

# End Of Error Correction, Flow Control

CS158a

Chris Pollett

Feb 28, 2007.

# Outline

- Finish up Error Detecting and Correcting Codes
- Sliding Window Protocols for Flow Control

# Cyclic Redundancy Check Codes

- These are also known as **polynomial codes**.
- A k-bit frame is viewed as specifying the coefficients of a degree k-1 polynomial.
- For example 101 code  $1x^2 + 1x^0$ .
- Polynomials can be added mod 2, they can also be multiplied and divided.
- In a CRC code, both sender and receiver agree on a generator polynomial  $G(x)$  which has both its high and low order coefficient 1.
- We assume the message m has length  $> \deg(G)=r$ . Let  $M(x)$  be the polynomial of the message.
- To compute a checksum for a message we (1) Compute  $x^r M(x)$ , (the bit sequence of this polynomial is that of  $M(x)$  shifted over r bits). (2) Divide  $x^r M(x)$  by  $G(x)$  to get a remainder  $R(x)$ . (3) Send the coefficients of  $T(x) = x^r M(x) - R(x)$ .

# More on CRC codes.

- Notice  $T(x)$  is divisible by  $G(x)$ .
- Suppose  $T(x)+E(x)$  arrives.
- Then  $(T(x) +E(x))/G(x) = E(x)/G(x)$ .
- The only way an error is undetected is if this value is 0.
- If there was a single bit error then  $E(x)=x^i$  for some  $i$  which can't be divisible by  $G(x)$ . So any single bit error will be detected.
- If there are two errors then  $E(x)= x^i+ x^j$  where  $i>j$ . So  $E(x)= x^j(x^{i-j} +1)$ . So this error will be detected provided that  $G(x)$  does not divide  $x^k+1$  for any  $k$  up to  $i-j$ .
- There are some low degree polynomials that will give this protection to very long frames. For instance,  $x^{15}+ x^{14}+1$  does not divide  $x^k +1$  for any  $k$  below 32768.
- If  $E(X)$  contains an odd number of terms (hence errors), then it cannot have  $x+1$  as a factor mod 2, so if  $x+1$  is a factor of  $G(x)$  we can catch these errors.
- To see this suppose  $E(x)=(x+1)Q(x)$  had an odd number of terms. Then  $E(1)=(1+1)Q(1) = 0 \text{ mod } 2$ . On the other hand substituting 1 for an odd number of terms should give 1 mod 2.
- CRC codes with  $r$  check bits will detect all burst errors of length  $\leq r$ . Such an error would look like  $x^i(x^k +x^{k-1} + \dots +x^0)$ . If  $G(x)$  has a  $x^0$  term, it will not have an  $x^i$  factor, so if the degree of the parenthesized expression is less than  $G(x)$ , the remainder won't be 0.
- IEEE 802 uses the polynomial  
$$x^{32}+ x^{26}+ x^{23}+ x^{22}+ x^{16} + x^{12} + x^{11}+ x^{10}+ x^8+ x^7+ x^5+ x^4+ x^2+ x^1 +1$$

# Data Link Layer Protocols

- We start with some assumptions:
  - We will view the physical layer, data link layer, and network layer as each running in its own process on a machine.
  - Initially we will assume two machine A and B and that A wants to send on a connection-oriented channel a message to B.
  - We assume neither machine ever crashes.
  - We will assume the data link layer does not look at the data inside packets and the network layer is always ready to give a packet to A data link layer.

