WASHINGTON UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CS423 Computer Communications: More Error Recovery
Fall 1995

This is the fourth of probably 7 lectures on the Data Link Layer. In the last lecture we studied the stop-and-wait protocol for error recovery. In this lecture, we will study more pipelined error recovery protocols, that are called *sliding window protocols.* These typically offer much larger throughput than the alternating bit or stop-and-wait protocol. We will also look at some protocol design lessons, especially the problem of initializing the sliding window protocol in the face of link and node crashes.

1 Latency and Throughput

We start by introducing two important measures that we use to compare the performance of protocols. Consider a system (like a network) where jobs (like messages) arrive, and after completion leave the network. *Throughput* roughly measures the number of jobs completed per second. *Latency* measures the time (worst-case or average, we will typically consider worst-case) to complete a job.

The owners of a system want to maximize throughput to maximize revenues, while users of a system want low latencies so they don't waste their time. Consider a doctor's office. Often they keep you waiting for a long time so that you will be ready when the doctor is ready. They are optimizing for throughput, not to minimize your latency. A busy traffic signal should typically maximize for throughput by having each signal direction stay on for a long time; this minimizes the startup overhead every time the signal changes. However, that means that even if there is nobody at the intersection, you may have to wait a long time till the signal light changes if you are unlucky and arrive just as your light changes to red.

Another interesting point (exemplified by the traffic light example) is throughput is more interesting for busy systems and latency is more important for idle systems.

One might think that throughput is just the reciprocal of latency. That is *not* true when the system is pipelined — i.e., the number of users that may being serviced at the same time inside the system. For example, if the system consists of 8 service stations that take 1 unit of time each. If each job must go through each station, the throughput can be 1 job per unit time, while the latency is 8. On the other hand, suppose each job needs only 1 service station and any service station will do. Then the throughput can be 8 jobs per unit time, and the latency is 1.

For networks, the jobs are messages and the system is a network. The service stations correspond to a series of hops. However, in this lecture we will confine ourselves to a single hop.

Two other important definitions we need to make for a Data Link are *transmission rate* and *propagation delay.* The transmission rate at the Data Link is the rate at which the physical later sends bits. Thus a physical layer that sends 1000 bits a second, sends a bit every msec. The propagation delay is the time it takes for a single bit that is sent to arrive at the receiver. Propagation delay is limited by the speed of light to at least 3 usec/ per km. (Electrical signals travel even slower) and depends on the distance of the link between the sender and receiver. Consider a sender and receiver that are connected by a 50 km link. Assume speed of light propagation, that amounts to a 0.15 msec propagation delay and 1000 bit/second transmission, a 10 bit frame will take 1.15 msec to be received completely: 10 msec to transmit all the bits, and .15 msec for the last bit to reach the receiver.

The throughput of a link can be made independent of its latency by pipelining. However, for obvious reasons, the throughput can never exceed the transmission rate. Finally, the propagation delay is always an unavoidable part of latency on a link.

For unfortunate reasons, many people use the word bandwidth to talk of the transmission rate on a link. At the physical layer, bandwidth is the range of frequencies that are passed through. At the data link layer, it is often used to mean the link transmission rate. Please try to understand by context which meaning is being used. Another phrase that is used often is *round-trip delay.* This is the delay to send a frame from a sender to a receiver and to receive a reply frame from the receiver. It includes the time to transmit both frames, as well as two propagation delays.

2 Why Pipeline?

Consider a satellite link that transmits at a speed of 100 Kbit/sec using 1000 bit data frames. Using a stop-and-wait protocol, and assuming a propagation delay of 250 msec (remember a satellite link must go all the way up to the satellite, a large distance), we can send only 1000 useful bits every 500 msec. This leads to a throughput of 2000 bit/sec. Thus stop-and-wait limits us to using only 2000/100,000 of the link bandwidth (new sense of the term bandwidth!!) which means that the link is being utilized at only 2 percent. If the satellite link costs several hundred thousand dollars a month, the users of this system should be unhappy.

The problem, of course, is that stop-and-wait limits us to sending one frame every round-trip delay. This is a useful point: the performance of a network can be greatly influenced by the choice of protocol.

