

Layer Optimization: Handling Loss

CS 118

Computer Network Fundamentals

Peter Reiher

consistency checks & redundancy checks - page 6

IPv4 checksum - page 9

receive window - page 25

ways to lose a message - page 12

stop and go - page 30

go back N - page 38

Intralayer examples

- Integrity
- Time

Integrity

- Error
 - Detect accidental alteration
- Loss
 - Detect missing packet(s)
- Reordering (related mechanism)

Detect or correct?

- If we can correct, there's no problem
- Here we focus on integrity failure
 - Based on detecting uncorrectable errors

can only detect, not correct

What do errors look like?

- Errors in destination address
 - The message ends up elsewhere
- Errors in source address
 - The message affects the wrong FSM
- Errors in contents
 - Impacts other layers (up to the user)
other layers, the AT NLP

It's important to know when you see an error

Making errors visible

- Consistency checks

- Portion of a message has a valid range
- Value is outside that range (invalid values)

sets a valid range for consistency within that range

- Redundancy checks

- Add redundancy in a message
- If error only alters one of the redundant values, the other indicates an error

redundancy to see if error alters one of the redundant values

Consistency checks

- Some values of some fields for some layers aren't allowed
- Examples:
 - Padding field in TCP header must be all zeros
 - Can't have unroutable IP address in Internet
 - IPv4 IHL field can't be less than 5

Redundancy checks

- **Mathematical constraints**
 - Treat contents as numbers
 - Use only values that satisfy an equation
 - Matching: $A = B$
 - Parity: $(\sum a_i) \bmod 2 = p$
 - Checksums
 - Cyclic Redundancy Check (CRC)

Example – IPv4 checksum

- 16-bit ones complement sum
 - Consider IPv4 header as set of 16 bit numbers
 - Add the numbers and hold the carry
 - Add the carry back in (carry-wraparound)
- Easy to implement in SW or HW
- IPv6 doesn't have header checksum
only IPv4 has header checksum

TCP checksum

- Independent of IPv4 checksum
- Taken over entire packet (except a few IP header fields)
- Basic method like IPv4 checksum
- Somewhat different computation method if running over IPv6
 - Interlayer dependency in this optimization . . .

Loss

- Detecting when a message should have arrived but did not
 - Message completely lost
 - Never sent
 - Never arrives
 - Message source/destination error
 - Message arrives but cannot reach the intended receiver (or relay) FSM

One way to lose a message

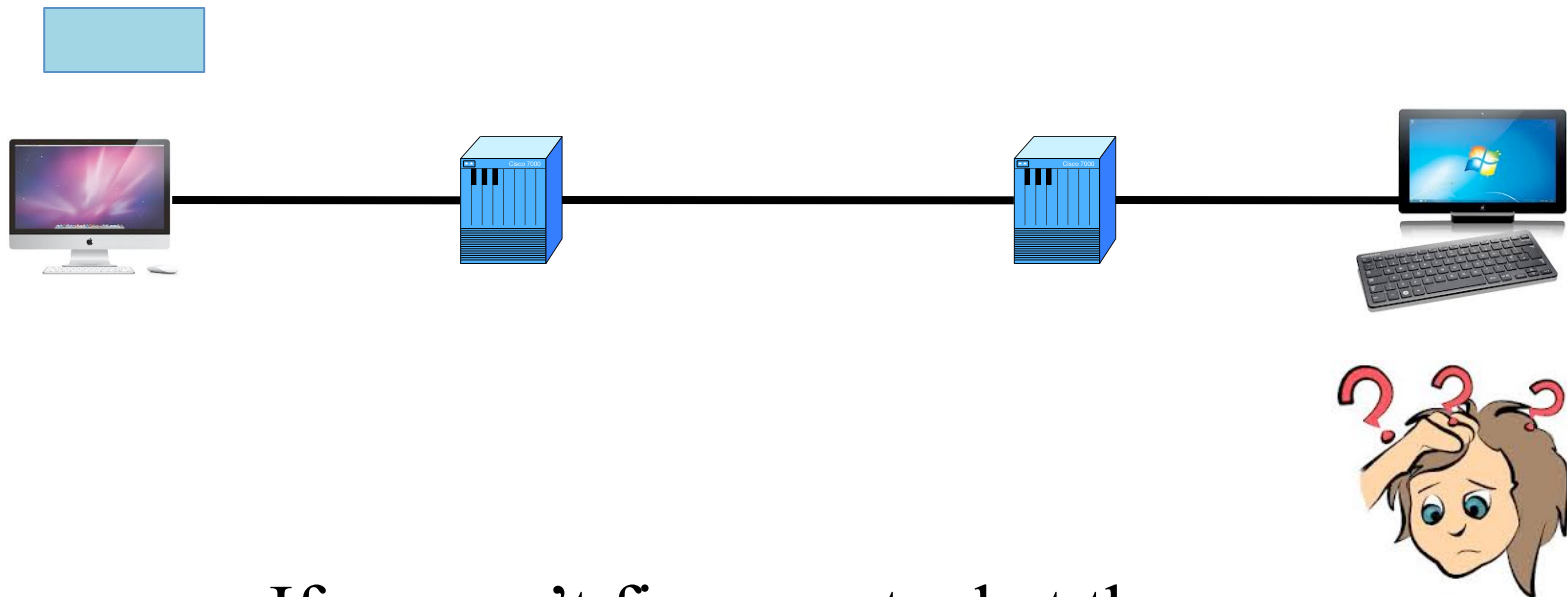
Timeout



Different Ways to Lose a Message
timeout
error detection / correction
flow control
congestion control

