ECE 333: Introduction to Communication Networks
Fall 2002

Lecture 8: Data Link Layer IV

- **ARQ algorithms**

In the last lecture, we began looking at ARQ protocols. Recall our assumptions are:

1. Packets on both the forward or reverse link may be delayed arbitrarily long and may not arrive at all (packets that are found to be in error are considered to not have arrived)
2. If packets do arrive they arrive in the order they were sent.

Given these assumption we want to develop a protocol to provide a ***reliable*** packet delivery service, meaning that each packet is delivered to the next layer correctly, in order and only one time.

Last time we proposed that the receiver send an Acknowledgement packet (ACK) to the transmitter for every correct packet received, and we saw that the transmitter needs to number the packets that are sent.

Where Do We Stand Now?

Current protocol:

Transmitter:

1. Set sequence number, SN = 0
2. Wait until packet available to be sent (from higher layer)
3. Send current packet in frame with number SN, start timer.
4. If time-out, go to 3.
5. If (error free) ACK received, set SN=SN+1 (mod 2), go to 2.

Receiver:

1. Set current number to be received, RN= 0.
2. Wait until packet arrives error free then:
   a. Send ACK to transmitter.
   b. If SN=RN, release packet to higher layer, set RN=RN+1(mod2).
3. Go to 2.

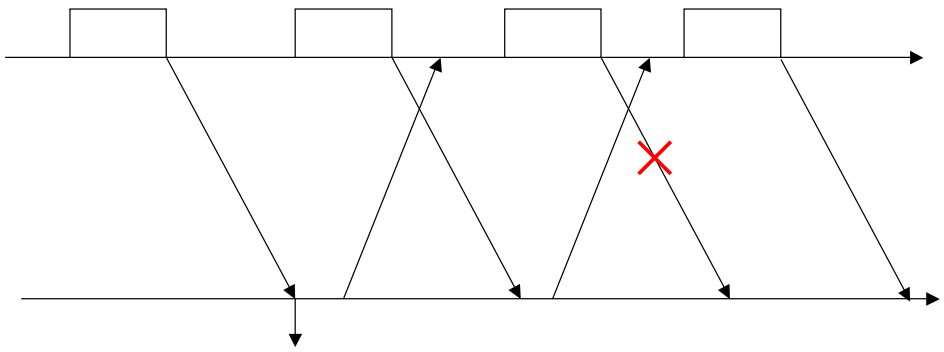
Is our protocol now reliable?

A Problem

How do we decide on the value to use for the timer?

- If it is too long, we waste time waiting to retransmit.
- What if it is too short?

Consider the following possibility: (what should the transmitter do next ?)



The ACK that arrives after packet 2 may be for the second packet 1 as shown, or it could be for packet 2 (if one of the ACKs for packet 2 was lost). Depending on which event occurred the transmitter should either resend packet 2 or send packet 3.

The solution to this is to include in the ACK packet the sequence number of the next frame expected. (Equivalently, the sequence number of the frame just received could be used, but in practice this is not done.)

We now have the following distributed protocol:

At transmitter:

1. Set sequence number, SN=0
2. Wait until packet available to be sent (from higher layer)
3. Send current packet in frame with number SN, start timer.
4. If time-out, go to 3.
5. If an error free ACK received with request number (RN) ≠ SN, set SN = RN, go to 2.

At receiver:

1. Set RN=0, repeat 2 and 3 forever.
2. If packet received with SN = RN, then release the packet to the higher layer, set RN = RN+1 (mod 2).
3. After receiving any data frame, transmit ACK containing RN.

Note: the receiver can not send back an ACK only after receiving a packet with SN=RN or the protocol may fail. Think about why.

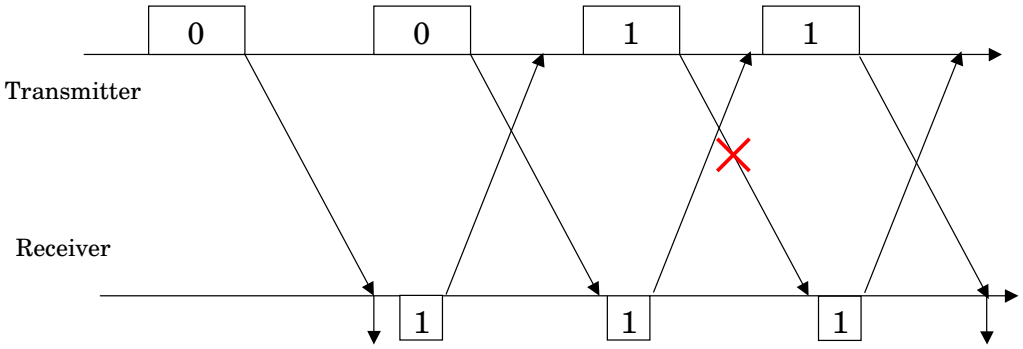
Stop-and-Wait Protocol

The protocol that we have just described is called a ***stop- and-wait protocol*** (It is also sometimes called an ***alternating bit protocol*** because of the use of 1-bit sequence numbers)

