UCLA

DEPARTMENT OF COMPUTER SCIENCE

# CS 118 **Computer Communications: Error Recovery for Point-to-Point Links**

This is the third of probably 7 lectures on the Data Link Layer. In the last two lectures we studied the two bottom sublayers of any Data Link: framing and error detection. Framing was used to transform a stream of bits into units called frames, and error detection added checksums to these frames to detect whether any bits in a frame are corrupted. In this lecture we will study *error recovery*, which goes beyond error detection by correcting for lost and corrupted packets on each hop.

## 1 Why Error Recovery?

In our sublayer model of a Data Link, we said that error recovery was an optional sublayer of some older point-to-point links. Thus in this lecture we will restrict ourselves to point-to-point Data Links. However, if hop-by-hop error recovery is largely historical, why bother to study it. Here are some reasons:

- The effort required to understand error recovery will not be wasted: transport protocols use essentially the same ideas for end-to-end recovery. When we come to transport protocols we will only talk about the differences in end-to-end error recovery.
- Many older protocols like HDLC use these techniques and are still being used.
- Error recovery furnishes us with the first non-trivial example of a protocol and its possible problems due to varying message delay and errors (e.g., frame loss and node crashes).
- End-to-end recovery is no longer popular because of the end-to-end argument and the fact that modern fiber links have very low error rate. This may change on certain low-bandwidth links between mobile computers because the error rate may be quite high on such links. When the wheel of time moves around again, you should be ready. (In fact, in this year's SIGCOMM, there were several proposals for doing hop-by-hop error recovery on mobile links.)

## 2 What must Error Recovery Guarantee?

As in the figure, error recovery must guarantee that the packets given for transmission to the sending data link must be delivered to the receiver without duplication, loss, or misordering. Duplication means including extra copies — i.e., delivering $a, a, b, c$. Loss means missing a data item — i.e., delivering $b, c$. Misordering means permuting the sending order i.e., delivering $b, c, a, d$. Recovery at the transport layer means much the same thing. (So far we are ignoring crashes at the sender and receiver; we will talk about node crashes next lecture.)

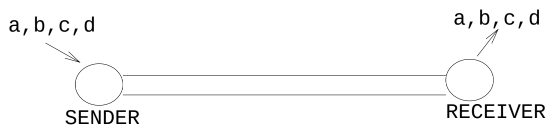

Figure 1: Reliable in-order transmission

Avoiding loss is, of course, useful on very high error-rate physical links. Avoiding reordering and duplication can avoid extra work for the end-to-end transport protocol to reject duplicates and to reorder the stream of packets. Some special applications like video and audio may prefer not to run above a transport protocol for efficiency and definitely cannot tolerate misordering.

## 3 Assumptions

We assume that error recovery is being done between a single sender and a receiver that are connected by two point-to-point physical links in each direction as shown in the figure. We assume that error recovery is layered above the error detection sublayer. We assume that:

- The undetected error rate (probability of a corrupted frame being passed by error detection) is small enough to be ignored.
- Whole frames can be lost in a way that may not be detected by error detection (i.e., if all bits in a frame are lost).
- However, the physical layer is FIFO. If bit is sent before $b'$, then if both bits are received, bit $b$ is received before $b'$.
- The delay on the physical links is arbitrary and can vary from packet-to-packet.

The last assumption needs to be explained. The delay on a single fibre-link is pretty constant. However, if the physical link is going through another level of switching (there is a way of switching fibre links known as SONET) then the delay can change. Also, if there is multiplexing going on at the physical level, the delay may vary. Second, the timers on most operating systems are fairly inaccurate and the result is that one cannot measure time very precisely. Third, its nice if you can have a protocol that does not depend on constant delays to work correctly.

## 4 Protocol Play

Suppose we try the following sequence of protocols:

- First the sending Data Link just sends the sequence of received packets as frames. Clearly if a frame is lost, we can have omissions.