The stop-and-wait problem is not limited to satellite links. It really applies to all links such that the link transmission rate multiplied by the propagation delay is large compared to the frame size. This is sometimes called the *bandwidth-delay product* or pipe size (because it measures the number of bits that can physically be stored on the physical link.) Since the speed of light has been constant for centuries, but transmission rates keep improving, the bandwidth-delay product keeps increasing. Consider a fibre optic link that links two locations on two coasts. Assuming a coast-to-coast propagation delay of 20 msec, and a transmission speed of 100 Mbit/sec, and a frame size of 1000 bits, we can store 2000 frames on the physical link! Thus stop-and-wait would restrict us to using only 1/4000 of the link bandwidth.

For all these reasons, pipelined data link protocols are essential. Of course, most high-speed data link protocols do not do error recovery, so this problem does not arise. However, transport protocols have the same problems end-to-end, and thus it is crucial that they use pipelined error recovery protocols. In the rest of the lecture we will study pipelined *sliding window protocols.*

3 Sliding Window Protocols

In a sliding window protocol, the sender can send a *window* of outstanding frames before getting any acknowledgements. The window size limits the degree of pipelining. Consider a sliding window protocol with a window size of w. As in the stop-and-wait protocol, lets start by using large sequence numbers first. The sender maintains a single variable, a *a lower window edge* L (sometimes called the lower window) which represents the lowest sequence number that the sender has not received an ack for. The sender can also be thought of as maintaining an upper window edge, which is equal to $L+w-1$, which is the maximum sequence number it is allowed to send. The receiver maintains a receive sequence number R that is the next number it expects to see (just as in stop-and-wait). L and R are initially equal to 0.

In the stop-and-wait protocol, the sender keeps retransmitting the frame with sequence number equal to the sender sequence number until it gets an ack. In sliding window protocols, the sender keeps retransmitting all frames in its current window until it gets an ack. The receipt of an ack numbered R implicitly acks all sequence numbers that are strictly less than R.

There are two important variants of sliding window protocols. In the first, called go-back-N, the receiver only accepts frames in order. Suppose the receiver is expecting frame number 0, and the sender sends frames 0 and 1. If frame 0 gets lost and frame 1 arrives at the receiver, a go-back-N receiver will discard frame 1 because it arrived out-of-order. It is only when the sender retransmits 0 and 1 and they both arrive in order that the receiver will accept both frames. This version is simple to implement but it implies that the loss of a single frame in a window can cause the sender to have to retransmit the entire window (hence the name, with n representing the window size).

A fairly obvious fix is to have the receiver not discard frame 1 in the above example but to buffer it at the receiver until the sender resends frame 0. WHen frame 0 arrives, the receiver can then release frames 0 and 1 to the client. This avoids the sender from having to unnecessarily retransmit frame 1. It has a slightly more complicated implementation, however, and so is as not used as often as one might like. The difference between the two is slight if the error rate is low and the window size is small. However, for large pipe sizes (common in high speed networks) and fairly large chances of losing frames, selective reject is important. In particular, for high-speed transport protocols where most frames are lost due to congestion (which can be quite common) and the window sizes are large, selective reject is very desirable.

4 Example

Figure 1 shows an example that contrasts the two variants of sliding window. On the left we show go-back-n operation for a window size of 3. When we use a pair of numbers $m, m+w-1$ next to the sender, it means that the sender can send frames in the range. Notice that after the first ack is received, the window slides to $1, 3$. If acks keep coming back smoothly and the window is sufficiently large, the pipeline can keep flowing smoothly with the ack for the first frame in the window being received just after the last frame in the window is transmitted. This would ensure that the transmitter and the transmission line are never idle.

However, the example shows that frame number 1 is lost. In the go-back-3 version, the sender uselessly transmits 2 and 3. Only when a timer expires, and the entire window is retransmitted (for go-back-n, it suffices to have a single timer that is refreshed whenever the window slides; if it expires, the entire window is transmitted in order). In the selective reject scheme, on the other hand, frames number 2 and 3 are buffered at the receiver until the retransmitted 1 arrives; at this point the receiver number jumps to 4, and the receiver delivers frames 1, 2, and 3.

