Transport Protocols

- UDP provides just integrity and demux
- TCP adds…
  - Connection-oriented
  - Reliable
  - Ordered
  - Point-to-point
  - Byte-stream
  - Full duplex
  - Flow and congestion controlled

UDP: User Datagram Protocol [RFC 768]

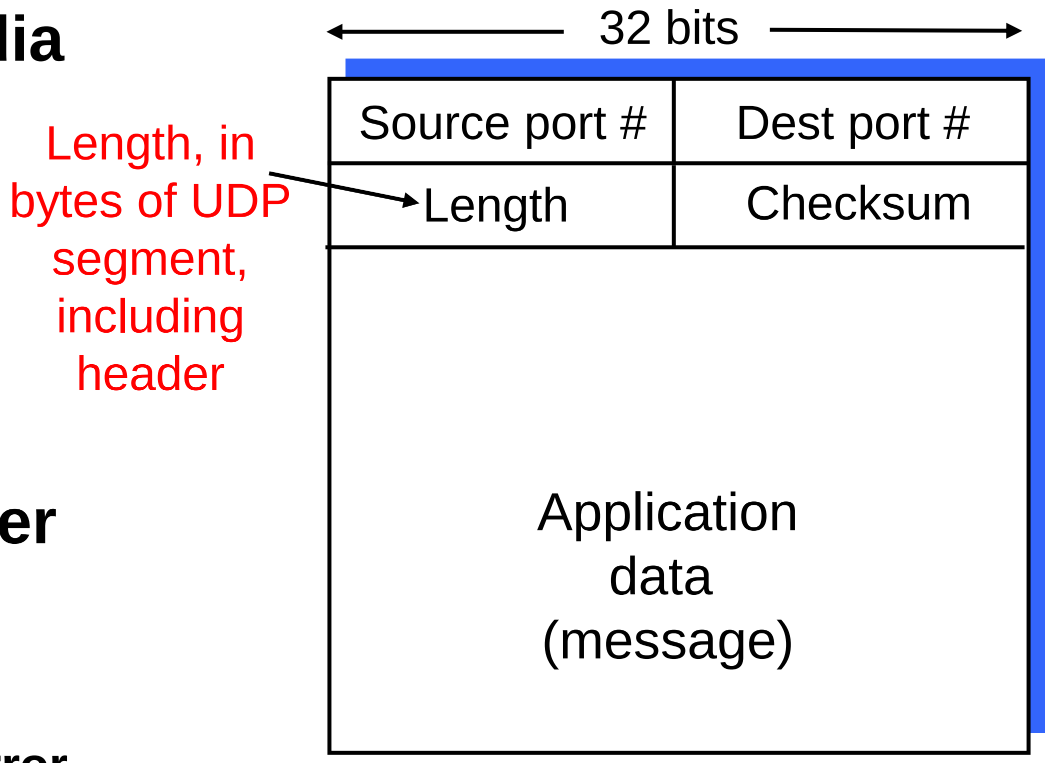
- "No frills," "bare bones" Internet transport protocol
- "Best effort" service, UDP segments may be:
  - Lost
  - Delivered out of order to app
- *Connectionless:*
  - No handshaking between UDP sender, receiver
  - Each UDP segment handled independently of others

Why is there a UDP?

- No connection establishment (which can add delay)
- Simple: no connection state at sender, receiver
- Small header
- No congestion control: UDP can blast away as fast as desired

UDP, cont.

- Often used for streaming multimedia apps
  - Loss tolerant
  - Rate sensitive
- Other UDP uses (why?):
  - DNS, SNMP
- Reliable transfer over UDP
  - Must be at application layer
  - Application-specific error recovery



32 bits
Source port #
Dest port #
Length
Checksum
Application data (message)
Length, in bytes of UDP segment, including header

UDP segment format

UDP Checksum

Goal: detect “errors” (e.g., flipped bits) in transmitted segment – optional use!

Sender:

- Treat segment contents as sequence of 16-bit integers
- Checksum: addition (1’s complement sum) of segment contents
- Sender puts checksum value into UDP checksum field

Receiver:

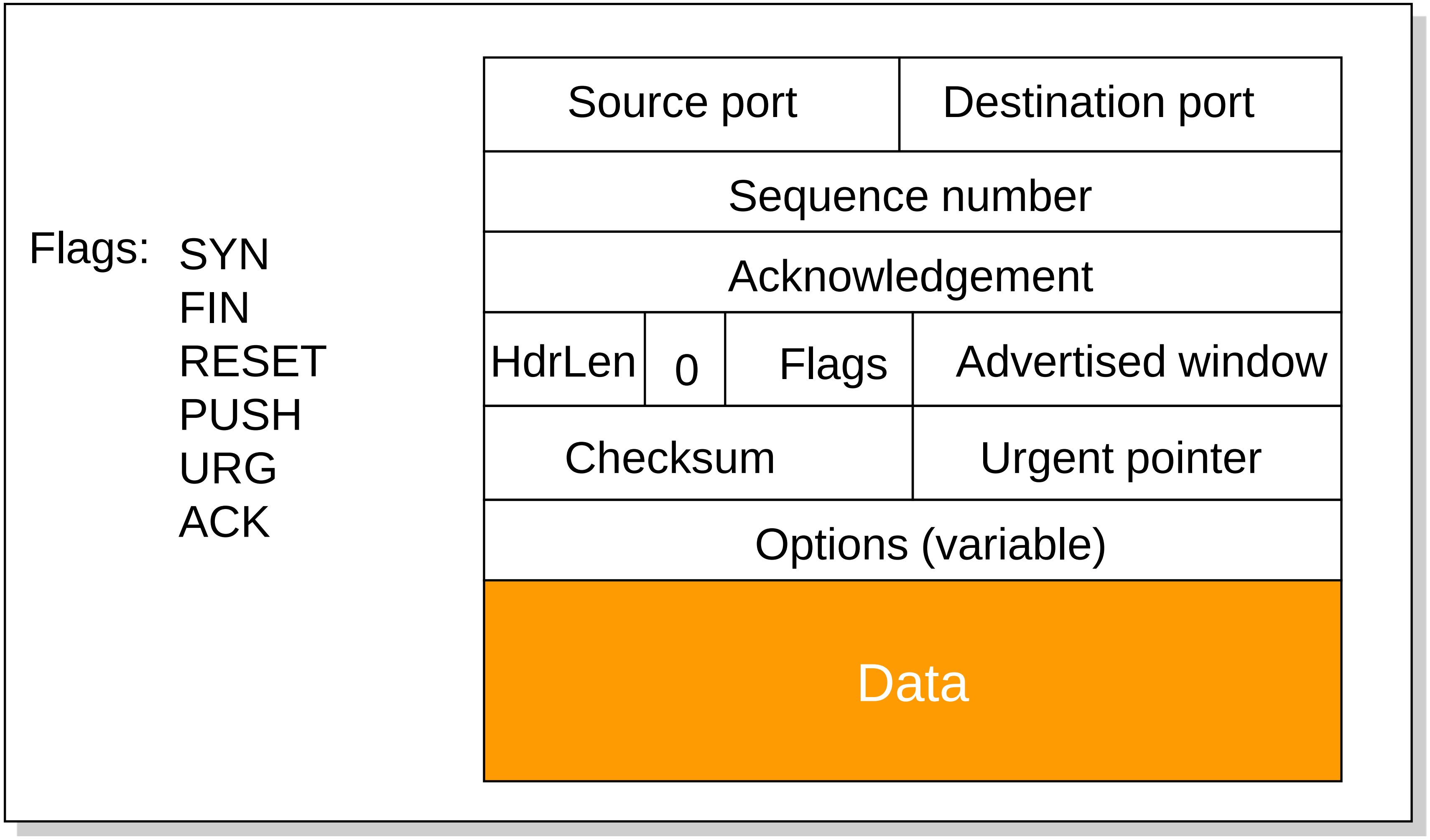
- Compute checksum of received segment
- Check if computed checksum equals checksum field value:
  - » NO - error detected
  - » YES - no error detected

    But maybe errors nonethless?

