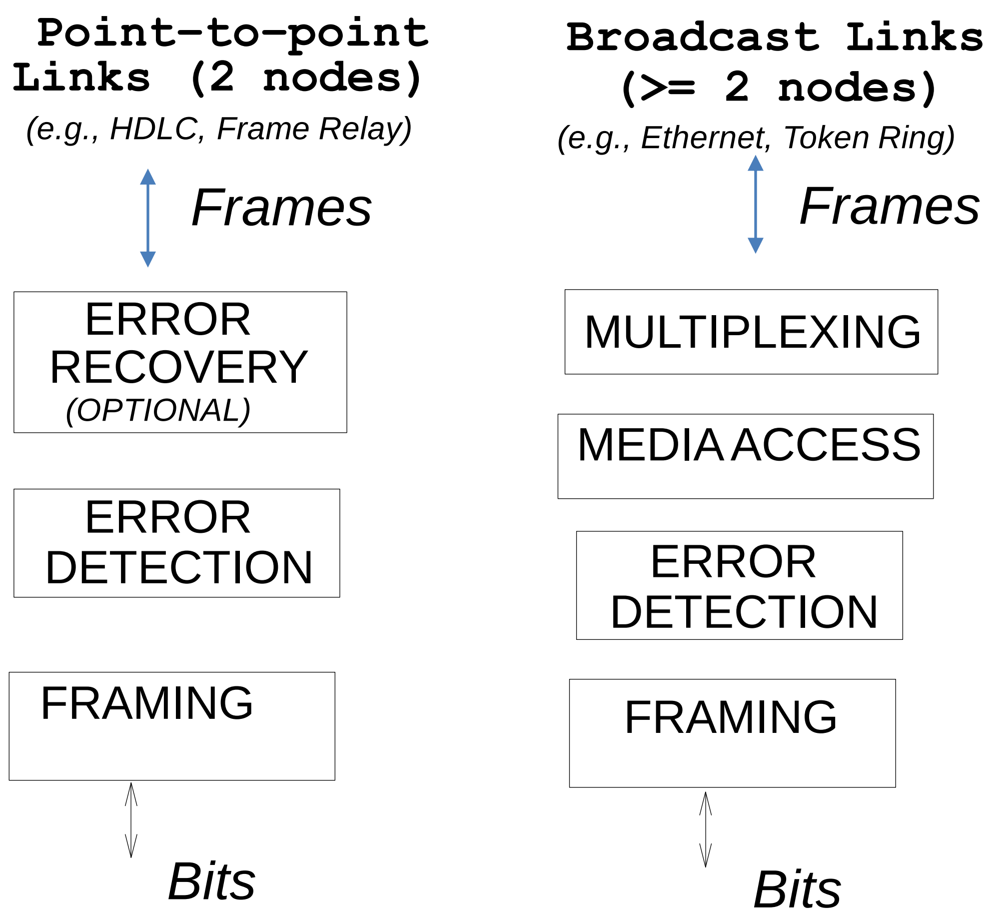
CS 118: Data Link, Error Recovery Slides

George Varghese

Data Link Sublayers



QUASI-RELIABLE 1 HOP FRAME PIPE

Why error recovery at Data Link?

- Used later: Protocols like TCP use the same ideas. When we get to TCP, we will only study differences.
- Still used at Data Link: some existing protocols like HDLC and Fiber Channel still do error recovery at Data Link.
- Non-trivial protocol: Our first example of the problems caused by varying message delay and errors (frame loss, crashes)
- Technology seesaw: Hop-by-hop becoming popular again to reduce latency. Wheel of time!

Correctness Specification

Packets given to the sending Data Link must be delivered to receiver without duplication, loss, or mis-ordering

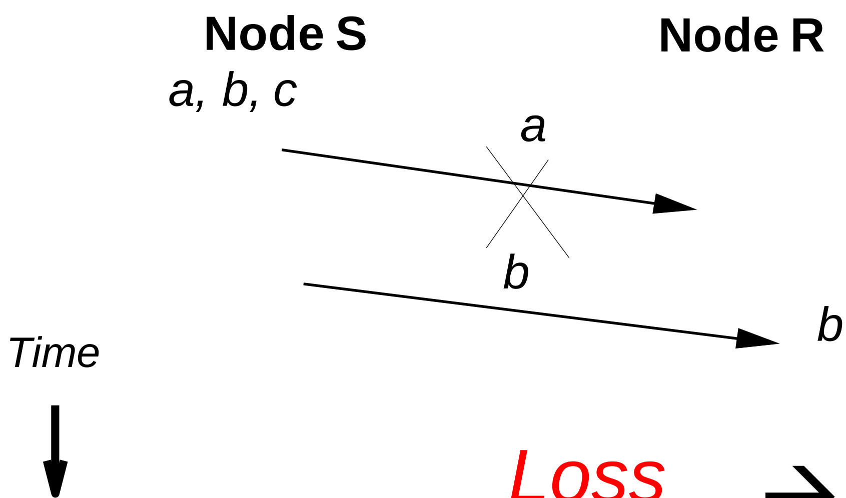
Assumptions

- Assumes error detection: Assumes undetected error rate small enough to be ignored
- Loss as well as errors: whole frames can be lost in a way not detected by error detection
- FIFO: Physical layer is FIFO
- Arbitrary Delay: Delay on links is arbitrary and can vary from frame to frame.

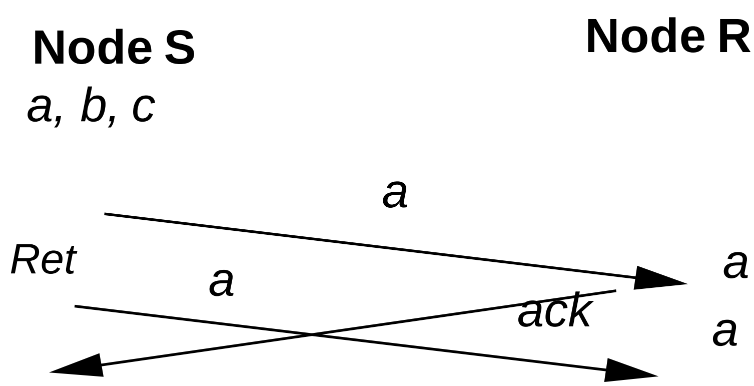
Assumptions

- Assumes error detection: Assumes undetected error rate small enough to be ignored
- Loss as well as errors: whole frames can be lost in a way not detected by error detection
- FIFO: Physical layer is FIFO
- Arbitrary Delay: Delay on links is arbitrary and can vary from frame to frame.