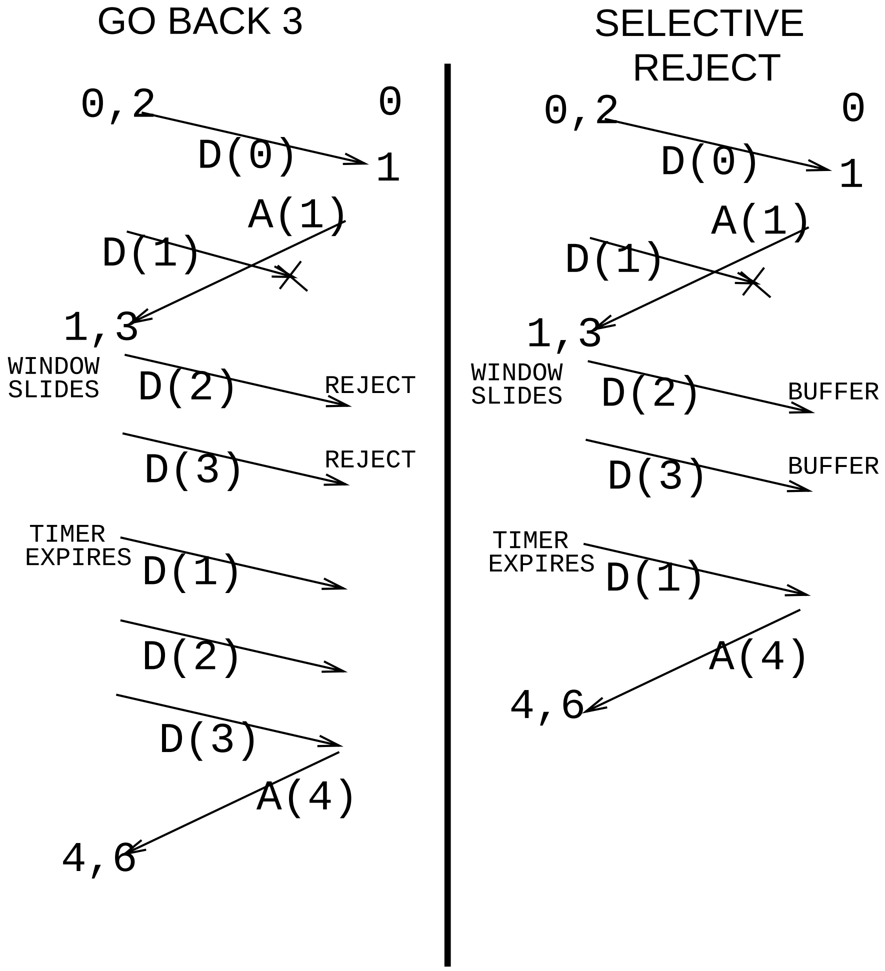


Figure 1: An example of Go-Back n and Selective Reject Operation. Note that selective reject can recover faster from errors as sender has only to retransmit lost frames.

5 Sliding Window Code

The code for Go-back-n sliding window protocols is given below. To make the code more general, we allow the receiver to send acks whenever it wants to, not just when it gets a data frame, though that is typical.

Assume that the sender has a long sequence of data packets given to it by its client, that it wants to send. The data packets are stored at the sender. Number the data packets from 0 onwards. We will send the s-th data packet with sequence number s attached.

```
Sender code for Go-back N:

Assume all counters are large integers that never wrap.
The sender keeps a lower window L, initially 0.

Send (s,m)  (* sender sends or resends s-th data packet *)
    The sender can send this frame if and only if:
       m corresponds to data packet number s given to sender by client AND
       L <= s <= L + w - 1 (* only transmit within current window *)

Receive(r, Ack) (* sender absorbs acknowledgement *)
   On receipt, sender changes state as follows:
      L := R

The receiver keeps an integer R which represents the next sequence number
it expects, initially 0.

Receive(s,m) (* receiver gets a  data frame *)
     On receipt, receiver changes state as follows:
        If s = R then (* next frame in sequence *)
          R := s + 1
          deliver data m to receiver client.

Send(r, Ack) (* we an allow receiver to send an ack any time *)
       r must be equal to receiver number R at point ack is sent
       most implementations send an ack only when a data frame is received

We assume that any unacknowledged frame in current window is periodically
resent.  In particular, the lowest frame in the current window must be
periodically sent to avoid deadlock.
```

The code for selective reject is given below. This time we assume that the sender keeps a table that records which numbers greater than L have been acked (while we assume a large array in the code, it suffices to keep a bitmap representing numbers from L to L + w - 1.) Similarly, we assume the receiver has a similar table that stores both a bit (indicating receipt) and a pointer to the data if the frame has been received.