- Second, we have the receiver send back a special ACK frame (with no other information) for every data message it receives. If the sender does not get an ACK within a timeout period, the sender retransmits. This can cause duplication. If $a$ is sent and is received but the ACK is lost, then the sender will resend $a$, and the receiver will output $aa$.
- Third we can try and detect duplicates by rejecting any frames whose data is the same as that of the previous frame. But the data sent may contain the same value in successive frames and so this would cause deadlock.
- Fourth, we add a sequence number that is attached to each frame. The sender starts with sequence number 0 and the receiver starts with sequence number 0. Each frame sent (or resent) by the sender carries the current sender sequence number. The receiver only accepts frames that have the same sequence number as the receiver number. On successful receipt, the receiver accepts the frame, increments its number, and sends an ACK. On receipt of an ACK, the sender increments its number and sends the next packet. This protocol can *deadlock* if the first ACK is lost; the sender will keep retransmitting but the receiver will not accept the new packet and send an ACK.
- The obvious way to fix the previous problem is to have the receiver send back an ACK frame even if it gets a duplicate. However, now the protocol may lose a packet. Suppose the sender wants to send $a, b$. The sender sends $a, 0$ and the receiver sends an ACK. The sender times out before the ACK is received and resends $a, 0$. The receiver gets the duplicate and does not accept it but sends a second ACK. Assume now that the first ACK is received by the sender and the sender sends $b, 2$. If $b, 2$ is lost and the second ACK arrives, the sender will assume $b$ was successfully sent and move on to $c$. Thus we have an omission.
- The last problem is because an ACK for an older frame is being mistaken for an ACK for a recent frame. An obvious way to fix this last problem is to add a sequence number to ACKs as well; an ACK should carry the receiver number. The sender should only accept ACKs with number greater than its own sequence number.

The protocol plays are shown schematically in Figure 2, Figure 3 and Figure 4.

Thus we conclude that we need retransmission for loss, sequence numbers to detect duplicates and misordering, we need to send acks back even on receipt of duplicate data, and the acks must also be numbered. Assume the sequence numbers are large integers (say 64 bits) that never wrap around. The resulting protocol is called the *stop-and-wait* protocol (because the sender has to stop further transmission of frames and wait for an ack.) The code for the protocol is given below.

## 5 Stop and Wait Code

Here is the sender code:

```
Sender has a variable SN (for sender number) initially 0.
Sender repeats the following loop:
```

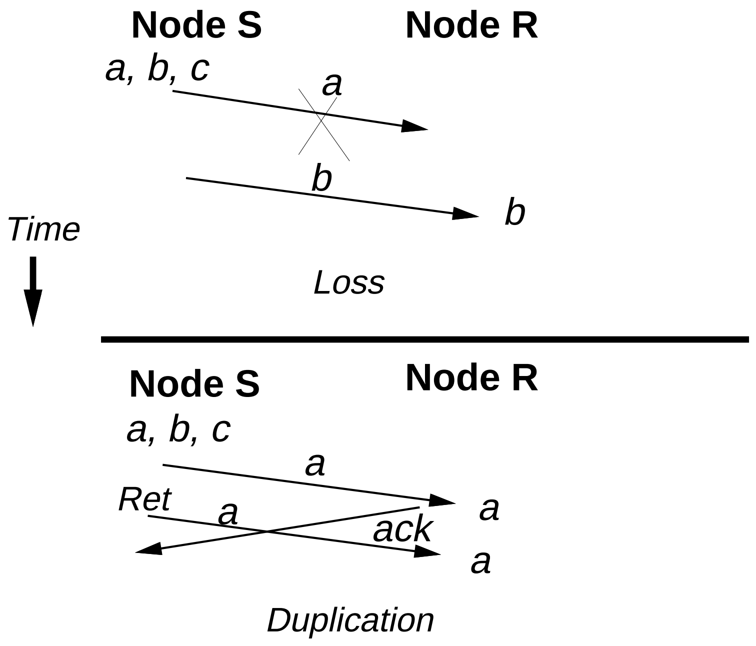

Figure 2: No retransmission leads to loss: retransmission without checks leads to duplications

```
1) Accept a new packet from the higher layer if available and store it
   in Buffer B.
2) Transmit a frame (SN, B)
3) If an error-free (ACK,R) frame is received and R is not equal to SN then
      SN = R
      Go to Step 1.
  Otherwise if the previous condition does not occur with an arbitrary
  timeout period,  go to Step 2 after the timeout period.
```

