CS 336 Lecture 8– 2/25/98

Announcements

- Exercise 3 is now available on the web (due Friday, Feb. 27).

The Cyclic Redundancy Check

1. The other aspect of the CRC that makes it valuable is the surprising fact that it is actually quite simple to calculate.

   - In a transmission system, the ideal is to be able to calculate the checksum bit by bit as the data arrives (rather than having to wait for the whole frame to arrive before we can start the process). This is how the calculation of the CRC is performed.
   - As a result, the best way to undertand the computation is to assume you have received a message and computed its CRC when to your surprise an extra bit of the message arrives. How could you calculate the CRC of the message with the added bit included based on the CRC you had computed before you knew about the extra bit?
   - Well, let's call the partial message T(X) and assume that the partial CRC is R(x) such that for some Q(X)

     $$T(X) = Q(x)G(x) + R(x)$$

     and lets call the extra bit "b".
   - By multiplying both sides of the equation above by x, adding b to both sides and then regrouping a bit we obtain

     $$(T(X)x + b) = (xQ(x))G(x) + (R(x)x + b)$$

   - If $R(x)x$ is of degree less than $G(x)$ then we are done. In this case, by the definition of polynomial division, $R(x)x + b$ is the remainder of dividing the polynomial corresponding to the complete message by G(x). Since multiplying a polynomial by x just raises the power of all its terms, this means we can obtain the CRC for the polynomial with b added by throwing away the first bit of the old CRC (which must have been 0) and adding b as the last bit.
   - If the first bit of the old CRC was 1, then $R(x)x$ will have degree k. Luckily, in this case

     $$R(x)x + b - Q(x)$$

     is guaranteed to have degree less than k since the coefficients of x^k in both Q(x) and $R(x)x$ must be 1 and 1+1 = 0.

     In this case, regrouping the terms of the equation above to yield

     $$(T(X)x + b) = (xQ(x) + 1)G(x) + (R(x)x + b - G(x))$$

     allows us to conclude that this time

     $$R(x)x + b - Q(x)$$

     must be the new remainder.

     Accordingly, when the first bit of the CRC is 1, we can compute the new CRC by first shifting the old CRC left by one, then appending the new bit b and then adding G(x). Since addition mod 2 is just the exclusive or operation, we can just exclusive or the bits of the shifted CRC with the bits corresponding to the coefficients of G(x).
   - As the text shows, these rules can be expressed by cute little diagrams that can ultimately be turned into simple and highly efficient circuits for computing and verifying CRCs.

Automatic Repeat Request (ARQ) Protocols

1. The acronym ARQ makes absolutely no sense to me, but it is a convenient shorthand for discussing the protocols and algorithms by which the actors at the two ends of a network connection can communicate to arrange reliable, ordered delivery of messages.

2. It is worth commenting that message loss/damage is not the only problems that need to be solved by these protocols. Once you decide that retransmissions will be used to deal with lost and damaged frames, then other issues arise:

   Duplication If I send a message and your attempt to tell me it reached you is lost, I will send the message again and you may think the second copy is a second, distinct message.

   Reordering If I send my messages in a hurry, I may not realize that the first message sent never made it until after you have accepted my second message.

3. To motivate the approach I take to discussing these protocols, I should explain that I feel that two details often cloud the presentation of these protocols:

   - The need to ultimately limit sequence numbers to finite ranges.
   - The two way nature of the communications involved.

4. In our examination of ARQ protocols we will:

- Initially assume the use of unbounded sequence numbers and then explain the range actually required once the behavior of the protocol is understood.
- Assume data only moves in one direction (most of the time).

5. The efficiency and complexity of ARQ protocols increases with the degree to which the sender and receiver are willing to reserve space in memory for message buffers.

   - If neither end is willing to buffer messages, then only one message may be outstanding at a time. This leads to stop and wait.
   - If only the sender is willing to keep multiple message buffered, receiver must insist on ordered deliver leading to Go-back-n protocol.
   - If both sides are willing to buffer, selective repeat protocols can be used.

   Stop and Wait is the simplest, so we start with it.

6. Our version of stop and wait will be slightly different from the text in that it will be designed to handle packet loss.

7. The protocol described as stop and wait in the text does not handle packet losses for two reasons.

   - Neither the sender nor the receiver uses timeouts to retransmit if the other side is unexpectedly silent.
     - If A sends a packet to B and then waits patiently for an acknowledgement, A may wait forever if either the packet or its acknowledgment is lost.
     - To deal with this we will assume the sender sets a timer after each transmission and retransmits if the timer goes off before an ack arrives.
   - Sequence numbers are not included in packets or acknowledgments.
     - Once timeouts are added, sequence numbers are necessary. Suppose A sends a packet to B, B receives the packet and sends an ack but the ack gets lost. When A times out and retransmits, B needs some hint to help it avoid duplication.
     - To deal with this, each packet will contain an identifying sequence number (unbounded for now) and each ack will contain the number of the next packet the sender expects.

8. The simplest way to imagine the algorithms used to ensure reliable transmission of messages in a network is to assume that one machine (the "sender") has an infinite supply of numbered messages to send and expects nothing in return except acknowledgements, while another machine (the "receiver") has nothing to send but is willing to accept messages as quickly as it can.

```
``sender''
      SendNext = 0;

      Get packet to send from higher layer

      do { TransmissionTime = now;

           Send Frame number ``SendNext''

           do { nothing }
              until ( ack is received or
                         now > TransmissionTime + TimeOut )

           if (an ack was received)
                 if Ack.SendNext > SendNext
                     SendNext = Ack.SendNext
                     Get packet to send from higher layer
       } while ( 1 + 1 == 2 )

``receiver''

     Receivenext = 0;

     do {
           do { nothing }
           until ( packet is received )

           If ( packet was intact && packet.num = ReceiveNext
)
                   deliver packet to higher layer
                   ReceiveNext ++ ;

           Send an ack containing ReceiveNext as ``SendNext''
     } while ( 1 == 1 )
```