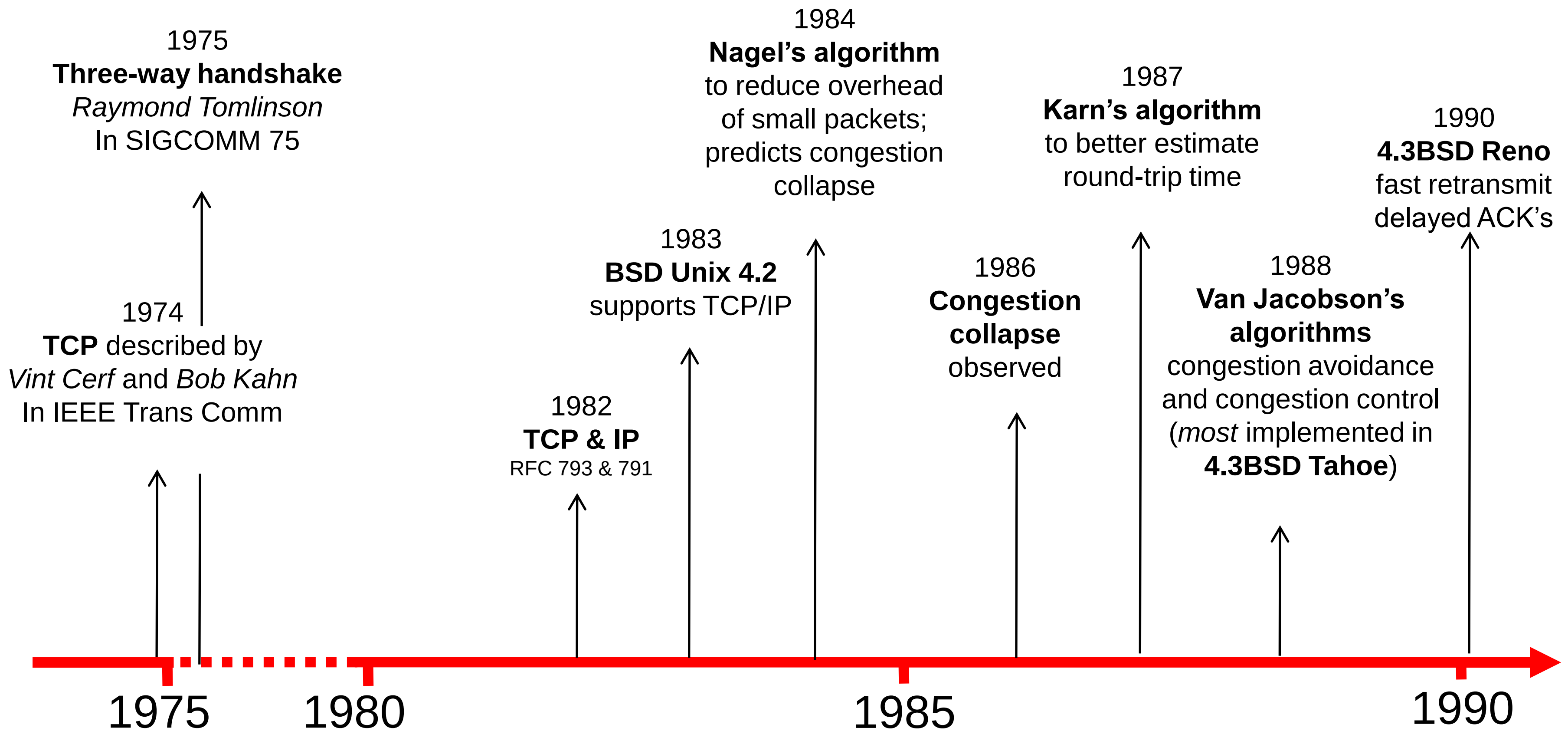
High-Level TCP Characteristics

- Protocol implemented entirely at the ends
  - » Fate sharing
- Protocol has evolved over time and will continue to do so
  - » Nearly impossible to change the header
  - » Use options to add information to the header
  - » Change processing at endpoints
  - » Backward compatibility is what makes it TCP

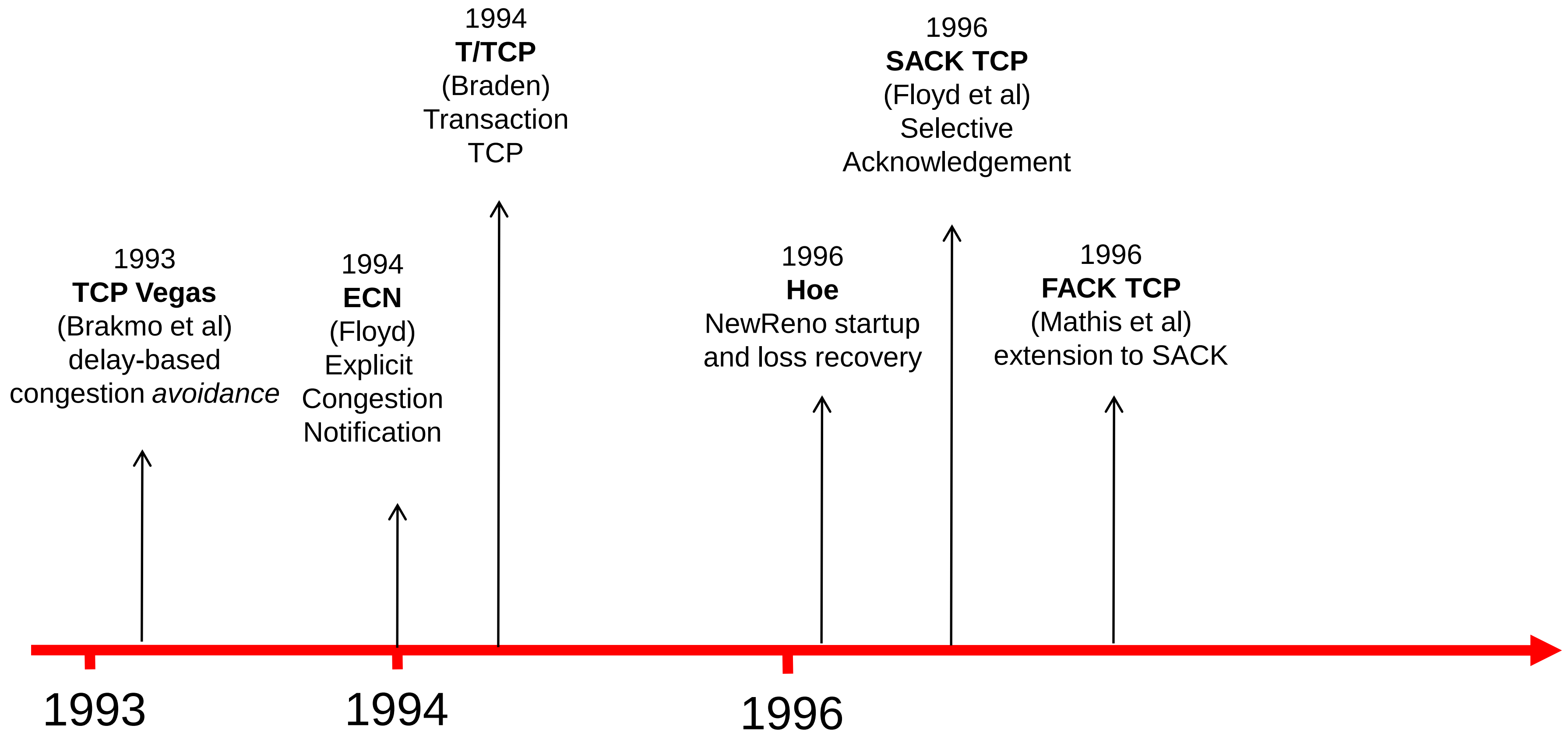
TCP Header


Flags: SYN
FIN
RESET
PUSH
URG
ACK
Source port
Destination port
Sequence number
Acknowledgement
HdrLen
0
Flags
Advertised window
Checksum
Urgent pointer
Options (variable)
Data