Protocol Plays to motivate Stop and Wait Protocol



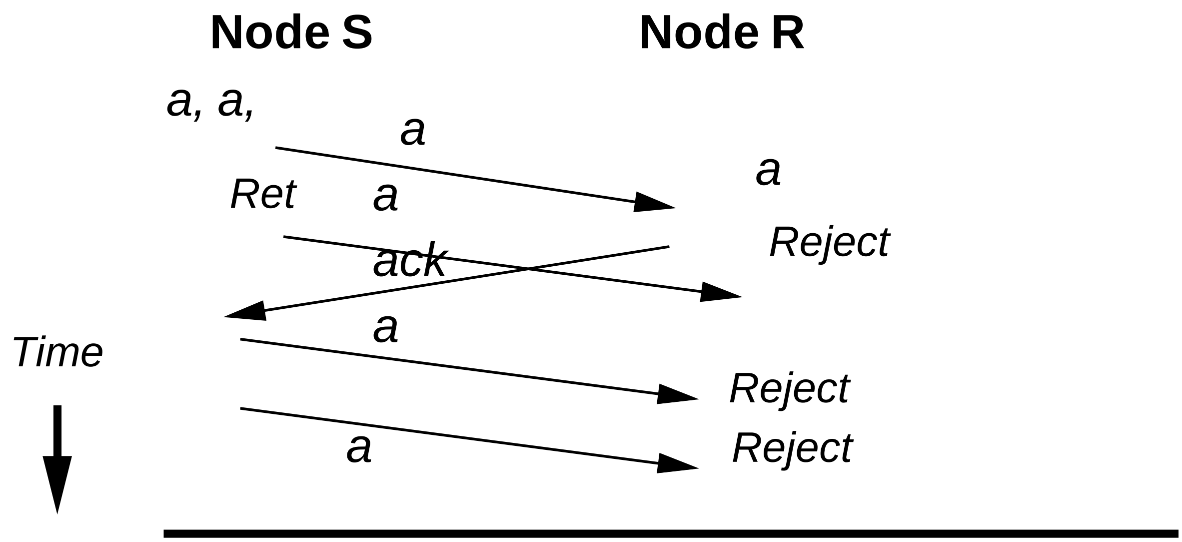
Loss → Must retransmit



Duplication

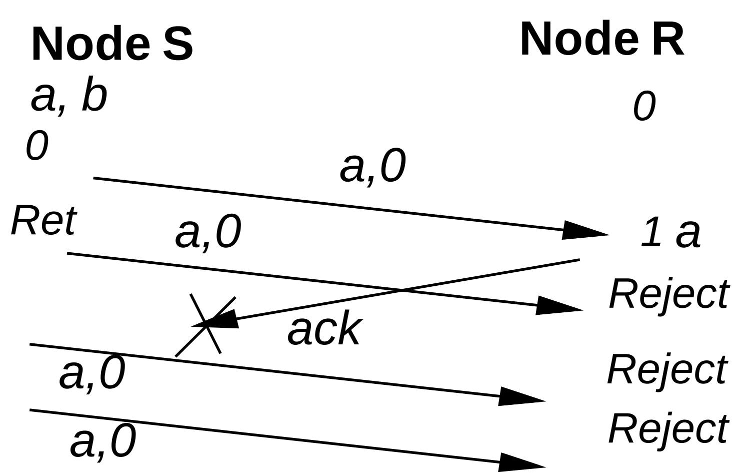
→ Must defend against early retransmits

More Protocol Plays



Livelock

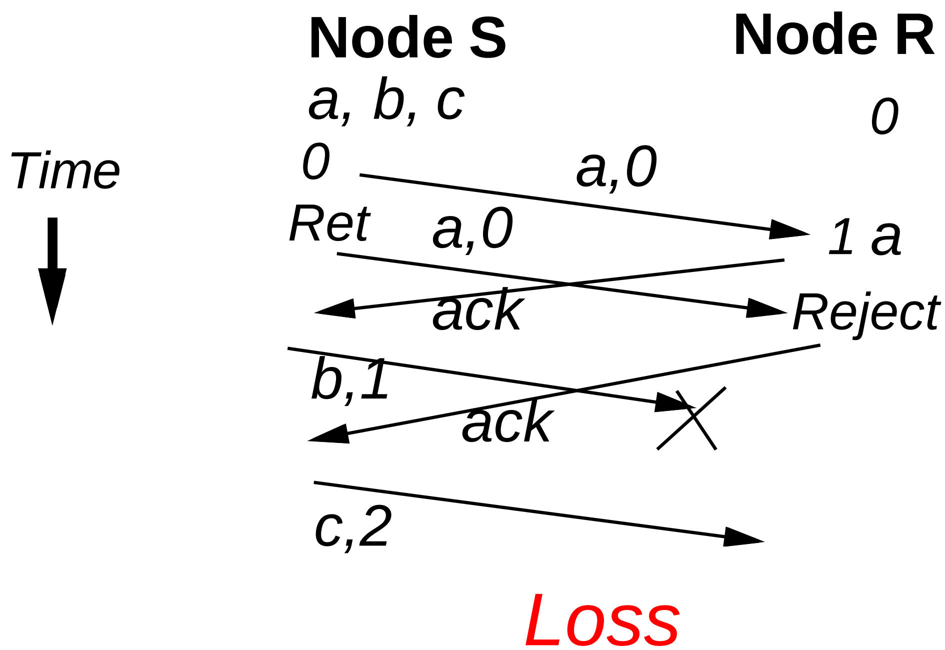
→ Need sequence numbers



Livelock

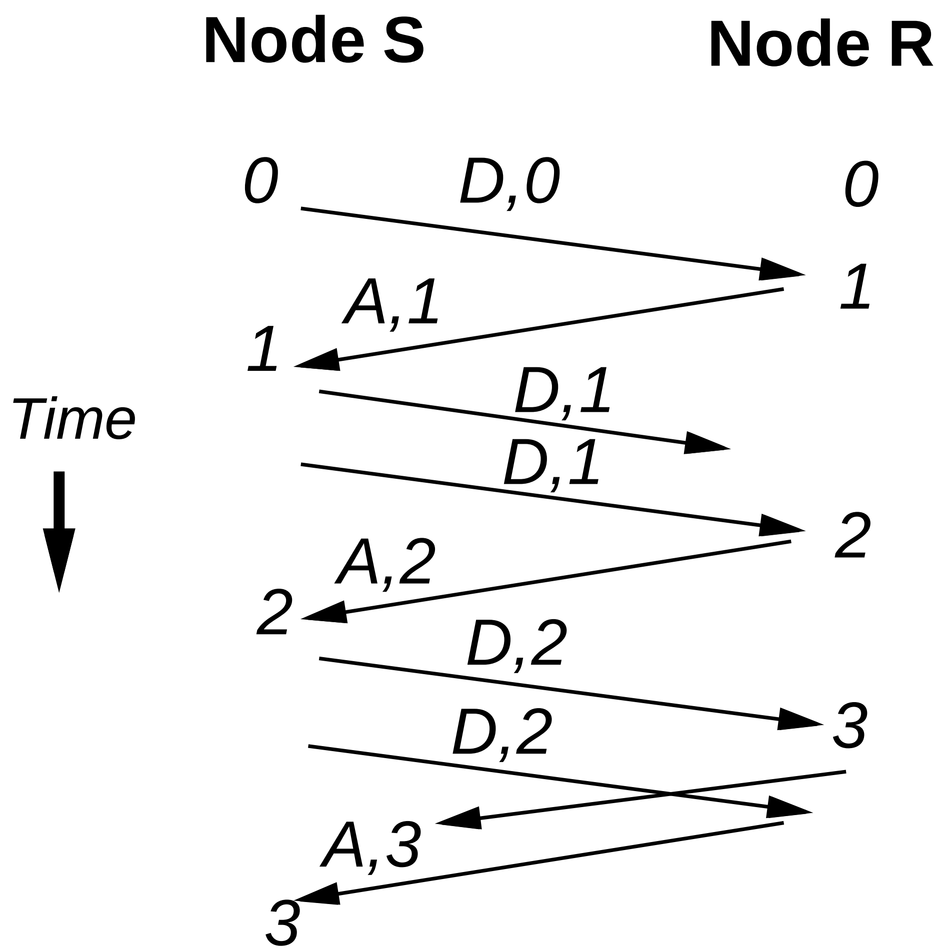
→ Must ack even duplicates

Final Protocol Play



→ Must number acks

Sketch of Final Protocol



Going deeper

- Correctness: How do we we show it works in all cases (infinite sequence of executions): invariants
- Sequence Number: Do we need an infinite number of bits? Heck, no
- Performance: send only one at a time. Bad over satellites. Can do better → sliding window.
- Initialization: How do we get started and synchronize sequence numbers in face of crashes

Code (buggy) of Stop and Wait

-----------Sender Code---------------------------

Sender keeps state variable SN, initially 0 and repeats following loop

1) Accept a new packet if available from higher layer and store it in buffer B

2). Transmit a frame Send (SN, B)

3). If error-free (ACK, R) frame received and R != SN then

    SN = R

    Go to Step 1

Else if the previous condition does not occur after T sec

    Go to Step 2

Receiver Code ------------------------------------

Receiver keeps state variable RN, initially 0

When an error free data frame (S, D) is received

On receipt:

    If S = RN then

        Pass D to higher layer

        RN = RN + 1;

        Deliver data m to client.

        Send (ACK, RN)

Correct Code of Stop and Wait

-----------Sender Code-------------------------------

Sender keeps state variable SN, initially 0 and repeats following loop

1) Accept a new packet if available from higher layer and store it in buffer B

2). Transmit a frame Send (SN, B)

3). If error-free (ACK, R) frame received and R != SN then

    SN = R

    Go to Step 1

Else if the previous condition does not occur after T sec

    Go to Step 2

Receiver Code -------------------------------------

Receiver keeps state variable RN, initially 0

When an error free data frame (S, D) is received

  On receipt:

    If S = RN then

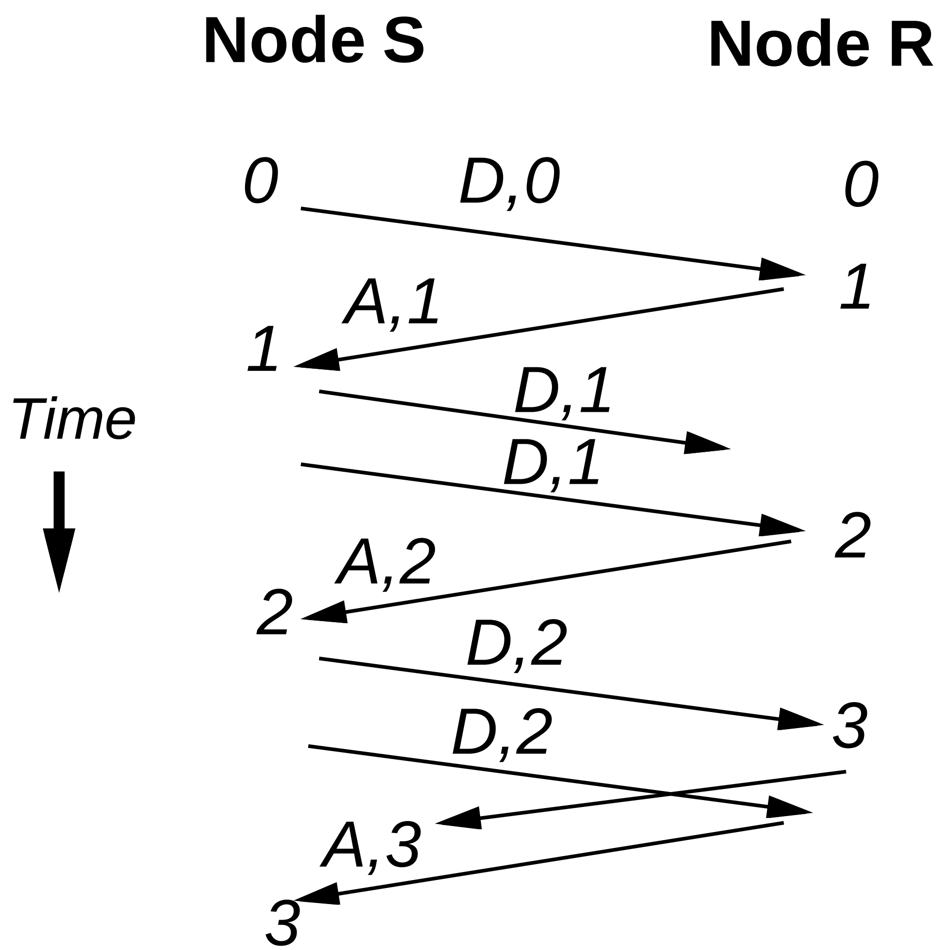
      Pass D to higher layer

      RN = RN + 1;

      Deliver data m to client.

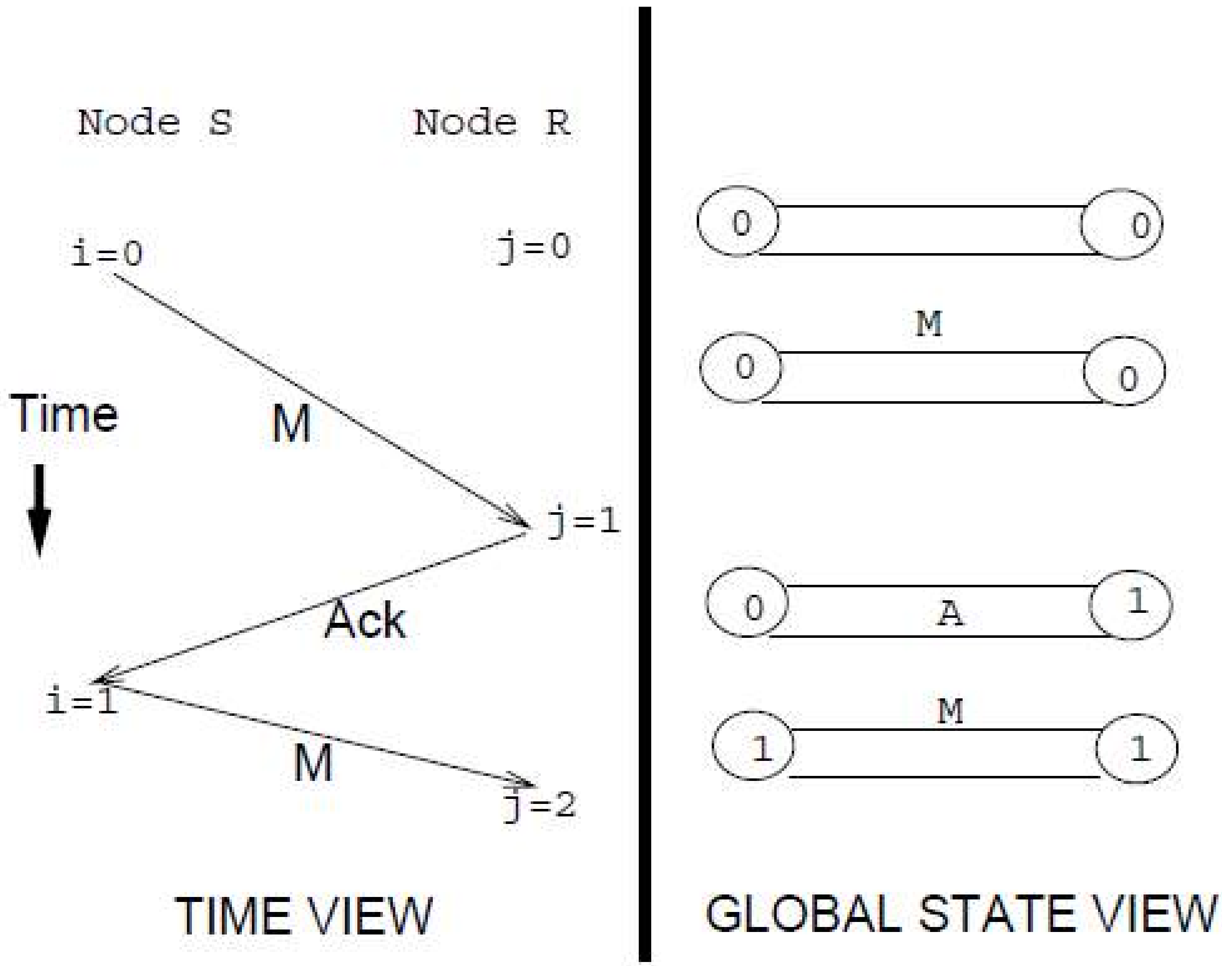
    Send (ACK, RN) // Send ack unconditionally!

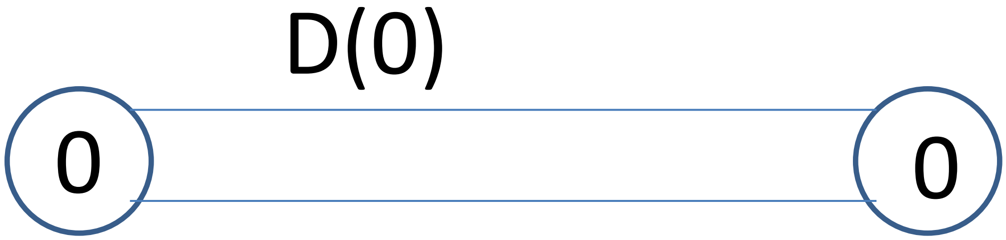
Correctness Observations



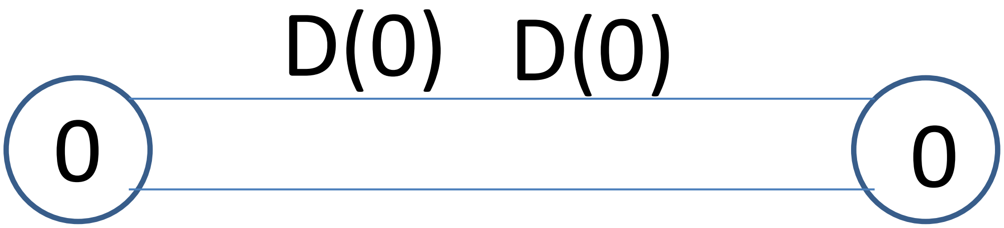
- When sender first gets to N, no frames with N or acks with N+1 and receiver is at N
- When receiver first receives frame N, entire system only contains number N→ only two numbers in system