Another way to lose a message

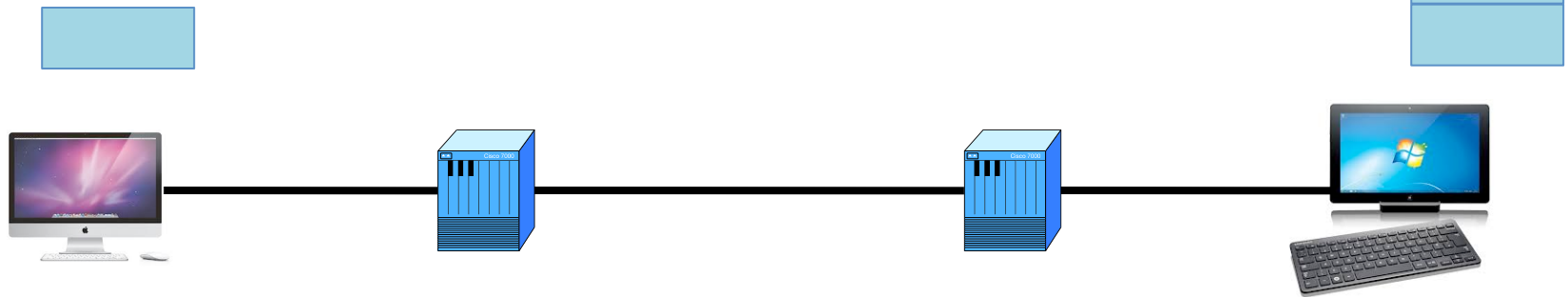
Error Detection/Correction



If we can't figure out what the message is, it's as good as lost

A third way to lose a message

Flow control

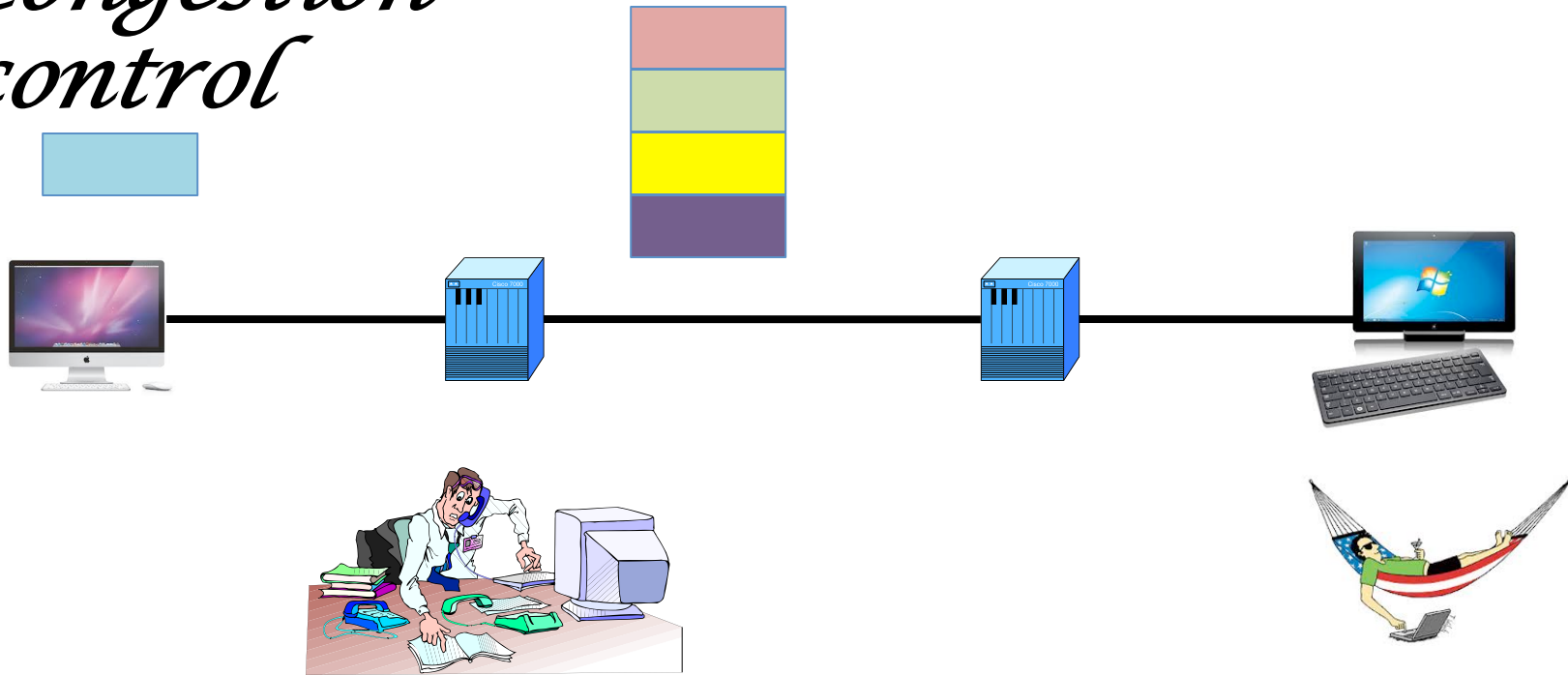


If the receiver is too busy and
has nowhere to put it, he'll
have to drop it



A fourth way to lose a message

Congestion control



If the network is too congested, it may be dropped

Even if the receiver isn't overloaded

Detecting loss

- We've talked about this before
 - Need to keep track of time
 - So let's say we do...



The key issue: what time?

- Different possible times:
 - Time expected
 - Latest time we want to wait
 - Time since last message sent
 - Time since last message received