Evolution of TCP



TCP Through the 1990s

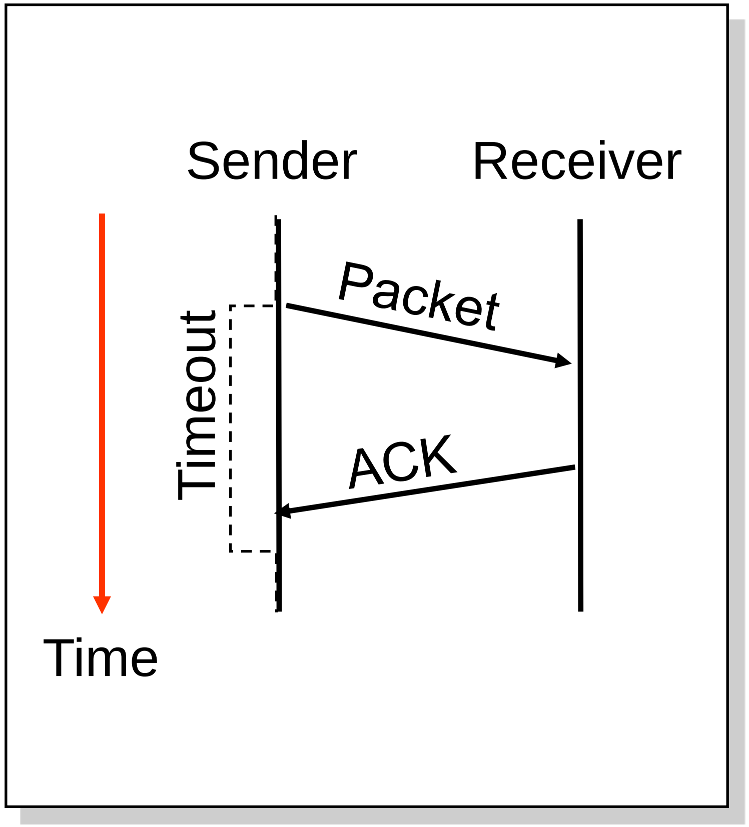


Outline

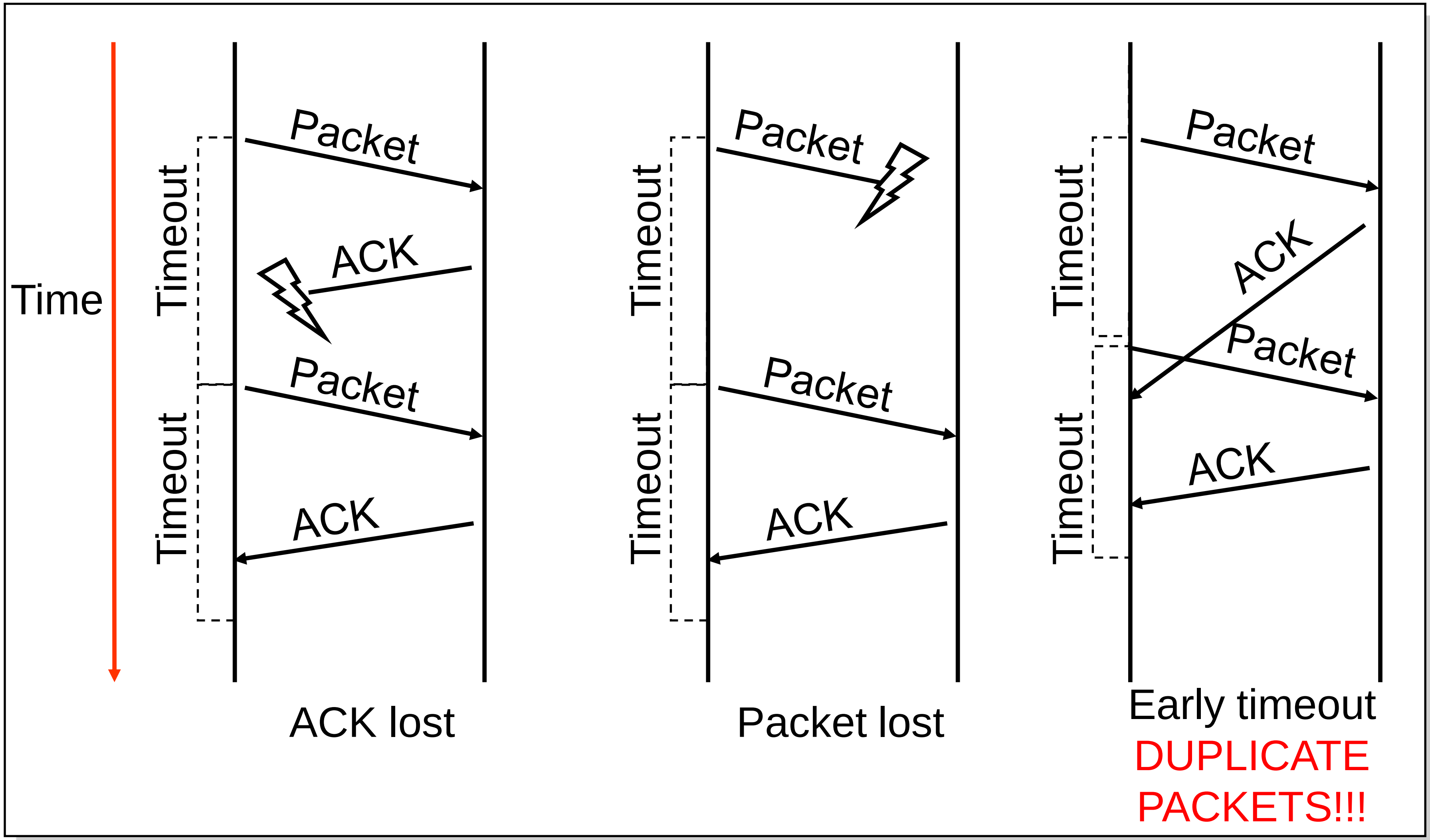
- Transport introduction
- Error recovery & flow control

Stop and Wait

- ARQ
  - » Receiver sends acknowledgement (ACK) when it receives packet
  - » Sender waits for ACK and timeouts if it does not arrive within some time period
- Simplest ARQ protocol
- Send a packet, stop and wait until ACK arrives



Recovering from Error

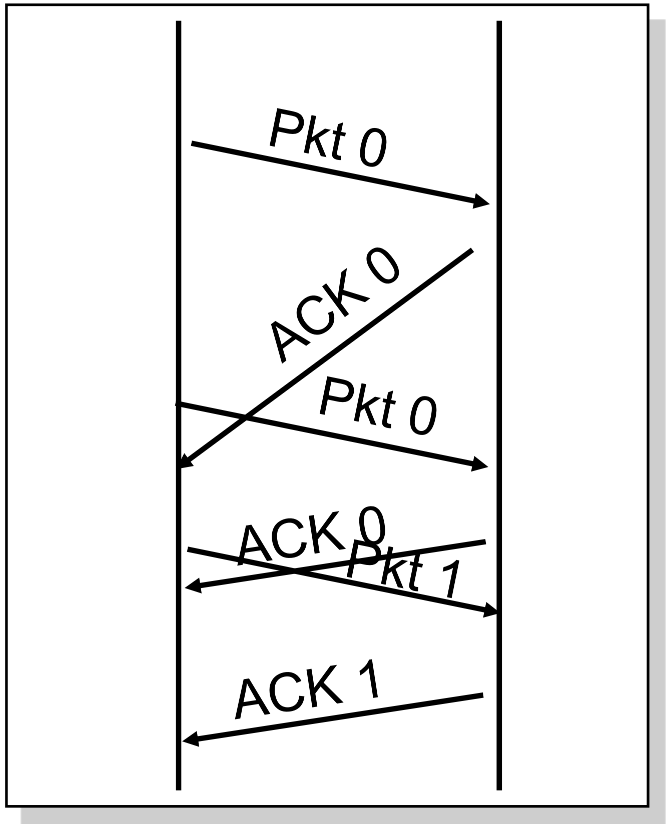


Problems with Stop and Wait

- How to recognize a duplicate
- Performance
  - Can only send one packet per round trip

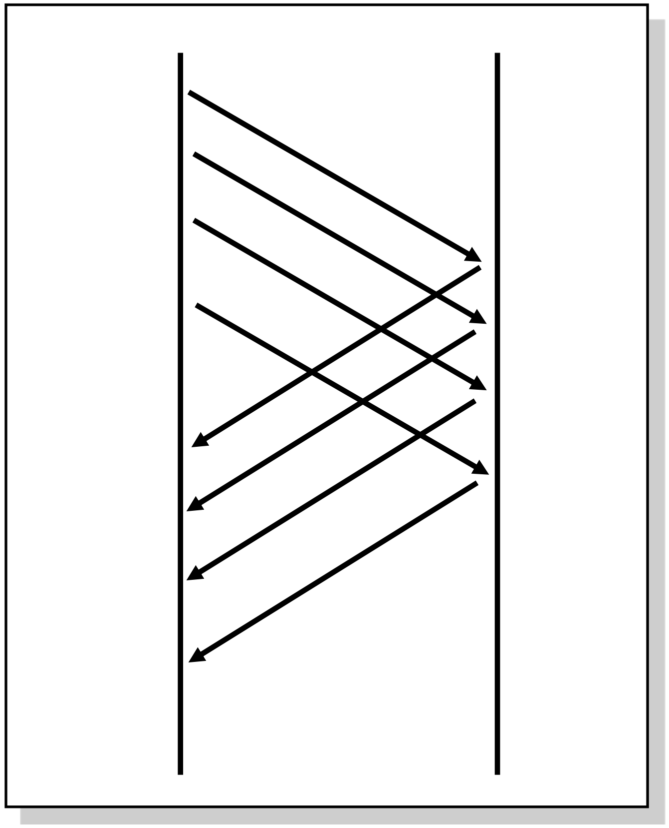
How to Recognize Resends?

- Use sequence numbers
  - both packets and acks
- Sequence # in packet is finite → How big should it be?
  - For stop and wait?
- One bit – won’t send seq #1 until received ACK for seq #0



How to Keep the Pipe Full?