Here is the receiver code:

```
Receiver has a variable RN (for receiver number) initially 0.
Receiver does the following code:
When an error-free data frame (S, D) is received:
  If S = RN then
      Pass D to higher layer
      RN = RN + 1
  Send (Ack, RN)
```

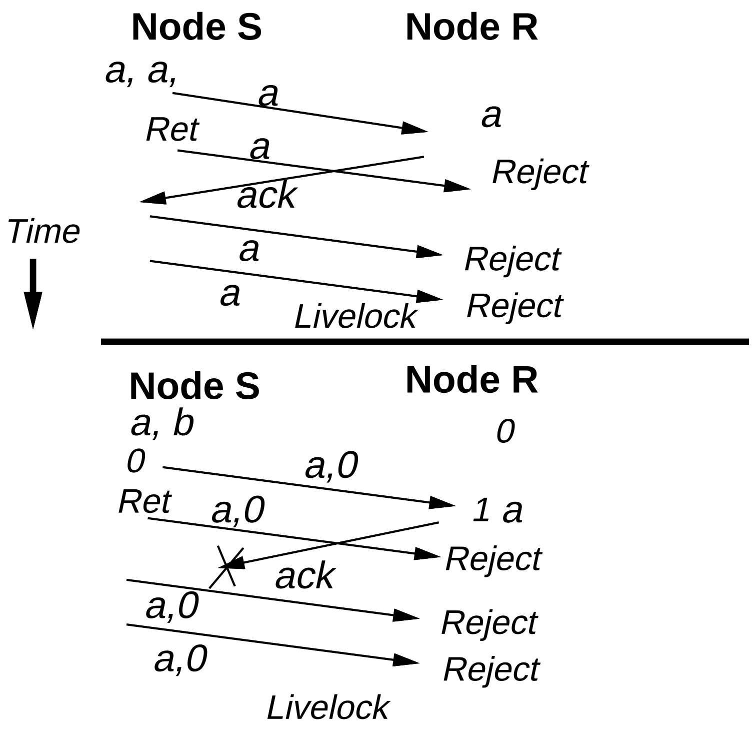

Figure 3: Trying to suppress duplicates by remembering previous data item leads to livelock when data to be sent contains the same data item twice. Not sending an ack for a message already received leads to livelock

## 6 Stop and Wait Example

Figure 5 shows a time-space diagram that depicts an example of stop and wait behavior. Time increases downwards in the figure (as in all such figures in this course unless explicitly said so). Thus a message sent from the sender $S$ to the receiver $R$ will be a line from left to right, and vice versa. In the figure notice that the second data item is lost and then retransmitted. (The convention for lost frames is that the line does not quite reach the other end; sometimes I add a cross to indicate the message was lost.) The third data item is retransmitted too early causing an extra ack. However, the first ack is lost but the second one is received.

We notice some interesting properties from the figure. Notice that by the time the sender first gets to number $n$, there are no frames with number $n$ or acks with number $n+1$ in the channel, and the receiver is at number $n$ (i.e., expecting the right number). If this is so, it is easy to see that frame $n$ will correctly be received. This is because the sender will keep retransmitting until a copy of the data corresponding to number $n$ is received (until that happens the receiver will not send an ack for $n+1$). Since the receiver keeps sending acks for every sender retransmission, the sender will eventually get an ack and go onto number $n+1$.

