ECE 333: Introduction to Communication Networks
Fall 2001

Lecture 8: Data Link Layer IV

- **ARQ algorithms**

Last time 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 packet do arrive they arrive in the order they were sent.

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

Last time we proposed have 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.

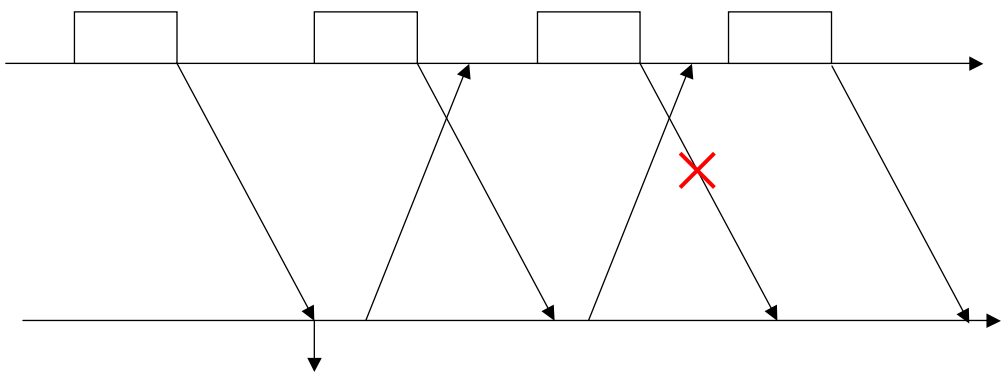
Is our protocol now reliable?

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:



The solution to this is to include in the ACK 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 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.

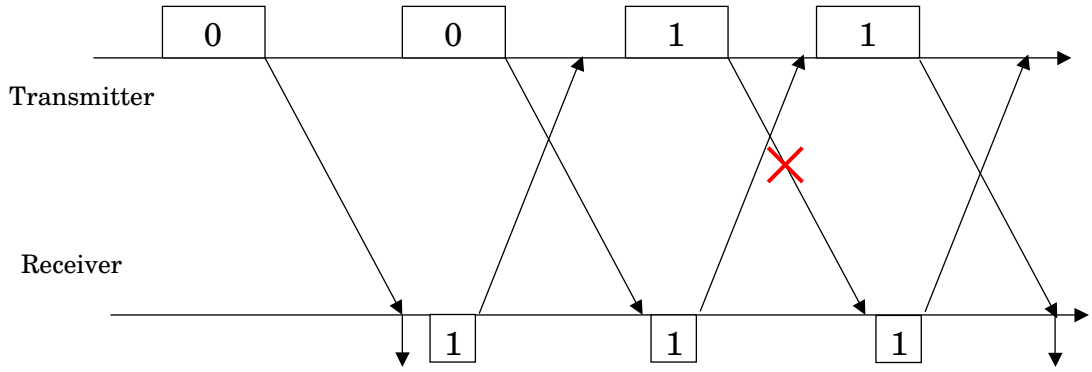
Stop-and-Wait Protocol

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)

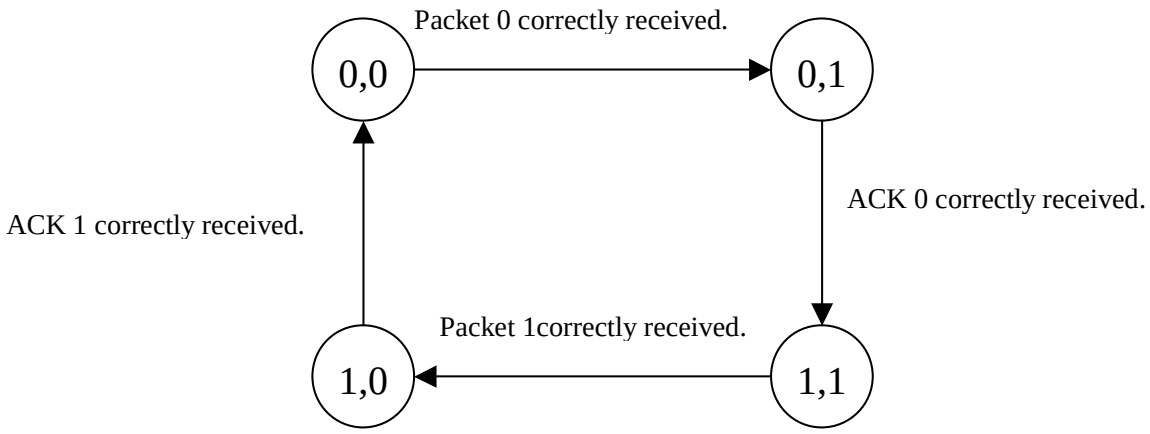
Notes:

- 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). These correspond 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 frame looks something like this:

SEQ	REQ	Message to be sent	Checksum

SEQ – The sequence number of the message being sent
REQ – The sequence number of the next message that is being requested.

Acknowledgements may be temporarily delayed to wait for data on the reverse link. But can't wait 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*** at times). 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, thus the maximum effective throughput seen by the network layer is

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

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 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 be 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 = 520 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.

For long propagation times and high transmission rates stop-and-wait becomes inefficient. Can this be improved?

Sliding Window Protocols

One way to improve the efficiency of stop-and-wait is to use longer packets. Problems?

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.

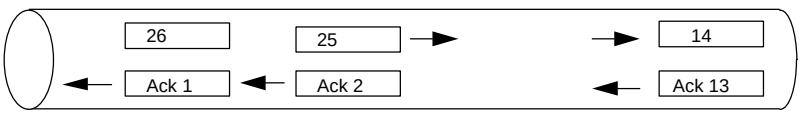
Such protocols are called ***Sliding Window Protocols***.

First question: How big should the window, *N,* be?

Filling the Pipe

To improve efficiency, the window should be chosen big enough to minimize the time the transmitter is idle. Returning to the above satellite example, recall the RTT is 520 msec, In this time the transmitter can send 26 frames. If no errors occur, the acknowledgment from the first frame should arrive immediately after the 26th frame is sent. This allows it to send frame 27. In this case a window size of 26 is said to ***fill the pipe.***

This technique is known as ***pipelining.*** In this case a window size of 26 is said to ***fill the pipe.***



With no errors, this gives good efficiency, because it allows the sender to transmit continuously.

What about when errors occur?

Error Recovery in sliding window protocols

When a sliding window protocol is used, we would like to have a retransmission protocol that still provides a reliable service, i.e. delivers packets in order, correctly, and only once. There are two classes of such protocols that are commonly used, the first is called ***go back n,*** and the second is called ***selective repeat***.

For both types of protocols, as with stop-and-wait, the sender numbers each packet, and the receiver returns numbered acknowledgments when a packet is correctly received. However with a sliding window protocol, as opposed to stop-and-wait, more than 1 bit will be needed for sequence numbers. In each case the sender has a ***maximum window size***, which is the maximum number of unacknowledged frames it is allowed to have sent at any time. We will refer to the ***sender window*** as the set of frames that have been sent but are not yet acknowledged. The number of frames in the sender window must be no greater than the maximum window size. (*Warning*: Tanenbaum uses "Sender Window" inconsistently to mean both of these concepts.)

Go Back N

In a ***Go Back N*** protocol, the transmitter has a maximum window size of N packets. The receiver behaves essentially the same as in stop-and-wait. That is it keeps a counter of the current frame it is waiting to accept, and will accept only that frame. When the receiver receives any frame it returns an ACK to the transmitter containing the number of the next packet expected.

When the transmitter receives an ACK with request number RN, it can assume that every packet with number less than RN has been correctly received. As with stop-and wait, the transmitter sets a timer for each packet that is sent. When a timer expires, the transmitter goes back and resends the first unacknowledged packet.