In the last example, we saw that the sender only need retransmit the frames that are actually lost. In order to prevent the sender from doing useless retransmissions, the receiver needs to send an ack containing its current number R as well as a list of numbers greater than R that have been received out of sequence at the receiver.

```
Sender code for selective-reject:

Assume all counters are large integers that never wrap.
The sender keeps a lower window L, initially 0 and a table that
indicates which numbers have been acked (this can be optimized to store
only a windows worth of such state).

Send (s,m)  (* sender sends or resends s-th data packet
    The sender can send this frame if and only if:
       m corresponds to data packet number s given to sender by client AND
       L <= s <= L + w - 1 (* only transmit within current window *) AND
       s has not been acked.

Receive(R, List, Ack) (* sender absorbs acknowledgement *)
   On receipt, sender changes state as follows:
      L = R
      Mark every number in List as being acked in table.

The receiver keeps an integer R which represents the next sequence number
it expects, initially 0.  The receiver also keeps a table that, for
each sequence number, stores a bit indicating whether it has been received
and a pointer to the data, if any, being buffered.  Once again, this table
can be optimized to reduce the amount of storage to be proportional to
a window size.

Receive(s,m) (* receiver gets a  data frame *)
     On receipt, receiver changes state as follows:
        If s >= R then
          Store m in table at position s and set bit in position s
          While the bit at position R is not set do
             Deliver data at position R
             R = R + 1

Send(R, List, Ack) (* we an allow receiver to send an ack any time *)
       R must be equal to receiver number R at point ack is sent
       List consists of numbers greater than L that have been received.
       most implementations send an ack only when a data frame is received

We assume that any unacknowledged frame in current window is periodically
resent.  In particular, the lowest frame in the current window must be
```

`periodically sent to avoid deadlock.`

6 Implementation Details

The code we have shown does not show timers that are used in any real implementation. As we said earlier, in go-back-N implementations, it suffices to have one outstanding timer. The timer is set to some multiple of the average round-trip delay. (The round-trip delay can be calculated by seeing how long it takes for acks to arrive; we"ll examine this in more detail when we look at transport protocols.) Notice, that as usual the protocol will work correctly if the timer values are set wrong; it will only cost in performance. In selective reject, we have to set a timer for every outstanding frame in order to retransmit any outstanding frame in the window.

The ack list can be represented as a bit map to save bits. In order to further reduce the ack bandwidth, some implementations often *piggyback* acks on data flowing in the reverse direction. In order to allow this, the data frames must have fields in their header that can be used to put in reverse ack information. While the saving in bits can be small (you only save the rest of the header and framing information), it does reduce the number of frames that the sender has to process. It turns out that there is a large fixed cost (e.g., interrupt processing) for processing frames, and it pays to reduce the number of frames.

7 Sequence Number Space for Sliding Window Protocols

For the stop-and-wait protocol, we showed that a sequence number space of two was sufficient. In other words, if we did all arithmetic mod n, where n is the size of the space, it all works out OK. Similar arguments hold for sliding window protocols, but the arguments differ for the different variants.

7.1 Go-back-n Modulus

Consider first go-back-n. First, notice that it is a strict generalization of stop-and-wait if we set the window size to 1. Thus if we can get away with a sequence number space of 2 (1 + 1) in stop-and-wait, perhaps we can get away with a space of $w + 1$. This is indeed true but the argument is more tricky. This is because in sliding window protocols we can show that there can be $2w$ distinct numbers simultaneously in the system (i.e., including both links) in some executions (try to find this example). For $w = 1$, this gives us 2, which is also equal to $w + 1$. However, that seems to indicate you may need a space of $2w$.

Figure 2 shows why a space of $w + 1$ is sufficient for go-back-n. Notice that in the code, the only comparison between sequence numbers is when the receiver checks whether an incoming frame with number s has number $s \neq R$. Suppose we can show that both s and R are always within w of each other. Then a space of $w + 1$ will suffice.

The proof is a little tricky. We will not use a proof that uses invariants through such a proof can be constructed. Consider the receipt of frame number s when the receiver number is R. Consider the state of the sender when s was sent; clearly s must be in the range L to $L + w - 1$, where L is the lower window at the time frame s was sent. Now if the sender number is L at this earlier point in time, it must be because it received an ack from the receiver with number L. This means that the receiver number was L some time in the past. But since the receiver number only increases or stays the same, it means that the receiver number when frame s is received is at least L, where L is the lower window at the point s is sent.