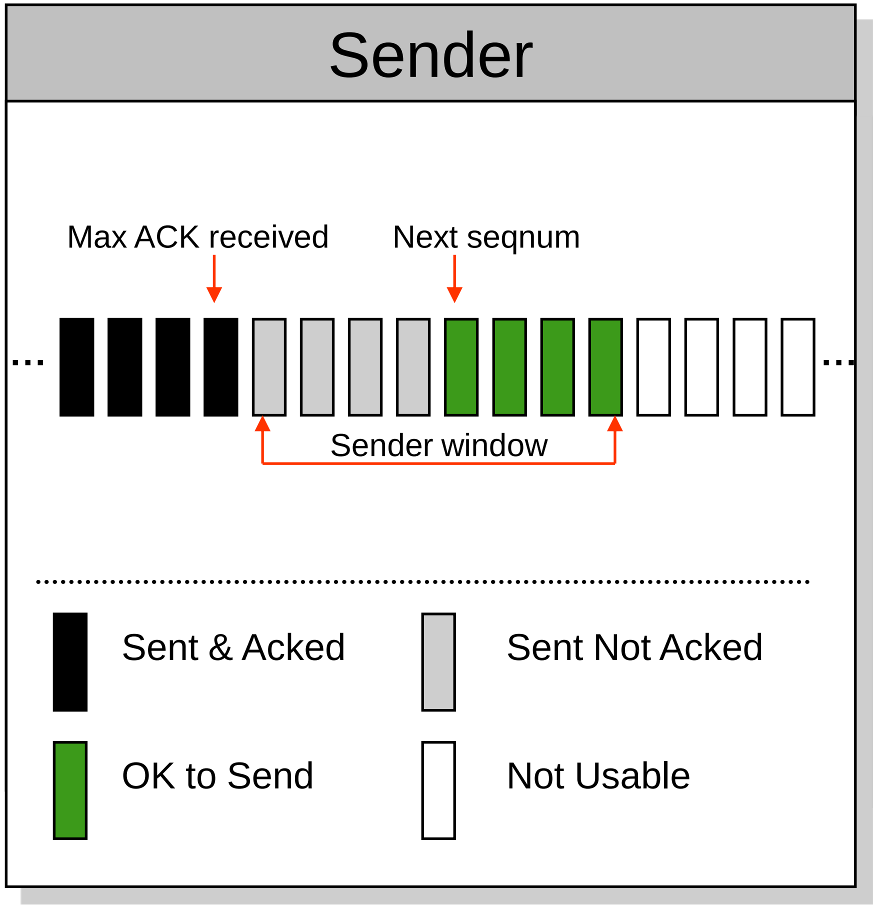
- **Send multiple packets without waiting for first to be acked**
  - » **Number of pkts in flight = window**
- **Reliable, unordered delivery**
  - » **Several parallel stop & waits**
  - » **Send new packet after each ack**
  - » **Sender keeps list of unack'ed packets; resends after timeout**
  - » **Receiver same as stop & wait**
- **How large a window is needed?**
  - » **Suppose 10Mbps link, 4ms delay, 500byte pkts**
    - – **1? 10? 20?**
  - » **Round trip delay * bandwidth = capacity of pipe**

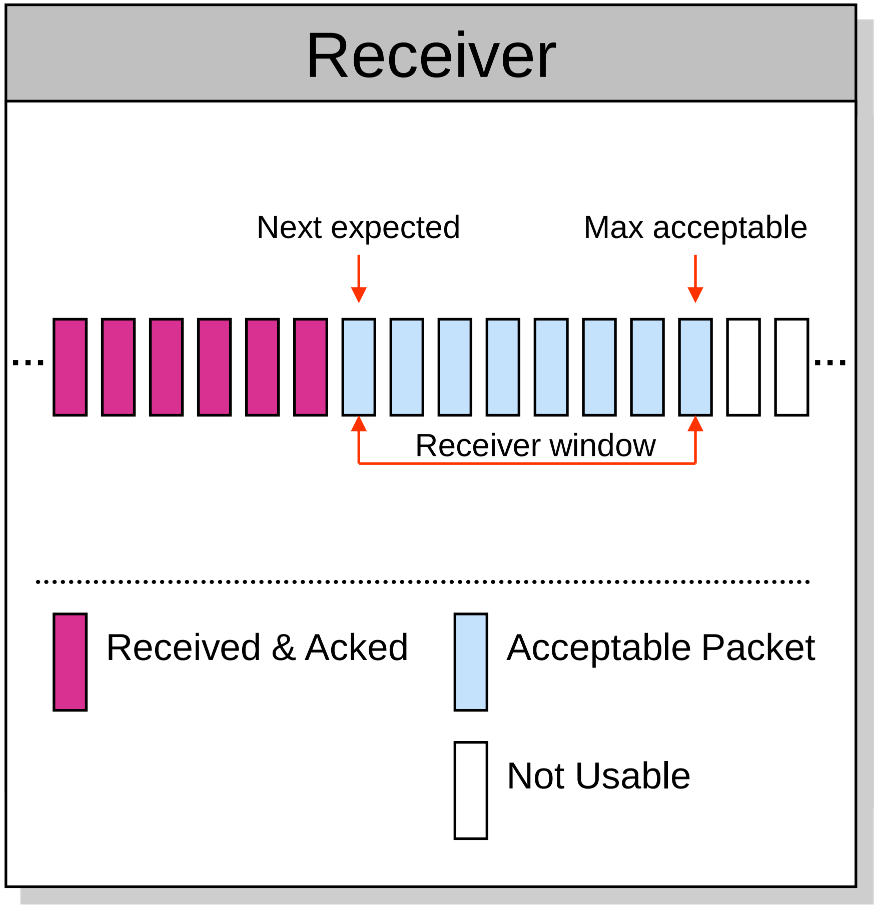


Sliding Window

- Reliable, ordered delivery
- Receiver has to hold onto a packet until all prior packets have arrived
  - » Why might this be difficult for just parallel stop & wait?
  - » Sender must prevent buffer overflow at receiver
- Circular buffer at sender and receiver
  - » Packets in transit ≤ buffer size
  - » Advance when sender and receiver agree packets at beginning have been received

Sender/Receiver State





Sequence Numbers

- **How large do sequence numbers need to be?**
  - » **Must be able to detect wrap-around**
  - » **Depends on sender/receiver window size**
- **E.g.**
  - » **Max seq = 7, send win=recv win=7**
  - » **If pkts 0..6 are sent succesfully and all acks lost**
    - – **Receiver expects 7,0..5, sender retransmits old 0..6!!!**
- **Max sequence must be $\geq$ send window + recv window**