Another thing to notice is that because the physical channels are FIFO, when the receiver receives frame $n$, the receiver can be sure that the sender has received a $n$ numbered ack which must have "flushed" out the reverse link of all lower numbered acks. Similarly, the data frame

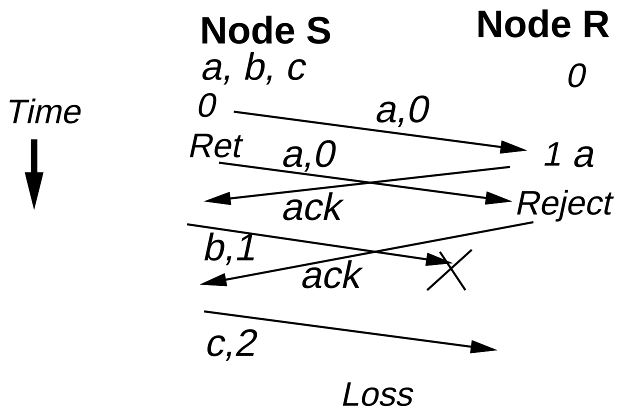

Figure 4: Not numbering acks with a sequence number leads to loss as old acks can be mistaken for more recent acks.

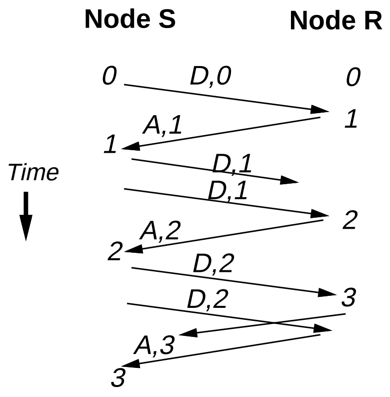

Figure 5: An example of Stop and Wait Behavior

numbered $n$ must have flushed the forward link of all lower numbered frames. Thus, at the instant the receiver receives a data frame with number $n$, the entire system only contains number $n$ (all lower numbers must have disappeared). Thus when the receiver creates a new value like $n+1$), all values that are $n-1$ or less have been flushed from the system.

Thus there are only 2 possible consecutive sequence numbers in any state. Now notice that in the sender and receiver protocols, when we compare two numbers we only check if they are equal or unequal. But if the two numbers being compared are always consecutive numbers, it suffices to check their Least Significant Bit (LSB). But in that case we can replace the large sequence number by a single bit that is incremented mod 2. This is indeed true; the resulting protocol is called the *alternating bit* protocol.

Our argument of the correctness of stop-and-wait was based on the statement that when the sender first reaches number $n$, there are no data frames numbered $n$ in the forward link, and no acks numbered $n+1$ in the reverse link, and the receiver is at $n$. But how do we prove such statements, especially across the infinite number of possible executions (recall that retransmission and message delivery times and losses are arbitrary, leading to an infinite number of possible combinations). Similarly, we argued informally that there are only two sequence numbers present in any state and so we can get by with a single bit. Can we prove such a statement with more confidence than the casual, error-prone way we argued in the last two paragraphs? There is a better way, known as invariants, which we study next.

## 7 Invariants

In order to understand what an invariant is, we need to understand what a global state of a protocol is. To do so we consider a simple protocol between a sender $S$ and a receiver $R$. The sender has only one variable $i$ and the receiver has only one variable $j$. Initially both variables are 0. This is not a Data Link Protocol; its just a trivial example, to help you understand invariants.

The protocol starts by $S$ sending a frame $M$ to $R$. Frames and acks are never lost. When $M$ arrives at $R$. $R$ increments $j$. Then $R$ sends back an ack to $S$. When the ack arrives (we assume in this simple protocol example that no frames ever get lost) at $S$, $S$ increments $i$ and sends another frame $M$ and the cycle continues.

A sample execution of this protocol is shown on the left in Figure 6. This is a time-space figure; as usual time flows downwards. However, there is another way to look at executions of a protocols: in terms of transitions between global states. This shown on the right in the same figure.