Similarly, it is easy to see that the sender has never sent frames with numbers larger than $L+w-1$ before frame s was sent (because of the window size restriction). Since the links are FIFO (this is the only place we use the FIFO property of the physical link), no data frame with number greater than $L+w-1$ can have arrived before frame s arrives. Thus the receiver number cannot be greater than $L+w$ (recall that the receiver goes to one higher number than the last frame number received in sequence), where L is the lower window at the point where frame s was sent.

Thus we know that $L \leq R \leq L+w$. But we know (because only frames within current window are sent) that $L \leq s \leq L + w - 1$. Thus the absolute value of $R - s$ is no greater than w. Hence we can get away with a modulus of size $w + 1$, because any two distinct integers that differ by at most w will be assigned distinct numbers in the modulus space.

If we use a smaller space, we can easily cause problems. For instance, if we use a space of 0 to 7 and a window size of 8. Then after the receiver receives 0 to 7 and then gets a 0, the receiver cannot tell whether the 0 is a retransmission from the previous window or a new frame from the next window. Accepting a retransmission will cause a duplicate to be delivered.

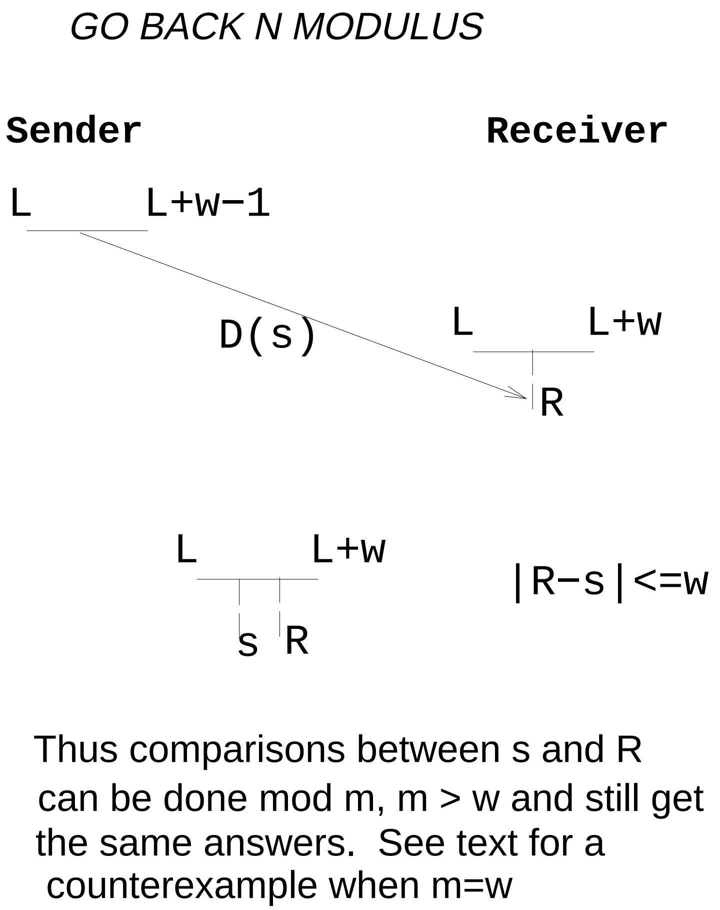
7.2 Selective Reject Modulus

The same inequalities hold for selective reject. However, recall from the code that selective reject involves a comparison of the form $s > R$ as opposed to $s = R$. Now from the previous inequalities, we know that s and R lie within a range w of each other. A slightly more careful analysis shows that $R - w \leq s \leq R + w - 1$. To find if $s > R$, the receiver needs to distinguish the previous w frames from the next w frames, which leads to a space of $2w$ numbers at least.

If we use a smaller space, we can easily cause problems. For instance, if we use a space of 0 to 7 and a window size of 5. Then after the receiver receives 0 to 5 and then gets a 0, the receiver cannot tell whether the 0 is a retransmission from the previous window or a new frame from the next window. Accepting a retransmission will cause a duplicate to be accepted. The text gives a more detailed example on Page 231.

8 Flow Control

Another problem that is often solved by transport protocols and some data link protocols is the problem of avoiding the sender from sending at too high a rate so that the receiver will run out of buffer space. This is typically solved in sliding window protocols by having the receiver have at least w buffers. If the receiver wants to dynamically adjust the window, the receiver can add a field to its acks, indicating that its willing to receive so many frames beyond R. This is most



Thus comparisons between s and R can be done mod m, m > w and still get the same answers. See text for a counterexample when m=w

Figure 2: Argument for go-back-n modulus looks back in time to the point that the frame s was sent and the lower window L at that point.

