

CS/ECE 4457

Computer Networks: Architecture and Protocols

Lecture 22 Reliable Transport and TCP

Qizhe Cai



Announcements

- Problem set 6 is released
- **Practice final exam will be released soon**

Goal of Today's Lecture

- Continue our understanding of reliable transport **conceptually**
- Understanding TCP will become infinitely easier
 - TCP involves lots of detailed mechanisms
 - **Knowing WHY TCP uses these mechanisms is most important**

Lets start with recapping last lecture

Recap: Best Effort Service (L3)

- Packets can be **lost**
- Packets can be **corrupted**
- Packets can be **reordered**
- Packets can be **delayed**
- Packets can be **duplicated**
- ...

Transport layer:

Enabling reliability over such a best-effort service model

Recap: Complete Correctness Condition for reliability

A transport mechanism is “reliable” if and only if

- (a) It resends all dropped or corrupted packets**
- (b) It attempts to make progress**

Recap: Four Goals for Reliable Transfer

- **Correctness**
 - As defined in the last slide
- **“Fairness”**
 - Every flow must get a fair share of network resources
- **Flow Performance (Latency-related)**
 - Latency, jitter, etc.
- **Utilization (Throughput-related)**
 - Would like to maximize bandwidth utilization
 - If network has bandwidth available, flows should be able to use it!

Recap: Solution v1

- **Send every packet as often and fast as possible...**
- Not correct
 - **if condition **not** satisfied: Transport must **attempt to make progress****
 - No way to check whether the packet was dropped or corrupted
 - So, must continue sending the same packet
- **What did we learn from this incorrect solution?**
 - **why we need receiver feedback**

Recap: Solution v2

- **Resend packet until you get an ACK**
 - **And receiver sends per-packet ACKs until data finally stops**
- Correct, fair, good (but suboptimal) latency, suboptimal utilization
 - A specific kind of under-utilization:
 - The source is unnecessarily sending the same packet
- **What did we learn from this solution?**
 - **why we must wait for an ACK after sending a packet**
 - But how long shall we wait for an ACK?
 - Indeed, the ACK may be lost as well

Recap: Solution v3

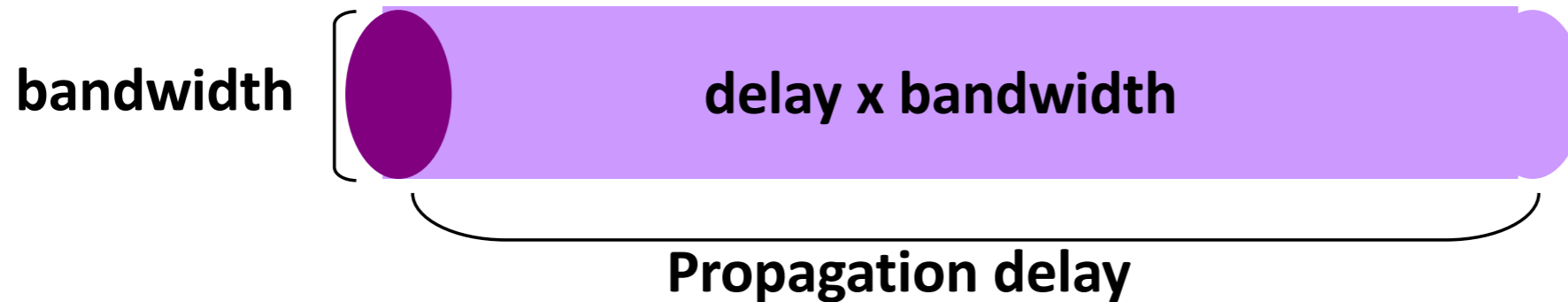
- **Send packet**
 - **But now, set a timer**
- **receiver sends per-packet ACKs**
- **If sender receives ACK, done**
- **If no ACK when timer expires, resend**
- Correct, fair, good (but suboptimal latency and utilization)
 - A different kind of under-utilization
 - source is not “work conserving”: could send, but is not
- **What did we learn from this solution?**
 - **We should not be just waiting; sender-side bandwidth wasted**
 - Keep more than one packet “in flight”
 - How many?

Recap: Window-based Algorithms

- Very simple concept
 - Send W packets
 - When one gets ACK'ed send the next packet in line
- **We want to set W such that:**
 - if I am sending at rate = link bandwidth, then
 - the ACK of the first packet arrives
 - exactly when I just finish sending the last of my W packets
 - **(assuming same transmission time for data and ACK packets)**
- **Lets me send as fast as the path can deliver...**

$RTT \times B \sim W \times \text{Packet Size}$

- Recall that **Bandwidth Delay Product**
 - $BDP = \text{bandwidth} \times \text{propagation delay}$



- **$B \times RTT$ is merely $2 \times BDP$**
- Window sizing rule:
 - Total bits in flight is roughly the amount of data that fits into forward and reverse “pipes”
 - Here pipe is complete path, not single link...
 - **This is not “detail”, this is a fundamental concept...**

Where Are We?