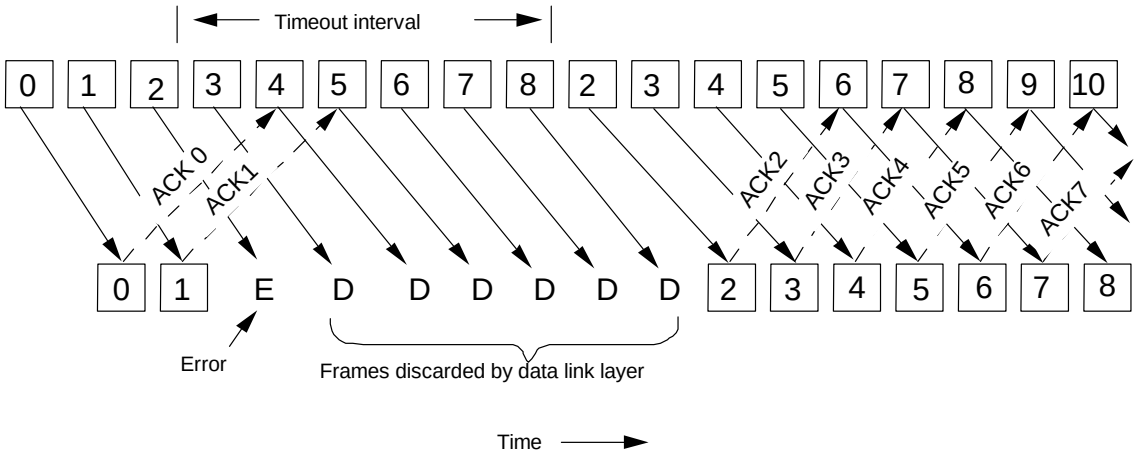
Actually, several variations of the rule used to decide when the transmitter goes back can be used, for example, it might go back before the timer expires if it receives two ACKs in a row containing the same request number. (Why?) The important point for the protocol to work correctly is that the transmitter eventually does go back and resend any unacknowledged frame. Also several variations of the rule used at by the receiver to return ACKs are possible, it may return ACK for every received frame, or possibly only after receiving error free frames.

Go Back N

Pictorially, an example of the go back N protocol is shown below.

Sender transmits frames until the window size stops it. If a frame times out, sender retransmits it and goes back to send all frames after retransmitted frame.

Receiver acknowledges the frame received. (here receiver only ACKs correct frames)



Sender's Window and Sequence Numbers

Suppose there are n bits available for sequence numbers, How big should the senders maximum window size be? (i.e., N= ?)

Clearly we must have $N \leq 2^n$. Can we have $N = 2^n$?

Consider 3 bit sequence numbers with a max. window size of 8:
Suppose the following occurs:

- Sender sends frames 0 – 7 (using sequence numbers 0 – 7).
- ACK for frames 0 -7 are received.
- Sender sends frames 8 – 15 (using sequence numbers 0 – 7)
- Sender receives an ACK with RN=0.

What should the sender do?

What does the RN=0 mean?

Sender's Window and Sequence Numbers

RN-0 could mean:

All 8 frames in second group (frames 8 – 15) arrived OK at the receiver, and only the last ACK got through.

Alternatively -
Frame 8 arrives at the receiver damaged, receiver sends an RN=0, Frames 9 – 15 are lost.

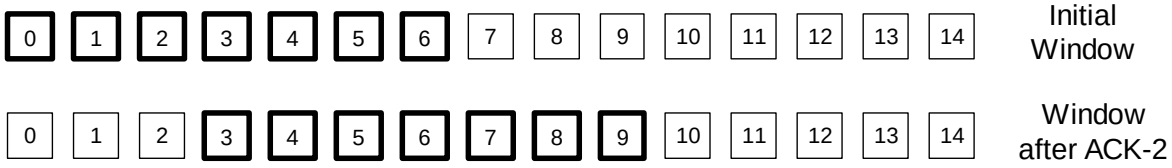
This ambiguity is clearly not a good thing. The ambiguity arises because the sender can receive an ACK to any new frame sent, and it can also receive a duplicate of the previous ACK. In this case that makes for 9 frames that can be ACKed, and there are only 8 sequence numbers.

⇒ To make the Go Back N protocol work correctly, with n bits for sequence numbers, we need to restrict the maximum window size to be at most $2^n - 1$.

Go Back N Protocol

Note that the sender will buffer outstanding frames until they are acknowledged, and thus must have enough buffer space for its window size.

As frames are acknowledged, the sender can advance its window allowing additional frames to be sent, i.e., a sliding window.



Performance?