the one-way transit time, or the bandwidth-delay product of the link.

- The technique of keeping multiple frames in flight is an example of pipelining.
- Pipelining frames over an unreliable communication channel raises some serious issues. First, what happens if a frame in the middle of a long stream is damaged or lost?

# Go-back-n

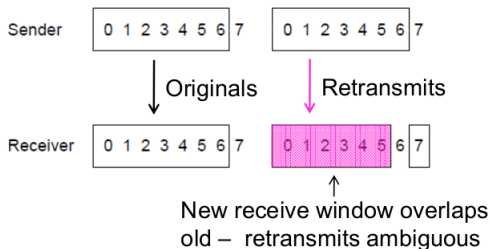


# Go-back-n: Selective Repeat

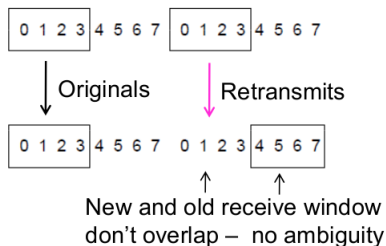
For correctness, we require:

- 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



# The End