# Elementary Data Link Protocols

- To start we will consider the following three protocols:
  - An Unrestricted Simplex Protocol
  - A Simplex Stop-and-Wait Protocol
  - 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.

```
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
```



# Some Comments

- In our struct *frame* the *kind* field is needed because some frames such as acknowledgements don't have any data in their info field to be forwarded to the network layer.
- We assume the channel is unreliable and sometimes loses entire frames.
- To recover from this, the data link layer must start an internal clock whenever it sends a frame.
- If no reply is received within a certain period, the clock times out and the data link layer receives an interrupt signal.
- Our *wait\_for\_event* function will in this case return *event=timeout*. *start\_timer* and *stop\_timer* are used to turn on and off this timer.
- The functions *start\_ack\_timer* and *stop\_ack\_timer* control an auxiliary timer used to generate acknowledgements in certain situation.
- The functions *enable\_network\_layer* and *disable\_network\_layer* will be used in some of our more advance protocols for flow control.

# Unrestricted Simplex Protocol

/\* 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 */
}

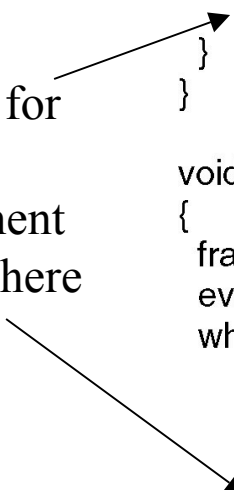
void receiver1(void)
{
    frame r;
    event_type event;                    /* filled in by wait, but not used here */

    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 */
    }
}
```

# Simplex Stop-and- Wait Protocol

Although data is only sent one way, both A and B send frames.

Here we wait for an acknowledgment which is sent here



*/\* 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;  
    packet buffer;  
    event_type event;
```

```
/* buffer for an outbound frame */  
/* buffer for an outbound packet */  
/* frame_arrival is the only possibility */
```

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

```
/* go get something to send */  
/* copy it into s for transmission */  
/* bye bye little frame */  
/* do not proceed until given the go ahead */
```

```
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);  
    }  
}
```

```
/* buffers for frames */  
/* frame_arrival is the only possibility */  
  
/* only possibility is frame_arrival */  
/* go get the inbound frame */  
/* pass the data to the network layer */  
/* send a dummy frame to awaken sender */
```

# A Simplex Protocol for a Noisy

In this example, unlike simplex stop and wait, frames may be lost, so we need to timeout if we don't get an acknowledgement

This is called a **positive acknowledgement with retransmission (PAR)** or **Automatic Repeat ReQuest (ARQ)** protocol because the sender waits for a positive ack before advancing to next frame.

```
/* 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 */
            }
        }
    }
}
```

Continued →

# A Simplex Protocol for a Noisy Channel

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

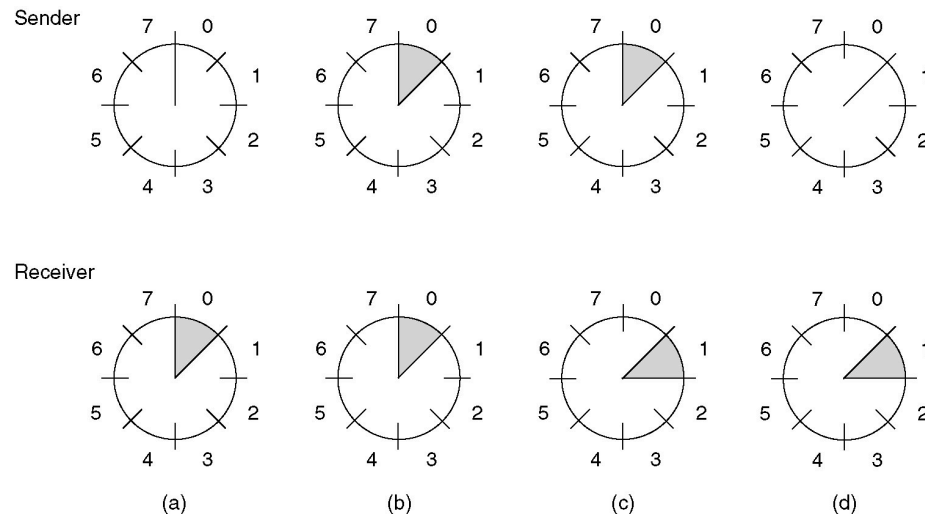
    frame_expected = 0;
    while (true) {
        wait_for_event(&event);           /* possibilities: frame_arrival, cksum_err */
        if (event == frame_arrival) {    /* a valid frame has arrived. */
            from_physical_layer(&r);     /* go get the newly arrived frame */
            if (r.seq == 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.