- **Figured out correctness condition:**
 - Always resend lost/corrupted packets
 - Always try to make progress (but can give up entirely)
- **Figured out single packet case:**
 - Send packet, set timer, resend if no ACK when timer expires
- **Some progress towards multiple packet case:**
 - Allow many packets (W) in flight at once
 - And know what the ideal window size is
 - $RTT \times B / \text{Packet size}$
- What's left to design?

Questions?

Three Design Considerations

- Nature of feedback
 - What should ACKs tell us when we have many packets in flight
- Detection of loss
- Response to loss

ACK Individual Packets

The receiver sends ACK for each individual packet that it receives

Example:

- Assume that packet 5 is lost, but no others
- Stream of ACKs will be
 - 1
 - 2
 - 3
 - 4
 - **6**
 - **7**
 - **8**
 - ...

ACK Individual Packets

- **Nature of feedback:** simple - the receiver ACKs each packet
- **Loss detection:** simple - ACKs tell the fate of each packet to the source
- **Response to loss: moderate:**
 - + Retransmit the packet for which ACK not received
 - + Reordering not a problem
 - + Simple window algorithm
 - W independent single packet algorithms
 - When one finishes grab next packet
 - - **Loss of ACK packet requires a retransmission**

Full Information Feedback

- **List all packets that have been received**
 - Give highest cumulative ACK plus any additional packets

Same Example (suppose packet 5 gets lost):

- Same story, except that the “hole” is explicit in each ACK
- Stream of ACKs will be
 - Up to 1
 - Up to 2
 - Up to 3
 - Up to 4
 - Up to 4, plus 6
 - Up to 4, plus 6,7
 - Up to 4, plus 6,7,8
 - ...

Full Information Feedback

- **Nature of feedback: complex** - feedback may have high overheads
 - If packets 1, 5, 6,, 100 received: ACK(1, 5, 6, ...,100)
- **Loss detection:** simple - the source still knows fate of each packet
- **Response to loss:** simple:
 - + Retransmit the packet for which ACK not received
 - + Reordering not a problem
 - + Simple window algorithm
 - + **Loss of ACK does not necessarily requires a retransmission**
 - **The next ACK will tell that the packet was indeed received**
 - Resilient form of individual ACKs

Cumulative ACK

- **Individual ACKs** can get lost, and require **unnecessary retransmission**
- **Full information feedback** can handle lost ACKs but has **high overheads**
- **Cumulative ACKs: a sweet spot between the two**
- Just the first part of full information feedback
- ACK the highest sequence number for all previously received packets

Cumulative ACKs (same example; say packet 5 lost)

Full information feedback:

- Stream of ACKs will be
 - Up to 1
 - Up to 2
 - Up to 3
 - Up to 4
 - Up to 4, plus 6
 - Up to 4, plus 6,7
 - Up to 4, plus 6,7,8
 - ...

Tells “**which**” packet arrived, and
which packet did not

Cumulative ACKs:

- Stream of ACKs will be
 - Up to 1
 - Up to 2
 - Up to 3
 - Up to 4
 - Up to 4
 - Up to 4
 - Up to 4
 - ...

Tells “**some**” packet arrived, and
which packet did not

Loss With Cumulative ACKs (cont'd)

- Duplicate ACKs are a sign of loss
 - The lack of ACK progress means 5 hasn't been delivered
 - Stream of duplicate ACKs means some packets are being delivered (one for each subsequent packet)
- Response to loss is trickier... When shall the source retransmit packet 5?
 - Packet may be delayed (so, source should wait)
 - Packet may be reordered (so, source should wait)
 - Or, packet may be dropped (source should immediately retransmit)
 - Impossible to know which one is the case
 - Life lesson: **be optimistic!**
 - Until optimism starts hurting
 - **Solution: retransmit after k duplicate ACKs**
 - **for some value of k, depending on how optimistic you feel!**

Cumulative ACKs (how is reordering handled; large k)

Receiver events:

- Packet 1 received
- Packet 2 received
- Packet 3 received
- Packet 4 received
- **Packet 6 received**
- Packet 7 received
- **Packet 5 received**
- **Packet 8 received**
- ...

Cumulative ACKs:

- Up to 1
- Up to 2
- Up to 3
- Up to 4
- **Up to 4**
- Up to 4
- **Up to 7**
- **Up to 8**
- ...