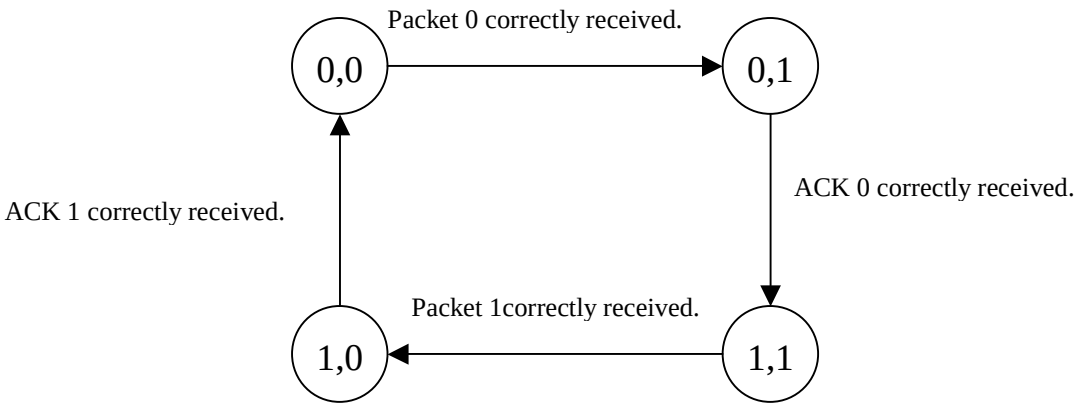
Notes:

- The two ends must agree to a starting "state" – in this case the first sequence number to be used – and must maintain that state as the protocol operates.
- Setting the timeout parameter is a trade-off between not requiring too many retransmissions and incurring long delays when frames are lost.
- The resulting protocol achieves the requirements for reliability under the assumption that all error can be detected and that packets stay in order.

Example:



The following figure illustrates the correctness of stop-and-wait. This figure shows four ***states*** labeled by (SN,RN). Each state corresponds to the values of these counters at the transmitters and receiver at any given time. The sequence of states shown in the figure is the only possible sequence that can occur under correct operation.



Piggybacking

The stop-and-wait protocol described above addressed one-way (simplex) communication of data packets. It is common for nodes at both ends of a communication link to have data to send. In this case, the above protocol could be implemented for each direction. However, rather than send separate ACK's on the reverse link, the acknowledgements for messages coming in one direction can be included with the data going the other direction. This is called ***piggybacking***.

When piggybacking is used the format of each packet looks something like this:

SN	RN	Message to be sent	Checksum

SN – The sequence number of the message being sent
RN – The sequence number of the next message that is being requested for the reverse connection.

Acknowledgements may be temporarily delayed to wait for data on the reverse link, but should not be delayed too long. Why?

Performance metrics

Now that we have a working ARQ protocol, we can start considering its performance. One measure of the performance is the ***effective throughput*** (also called the ***goodput***). This is the long-term average transmission rate seen by the next layer up. For example, suppose we are sending packets over a link with a transmission rate of R bps. If each packet contains d data bits and a header of h bits is added to the packet, then it will take $(d+h)/R$ seconds to sent each packet. In this case, the maximum effective throughput seen by the network layer is

$$\frac{d}{d+h} R \text{ bps.}$$

If a stop-and-wait protocol is used on this link then after sending a packet we have to wait for the ACK to return before we can send the next packet. The delay from when we finish sending a packet until an acknowledgement returns is referred to as the ***round trip time*** (RTT). Assuming no errors occur, the RTT is equal to 2 times the propagation delay, plus the transmission time for an ACK plus any additional processing time at the receiver. Suppose the round trip time is I sec. Thus it takes at least $(d+h)/R + I$ seconds to send d bits of data. Therefore the maximum effective throughput is given by

$$\frac{d}{(d+h)/R + I}$$

Note (for now) we have ignored retransmissions.

A related performance measure is the ***efficiency*** of the protocol. We define this as follows

$$efficiency = \frac{effecitve\ \ throughput}{link\ \ throughput}$$

(usually this is expressed as a percentage). For the above example, the efficiency, η, is

$$\eta = \frac{d/R}{(d+h)/R + I} = \frac{d}{d+h+IR}$$

Equivalently, efficiency can be defined in several other ways:

$$efficiency = \frac{data\ bits}{total\ bits\ possible} = \frac{time\ sending\ data}{total\ time}$$

Here "total bits possible" includes the number of bits that could have been sent during the idle time, *i.e. IR*.

Protocol Efficiency Example

Consider a Stop-and-Wait Protocol used over a 50 kbps satellite channel with a propagation delay of 250 msec each way. Further assume a 1000 bit frame. What is the efficiency?

Answer: If we start transmitting at t = 0, the last bit of the frame is transmitted at t = 20 (msec). The last bit arrives at the receiver at t = 270 msec. If we assume a very short acknowledge frame and turnaround time, the acknowledgment is received by the sender at t = 520 msec, and the next frame can be sent. (RTT = 500 msec)

Thus, we are sending data 20 out of 520 msec, for an efficiency of 3.85%, and this ignores the loss of efficiency due to header overhead, retransmission or any other problems. In this case the effective throughput is given by (.0385)50 kbps ≈ 1.9 kbps.

For long propagation times and high transmission rates stop-and-wait clearly becomes inefficient. How can we improve this, while still providing a reliable service?

Sliding Window Protocols

One way to improve the efficiency of stop-and-wait is to use larger packets, i.e., more data bits per packet. One problem with this is that the data may not arrive from the higher layer in large units. Hence, to send larger packets, the data would have to be delayed until enough accumulated. In general, this is also undesirable.

A better way to improve the efficiency is to allow the sender to send more than 1 frame before receiving an ACK.

For example, let the transmitter send up to N packets before it receives an ACK for the first packet.

Such protocols are called ***Sliding Window Protocols***. In the next lecture, we will begin to study this type of protocol.