- Let's say we know (or pick) one
 - Still need to know the value to use

Estimating time

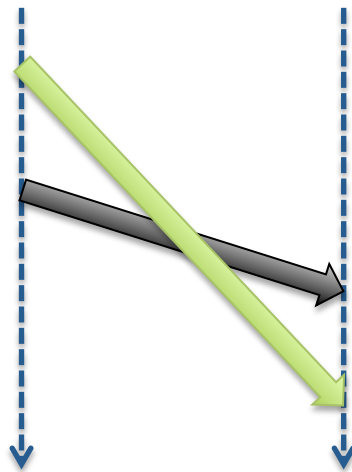
- Based on a network property
 - Number of hops (diameter)
 - Longest transmission time
 - Longest time held per relay * # relays
- Based on past measurements
 - How do we measure time?
 - How do we aggregate measurements?

Measuring time

- Need a timer
 - Can't wrap-around too quickly
 - Good resolution (time of smallest increment)
- Need to timestamp messages
 - Measure differences

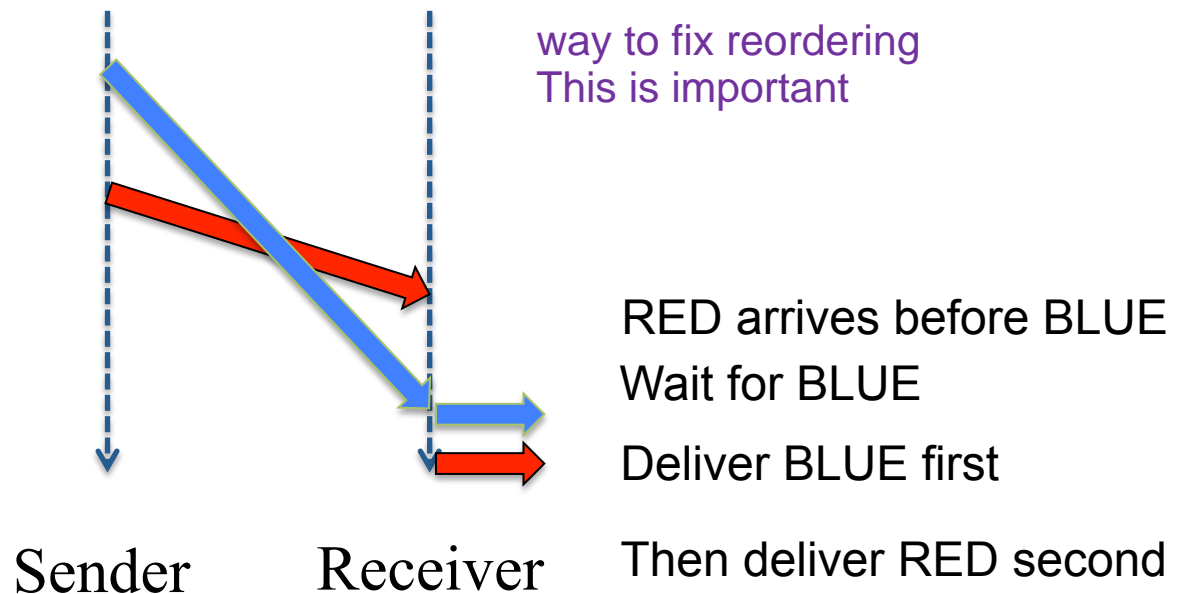
Reordering

- Messages arrive without loss
 - But out of order



Fixing reordering

- Hold onto messages
 - Keep the ones that come too early
 - Process once sequence gaps are filled



What's hard about reordering?