How to reason: from time-space to Global States

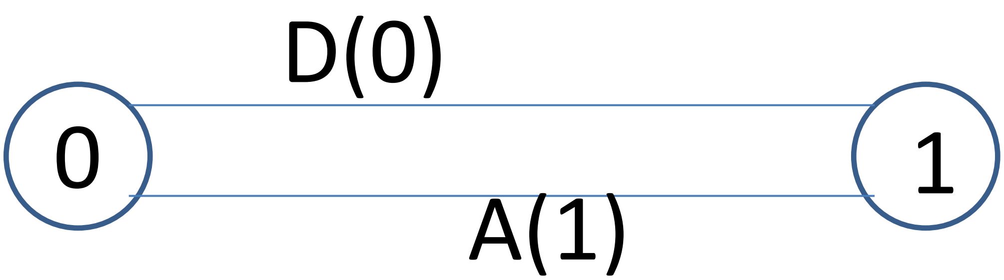




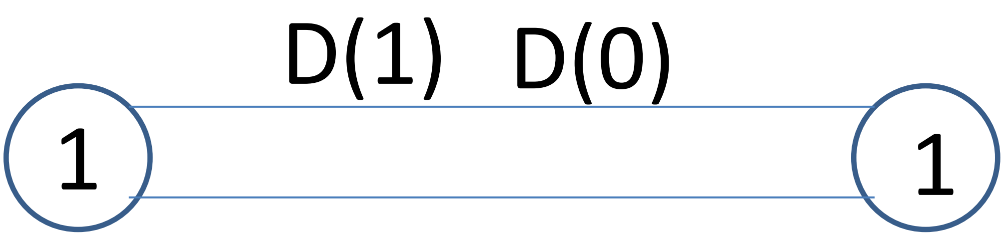
Send frame 0



Retransmit early

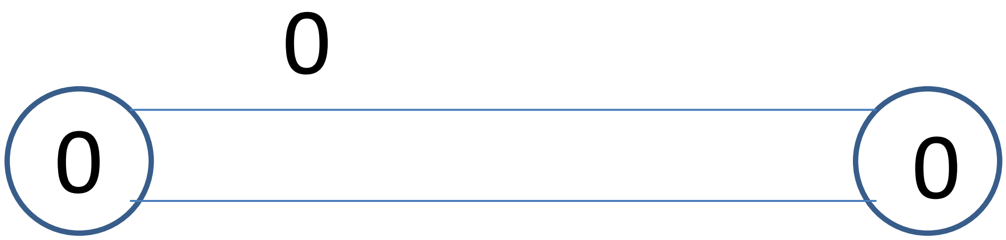


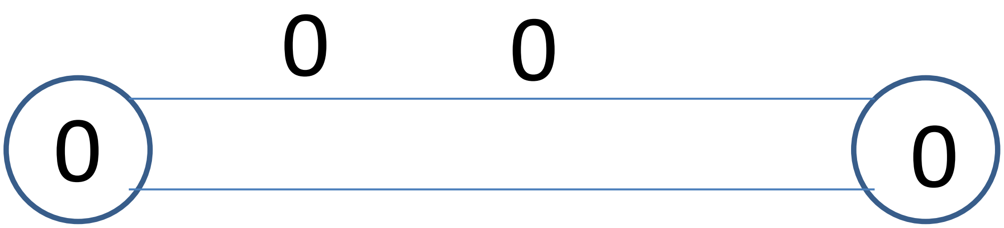
Receive D(0); Send Ack

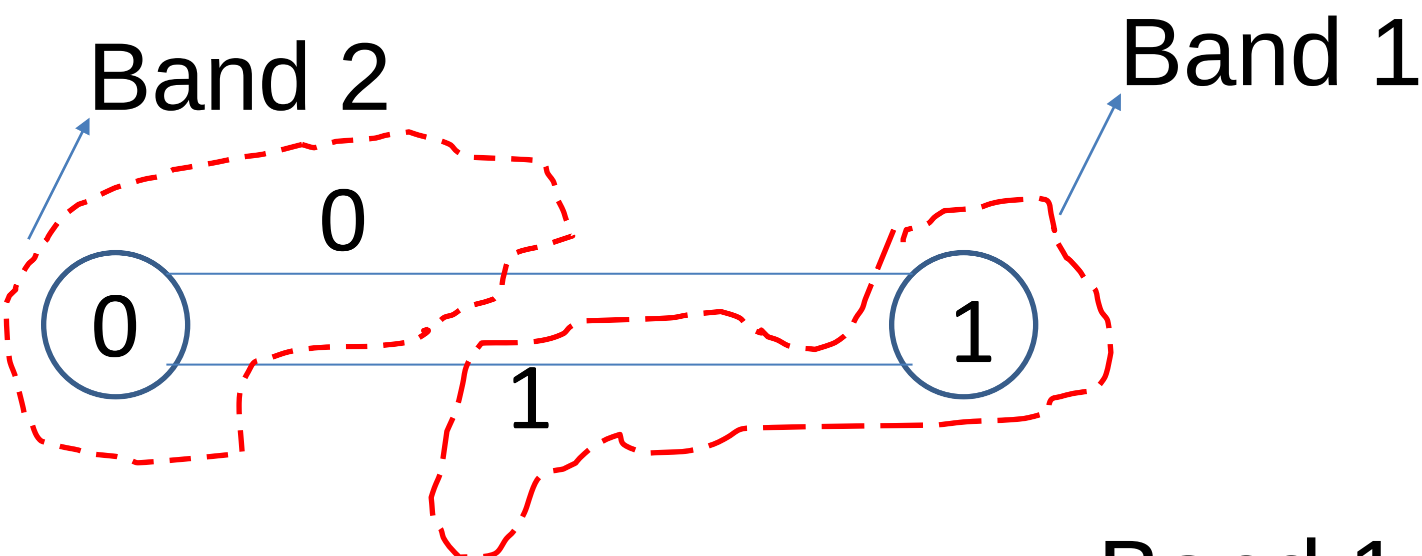


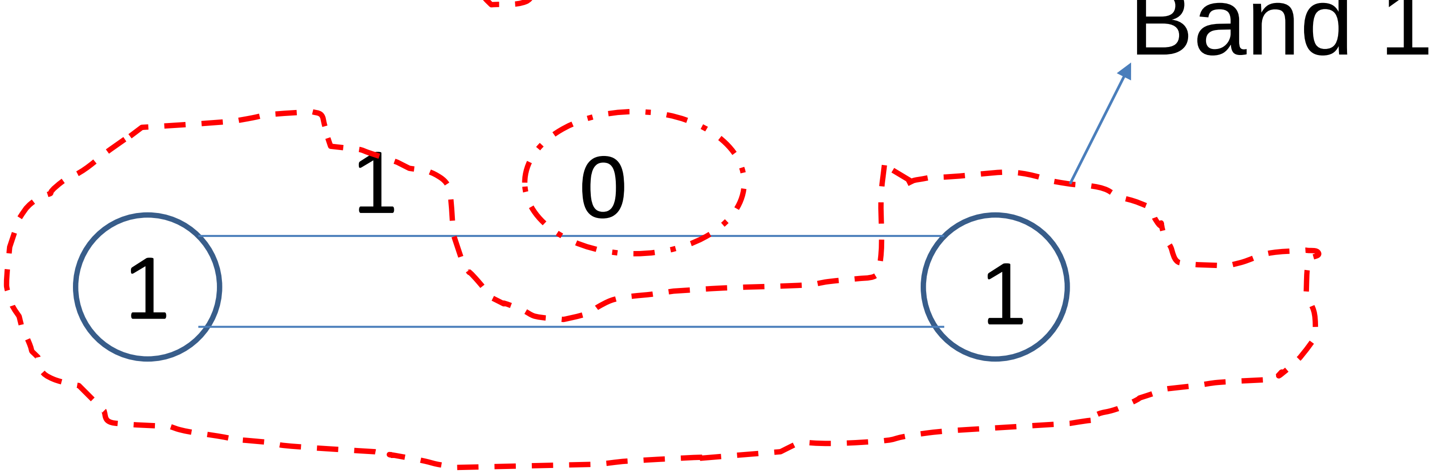
Receive ack; send new frame

Retain only the numbers

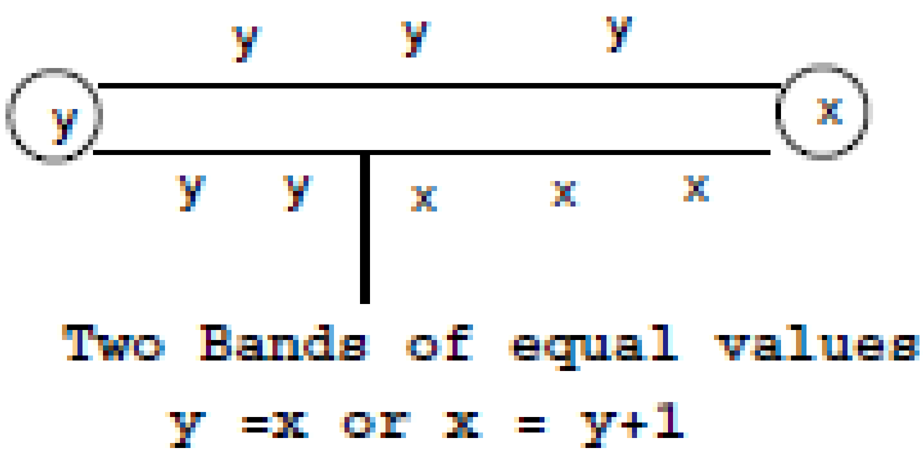






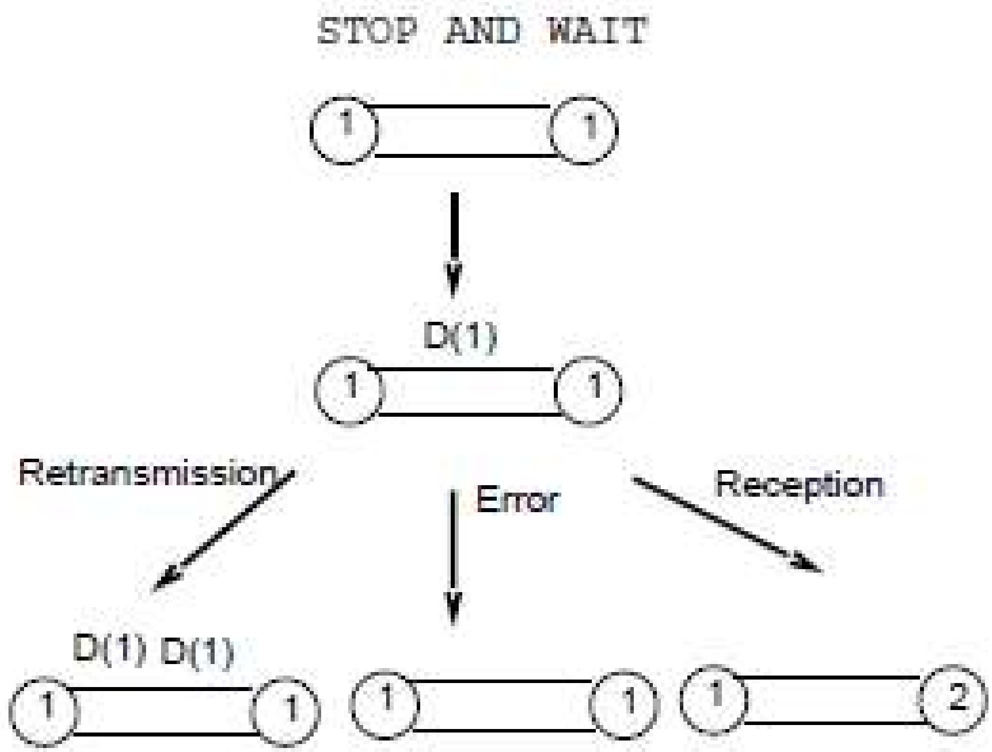


Band Invariant



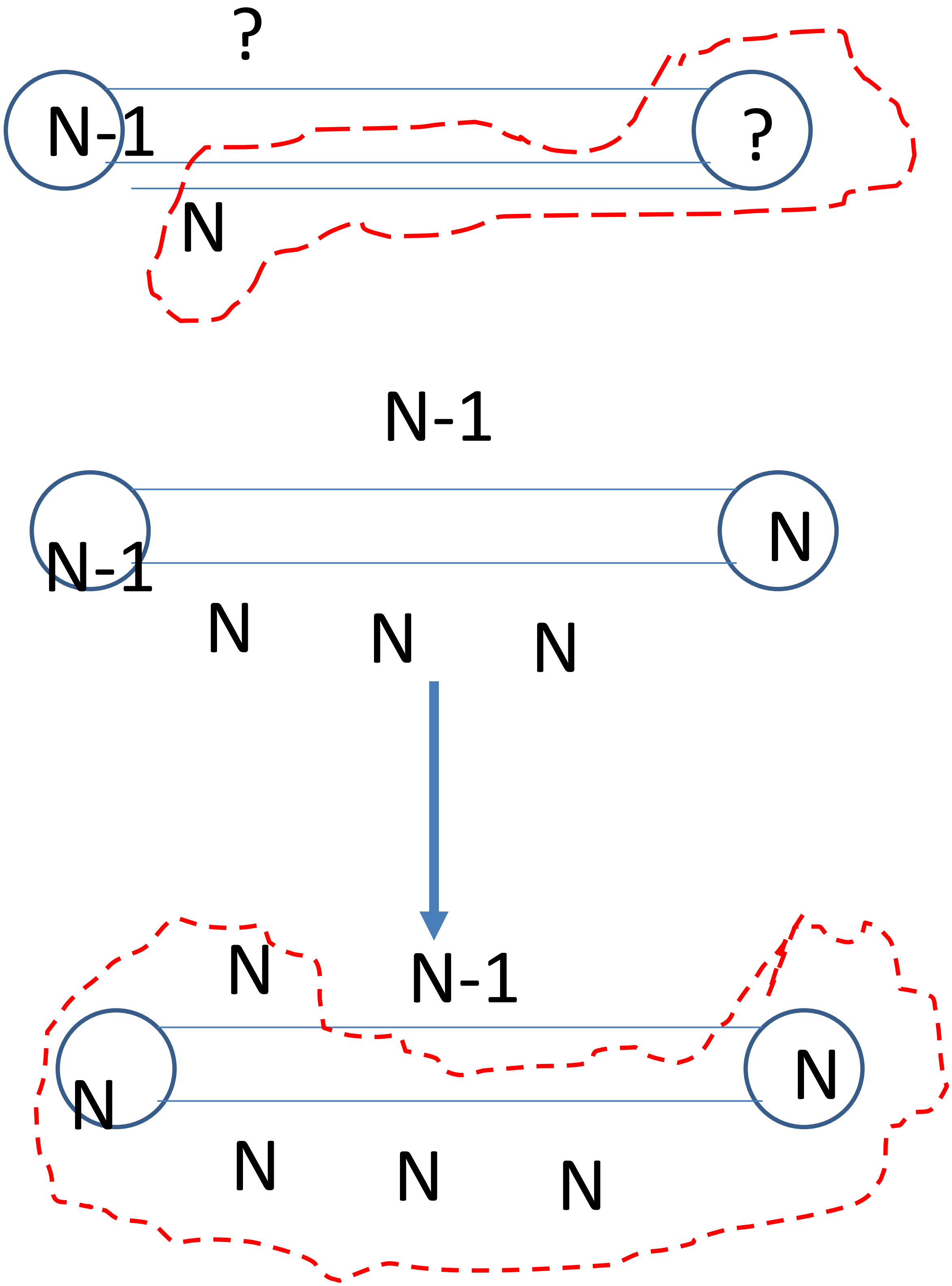
- When sender first gets to N, no frames with N or acks with N+1 and receiver is at N
- When receiver first receives frame N, entire system only contains number N→ only two numbers in system → So what?

Prove Invariant by checking a small number of State Transitions



- 3 other cases: Receive Ack, Send Ack, and Send new frame