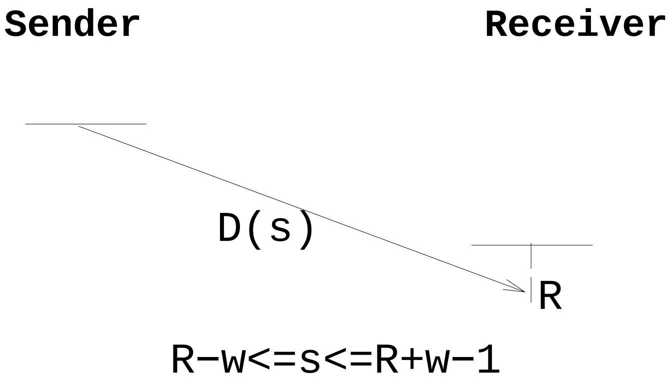
useful when the receiver is sharing buffers among multiple senders; we will study this technique of dynamic window sizes when we study transport protocols.

For protocols that do not do reliable error recovery (for example, ATM data links), a simple method is to have the sender never send more than x bytes in every T seconds. If the receiver can keep up with this rate, this works well. Once again, there are variants in which the rate can be adjusted dynamically.

9 Initializing Sliding Window Protocols

So far we have assumed that the receiver magically starts with sequence number 0 and the sender with number 0. What if the sender crashes while the receiver is at number 55. If the sender starts with number 0 again, the receiver will reject all frames sent, and the protocol will deadlock. What is needed is a way for the sender to *reset* the protocol after the sender crash (and vice versa, if the receiver crashes).

SELECTIVE REJECT MODULUS



Since receiver must buffer n packets ahead, simple equality testing is no longer enough. Must be able to tell apart the last w frames from next w frames. Needs modulus m >=2w

See text for counterexample if m < 2w

Figure 3: Selective Reject Modulus satisfies same inequalities as Go-back-n. However, the required modulus size is greater because slective reject needs to know if the received sequence number is greater than the receiver number.

Assume that the sender is the leader of the protocol. The sender can initialize whenever it wants to; if the receiver crashes, it must send a request for a reset to the sender. If the sender's reset message arrives before the receiver sends a request, the receiver does not send a request because it assumes the sender is already resetting. The normal procedure one might think of for a reset is shown on the left in Figure 4. The sender sends a Reset message and waits for an ack (RA); when the receiver gets a Reset message it resets its sequence number. When the sender gets the RA, the sender resets its sequence number and starts sending data items.

This procedure can fail as shown on the right. Essentially an alternating series of crashes at the sender and receiver force each node to respond to messages sent by previous incarnations. Finally, the sender sends a data frame that is lost, but an ack from an earlier incarnation arrives and fools the sender into thinking that it has arrived.

We can show that sending more restart messages cannot help. Essentially the alternating crashes force a node to receive the first i messages from a previous incarnation and send the first $i + 1$ messages before crashing again. By continuing inductively, we force the receiver to emit an ack for the first data frame and this "old" ack can fool the sender. Thus dallying further can only delay your doom.

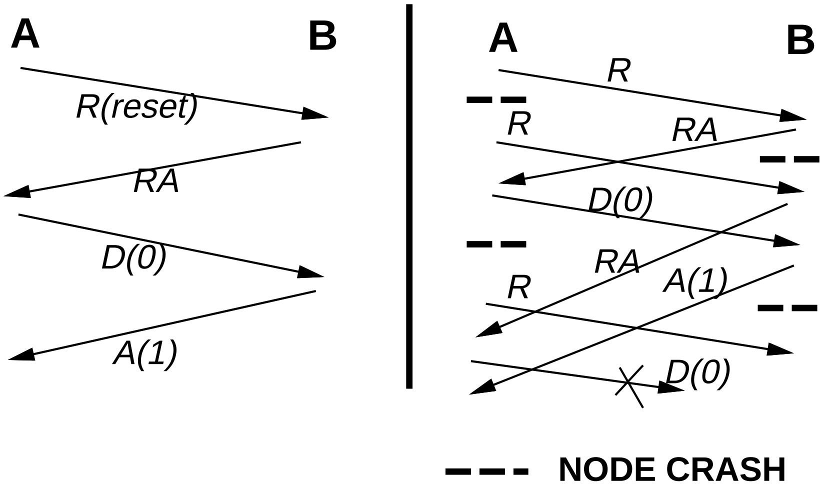


Figure 4: Naive Procedure to restart a protocol and why it can fail after crashes

9.1 Impossibility Result

The problem in Figure 4 arises because nodes are "fooled" by older responses to their restart messages. An easy way to fix this is by numbering all restart messages. However, that does not help if the sender does not keep memory after a crash (i.e., a diskless workstation). In that case, the sender cannot reliably number its restart messages to be different from previous restart messages.

