Chapter 3

The Data Link Layer (cont…)

So far…..

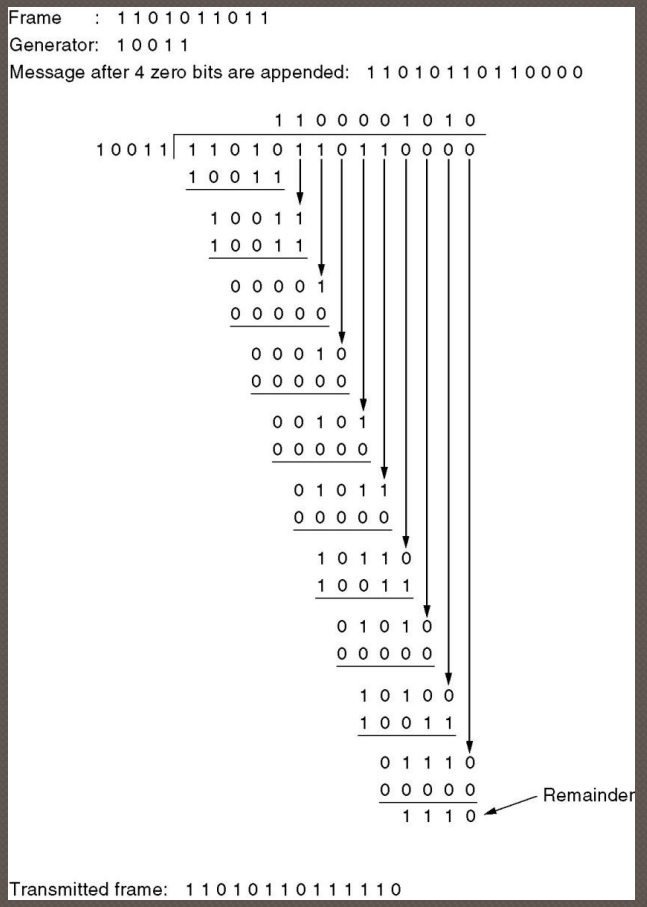
- Framing Techniques
  - Character Count
  - Flag bytes with byte Stuffing
  - Bit stuffing
- Error-Correcting Codes
  - Hamming's Code
- Error-Detecting Codes

Error-Detecting Codes

Calculation of the polynomial code checksum -CRC (Cyclic Redundancy Check)

1. Sender and receiver agree on a generator polynomial , degree *r*
   - Ex. G(x) =x^4+x+x^0 i.e.10011
   1. Generator polynomial must start and end with 1
2. Append *r* zero bits to the frame M(x) and divide it by G(x)
3. Subtract the remainder from the dividend. This is T(x) the transmitted bits
4. If T(x)+E(x) is received, error occurred

- Complex calculation but a simple shift register circuit is used in hardware.



Error Detection

- $[T(x)+E(x)]/G(x) = T(x)/G(x)+ E(x)/G(x) = 0 + E(x)/G(x)$
- If $E(x)/G(x) \neq 0$

How to select a good G(x)?

If G(x) has at least two terms all single bit-errors will be detected.

If G(x) has at least one factor with 3 terms all double bit-errors will be detected.

If G(x) has the factor (x+1) any odd number of errors will be detected.

Any burst of errors can be detected as long as it is less than k.

How to select a good G(x)? - Proof

If G(x) has at least two terms all single bit-errors will be detected.

- $E(x)/G(x) \neq 0$ if $E(x) = x^i$

If G(x) has at least one factor with 3 terms all double bit-errors will be detected.

- $E(x) = (x^i + x^j) = x^j(x^{i-j}+1)$
- Thus for 2 bits errors will have factors with at most 2 terms
- Any G(x) with 3 terms will not divide by x^k or $(x^{kj}+1)$

How to select a good G(x)? - Proof

If G(x) has the factor (x+1) any odd number of errors will be detected.

- No module 2 polynomial exists which adds odd number of terms and divides by (x+1)
- Assume that E(x) has an odd number of terms and it is divisible by (x+1), then E(x) can be written as

  E(x)= (x+1)/Q(x) and E(1)=(1+1)Q(1) mod 2 = 0
- But if E(x) has odd number of terms then E(I)= 1+1+1..+1 (odd number of ones) = 1, so no polynomial with a odd number of terms is divisible by (x+1)

Any burst of errors can be detected as long as it is less than k.