- Need more state – a reordering buffer takes up space
- How much space?
 - Enough to store maximum displacement
- Where?
 - At the receiver, where things arrive too early
- Do you just wait for late messages?
- Or request a resend?
 - If so, how aggressively?

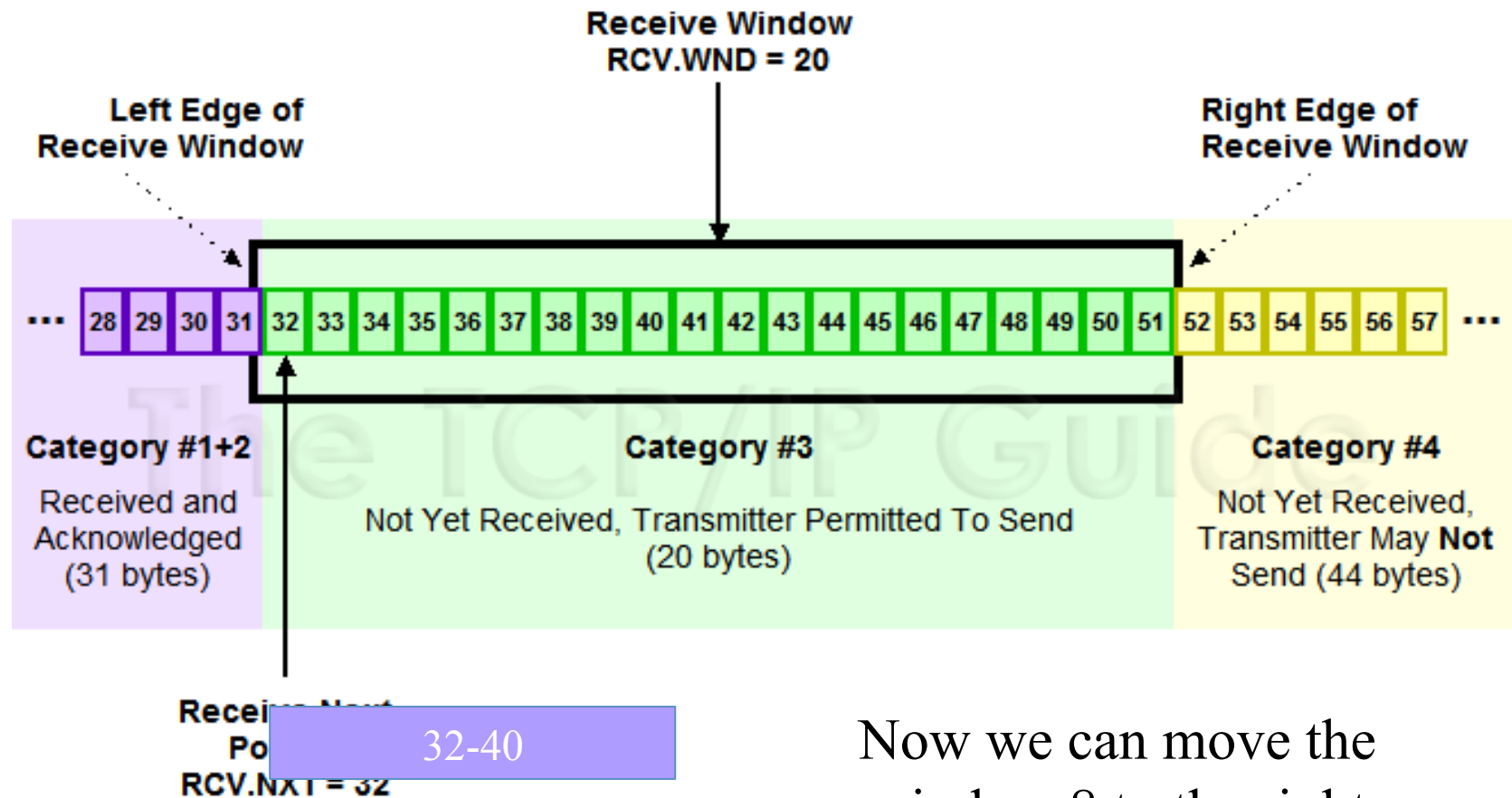
TCP receive window

- TCP segments have a sequence number
 - Number indicates byte offset of each segment
- TCP receive window enables reordering
 - Make it large to handle large displacements
 - Left side = smallest offset not yet here
 - Right side = largest offset that can be held