In general, we can show that there is no deterministic (i.e., that works always with zero probability of failure) protocol to initialize a Data Link assuming that the physical links take unbounded and varying delays, and that the sender has no memory after a crash. This result is due to Lynch, Mansour, Fekete, and Spinelli. Amazingly, it was discovered only recently. Older protocols like HDLC are indeed susceptible to this problem. However, the error scenarios are extremely unlikely and so no damage has probably been done. But there is a way to fix the problem!

9.2 Reliable Restarts: fixing the problem

The problem can be fixed by changing any of the assumptions in the impossibility result:

- *Non-volatile memory:* If we have non-volatile memory after a crash, the sender can keep a crash-count on disk that is incremented after every crash. RESTART messages are labeled with this counter, and RESTART-ACKS are accepted only if they match the current crash counter.

- *Probabilistic Protocol:* Instead of using a crash counter, use a random number to label RESTART messages. This should succeed with high probability if the number of possible random numbers is much larger than the number of frames that can be stored in the links.

- *Assume a time bound:* Most real protocols can assume that all frames on the link will either be lost or delivered in some maximum time delay, say 2 minutes. This delay can be set conservatively. Then after a crash the sender must wait for this time before sending RESTART messages.

These techniques and problems also apply to transport protocols as we will see later.

10 What makes Protocols so Hard

Suppose you want to meet your friend at the student center in five minutes. Just as you tell your friend "Let's meet in 5 minutes", the phone goes dead. You can't tell whether your friend heard your last statement. If he did, he might go to meet you; if he didn't you might go alone. It turns out that there is no way to avoid the problem, if communication can fail at the crucial juncture. Since one side must change state before the other, and communication can fail at that point, there is no way to prevent this possibility. It is sometimes called the Coordinated Attack or Two Generals problem in which 2 generals need to attack a distant fort at the same time, and they can communicate only by unreliable messengers. It is always possible for disaster to occur, wherein one general attacks alone (unless both general always don't attack, which is not very useful).

For network protocols, it means that since links can lose messages, it is impossible for two nodes in a protocol to change state at the same time. One node must change state before the other. So if we have two banks and we want to transfer 1000 dollars from one bank to another, we have to be prepared for an intermediate state in which one bank has withdrawn money and the other has not received the deposit. More practically, if a router changes over to a different route, there will be a period in which the other routers may be using the old route. One has to be prepared for periods of inconsistency and prevent damage during such periods. For instance, during periods of inconsistent routes, packets can loop, and we need hop-counts to prevent packets from looping forever.

Another way to state this limitation, is that it is impossible for nodes (in a non-trivial protocol that works over links that can lose messages) to reliably know the state of other nodes at all times. This happens in the Data Link protocols. There is always one node that does not know the state of the other node. For instance, in Stop-and-Wait, when the sender gets an ack, it knows the receiver number; however, at that instant, the receiver does not know the sender number (because it may or may not have received the ack as far as the receiver is concerned.)

11 Protocol Design Lessons

We enumerate the following protocol design lessons:

- If you can, design protocols whose correctness does not depend on timer settings; if you have to, make sure that the timer values are set conservatively to handle different installations.
- Don't add complexity until its justified. We went to sliding windows and selective reject only after we figured out the performance degradation of using stop-and-wait.

- Design Simple Protocols first and then optimize later. Separate out implementation details from the essential features. We did this when designing the selective reject and go-back-n protocols.

- It is insufficient to understand protocols when they are operating correctly. You need to understand how they recover after faults, including node crashes. Consider using the restart methodology we described to completely restart the states of the nodes after a crash, especially if its a 2-node protocol.

- Impossibility Results should teach us what we have to change to do our jobs, and not what we cannot do. Thus the crash impossibility result motivated three solutions, each of which was a consequence of changing some assumption in the impossibility result.