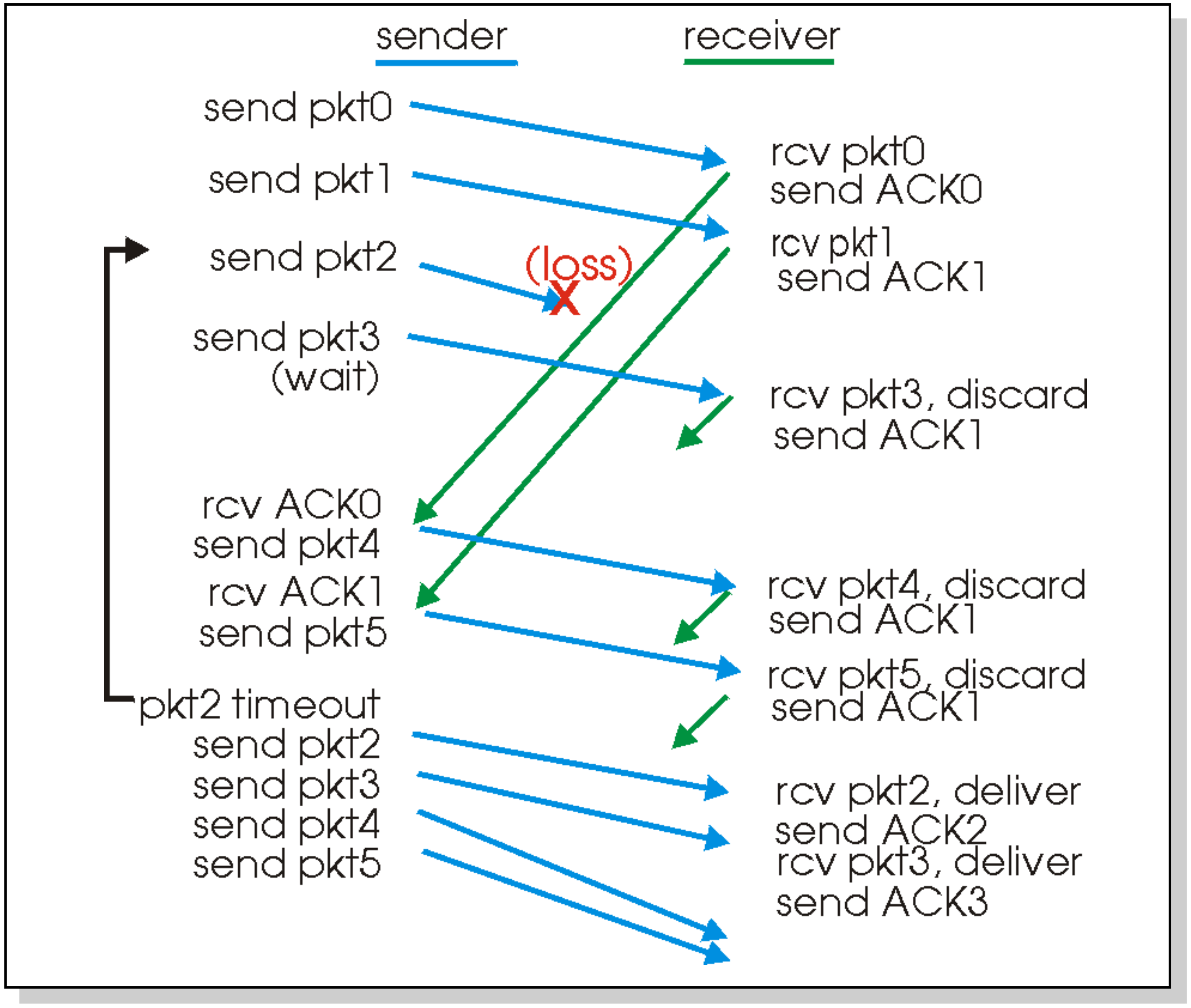
Window Sliding – Common Case

- On reception of new ACK (i.e. ACK for something that was not acked earlier)
  - » Increase sequence of max ACK received
  - » Send next packet
- On reception of new in-order data packet (next expected)
  - » Hand packet to application
  - » Send cumulative ACK – acknowledges reception of all packets up to sequence number
  - » Increase sequence of max acceptable packet

Loss Recovery

- On reception of out-of-order packet
  - Send nothing (wait for source to timeout)
  - Cumulative ACK (helps source identify loss)
- Timeout (Go-Back-N recovery)
  - Set timer upon transmission of packet
  - Retransmit all unacknowledged packets
- Performance during loss recovery
  - No longer have an entire window in transit
  - Can have much more clever loss recovery

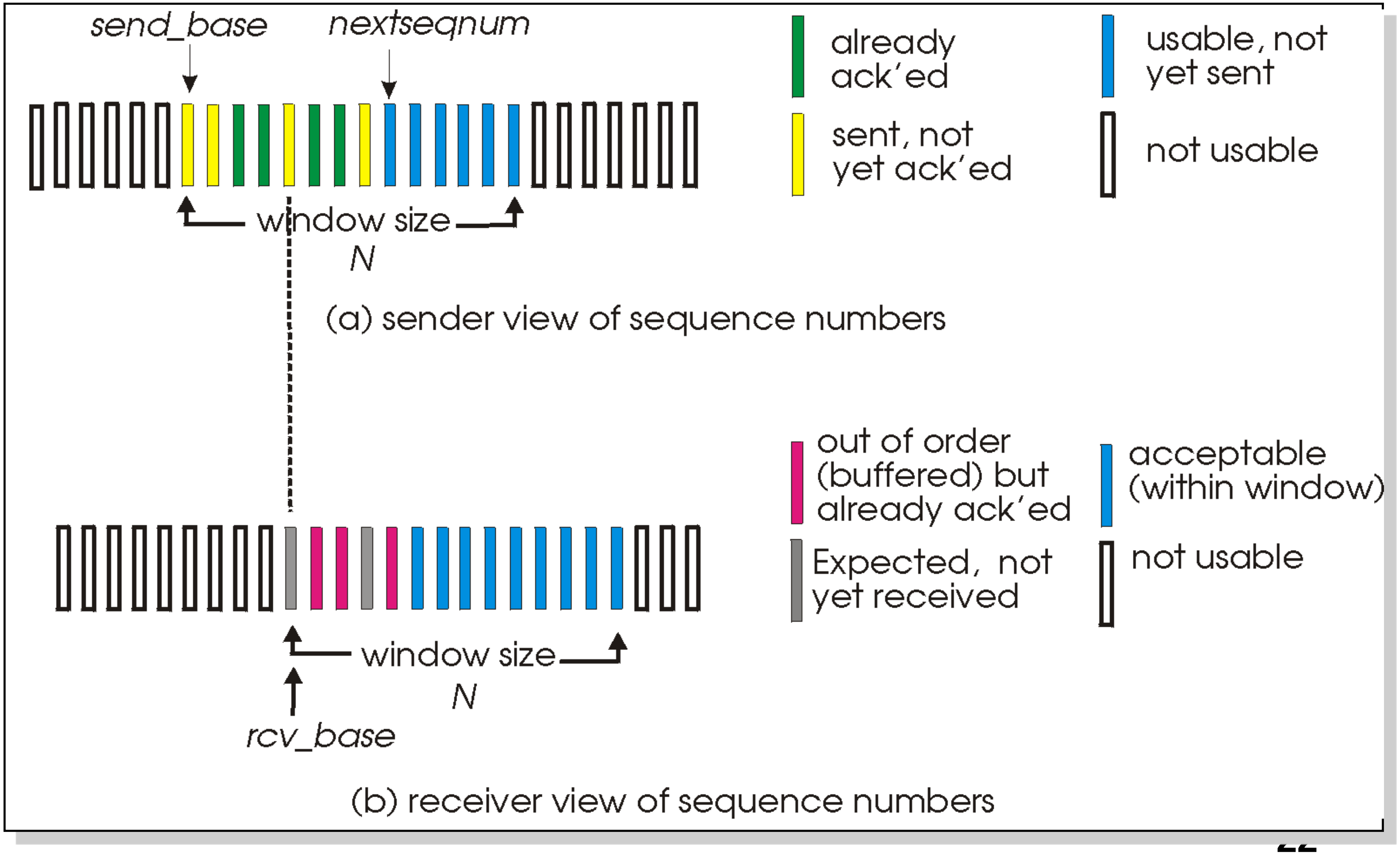
Go-Back-N in Action



Selective Repeat

- Receiver *individually* acknowledges all correctly received pkts
  - » Buffers packets, as needed, for eventual in-order delivery to upper layer
- Sender only resends packets for which ACK not received
  - » Sender timer for each unACKed packet
- Sender window
  - » N consecutive seq #'s
  - » Again limits seq #s of sent, unACKed packets

Selective Repeat: Sender, Receiver Windows


send_base
nextseqnum
window size
N
already ack'ed
sent, not yet ack'ed
usable, not yet sent
not usable
(a) sender view of sequence numbers
out of order (buffered) but already ack'ed
Expected, not yet received
acceptable (within window)
not usable
window size
N
rcv_base
(b) receiver view of sequence numbers

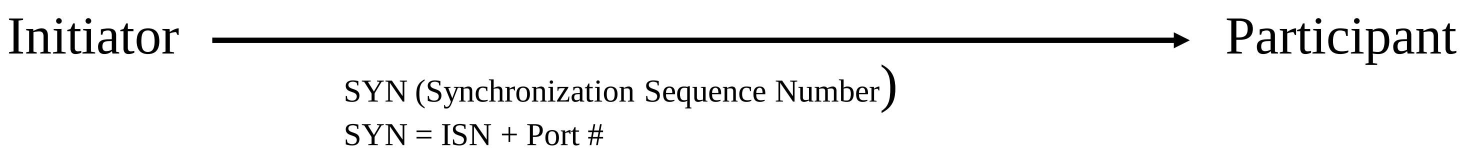
Important Lessons

- Transport service
  - UDP → mostly just IP service
  - TCP → congestion controlled, reliable, byte stream
- Types of ARQ protocols
  - Stop-and-wait → slow, simple
  - Go-back-n → can keep link utilized (except w/ losses)
  - Selective repeat → efficient loss recovery
- Sliding window flow control
  - Addresses buffering issues and keeps link utilized

Transmission Control Protocol (TCP)

- Reliable
- Connection-oriented
- Point-to-point
- Full-duplex
- Streams, not messages

Initialization: 3 Way Handshake

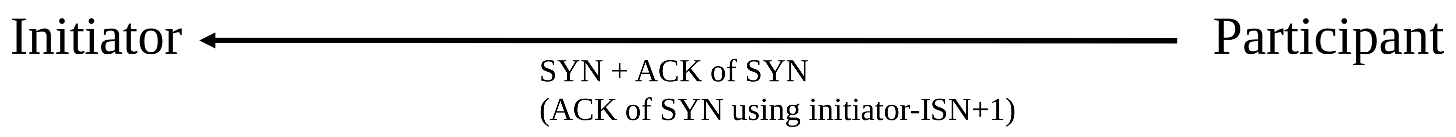


- The client begins it's active open by sending a SYN to the server. SYN stands for "Synchronization Sequence Number", but it actually contains much more.

- The SYN message contains the initial sequence number (ISN). This ISN is the starting value for the sequence numbering that will be used by the client to detect duplicate segments, to request the retransmission of segments, &c.