Receive window management

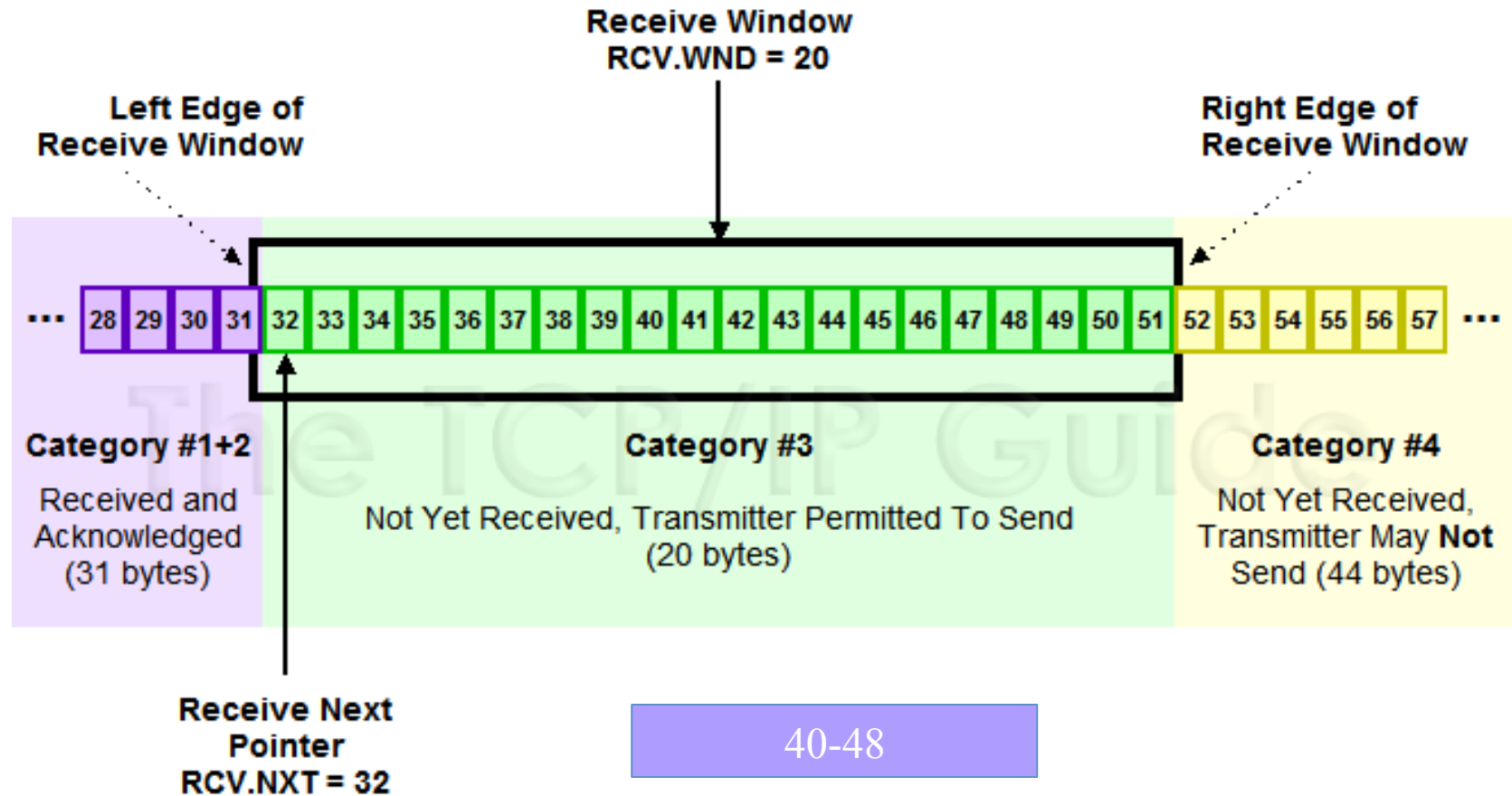
- Message arrives
 - Put it in the buffer if it fits
 - Buffer size = RCV.WND
- RCV.NXT (left side)
 - Move right to deliver info to the user (upper)
 - Stop and wait at the first gap
- $\text{RCV.NXT} + \text{RCV.WND}$ (right side)
 - Moves right (allow higher offsets) whenever left side moves right

Receive window



Now we can move the window 8 to the right

Receive window with misordering



Can't move the window yet

Hmm – what’s that about the sender?

- That figure shows how the receive buffer is tied to the send buffer
- The receive buffer is for reordering
 - So what’s the send buffer for?
 - Why (and how) are the two related?

Time

- Flow control
- Congestion Control
- Latency management

Flow control I

- Not overrunning the receiver
 - The receiver always can handle one message
 - Send that,
wait for confirmation,
send the next, ...
 - Why does the sender need to wait?
 - Message was lost
 - Message is waiting at the receiver
 - If receiver can handle messages as fast as the sender, not an issue
 - Solution: send one at a time
 - One message outstanding at any given time
- avoid waiting, send one message at a time