**Cumulative ACKs naturally handle packet reordering
(Packet delays are similar to reordering)**

Cumulative ACKs (confusion with duplication)

- Produce duplicate ACKs
 - Could be confused for loss with cumulative ACKs
 - But duplication is rare...

Source events:

- Packet 1 sent
- Packet 2 sent
- Packet 3 sent
- Packet 4 sent
- Packet 5 sent
- Packet 6 sent
- **Packet 3 resent**
- Packet 7 sent
- ...

Receiver events:

- Packet 1 received
- Packet 2 received
- **Packet 4 received**
- Packet 5 received
- Packet 6 received
- **Packet 3 received**
- **Packet 3 received**
- Packet 7 received
- ...

Cumulative ACKs:

- Up to 1
- Up to 2
- **Up to 2**
- **Up to 2**
- **Up to 2**
- Up to 6
- **Up to 6**
- **Up to 7**
- ...

Possible Design For Reliable Transport

- Cumulative ACKs
- Window based, with retransmissions after
 - Timeout
 - k subsequent ACKs
- This is correct, high-performant and high-utilization
 - At least as much as we can efficiently
- How about fairness?

Fairness? (Come back to later)

- The question of fairness comes up when:
 - Senders want to send data at rate higher than bandwidth
 - There will be packet loss!
- Adjust W based on losses...
- In a way that flows receive same shares
- Short version:
 - Loss: cut W by 2
 - Successful receipt of window: W increased by 1

Overview of Reliable Transport

- Window based self control separate concerns
 - Size of W
 - Nature of feedback
 - Response to loss
- Can design each aspect relatively independently
- Can be correct, fair, high-performant and high-utilization
- All of these are important concerns
 - **But correctness is most fundamental**
- Design **must** start with correctness
 - Can then “engineer” its performance with various hacks
 - These hacks can be “fun”, but don’t let them distract you

What Have We Done so far?

- Started from first principles
 - Correctness condition for reliable transport
- ... to understanding **why feedback from receiver is necessary** (sol-v1)
- ... to understanding **why timers may be needed** (sol-v2)
- ... to understanding **why window-based design may be needed** (sol-v3)
- ... to understanding **why cumulative ACKs may be a good idea**
 - Very close to modern TCP
- **You are now ready to learn TCP**

Lets learn TCP

Transport layer

- Transport layer offer a “pipe” abstraction to applications
- Data goes in one end of the pipe and emerges from other
- **Pipes are between processes, not hosts**
- There are two basic pipe abstractions

Two Pipe Abstractions

- **Unreliable packet** delivery (UDP)
 - Unreliable (application responsible for resending)
- **Reliable byte stream** delivery
 - Bytes inserted into pipe by sender
 - They emerge, in order at receiver (to the app)
- What features must transport protocol implement to support these abstractions?

UDP (Datagram Messaging Service)

- Sources send packets
- **Destinations do nothing**, but receive packets
- If packets delayed/reordered/lost:
 - Meh!
 - Let application handle packet loss (or be oblivious to drops)
 - If application needs reliable delivery, it must use reliable transport
- Discarding corrupted packets (optional)
- Nothing else!
- A minimal extension of IP

Transmission Control Protocol (TCP)

- Full duplex stream of **byte service**
 - **Sends and receives stream of bytes (segments), not messages**
- **Reliable, in-order delivery**
 - Ensures byte stream (eventually) arrives intact
 - In the presence of corruption, delays, reordering, loss

From design to implementation: major notation change

- Previously we focused on packets
 - Packets had numbers
 - ACKs referred to those numbers
 - Window sizes expressed in terms of # of packets
- TCP focuses on bytes, thus
 - Packets identified by the bytes they carry
 - ACKs refer to the bytes received
 - Window size expressed in terms of # of bytes

Basic Components of TCP

- **Connections:** Explicit set-up and tear-down of TCP sessions/connections
- **Segments, Sequence numbers, ACKs**
 - TCP uses byte sequence numbers to identify payloads
 - ACKs referred to sequence numbers
 - Window sizes expressed in terms of # of bytes
- **Retransmissions**
 - Can't be correct without retransmitting lost/corrupted data
 - TCP retransmits based on timeouts and duplicate ACKs
 - Timeouts based on estimate of RTT
- **Flow Control:** Ensures the sender does not overwhelm the receiver
- **Congestion Control:** Dynamic adaptation to network path's capacity

Connection/Session

Connections (Or Sessions)

- Reliability requires keeping state
 - Sender: packets sent but not yet ACKed, and related timers
 - Receiver: packets that arrived out-of-order
- Each byte stream is called a **connection** or **session**
 - Each with their own connection state
 - State is in hosts, not network