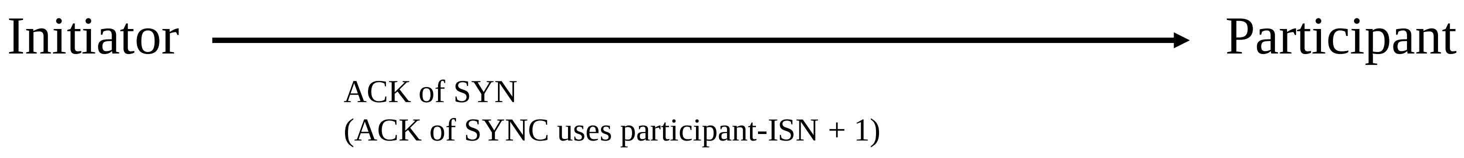
- The message also contains the *port number*. Whereas the hostname and IP address name the machine, the port number names a particular processes. A process on the server is associated with a particular port using bind().

Initialization: 3 Way Handshake



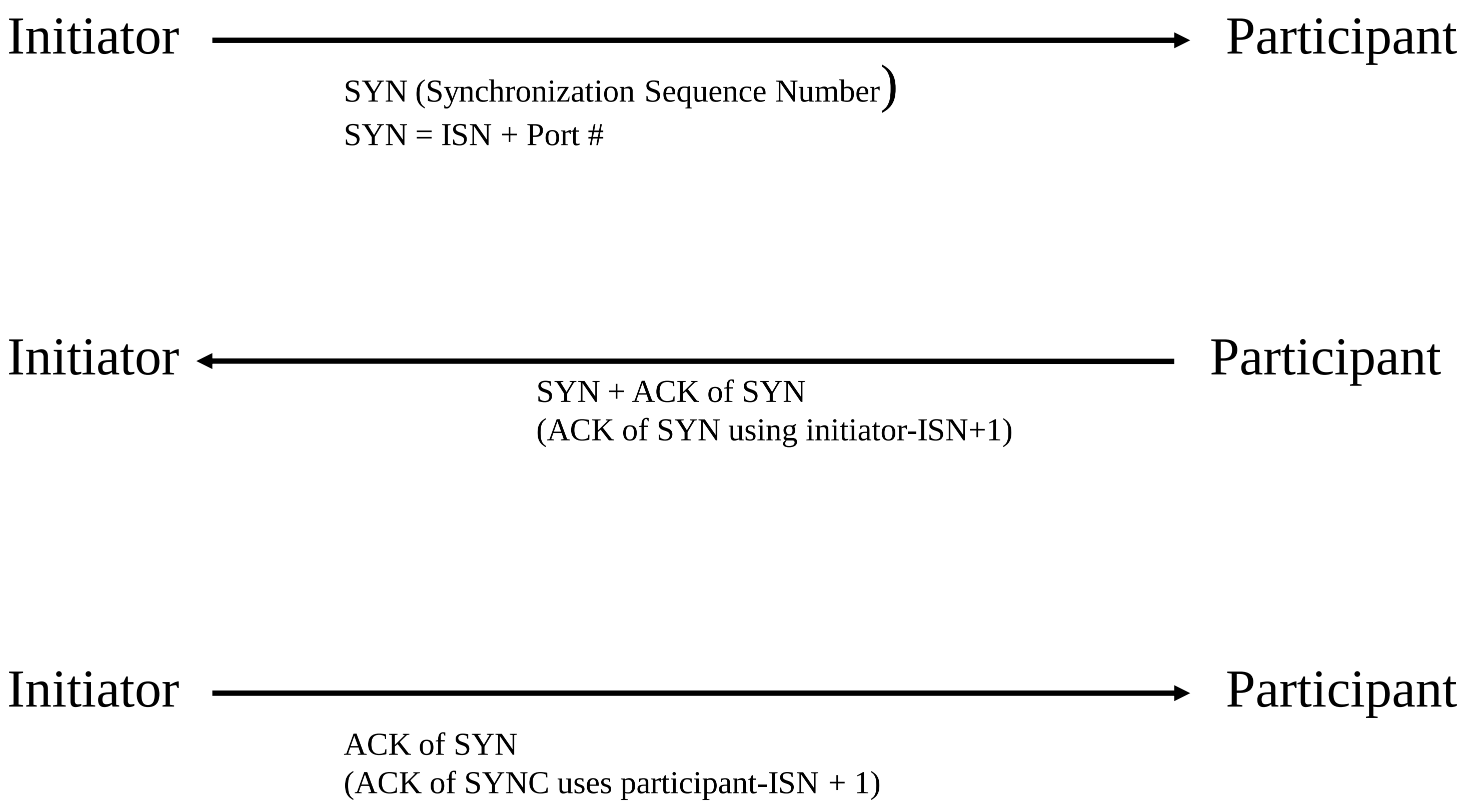
- The server performs the passive open, by sending its own ISN to the client. It also sends an Acknowledgement (ACK) of the client's SYN, using the ISN that the client sent plus one.

Initialization: 3 Way Handshake



- The last step is for the client to acknowledge the server's SYN

Initialization: 3 way Handshake


Initiator
Participant
SYN (Synchronization Sequence Number)
SYN = ISN + Port #
Initiator
Participant
SYN + ACK of SYN
(ACK of SYN using initiator-ISN+1)
Initiator
Participant
ACK of SYN
(ACK of SYNC uses participant-ISN + 1)

How and Why is the ISN Chosen?

- **Why do we send the ISN, instead of just always start with 1?**
- **The answer to this is that we don't want to misinterpret an old segment. For example, consider a short-lived client process that always talked to the same server. If the ISN's would always start with one, a delayed segment from one connection might be misinterpreted as the next segment for a newer instance of the same client/server-port combination. By doing something more random, we reduce the bias toward low sequence numbers, and reduce the likelihood of this type of situation.**
- **RFC 793 specifies that the ISN should be selected using a system-wide 32-bit counter that is incremented every 4 microseconds. This approach provides a "moving target" that makes segment number confusion unlikely.**
- **4.4BSD actually does something different. It increments the counter by 64K every half-second and every time a connection is established. This amortizes to incrementing the counter by one every 8 microseconds.**

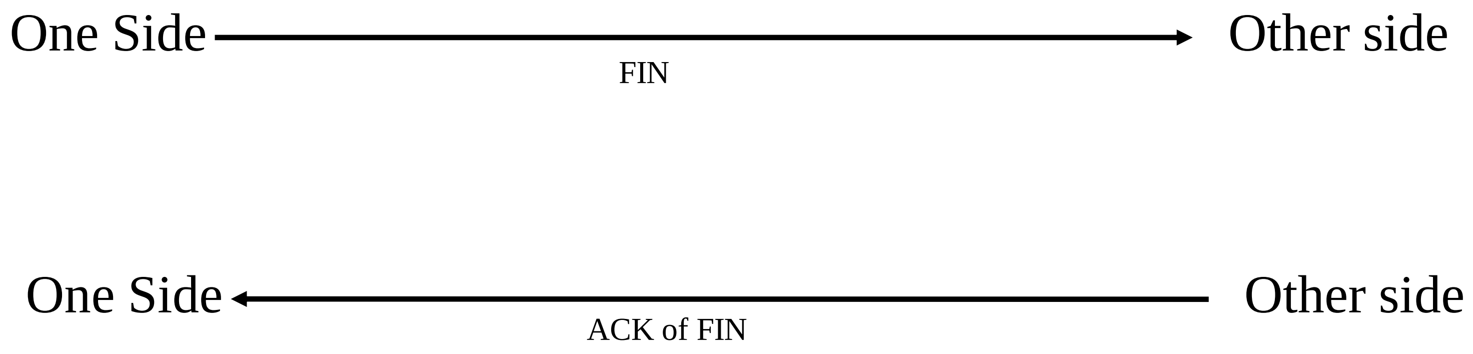
Connection Termination

- When either side of a TCP connection is done sending data, it sends a FIN (finished) to the other side. When the other side receives the FIN, it passes an EOF up the protocol stack to the application.

- Although TCP is a full-duplex protocol, the sending of a FIN doesn't tear down the whole connection. Instead it simply indicates that the side sending the FIN won't send any more data. It does not prevent the other side from sending data. For this reason, it is known as a *half-close*. In some sense, a half-closed connection is a half-duplex connection.

- Although TCP allows for this half-closed state, in practice, it is very rarely used. For the most part, when one side closes a connection, the other side will immediately do the same. It is also the case that both sides can concurrently sends FINs. This situation, called a *simultaneous close* is perfectly legal and acceptable.

Half Close



Maximum Segment Life

- MSL stands for *Maximum Segment Life*.
- Basically, MSL is a constant that defines the maximum amount of time that we believe a segment can remain in transit on the network.
- 2MSL, twice this amount of time, is therefore an approximation of the maximum round trip time.
- We wait 2MSL after sending the ACK of the FIN, before actually closing the connection, to protect against a lost ACK.
- If the ACK is lost, the FIN will be retransmitted and received. The ACK can then be resent and the 2MSL timer restarted.