Stop and Go

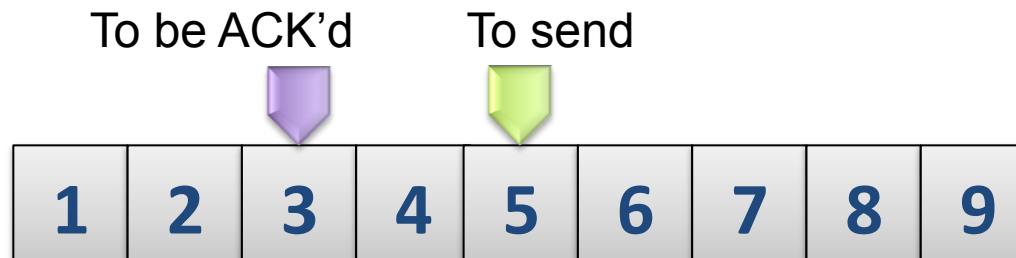
- Assume a sequence number
 - One number per message
- Sender and receiver work in lockstep
 - Both “walk” the number space
 - Both have “inchworm” behavior

Let's take a walk

- Messages are numbered

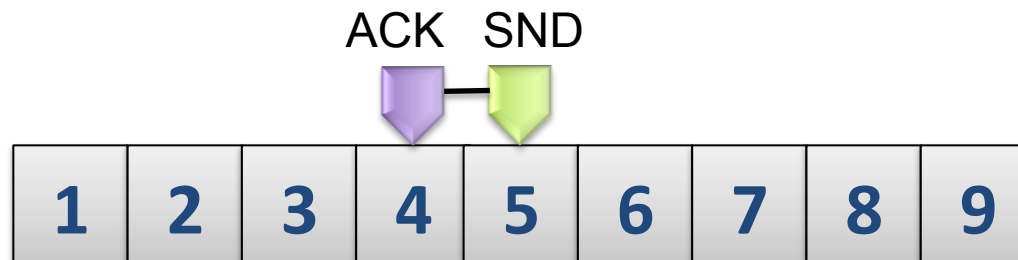


- Sender “walks” the line via receiver ACKs



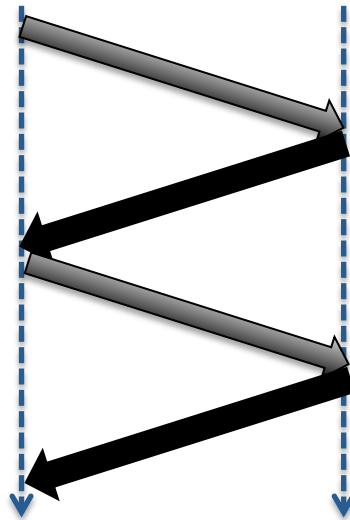
Limiting the walk

- **Send and receiver ACK linked by a limit**
 - $SND - ACK \leq N$ equation for linking send and receive
 - For stop-and-go, $N=1$



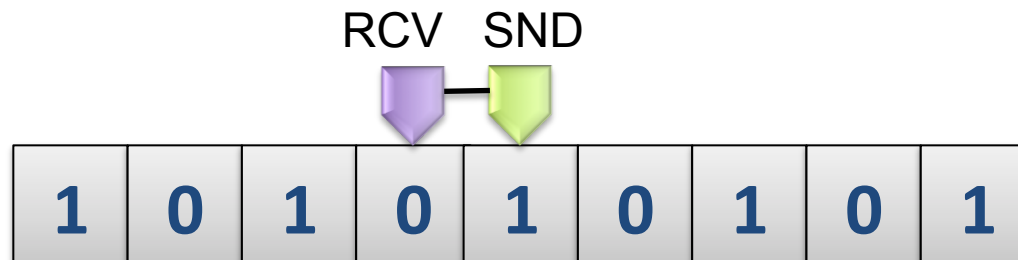
A look at the exchanges

- One message per round trip
 - ACK indicates received and ready for next



A look at the numbering

- If SND and ACK differ by at most 1, we don't need to number 1..999
 - OK to just number 0,1,0,1,0,1
 - Also known as “alternating bit” protocol



Why do anything else?

- Receiver might be faster than sender
 - In which case it could handle more messages
- Learning that receiver has handled a message takes time
 - At least time to get acknowledgement to sender
- During that time, channel is not in use, receiver is not busy, sender is not busy
 - Lots of wasted resources

What if we have more than 1 outstanding message?

- Messages could arrive out of order
 - As we've already discussed
- A message could arrive while receiver is busy handling an earlier message
 - Not possible with 1 message window Stop and Go
- If we allow multiple outstanding messages, we must handle both problems

Flow control II

- How to handle these problems?
- As discussed, use a buffer to handle misordered messages
 - The receive window indicates max disorder
 - Also max “outstanding” held messages
- Hmm – let’s use that buffer two ways!
 - If receiver is busy when message arrives, put it in the buffer
 - Even if it is the next message to be received
- So we can send more than one at a time...