# Sliding Window Protocols

- We next want to consider protocols that allow for a full duplex channel. i.e., both A and B can send data.
- We will consider:
  - A One-Bit Sliding Window Protocol
  - A Protocol Using Go Back N
  - A Protocol Using Selective Repeat
- In these protocols we both A and B may be sending data frames and acknowledgement frames. We will use the kind field of our *frame* struct to say which.
- To further speed things up, when a data frame arrives rather than immediately sending a control frame acknowledging it, the receiver restrains itself and waits until its network layer want to send a packet. Then it sends both this packet and the acknowledgement in the same frame. (**piggybacking**).
- If no frame controls from the receiver's network layer up till some timeout period then it just sends an acknowledgement.

# Sliding Window Protocols (2)



- In a sliding window protocol each outbound frame contains a sequence number from 0 to  $2^n-1$ .
- The sender maintains a set of sequence numbers it is permitted to resend.
- The receiver maintains a set of sequence numbers it is permitted to accept and pass on to its network layer.

A sliding window of size 1, with a 3-bit sequence number.

- (a) Initially, the sender has sent no frames so it is not allowed to resend any; the receiver is permitted to accept a frame with seq number 0. The sender can send a frame with seq=0 for the first time.
- (b) After the first frame has been sent, before it is received. Sender is allowed to resend this frame with seq=0 in the event of a timeout.
- (c) After the first frame has been received, it is passed to the receiver's network layer. So that the receiver does not give the same frame to its network layer twice the window is advanced.
- (d) After the first acknowledgement has been received. Now sender is no longer allowed to resend frame 0. So it will try to send frame 1 for the first time and the whole process starts over.

# A One-Bit Sliding Window Protocol

- Here both the window size and number of bits for the sequence number

```
is 1
/* Protocol 4 (sliding window) is bidirectional. */
#define MAX_SEQ 1 /* must be 1 for protocol 4 */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
    seq_nr next_frame_to_send; /* 0 or 1 only */
    seq_nr frame_expected; /* 0 or 1 only */
    frame r, s; /* scratch variables */
    packet buffer; /* current packet being sent */
    event_type event;

    next_frame_to_send = 0; /* next frame on the outbound stream */
    frame_expected = 0; /* frame expected next */
    from_network_layer(&buffer); /* fetch a packet from the network layer */
    s.info = buffer; /* prepare to send the initial frame */
    s.seq = next_frame_to_send; /* insert sequence number into frame */
    s.ack = 1 - frame_expected; /* piggybacked ack */
    to_physical_layer(&s); /* transmit the frame */
    start_timer(s.seq); /* start the timer running */
}
```