What About Crashes, &c.

- But wait, if both sides need to close the connection, what happens if the power fails on one side? Or a machine is shut off? Or the network goes down?
- Well, the answer to this is very simple: Nothing. Each side will maintain at least a half-open connection until the other side sends a FIN. If the other side never sends a FIN, barring a reboot, the connection will remain at least half-open on the other side.
- What happens if neither process ever sends data? The answer to this is also very simple: Nothing. Absolutely nothing is sent via TCP, unless data is being sent.

TCP Keep-Alive Option

- Well, some people were as upset as you were by the idea that a half-open connection could remain and consume resources forever, if the other side abruptly died or retired. They successfully lobbied for the *TCP Keepalive Option*.

- This option is disabled by default, but can be enabled by either side. If it is enabled on a host, the host will probe the other side, if the TCP connection has been idle for more than a threshold amount of time.

- This timer is system-wide, not connection wide and the RFC states that, if enabled, it must be no less than two hours.

- Many people (including your instructor) believe that this type of feature is not rightfully in the jurisdiction of a transport layer protocol. We argue that this type of session management is the rightful jurisdiction of the application or a session-level protocol.

- Please do realize that this is a religious issue for many and has received far more discussion than it is probably worth. Independent of your beliefs, please don't forget that the timer is system-wide -- this can be a pain and might even lead many keepalive-worshipers opt for handling this within the applications.

Reset (RST)

- TCP views connections in terms of *sockets*. A popular author, Richard Stevens refers to these as *connections* -- this is wrong, but has worked its way into the popular vernacular.

- A socket is defined as the following tuple:

  <destination IP address, destination port #, source IP address, source port number>

- A RST is basically a suggestion to abort the connection.

- A reset will generally be sent by a host if it receives a segment that doesn't make sense. Perhaps the host crashed and then received a segment for a port that is no longer in use.

- In this case, the RST would basically indicate, "No one here, but us chickens" and the side that received the RST would assume a crash, close its end and roll-over or handle the error.

Transferring Data

- TCP operates by breaking data up into pieces known as ***segments***.
- The TCP packet header contains many pieces of information. Among them is the Maximum Segment Length (MSL) that the host is willing to accept.
- In order to send data, TCP breaks it up into segments that are not longer than the MSL.

Acknowledgement

- Fundamentally, TCP sends a segment of data, including the segment number and waits for an ACK. But TCP tries to avoid the overhead involved in acking every single segment using two techniques.
- TCP will wait up to 200mS before sending an ACK. The hope is that within that 200 mS a segment will need to be sent the other way. If this happens, the ACK will be sent with this segment of data. This type of ACK is known as a *piggyback ACK*.
- Alternatively, no outgoing segment will be dispatched for the sender within the 200mS window. In this case the ACK is send anyway. This is known as a *delayed ACK*.
- *Note*: My memory is that the RFC actually says 500mS, but the implementations that I remember use a 200mS timer. No big deal, either way.

More About the ACKs

- TCP uses *cumulative acknowledgement*.
- Except, if a segment arrives out of order, TCP will use an *immediate acknowledgement* of the last contiguous segment received.
- This tells the sender which segment is expected.
- This is based on the assumption that the likely case is that the missing segment was lost not delayed.
- If this assumption is wrong, the first copy to arrive will be ACKed, the subsequent copy will be discarded.

Nagle Algorithm

- One interesting observation is that it takes just as much overhead to send a small amount of data, such as one character, as it does a large amount of data, such as a full MSL of data.

- The massive overhead associated with small segments can be especially wasteful if the network is already bogged down.

- One approach to this situation is to delay small segments, collecting them into a full segment, before sending. This approach reduces the amount of non-data overhead, but it can unnecessarily delay small segments if the network isn't bogged down.

- The compromise approach that is used with TCP was proposed by Nagle. The Nagle Algorithm will send one small segment, but will delay the others, collecting them into a larger segment, until the segment that was sent is acknowledged. In other words, the Nagle algorithm allows only one unacknowledged small segment to be send.

Nagle Algorithm

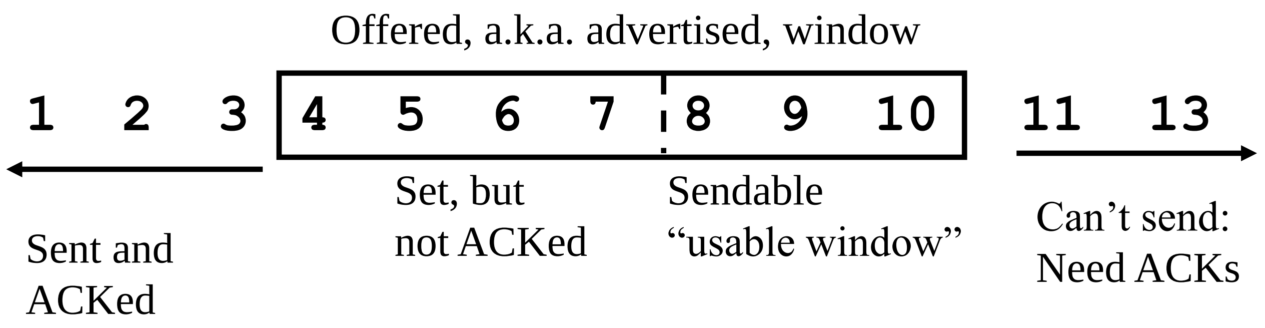
- **This approach has the following nice property. If the network is very bogged down, the ACK will take a long time. This will result in many small segments being collected into a large segment, reducing the overhead. If the network isn't bogged down, the ACK will arrive very rapidly, allowing the next small segment to be sent without much delay. If the network is fast, fewer small segments will be concatenated, but who cares? The network isn't doing much else.**

- **In other words, the Nagle algorithm favors the sending of short segments on a "fast network" and favors collecting them into larger segments on a "slow network." This is a very nice property!**

- **There are certain circumstances where the Nagle approach should be disabled. The classic example is the sending of mouse movements for the X Window system. In this example, it is critically important to dispatch the short packets representing mouse movements in a timely way, independent of the load on the network. These packets need a response in soft real-time to satisfy the human user.**

The Sliding Window Model

- As we mentioned earlier, TCP is a sliding window protocol much like the example protocol that we discussed last class. The sliding window model used by TCP is almost identical to model used in the example.
- In the case of TCP, the receiver's window is known as the *advertised window* or the *offered window*. The side of the window is advertised by the receiver as part of the TCP header attached to each segment. By default, this size is usually 4096 bytes.
- The *usable window* is the portion of the advertised window that is available to receive segments.
- The only significant difference is the one that we mentioned before: TCP uses a cumulative ACK instead of a bit-mask.



Slow Start and Congestion Avoidance

- **The advertised window size is a limit imposed by the receiver. But the sender doesn't necessarily need or want to send segments as rapidly as it can in an attempt to fill the receiver's window.**
- **This is because the network may not be able to handle the segments as rapidly as the sender can send them. Intermediate routers may be bogged down or slow. If the sender dispatches segments too rapidly, the intermediate routers may drop them requiring that they be resent.**
- **In the end, it would be faster and more bandwidth efficient to send them more slowly in the first place.**
- **TCP employs two different techniques to determine how many segments can be sent before acknowledgement: *slow start* and *congestion avoidance*.**
- **These techniques make use of a sender window, known as the *congestion window*. The congestion window can be no larger than the receiver's advertised window, but may be smaller. The congestion window size is known as *cwnd*.**

Slow Start

- Initially, the congestion window is one segment large. The sender will send exactly one segment and wait for an acknowledgement.
- Then the sender will send two segments. Each time an ACK is received, the congestion window will grow by two. (This results in 1,2,4,8,16,… growth)
- This growth will continue until the congestion window size reaches the smaller of a threshhold value, *ssthresh* and the advertised window size.
- If the congestion window reaches the same size as the advertised window, it cannot grow anymore.
- If the congestion window size reaches ssthresh, we want to grow more slowly -- we are less concerned about reaching a reasonable transmission rate than we are about suffering from congestion. For this reason, we switch to congestion avoidance.
- The same is true if we are forced to retransmit a segment -- we take this as a bad sign and switch to congestion avoidance.

Congestion Avoidance

- Congestion avoidance is used to grow the congestion window slowly.
- This is done after a segment has been lost or after ssthresh has been reached.
- Let's assume for a moment that ssthresh has been reached. At this point, we grow the congestion window linearly. This rate or growth is slower than it was before, and is more appropriate for tip-toeing our way to the network's capacity.