Go Back N

- Assume a sequence number
 - One number per message
- Sender and receiver work in lockstep
 - Both “walk” the number space
 - Both have “inchworm” behavior

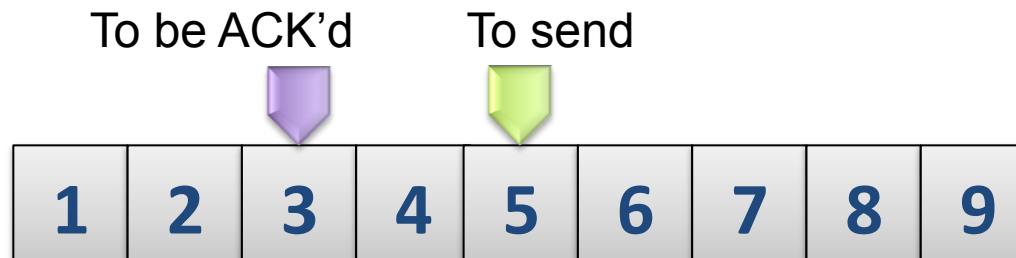
Just like stop-and go, but with N messages

Let's take another walk

- Messages are numbered



- Sender “walks” the line via receiver ACKs



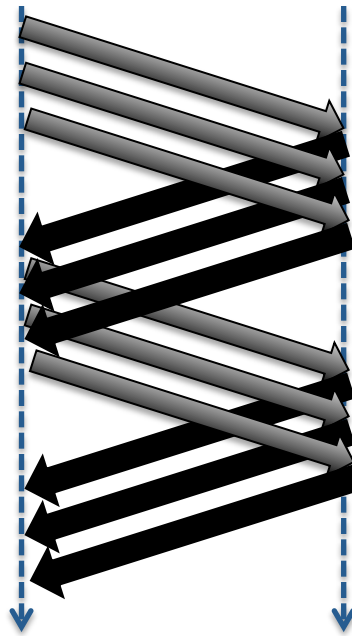
Limiting the walk

- Send and receiver ACK linked by a limit
 - $SND - ACK \leq N$
 - For Go-back-N, $N > 1$



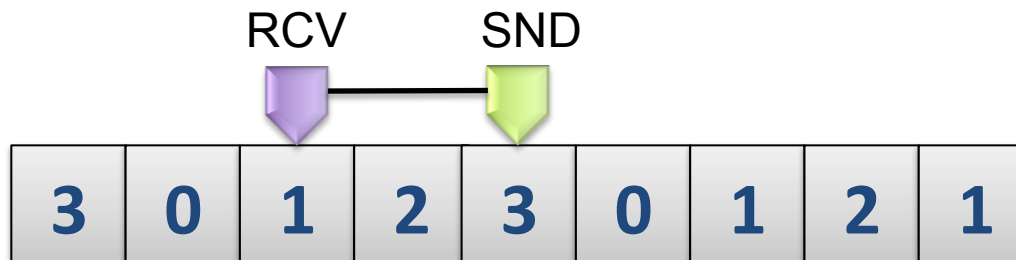
A look at the exchanges

- N messages per round trip
 - ACKs indicates received and ready for next



A look at the numbering

- If SND and ACK differ by at most N , we don't need to number 1..999
 - OK to just number $0, 1, 2, 3, \dots, 2N-1$



Why $2N$ values?

- N outstanding values
 - Each RTT, the window can slide forward by N
 - Need to prevent overlap from one RTT to next

what is an RTT?

How big is N?

- How many messages before getting ACK?
 - Once you get the ACK for the first, you can send $N+1$
 - ACK provides a “clock” to the pipeline
 - Every ACK/ $N+1$ pair acts like stop-and-go
 - Go-back-N is like N overlapping stop-and-go

N overlapping stop and go is a good analogy for stop and go

About the receive window

- What if the receiver isn't fast enough?
 - Info (message) has to go into the buffer as fast as it arrives (or we have other problems!)
 - If the FSM doesn't release the info to the upper layer as fast as it comes in, there's a delay