- For burst error <k bits at ith location from right $E(x)=x^i(x^{k-1}+\ \ +1)$. It is not divisible by G(x) which has last term 1 and first term x^k

Some good generators

- IEEE 802: $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4++x^2+x+1$
- CRC-8: x^8+x^2+x+1
- CRC-10: $x^{10}+x^9+x^5+x^4+x+1$
- CRC-12: $x^{12}+x^{11}+x^3+x^2+x+1$
- CRC-16: $x^{16}+x^{15}+x^2+1$
- CRC-CCITT : $x^{16}+x^{12}+x^5+1$

- This seemingly complex scheme can be implemented by very simple hardware [Peterson and Brown,1961] with X-OR and shift register.
- This is the most commonly used Error Handling mechanism in Networking
- Six versions of CRC are widely used in link-level protocols. Ethernet and FDDI networks use CRC-32. HDLC use CRCCCITT. ATM use CRC-8 and CRC-10.

Elementary Data Link Protocols

1. An Unrestricted Simplex Protocol
2. A Simplex Stop-and-Wait Protocol
3. A Simplex Protocol for a Noisy Channel

Protocol Definitions

```
#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 a frame is it? */
  seq_nr seq;                                         /* sequence number */
  seq_nr ack;                                         /* acknowledgement number */
  packet info;                                        /* the network layer packet */
} frame;
```

Continued →

Some definitions needed in the protocols to follow.
These are located in the file protocol.h.

Protocol Definitions (ctd.)

Some definitions needed in the protocols to follow. These are located in the file protocol.h.

```
/* Wait for an event to happen; return its type in event. */
void wait_for_event(event_type *event);

/* Fetch a packet from the network layer for transmission on the channel. */
void from_network_layer(packet *p);

/* Deliver information from an inbound frame to the network layer. */
void to_network_layer(packet *p);

/* Go get an inbound frame from the physical layer and copy it to r. */
void from_physical_layer(frame *r);

/* Pass the frame to the physical layer for transmission. */
void to_physical_layer(frame *s);

/* Start the clock running and enable the timeout event. */
void start_timer(seq_nr k);

/* Stop the clock and disable the timeout event. */
void stop_timer(seq_nr k);

/* Start an auxiliary timer and enable the ack_timeout event. */
void start_ack_timer(void);

/* Stop the auxiliary timer and disable the ack_timeout event. */
void stop_ack_timer(void);

/* Allow the network layer to cause a network_layer_ready event. */
void enable_network_layer(void);

/* Forbid the network layer from causing a network_layer_ready event. */
void disable_network_layer(void);

/* Macro inc is expanded in-line: Increment k circularly. */
#define inc(k) if (k < MAX_SEQ) k = k + 1; else k = 0
```

Protocol 1: Unrestricted Simplex Protocol

Very simple data layer protocol

1. One directional communication only
2. Assume that communication channel is error free
3. Assumes that receiver can process all input infinitely fast.

```
/* Protocol 1 (utopia) provides for data transmission in one direction only, from
   sender to receiver.  The communication channel is assumed to be error free,
   and the receiver is assumed to be able to process all the input infinitely quickly.
   Consequently, the sender just sits in a loop pumping data out onto the line as
   fast as it can. */

typedef enum {frame_arrival} event_type;
#include "protocol.h"

void sender1(void)
{
  frame s;                          /* buffer for an outbound frame */
  packet buffer;                    /* buffer for an outbound packet */

  while (true) {
      from_network_layer(&buffer);  /* go get something to send */
      s.info = buffer;              /* copy it into s for transmission */
      to_physical_layer(&s);        /* send it on its way */
  }                                 /*  * Tomorrow, and tomorrow, and tomorrow,
                                        Creeps in this petty pace from day to day
                                        To the last syllable of recorded time
                                            - Macbeth, V, v */
}

void receiver1(void)
{
  frame r;
  event_type event;                 /* filled in by wait, but not used here */

  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 */
  }
}
```

Protocol 2: Simplex Stop-and-Wait

1. After receiving each frame the receiver sends an ACK frame
   - When ACK arrives next packet can be sent
2. Sender cannot flood the receiver

