ECE 333: Introduction to Communication Networks
Fall 2002

Lecture 7: Data Link Layer III

- **CRC codes**
- **Retransmissions**

Notes:

In the last lecture we began discussing the problem of error control. We gave examples of simple codes that could be used to either correct errors or detect errors in a received packet. The common approach at the DLL and higher network layers is to use codes for detecting errors and then retransmit packets that are found in error.

In this lecture we will first look at the most common type of code used for detecting errors in a network, called a ***cyclic redundancy check (CRC***) code or often simply a CRC. Such codes have several good error detection properties and are simple to implement. After discussing CRC's, we will begin to consider approaches for retransmitting packets that are found to be in error.

Cyclic Redundancy Check (CRC) codes

The most common type of error detection codes used in the DLL is a CRC code. These are used in part because they have good burst error detection characteristics. By a ***burst error*** of length n we mean that an error occurs at some bit position x, an error occurs at bit position $(x + n - 1)$, and the bit positions in between may or may not be in error.

A CRC adds L check bits to each message (packet). Assuming the packet size is less than 2^L -1, then a CRC can detect any burst error of length L as well as all patterns of 1,2, or 3 bit errors.

Furthermore, a CRC will detect a random large numbers of errors with probability $1-2^{-L}$. (Compare this with a parity check code in which a random received sequence will be accepted as correct with probability 1/2)

CRC's with 16 or 32 check bits are commonly used.

To understand CRC's, it is useful to represent bit strings as polynomials with binary coefficients. For example the string 10010 is represented as:

$$1x^4 + 0x^3 + 0x^2 + 1x^1 + 0x^0 = x^4 + x$$

It is also useful to add, subtract, multiply, and divide these polynomials using (mod 2) arithmetic, i.e. this is done in the same manner as with usual polynomials, except the coefficients are treated using the following rules:

Addition: $0 + 0 = 1 + 1 = 0$
$0 + 1 = 1 + 0 = 1$

Subtraction = addition

Multiplication: $0 \times 0 = 1 \times 0 = 0 \times 1 = 0$
$1 \times 1 = 1$

For example, using these rules we have: $(x^3+1) + (x^2+1) = (x^3+x^2)$ and $x^3(x^2+1) = (x^5 + x^3)$.

Cyclic Redundancy Check Codes

A CRC code is defined in terms of a ***generator polynomial*** $G(x)$, let L be the degree of this polynomial, i.e. $G(x) = x^L + \ldots$ (as we will see L is the number of check bits added to each packet.)

We represent the bit string to be encoded as a polynomial, $M(x)$.

To encode this bit stream we perform the following steps:

1. Form the product $x^L M(x)$, note this corresponds to appending L zeroes to $M(x)$.
2. Divide the product $x^L M(x)$ by $G(x)$, using polynomial division mod 2, let $R(x)$ denote the remainder of this division.
3. Set the transmitted codeword $T(x) = x^L M(x) - R(x)$.

Note that $R(x)$ will have a degree of at most $L - 1$, and $T(x)$ will correspond to the polynomial representing the original bit string plus L additional bits. Also, $T(x)$ must be evenly divisible by $G(x)$ (with no remainder). To see this note that $x^L M(x) = H(x)T(x) + R(x)$, for some polynomial $H(x)$. Therefore, the receiver can check if the received sequence is divisible by $G(x)$, and if not it will know that an error has occurred.

CRC's (continued)

Example: Assume message is 1 1 1 1 1 0 1 1 1 0 0 0 1

$$\Rightarrow \quad M(x) = x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^6 + x^5 + x^4 + 1$$

Generator Polynomial $G(x) = x^4 + x^3 + x^2 + 1$ (L=4)

$$x^L M(x) = x^{16} + x^{15} + x^{14} + x^{13} + x^{12} + x^{10} + x^9 + x^8 + x^4$$

Using long division:

$$\begin{array}{r} 1001100000001 \\ 11101 \overline{)\,11111011100010000} \\ \underline{11101000000000000} \\ 00010011100010000 \\ \underline{\quad 11101000000000} \\ 01110100010000 \\ \underline{\quad 11101000000000} \\ 00000000010000 \\ \underline{\quad\quad 11101} \\ 1101 \end{array}$$