- Just need to show that invariant is preserved by these 6 protocol actions/state transitions.

Case: sender gets new ack



Which preserves invariant

Code of Alternating Bit

-----------Sender Code--------------------------

Sender keeps state bit SN, initially 0 and repeats following loop

1) Accept a new packet if available from higher layer and store it in buffer B

2). Transmit a frame Send (SN, B)

3). If error-free (ACK, R) frame received and R != SN then

SN = R

Go to Step 1

Else if the previous condition does not occur after T

Go to Step 2

Receiver Code ------------------------------------

Receiver keeps state bit RN, initially 0

When an error free data frame (S, D) is received

On receipt:

If S = RN then

Pass D to higher layer

RN = ~ RN ; //flip bit!

Deliver data m to client.

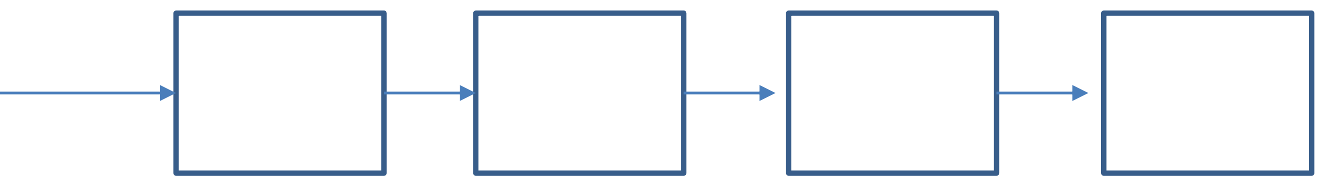
Send (ACK, RN) // Send ack unconditionally!