Congestion Avoidance

- **Eventually, a packet will be lost. Although this could just be bad luck, we assume that it is the result of congestion -- we are injecting more packets into the network than we should.**

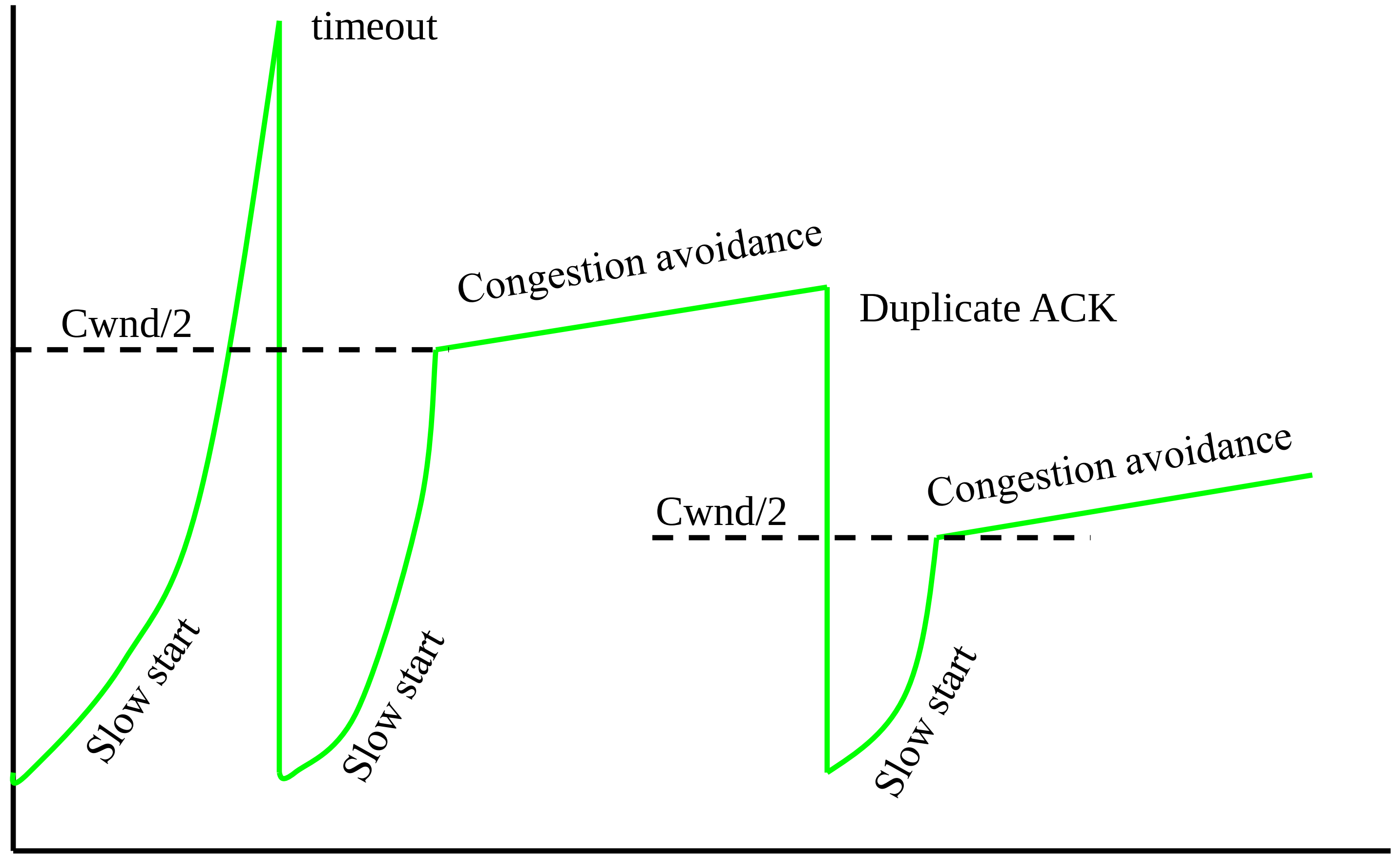
- **As a result, we want to slow down the rate at whcih we inject packets into the network. We want to back off a lot, and then work our way to a faster rate. So we reset ssthresh and cwnd:**

```
ssthresh = MAX (2, cwnd/2)
cwnd = 1
```

After Congestion Avoidance

- After reducing the congestion window, we reinvoke slow start.
- This time it will start with a cwnd size of 1 and grow rapidly to half of the prior congestion window size. At that point congestion avoidance will be reinvoked to make tip-toe progress toward a more rapid transmission rate.
- Eventually, a packet will be lost, ssthresh will be cut, cwnd will be reset to 1, and slow start will be reinvoked.
- It is important to notice that ssthresh doesn't always fall -- it can grow. Since ssthresh is set to (cwnd/2), if the new value of cwnd is more than twice the old value of ssthresh, ssthresh will actually increase.
- This makes sense, because it allows the transmission rate to slow down in response to a transient, but to make a substantial recovery rapidly. In this respect, the exponential

An Example of Slow Start and Congestion Avoidance



Tahoe, Reno, Vegas, and Friends

- Early TCP revisions focused on functionality, e.g. Nagle
- Recent TCP revisions focus on congestion.
- It is easy to see that Slow-Start/Congestion avoidance, as described are neither provable optimal nor sophisticated heuristics. They are functional, intuitive – but clearly far from perfect – hacks
- There are many newer tweaks to improve performance. But, the philosophy doesn’t change
- Fundamentally, lost ACKS are interpreted as messages from the router to the sender that there is congestion and that it should slow down.
- Various revisions are more agile in that they don’t necessarily assume the first missing ACK is a sign of congestion, so they apply a reduced penalty until it becomes really clear. They also may change the details of how the slow down and speed up phases work.

Does This Really Work? Can We do Better?

- Yes. It is far better than what we'd see if TCP were naïve to congestion. And, it is fully backward compatible.
- We could probably do better if we added some explicit message, such as an ICMP message, that communicated congestion explicitly.
- But, such a message is problematic. It adds work to routers that, as we discussed, are already super-busy and throughput limited
- But, more importantly, to whom would the router send such a message? It does not understand sessions or flows.
- If it would send it to any sender in the queue (or recently in the queue), it would unnecessarily punish old senders – new senders could show up late and proceed at an unrestricted rate.
- It would also generate wasted traffic to senders that would never send through it again, anyway.
- And, what would the sender do with it? Slow down forever? For just that session? Cache it for "a while"?
- But, what we've got clearly ain't good. This is an active research area!

Evidence That It Works The "New Sender Penalty"

- So, even if we'd explore the newest tweaks, we'd see that these are all hack-ish heuristics. And, we've discussed that none of this is provably optimal, if such a thing even exits. Intuitively, it is all better than nothing. But, is there any evidence that it is good?
- Here's something really cool. And, something that demonstrates an important lesson: It is hard to statically analyze and understand dynamic behavior.
- Congested routers favor "old senders" and, in effect, penalize old ones.

Evidence That It Works The "New Sender Penalty",cont

- Here's why: Because this mess actually works, old senders, those with long, established sessions, actually tend to mutually settle in on time slots and transmission rates.
- This happens because if they show up and the router queue is full, the dropped segment gets resent at a random time. And, the algorithms leave the senders sending at approximately the same rate for a long time (the linear portion).
- So, old senders bounce off each other until they find the right time and rate to send. New senders are more likely to show up to a full queue than the old senders which have reached a de facto agreement with each other as to the periodicity necessary to "jump right in"
- If we assume that the queue is full, the total number of messages queued per unit time can equal no more than the total number of messages that can be dispatched. So, if the protocol is efficient, the old senders will optimize to find this – sending at the right rate and synchronizing to hit the queue at different times.
- So, old senders tend to get through more often than new senders. And, we really do (sometimes) observe this in practice. Who'd have thunk it?
- To defeat this, some routers drop preemptively, before the queue is full. They drop *random* messages from their queue. This hurts old senders, as new senders aren't queued. It thereby gives new senders a chance.

Being Not-So-Nice

- It is surely possible to open multiple, simultaneous pipes between a sender and a receiver.
- Since TCP maintains per-session state, each pipe has its own state variables – its own window sizes, and its own understanding of congestion
- One can be anti-social and achiever higher data rates this way. For example, consider a large file. If we request it over a single pipe, our data rate will go up and down as per congestion control and slow start
- But, if we use a client that organizes itself to get various blocks of the file over multiple pipes, we won’t necessarily slow down as much.
- First, even when slow, we might be able to scale linearly with multiple slow pipes
- Second, it is possible that, with enough slow pipes, we’ll be able to keep some going fast, while others have gotten knocked down and are growing their rate slowly.
- This is especially true when the penalty was due to random loss, rather than actual congestion or when transient congestion has ended.
- But, I say “might be able…”, because if the congestion is real, our anti-social behaviors may end up hurting everyone – including ourselves. At the end of the day, there is only so fast one can suck through a straw. And, trying to suck faster just hurts (more resends=more congestion, equals bigger problem)