← remainder →

$$\begin{array}{r} x^{12} + x^9 + x^8 + 1 \\ x^4 + x^3 + x^2 + 1 \overline{)\,x^{16} + x^{15} + x^{14} + x^{13} + x^{12} + x^{10} + x^9 + x^8 + x^4} \\ \underline{x^{16} + x^{15} + x^{14} + x^{12}} \\ x^{13} + x^{10} + x^9 + x^8 + x^4 \\ \underline{x^{13} + x^{12} + x^{11} + x^9} \\ x^{12} + x^{11} + x^{10} + x^8 + x^4 \\ \underline{x^{12} + x^{11} + x^{10} + x^8} \\ x^4 \\ \underline{x^4 + x^3 + x^2 + 1} \\ x^3 + x^2 + 1 \end{array}$$

CRC example (continued)

The remainder is appended to the original message and

1 1 1 1 1 0 1 1 1 0 0 0 1 1 1 0 1 is sent.

The receiver will divide this by $G(x)$ and if the remainder is not zero, an error has occurred.

To ensure good performance, generator polynomials with particular characteristics must be chosen.

One characteristic is that $G(x)$ has the form x^L + ... + 1, i.e. the 0th order term is a one. Next we argue that if $G(x)$ has this form then the CRC can detect any single bit error.

Let $T(x)$ be the transmitted codeword. The received codeword can be represented as $T(x) + E(x)$, where $E(x)$ is a polynomial representing the error sequence.

Suppose a single error occurs. The error sequence is then given by $E(x)=x^i$ for some i.

If $G(x)$ has the above form, then note that $G(x)H(x)$ must contain at least 2 non-zero terms for any non-zero polynomial $H(x)$. Thus, it can not be that $G(x)H(x) = x^i$ for any i= 1,2,…

Using this it follows that $T(x) + x^i$ will not be divisible by $G(x)$. Therefore the code can detect all single bit errors.

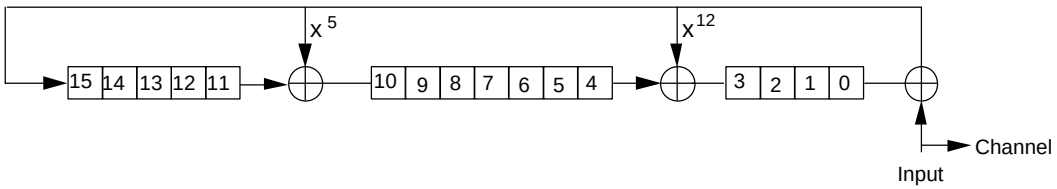
The other error correcting properties of the CRC code can be shown using similar ideas.

Both the encoding and decoding of CRC's can be easily implemented in either hardware (in VLSI chips) or software. In hardware, one simple implementation uses a feedback shift register, as shown next.

The number of registers is equal to L and the feedback taps correspond to the coefficients of the generator polynomial.

Cyclic Redundancy Code Implementation

$G(x) = x^{16} + x^{12} + x^5 + 1$ (CRC - CCITT)



Implementation:

Initialize shift register to all zero.

Input is directed to both the channel and the shift register.

When the last bit of input has passed, the shift register contains the checksum.

Complexity: 16 bits of shift register, 3 exclusive OR gates (note this is independent of the packet size.)

Retransmissions

Recall, once an error is detected, either:

- This information is sent up to a higher layer.

-or-

- The receiving DLL entity requests that the frame be re-transmitted.

Now we begin discussing the later case.

Algorithms for retransmitting packets are called ***Automatic Repeat reQuest*** (***ARQ***) algorithms.

We are interested in ARQ protocols for the DLL that provide a ***reliable*** serivce, this means

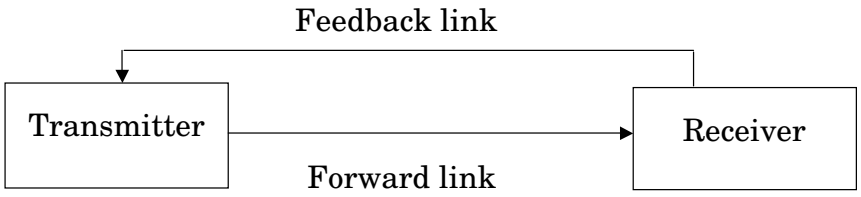
- § All frames delivered correctly
- § All frames delivered only once
- § All frames delivered in the correct order

Again we emphasize that these algorithms are also used at higher layers.