So far we have shown

- Correctness: based on band invariant
- Sequence Number: one bit suffices
- Performance: send only one at a time. Bad over satellites or any link were the bandwidth-delay product is large. So now we see how to do better
- Initialization: Coming up after performance

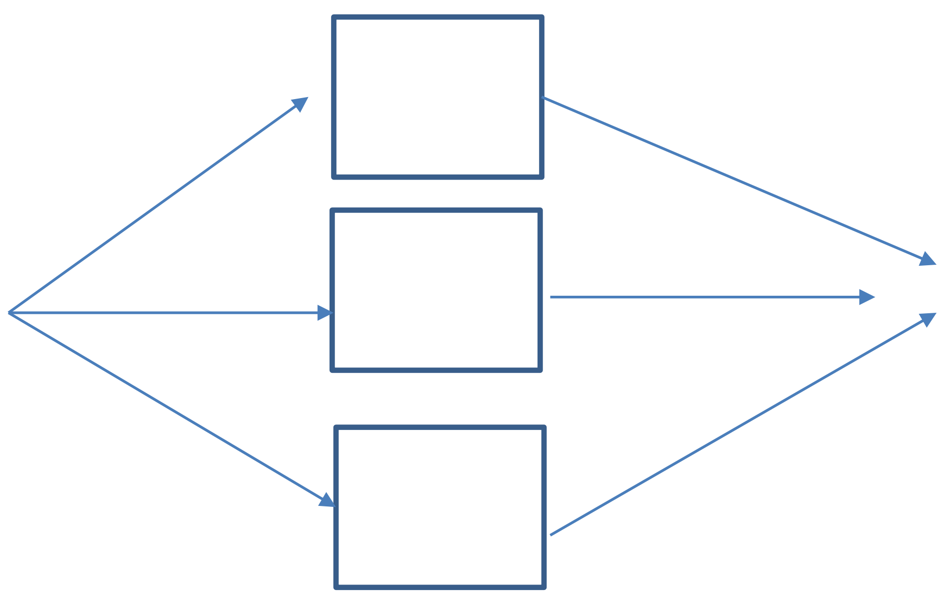
General Performance Measures

- Throughput: jobs completed per second. System owners want to maximize this.
- Latency: worst-case time to complete a job. Users want to minimize.