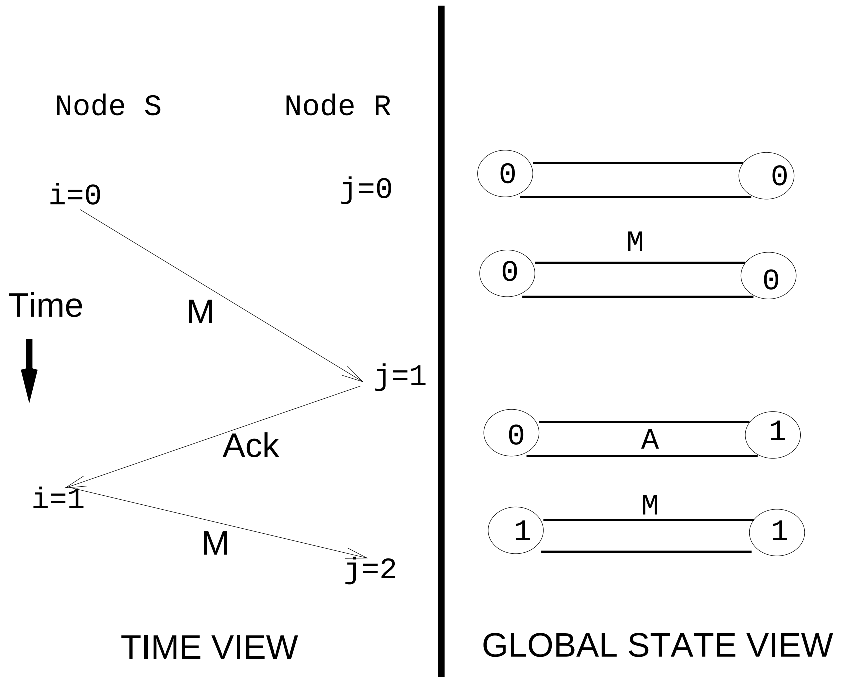

Figure 6: Time-Space and State Transition Views of a Simple Protocol.

The state of a node at any instant is simply the values of all node variables that are relevant to the protocol. For example, the state of node $S$ is the value of $i$, and the state of $R$ is the value of $j$. The state of a link is the sequence of frames (assuming the link is FIFO) that have been sent on the link but not received. Thus between the time a frame $M$ is sent by $S$ and the time frame $M$ is received by $R$, $M$ is "stored" on the link. During this time, the state of the link is the frame $M$. If you like, you can think of a link as a queue variable; a send inserts at the rear of the queue, a receive removes the head of the queue. The state of the queue is just the sequence of items (i.e., frames) stored in the queue. Finally the global state of a protocol at any instant is the state of all nodes and all links in the network. On the right of Figure 6 we show the global states corresponding to the time space picture on the left. Notice that during large portions of time (when no protocol actions occur), the global state remains unchanged.

Now let us ask ourselves a question. Is it ever possible to have a global state in which $i = 5$ and $j = 20$ when the protocol is working correctly? After some thinking, you will probably answer no. *Thus some global states are possible while some are not.* This is where invariants come in. The invariants of a protocol describe the possible states of a protocol when it is working correctly. If you really understand a protocol, you should be able to write down its invariants.

In the example above a possible invariant is:

```
If there is a frame M in the channel from S to R,
 OR if both channels are empty
   i = j
Else if there is an ack in the channel from R to S THEN
   j = i + 1
```

This is correct as far as it goes but it is not the strongest possible invariant one could use. For example, the previous invariant allows us to have a state in which there is both a frame $M$ on the link from $S$ to $R$ and an Ack on the link from $R$ to $S$. This is clearly impossible, and, if possible, would cause a contradiction in the invariant stated above. So we need one more invariant.

```
The channel from S to R is either empty or contains a frame M.
The channel from R to S is either empty or contains a frame A.
If a channel contains a frame, then the other channel is empty.
```

Now we have stated a complete description of the protocol's invariants. Often we cannot do as complete a job as we do here; we just settle for describing some important invariants. Similarly, when you do the problem in HW-4, just ask yourself whether certain global states are possible. These and similar questions should help you write down some useful invariants.

Once you have written down invariants, you can prove them by induction. You assume the invariant is initially true. (This is what protocol initialization must guarantee.) Then assume it is true in a state $s$ and show it is true in any possible state that can follow $s$. But any such state transition must be the result of a protocol action, so we need only consider protocol actions. This is where the power of invariants come in. Instead of arguing about a potentially infinite set of actions, we argue about the effect of a finite set of actions.