ARQ Algorithms

Basic problem:

Given that a packet is detected in error, the receiving DLL peer needs to request a retransmission from the transmitting DLL peer.



Perfect world:

- Every packet arrives on the forward link in order, within a specified time.
- Every packet arrives on the reverse link in order, within a specified time.
- Feedback link is error free.

Reality: Bad Things can Happen.

Depending on the situation:

- Packets on the forward link may not arrive at all, or may be delayed arbitrarily long.
- Packets on the reverse link may not arrive at all, or may be delayed arbitrarily long.
- Packets on the reverse link can also be in error.

In this environment, we want to design an ARQ algorithm so that each packet is delivered once, in order and without *detected errors*. (There will always be some errors that will not be detectable, but we cannot do anything about these.)

In some cases, in particular at higher layers, yet another issue can arise, that is that packets on both the forward or reverse link may not arrive in the order they where sent. We will **not** consider this issue right now.

Notification

Suppose that when an error is detected, the receiver sends a special "retransmit" packet to the transmitter requesting retransmission.

Even with a "perfect" feedback link there is a problem with this. What?

(Answer: the "retransmit" packet may not arrive.)

Instead, we will assume that the receiver tells the sender when a message is correctly received. The return message is called an ***acknowledgement (ACK)*****.**

A Simple ARQ algorithm

- The transmitter gets a packet from layer above, sends it, and starts a timer.
- At receiver, if the packet is received without any (detected) errors then it is delivered to the higher layer and an ACK is sent to the transmitter.
- Otherwise the receiver does nothing.
- If the transmitter receives ACK - it gets another packet, sends it, and resets the timer.
- If No ACK is received, then the transmitter times out and resends the packet.

Etc.

Note: the timer is needed so that if the packet isn't acknowledged, it will eventually be resent.

Problems ?

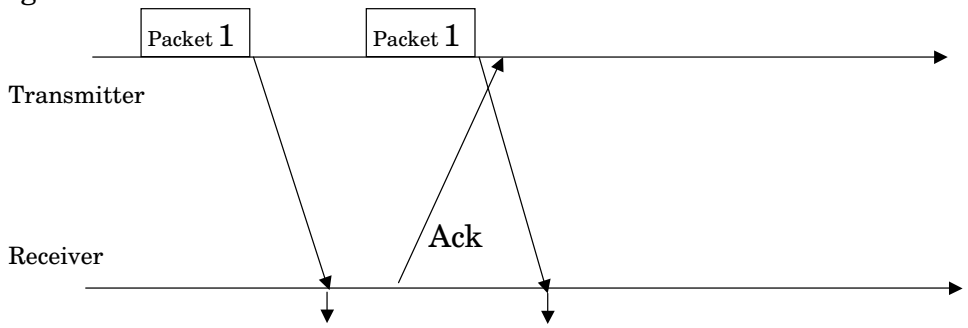
Duplicate Frames

What can go wrong with this protocol?

1. The ACK can get lost or damaged
2. The message can be excessively delayed
3. The timeout can be set too short

All of these can result in this layer delivering **duplicate** frames to the layer above, and the layer above might have no way of determining that this has happened. (Recall we want the DLL to provide a reliable service.)

E.g.



Timing Diagrams

The figure in the above example is called a ***timing diagram;*** such diagrams are useful for understanding the behavior of distributed protocols like ARQ..

The top line in the figure indicates time as measured at the transmitter, the bottom line, time at the receiver. The width of the rectangles at the top of the figure indicates the transmission time of each packet. The diagonal arrows indicate the propagation time of the packets. The point where the diagonal arrowheads intersect the time-lines indicates the time at which the entire packet is received. The downward, vertical arrows indicate the time at which the received packet is released to the next layer. In the above figure it is assumed that the ACK's are small packets and their transmission times are negligible.

Sequence Numbers

We cannot prevent duplicate frames from arriving at the receiver, since something as simple as a lost ACK can cause this to happen.

Thus, we need a way to keep the receiver from delivering duplicates.

To do this, we need a way of recognizing a duplicate.

A simple solution to this problem is to number each frame. For the simple protocol only a single bit is needed (again we are assuming that physical layer delivers all packets in order, without this assumption more bits for sequence numbers would be needed.).

A single bit is set to "1" or "0" in an alternating manner. This bit is called a ***sequence number***.

SEQ	Message to be sent	Checksum