Continued →



# A One-Bit Sliding Window Protocol

```
while (true) {
    wait_for_event(&event);           /* frame_arrival, cksum_err, or timeout */
    if (event == frame_arrival) {    /* a frame has arrived undamaged. */
        from_physical_layer(&r);     /* go get it */

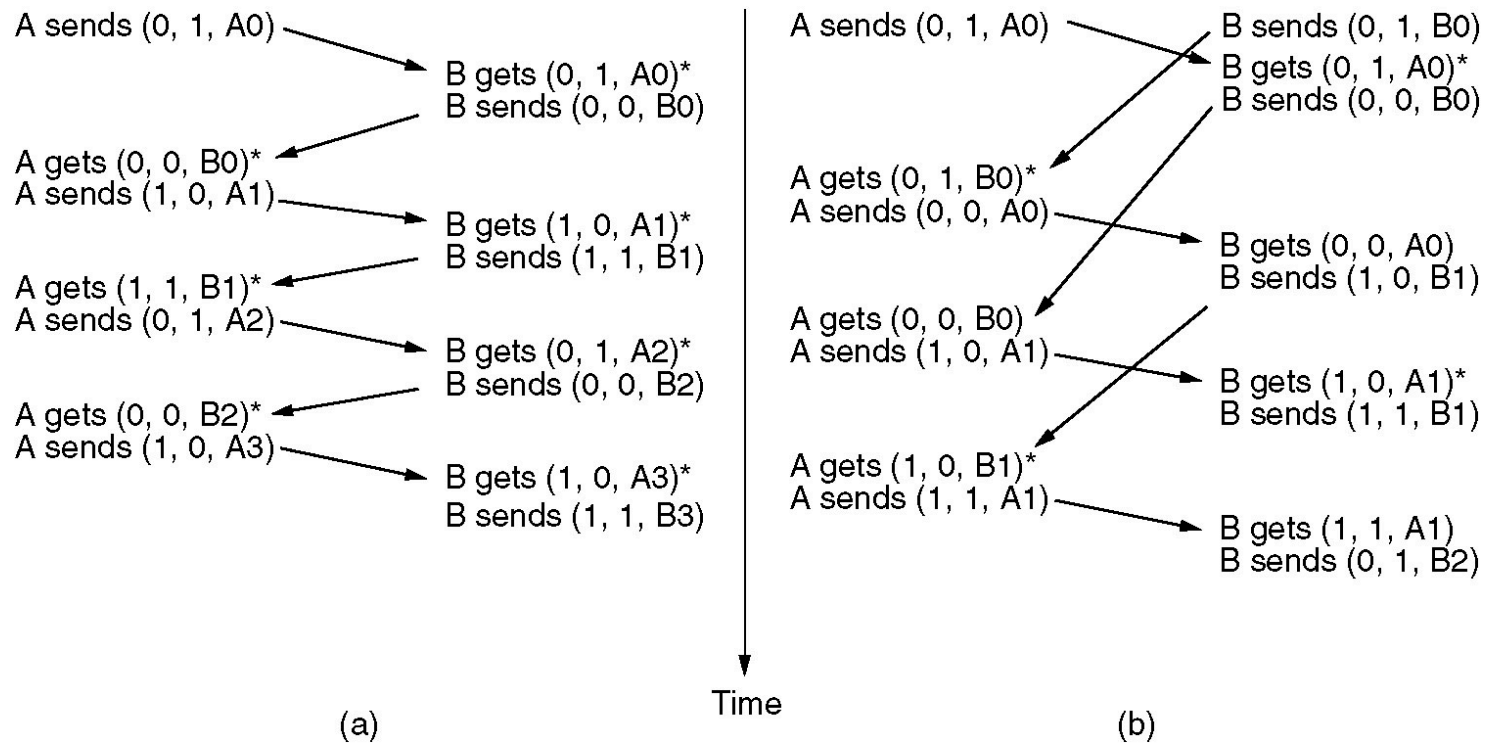
        if (r.seq == frame_expected) /* handle inbound frame stream. */
            to_network_layer(&r.info); /* pass packet to network layer */
            inc(frame_expected);      /* invert seq number expected next */
        }

        if (r.ack == next_frame_to_send) { /* handle outbound frame stream. */
            stop_timer(r.ack);         /* turn the timer off */
            from_network_layer(&buffer); /* fetch new pkt from network layer */
            inc(next_frame_to_send);   /* invert sender's sequence number */
        }

        }

    s.info = buffer;                 /* construct outbound frame */
    s.seq = next_frame_to_send;      /* insert sequence number into it */
    s.ack = 1 - frame_expected;      /* seq number of last received frame */
    to_physical_layer(&s);           /* transmit a frame */
    start_timer(s.seq);              /* start the timer running */
}
}
```

# A One-Bit Sliding Window Protocol



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.