Factory: Each pipeline stage = 1 second

Throughput = 1 , Latency = 4

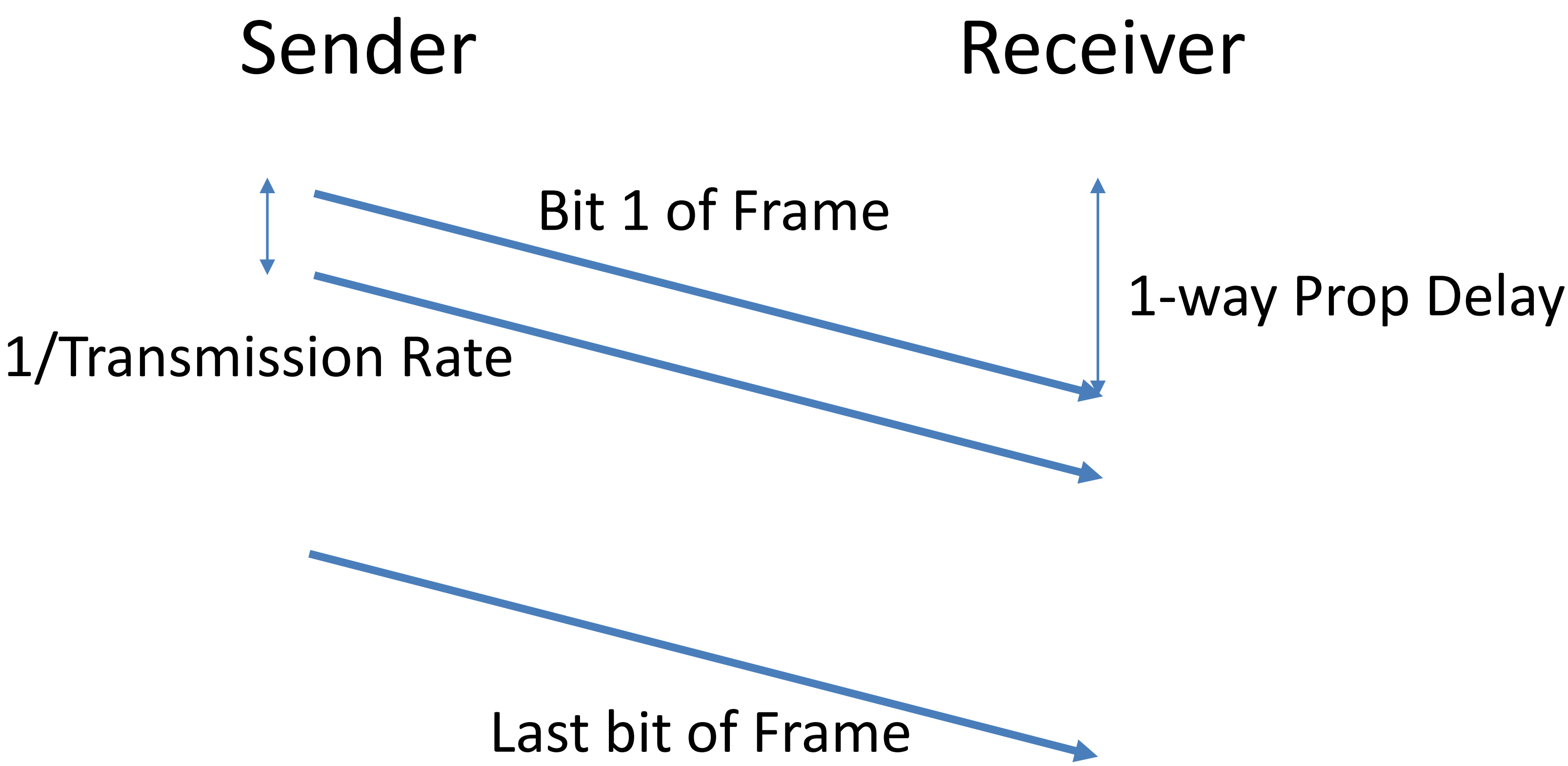


Bank: Each teller = 1 second

Throughput = 3, Latency = 1

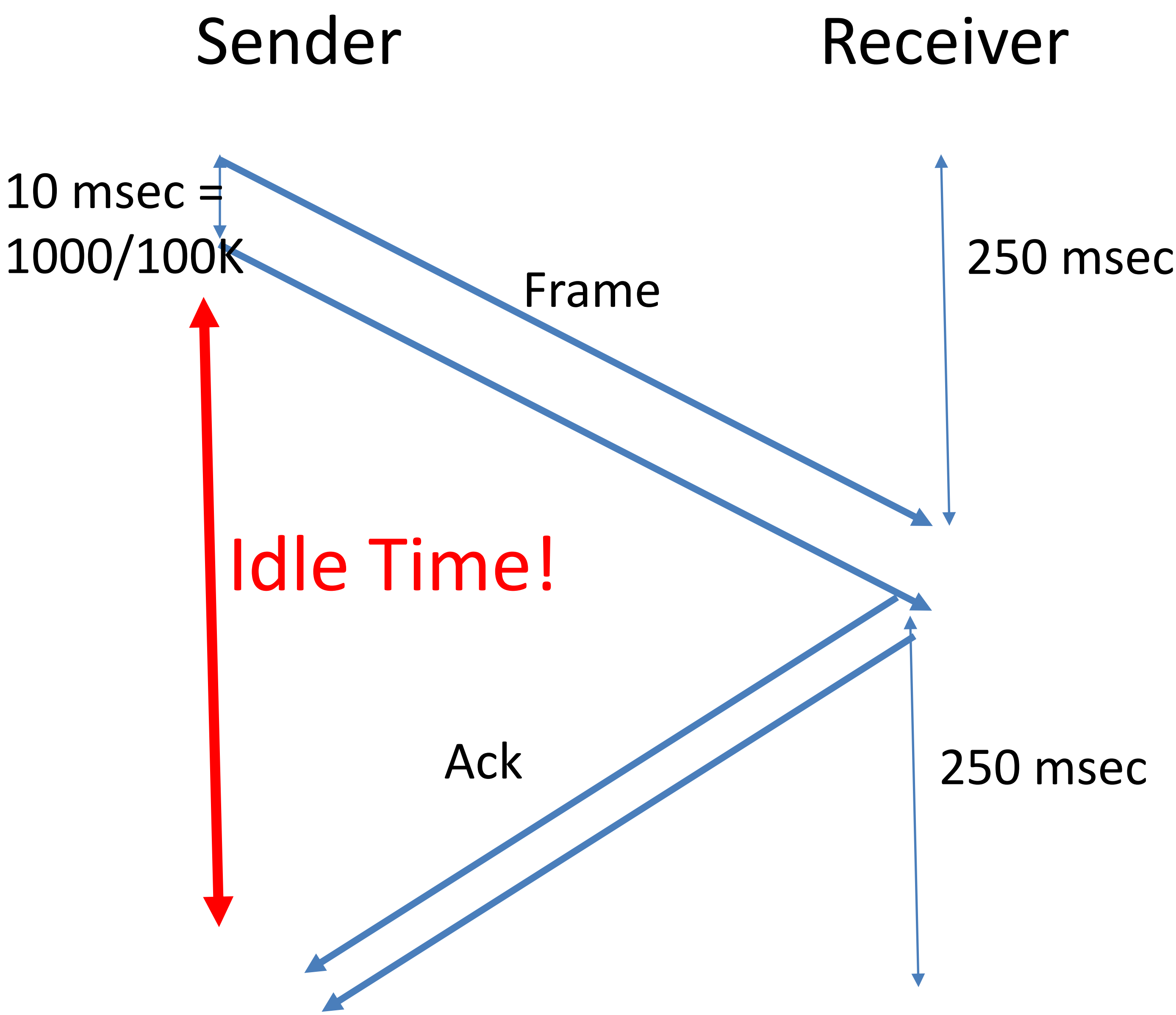
Network Specific Measures

- 1-way Propagation Delay: Time for transmitted bit to reach receiver. Contrast to transmission rate.
- Pipe Size: Transmission Rate * Round-trip Propagation Delay. Need to pipeline if pipe size is large. Alternating bit does not. Sometimes called the *bandwidth-delay* product



Stop and wait on Satellite Link

- 1-way Propagation Delay: 250 msec
- Transmission speed: 100 kbit/sec
- Frame size: 1000 bits.
- What is throughput? 2000 bits per second, which is 2% of a 100,000 bit per second link.



If you don’t want to be fired for using 2% of capacity . . . .

you got to send more frames before the ack for first frame arrives

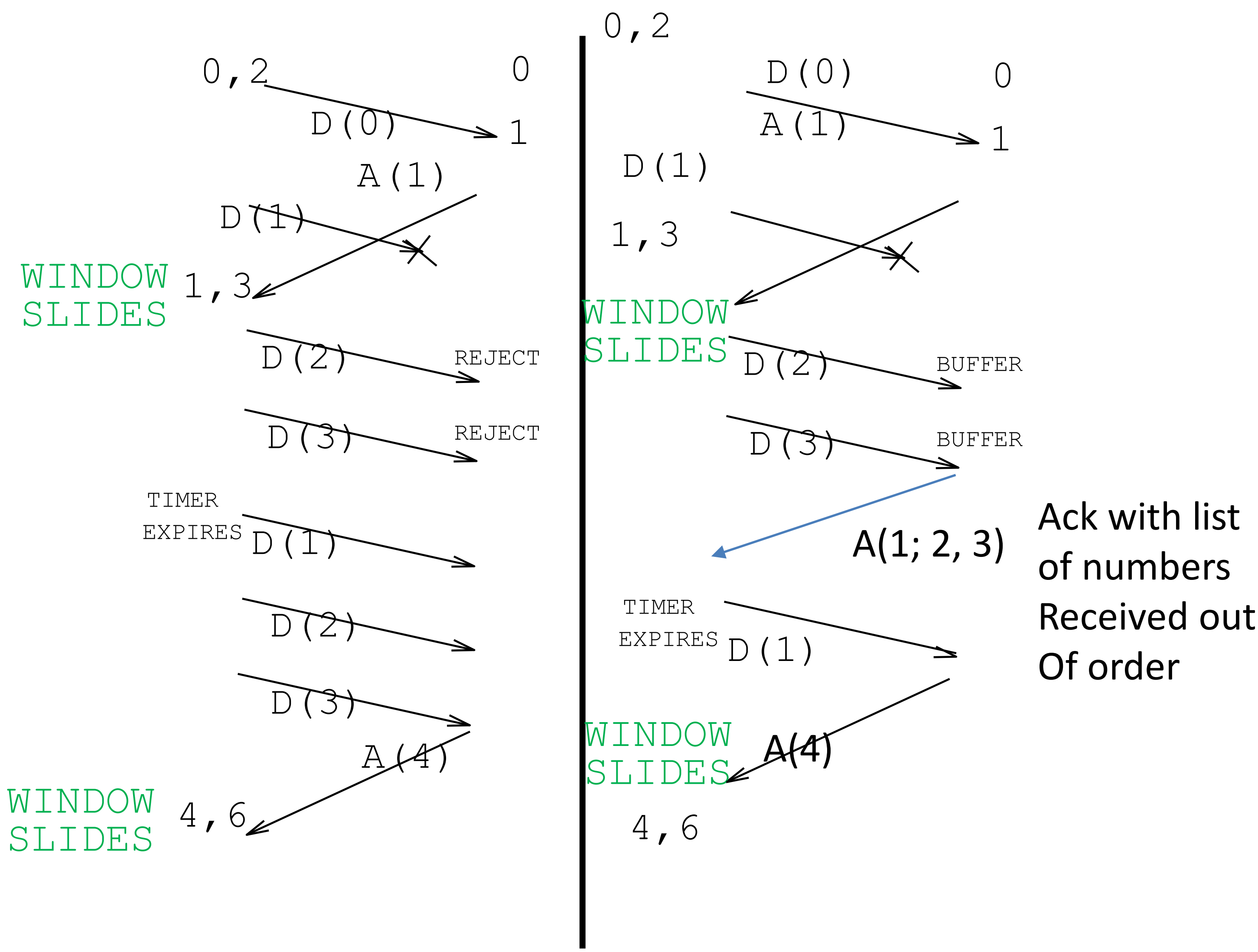
How? Using Sliding Window Protocols

Sliding Window Protocols

- Window: Sender can send a *window* of outstanding frames before getting any acks. Lower window edge L, can send up to L + w - 1.
- Receiver numbers: receiver has a receive sequence number R, next number it expects. L and R are initially 0.
- Sender Code: Retransmits all frames in current window until it gets an ack. Ack numbered r implicitly acknowledges all numbers < r.
- Two variants: receiver accepts frames in order only (go-back-N) or buffers out-of-order frames (selective reject}

GO BACK 3

SELECTIVE REJECT



Go Back N Code

------------Sender Code---------------------------

Sender keeps state variable L, initially 0

Send (s,m) // send data message m with number s

The sender can send this frame if:

m corresponds to s-th data item

given to sender by client AND

L <= s <= L + w - 1 r // in allowed send window

Receive(r, Ack) // receive an ack number r

On receipt:

L := r // slide lower window edge to ack number

Receiver Code ------------------------------------

Receiver keeps state variable R, initially 0

Receive(s, m) // receive data message m with number s

On receipt:

If s = R then

R := s + 1

Deliver data m to client.

Send(r, Ack) // send ack with number r

// receivers typically send acks in response to data

// messages but our code can send acks anytime

r must equal R

Selective Reject Sender code

Sender keeps a lower window edge L initially 0 but also an array with a bit set for all numbers acked so far. Initially, all bits are clear. In practice, we implement this array by a bitmap of size w which we shift

Send (s, m) // send data message m with number s

The sender can send this frame if:

m corresponds to s-th data item given to sender by client AND

L <= s <= L + w - 1 AND

s has not been acked // new for selective reject

Receive(r, List Ack) // receive an ack number r with List

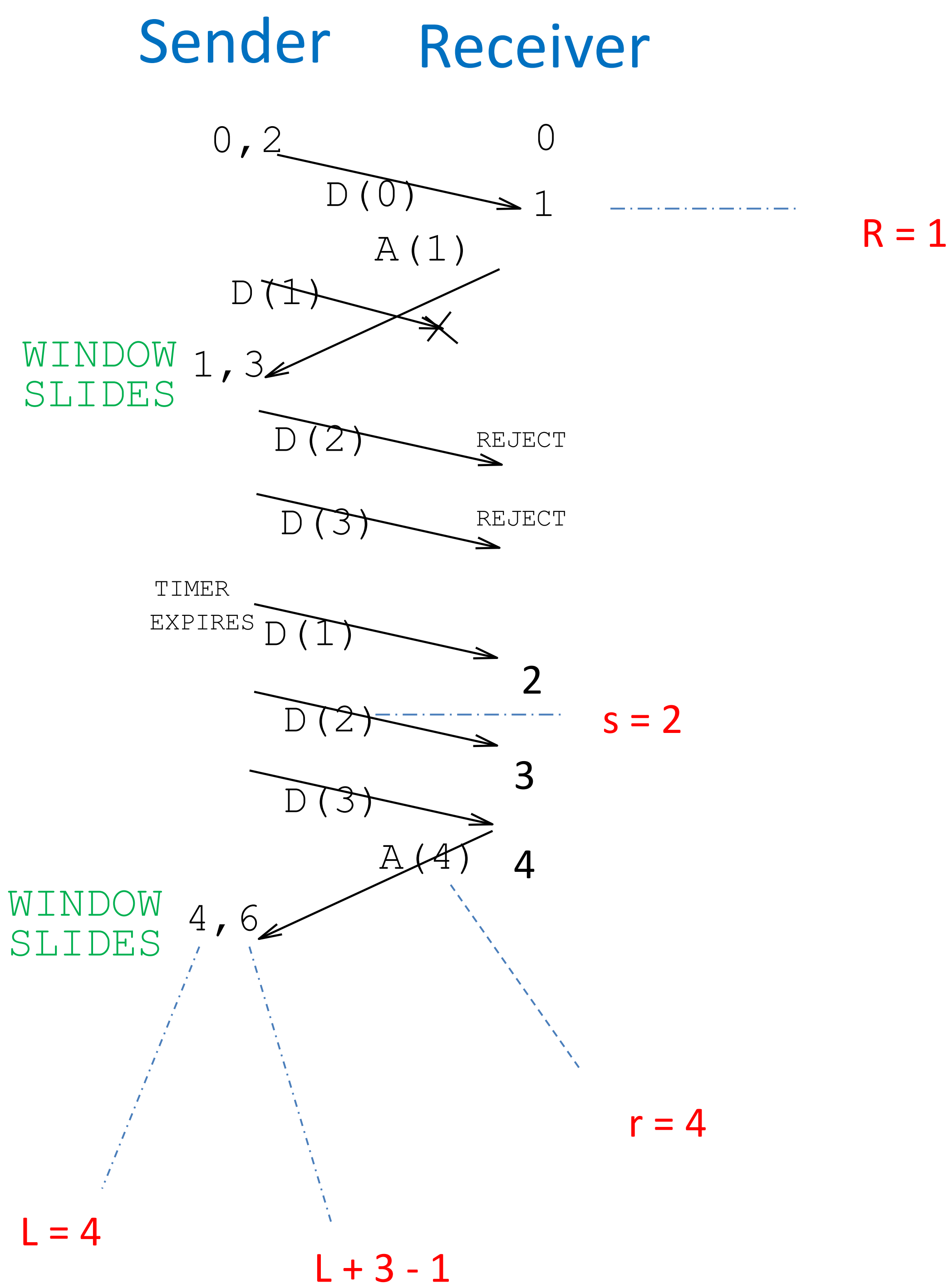
// of received numbers > r

On receipt:

L := r // slide lower window edge to ack number

Mark numbers in List as acked at sender

GO BACK 3 EXAMPLE WITH SOME VARIABLE VALUES



Selective Reject Receiver Code

Receiver keeps a receiver number R initially 0 but also an array with a bit set for all numbers received so far. Initially, all bits are clear. In practice, we implement this array again by a bitmap of size w which we shift.
In addition to the bitmap, we have a buffer for each number where we can store out of order messages

Receive (s, m) // receive data message m with number s
  On receipt:
    If s >= R then
      Mark s as acked and Buffer m
      While R acked do
        Deliver data message at position R
        R := R + 1

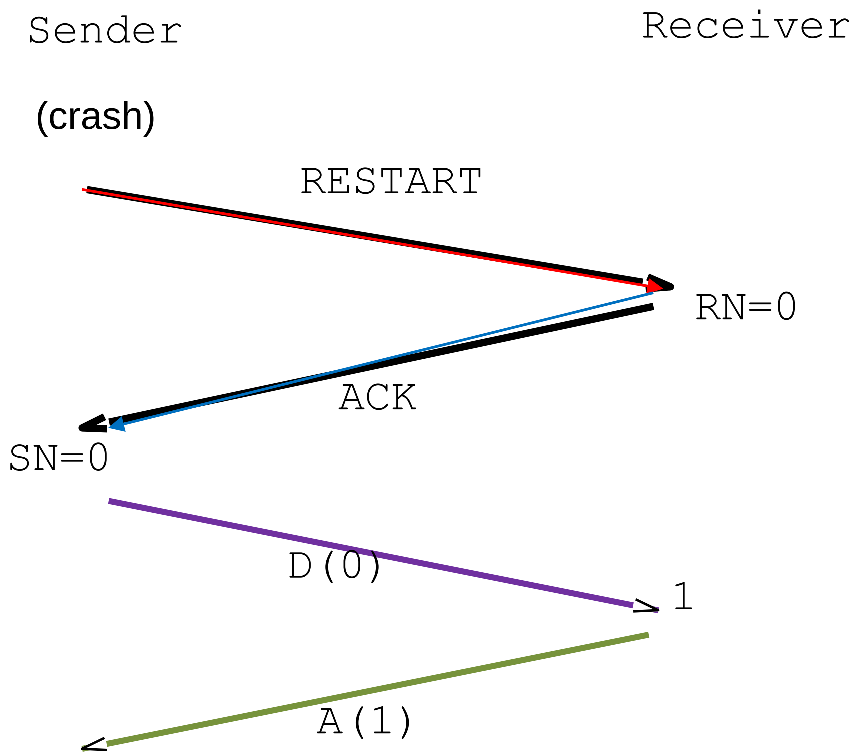
1 0 0 0 0 0

1 0 1 0 0 0

Send (r, List Ack) // send ack with number r and List
// of received numbers > r
  r must equal R
  List contains received numbers > R

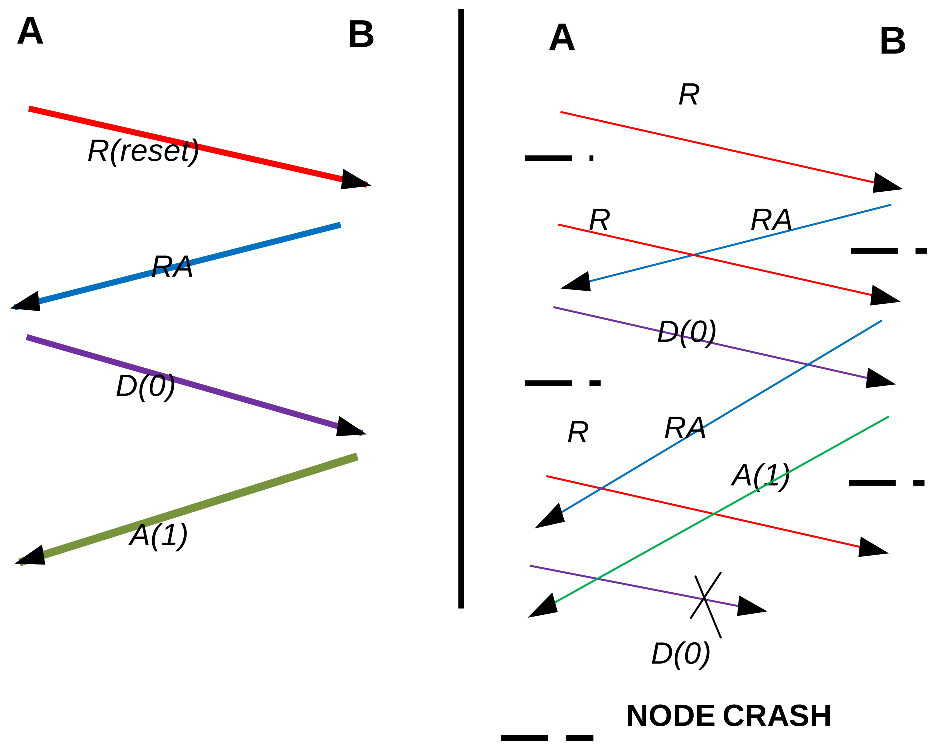
INITIALIZING LINK PROTOCOLS

(in the face of link and node crashes)



EXAMPLE: SETM = RESTART in HDLC

How naïve restarts can fail



Can prove that there is no reliable initialization protocol if we assume no non-volatile memory that survives crashes, the protocol is deterministic, and message can stay on wire indefinitely

So? Do we give up? No *change assumptions*

How to design a reliable initialization protocol

Change one of the assumptions in impossibility theorem:

- No memory after a crash: Can do correctly if sender keeps even one bit that can survive a crash
- Determinism: Can send restart messages with random numbers and only send data when random numbers are acked. High probability only
- Message lifetimes: If no message can live on a link for more than T seconds, simply wait T seconds after a crash for all old messages to die out

Protocol Design Lessons

- Start simple: Start with simple protocols like Stop and Wait. Optimize later based on invariants
- Minimal Specification: Understand what parts of protocol need to be specified for correctness. Some parts can be left to implementers to optimize
- Be an engineer: Our Impossibility results tell us what we need to change to get our job done, not just what we can't do.
- Next time: Media Access Control and LANs