For example, in our example, the only interesting actions are that of initially sending a frame, receiving a frame, and receiving an ack. For example consider receiving an ack; by the first invariant we know that $j = i + 1$ in the previous state, and by the second invariant we know that there is no frame in the link from $S$ to $R$. But after receiving an ack we remove the ack from the channel, increment $i$ and send a frame. It is easy to see that this action leaves the invariant true because after the action $i = j$, there is a frame from $S$ to $R$, and the reverse channel is empty.

Finally, what use are invariants? They do help us understand how a protocol behaves. Far more importantly, they can be used to prove properties about the services the protocol provides to external users (which is all that users care about!). For instance, in the stop-and-wait bit protocol the band invariant (shown next) can be used to show that the protocol guarantees reliable FIFO delivery.

## 8 Band Invariant for Stop-and-Wait

First you should notice that the global states of even a simple protocol like stop-and wait can be quite complex. When drawing the global states, consider only the numbers in transit frames and acks. If you look at Figure 7, you see that from a single global state where a frame is still in transit, we can go to three different global states. We can lose a frame, retransmit a frame (recall that the retransmit timers are arbitrary), or deliver the first frame. Each takes us to a different global state. With arbitrary retransmission timers we can get to states where there are an unbounded number of copies of a frame and arbitrary numbers of acks.

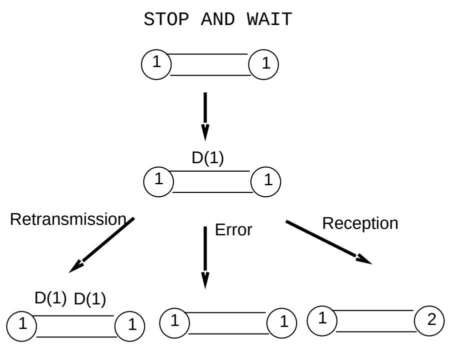

Figure 7:

However, not all global states are possible. After some thought, we can describe the possible global states (see Figure 8) as two bands of equal numbers. The first band starts at the receiver and goes clockwise (assuming the lower link is the reverse link on which acks are sent) until it either goes all way around or meets a second band of numbers. The numbers in the first band are always equal to or greater than the number in the second band. The second band grows larger until it swallows up the entire ring and then starts again as a second band. Its a little like the song, Band on the Run.

We can prove this invariant by induction. We assume it is true initially (both numbers equal to 0 and the links empty). We need to show that all protocol actions preserve this invariant. There are five cases: sending (or resending) a frame, receiving a frame, sending an ack, receiving an ack, and losing a frame or ack. The most interesting cases are receiving a frame or ack. Consider the

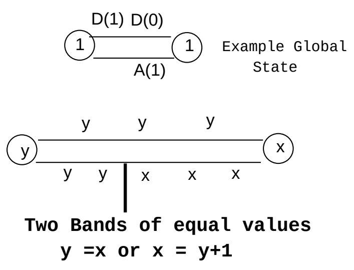

Figure 8: Band Invariant

case when the receiver gets a frame. If the number in the frame ($s$) is not equal to receiver number $RN$, then the frame is rejected. This only corresponds to removing a number from either band, which preserves the invariant. The more interesting case is when $s = RN$. In that case, if the invariant is true in the previous state, the entire band must be equal to $s$. Thus when the receiver increments to $r+1$ and sends an ack, we now create 2 bands. A smaller one starting at the receiver with values equal to $s+1$, and the rest of the circle, equal to $s$. We can argue similarly about all other cases. Try to do so yourself for practice! (You can easily dispose of many of the simple cases by observing that removing a number or duplicating a number does not affect the invariant: thus frame loss and frame sending does not affect the invariant.)