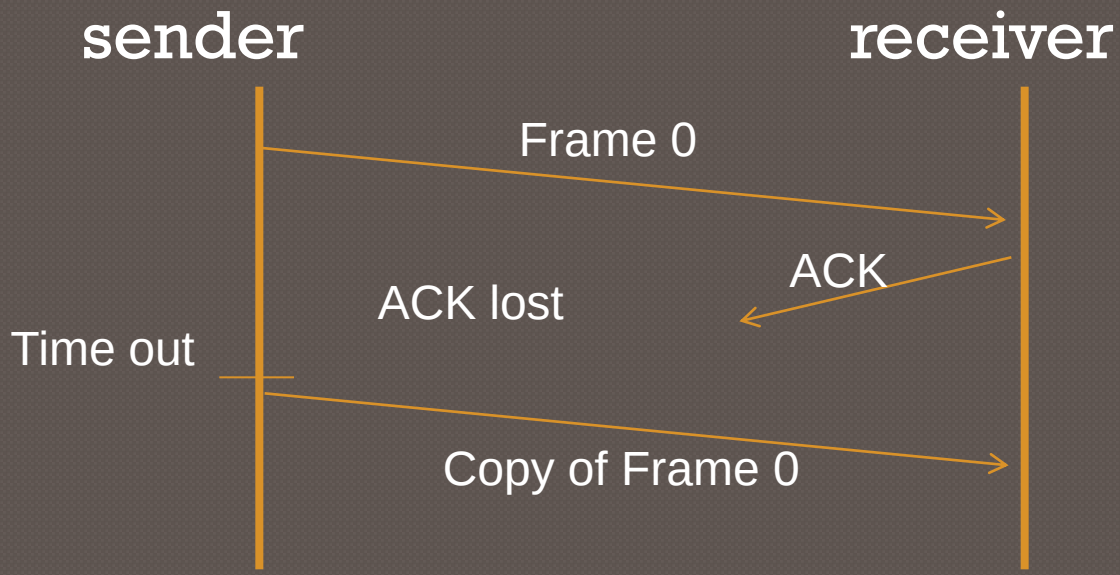
Transmission is still not reliable

What if an ACK is lost?

What is we add a timer?

```
/* Protocol 2 (stop-and-wait) also provides for a one-directional flow of data from
   sender to receiver. The communication channel is once again assumed to be error
   free, as in protocol 1. However, this time, the receiver has only a finite buffer
   capacity and a finite processing speed, so the protocol must explicitly prevent
   the sender from flooding the receiver with data faster than it can be handled. */

typedef enum {frame_arrival} event_type;
#include "protocol.h"

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 */
  }
}
```

The problem of the missing ACK



Frame 0 is sent (and received) twice – the protocol fails!

Protocol 3 with Sequence Number
A Simplex Protocol for a Noisy Channel

1. Sequence numbers are used for frames.
2. After receiving a frame the receiver sends an ACK
   - When ACK arrives next packet in the sequence can be sent
3. When a frame is received, the receiver checks if is a duplicate.
   - Duplicates and damaged frames are discarded.
4. Sender sends copy of the frame, if ACK frame is either damaged or timer has expired

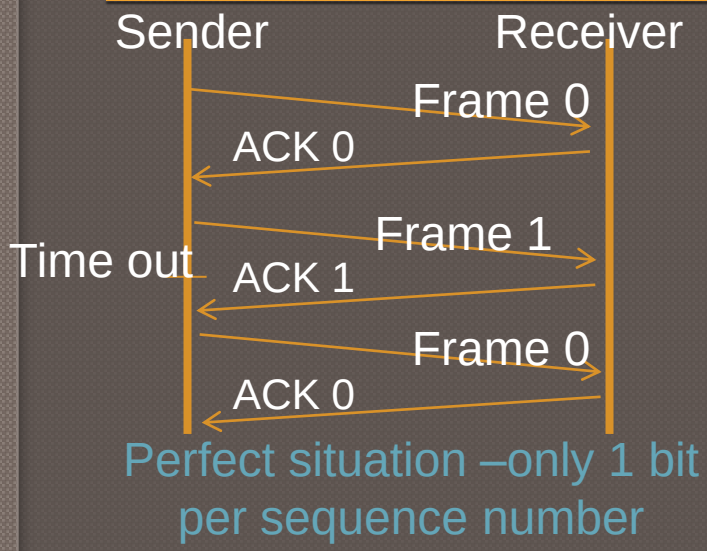
```
/* Protocol 3 (par) allows unidirectional data flow over an unreliable channel. */

#define MAX_SEQ 1                                   /* must be 1 for protocol 3 */
typedef enum  {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

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 */
          }
      }
  }
}
```

A Simplex Protocol for a Noisy Channel (ctd.)

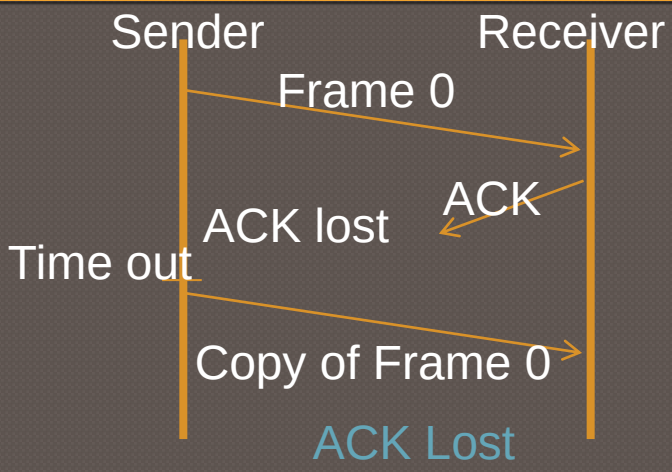
```
void receiver3(void)
{
  seq_nr frame_expected;
  frame r, s;
  event_type event;

  frame_expected = 0;
  while (true) {
       wait_for_event(&event);                  /* possibilities: frame_arrival, cksum_err */
       if (event == frame_arrival) {            /* a valid frame has arrived. */
            from_physical_layer(&r);            /* go get the newly arrived frame */
            if (r.seq == frame  expected) {     /* this is what we have been waiting for. */
                 to_network_layer(&r.info);     /* pass the data to the network layer */
                 inc(frame_expected);           /* next time expect the other sequence nr */
            }
            s.ack = 1 – frame_expected;         /* tell which frame is being acked */
            to_physical_layer(&s);              /* send acknowledgement */
       }
  }
}
```

A positive acknowledgement with retransmission protocol.