Recall receive rules

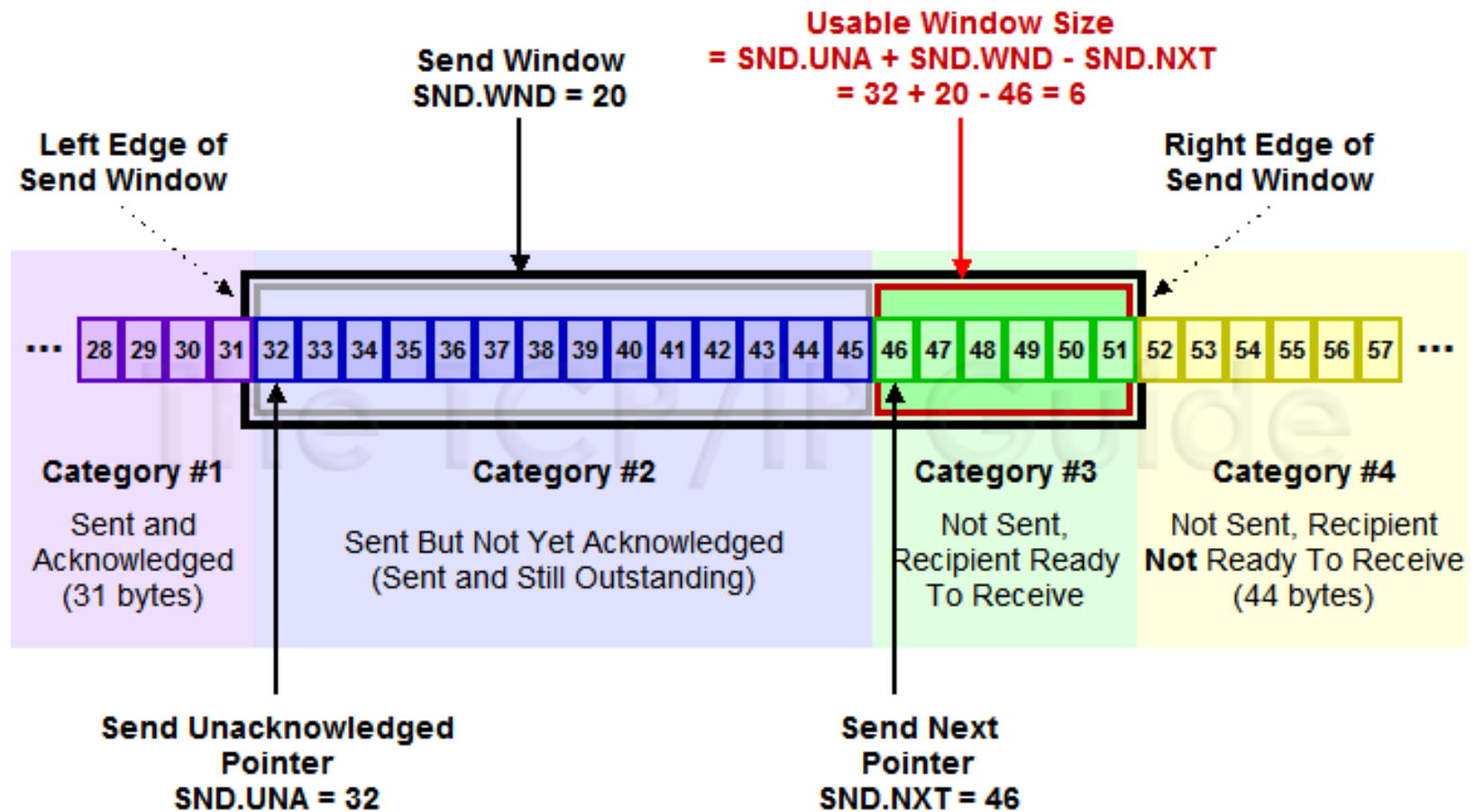
- Info (message) arrives
 - Place message in sequence
 - Move left side to the right until a gap
 - Pass that info to the next layer up
 - Right side moves to the right at the same time
- If the FSM is fast enough
 - The left side doesn't move immediately
 - Takes time – time to process the message

Left and right

- If left side doesn't move, right doesn't
 - I.e., receiver isn't ready for new offset info
- How do we coordinate with the sender?
 - Sender has a similar buffer
 - $SND.WIN = RCV.WIN$
 - If smaller, we could have sent messages that could have been held by receiver – wasted resources
 - If bigger, we won't be able to send it anyway (we can only fill the receive buffer!)

- flow control - uses a window --> at sender, permitted to send out certain amount of data to the receiver; receiver checks what they have seen
- deal with misordered messages
 - receiver is capable of holding a certain amount of data until his FSM is capable of catching up with the different layers; ensure no overflow

Coordinating SND and RCV



guarantee that we can send you all the data we have

Combining loss + windowing

- Positive feedback (ACK)
 - Indicate what was received
- Negative feedback (NACK)
 - Indicate what is missing but expected
 - Always a gap after the last msg received

use NACK when we know something is missing

Use this info to coordinate retransmission

if negative ack, we can go ahead and do a retransmission

Loss / windowing variants

- Stop and Go one at a time request for retransmit
 - On timeout, send retransmit request
 - Only one message to ever request
- Go back N back up to lowest ACK every trip
 - On timeout, ACK lowest missing sequence number
 - Sender “backs up” to where ACK indicates
 - Every round trip, backs up to the lowest gap
- Selective ACK ACK everything you send ; selective repeat
 - ACK everything you get, ask to fill in the holes
 - Sender fills in only the holes

resend one packet at a time

Congestion control

- Receiver might be ready, but is the net?
 - Don't want to overwhelm the network

control the congestion of data

when sender and receiver are both okay, but network is too busy

- We have some windows
 - Send = how much info can be outstanding
 - Recv = how much info can be reordered
- *Can* isn't the same as *should*

How much SHOULD be outstanding?

Solution: congestion window

receive window stays fixed

- Receive window
 - Stays fixed (no benefit to adjusting)
 - As large as reordering max
 - As large as send pipelining too
- Send window
 - No larger than the reordering max
 - As large as is needed to keep up with the receiver
 - Not so large that messages are lost in the net
- OK, how big is that?

increase the send window to keep up with the receiver