## 8.1 Getting the strongest invariant needed

One can often state too weak an invariant that cannot be proved. For example, suppose instead of the band invariant we stated only the weaker invariant that $RN \geq S$. While this is true its hard to prove by induction. Consider the case when the sender receives an ack with number $k+1$ while the receiver is at $R = k$. If this case is possible, then the sender will receive this ack and change $S = k+1$ which will cause $S > RN$.

Now in the actual protocol, this case (where an ack with number $k+1$ is on the link while the receiver is at $k$) cannot occur. Common sense tells us that the receiver number always increases and acks are only old receiver numbers. *But we cannot use common sense in an inductive proof.* We can only use the inductive hypothesis which is just the assumption that all invariants hold in the previous state.

Thus in the above example, we would (at the very least) have to add another invariant: i.e., that any acks on the reverse link have numbers that are no more than the receiver number (which rules out the crazy case which was causing us problems). The band invariant, of course, is a more succinct invariant that includes these two invariants.

In general, please note two things: a) In doing a proof you can only use the fact that all your listed invariants hold in the previous state b) If you find that there is some case in which your proof fails, you may need to add another invariant to rule out the case that fails.

## 9 So what good is the invariant

The result of the inductive proof is as follows. If the invariant is true initially and every action preserves the invariant, assuming it was true before, it is true in all states. Notice that the invariant immediately implies the two properties we had argued informally about earlier.

When the sender first goes to number $n+1$, on receiving an ack with number $n+1$, the invariant implies that the receiver is at $n+1$, any acks in transit have number $n+1$, and any data frames in transit have number $n$. But this is what we want to ensure that frame $n+1$ will be received by the receiver. Similarly, the invariant guarantees that there at most two consecutive numbers in any state which justifies the replacement of a large sequence number by a single bit.

## 10 Alternating Bit Protocol

This protocol is the same as stop-and-wait, except it uses a bit instead of an integer.

Here is the sender code:

```
Sender has a bit SN (for sender number) initially 0.
Sender repeats the following loop:
1) Accept a new packet from the higher layer if available and store it
   in Buffer B.
2) Transmit a frame (SN, B)
3) If an error-free (ACK,R) frame is received and R is not equal to SN then
      SN = R
      Go to Step 1.
  Otherwise if the previous condition does not occur with an arbitrary
  timeout period,  go to Step 2 after the timeout period.
```

Here is the receiver code:

```
Receiver has a bit RN (for receiver number) initially 0.
Receiver does the following code:
When an error-free data frame (S, D) is received:
  If S = RN then
      Pass D to higher layer
      RN = RN + 1 mod 2
  Send (Ack, RN)
```

Remember that the use of a single bit works only on FIFO links! Transport protocols, on the other hand, have to work over non-FIFO networks, and hence cannot get away with only a single bit.

## 11 Retransmission Timers

We can prove that stop-and-wait works correctly regardless of retransmission timer values. That is a nice property, because timer settings are often incorrect and if you set it one way and the link changes, the protocol will still work. However, retransmission timer values affect *performance*. Long timers will mean unnecessary delays in recovering from errors; short delays will mean extra, unnecessary retransmissions.

## 12 What Next

It is easy to see that stop-and-wait is inefficient on links with large delays (like satellite links) as you cannot send a second frame until one round-trip delay elapses. You can do better by pipelining several frame transmissions before a single ack arrives. We will study this in the next lecture.

Second, we assumed that nodes and links did not crash and the link was initially empty, a bad assumption. In the next lecture, we will show to initialize an error recovery protocol in the face of node and link crashes. All of this discussion applies equally well to transport protocols.

## 13 Some Protocol Design Lessons

Here are some ways of thinking that we used in this lecture:

- **Reduction**: We started with a simpler protocol (stop-and-wait) and reduced it to alternating bit by showing some relations among the state variables. Often its a good lesson to design a simple protocol first, and then optimize later, after understanding the simple protocol.
- **Invariants**: Rather than reason in time and say "That must have occurred because that must have occurred and keep trying to track multiple cases backwards in time" we use invariants. Invariants are facts that are "frozen in time". Rather than deal with an infinite number of possible executions, we deal with a small fixed number of possible protocol actions that could affect the invariant.