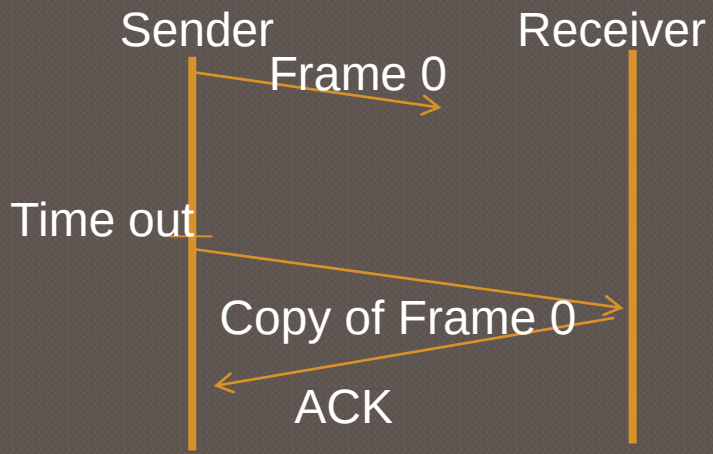
Protocol 3 – Possible situations



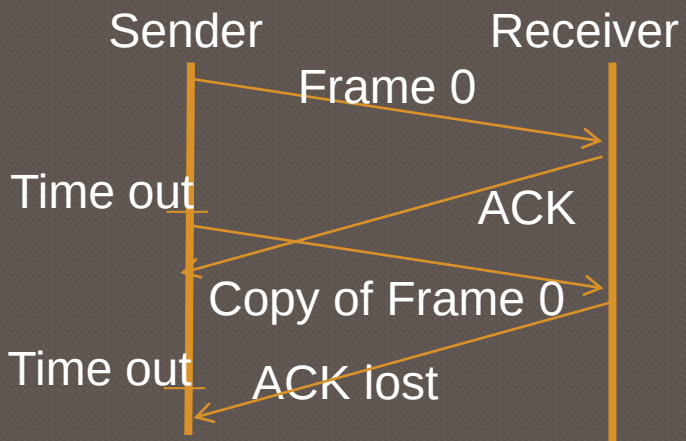
Perfect situation –only 1 bit per sequence number



ACK Lost

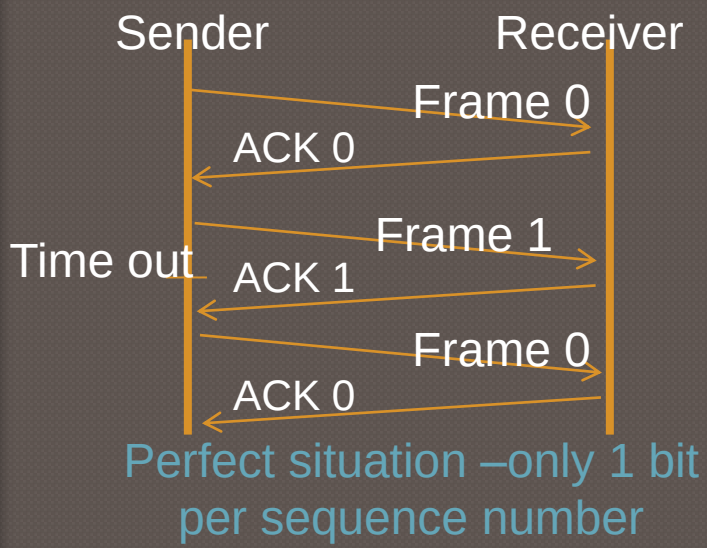


Frame lost



Low Time out value

Transmission Inefficiency of Stop-and-Wait protocols



Perfect situation –only 1 bit per sequence number

- Assume that 1.5Mbps link with 30ms round trip time (RTT)
  - Assume packet of size 1KB
  - A packet can be sent every 45ms
    - i.e.1024*8 every 30ms
    - i.e. (1024*8)/.030 = 273 Kbps
  - Only 1/5th (resp. 2/5th) of the link capacity is used for simplex (resp. duplex)

Next Time

- Exam
- After the exam we will complete the Chapter 3 and start Chapter 4.