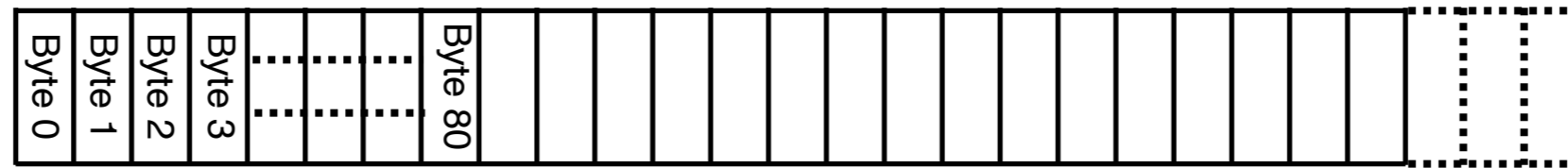
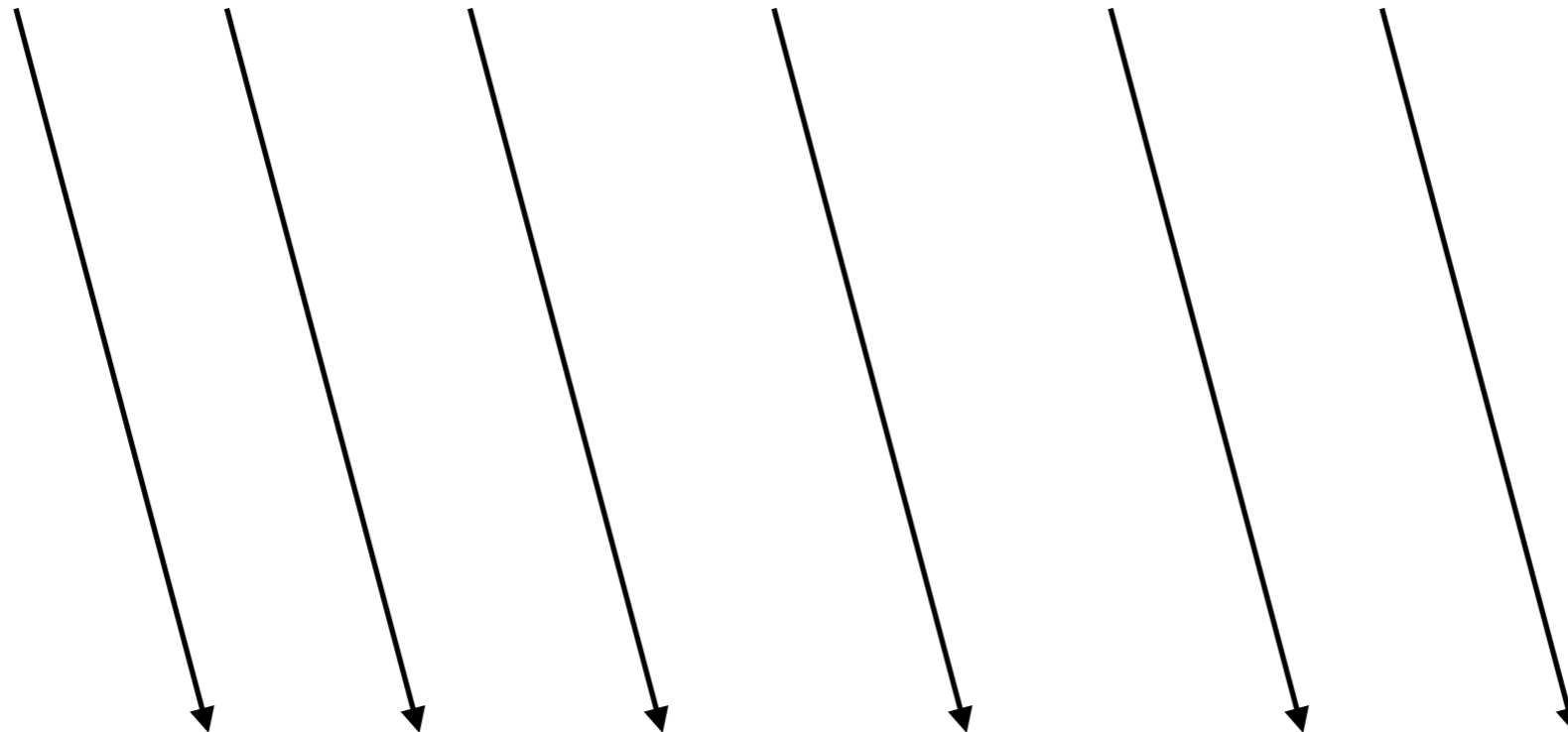
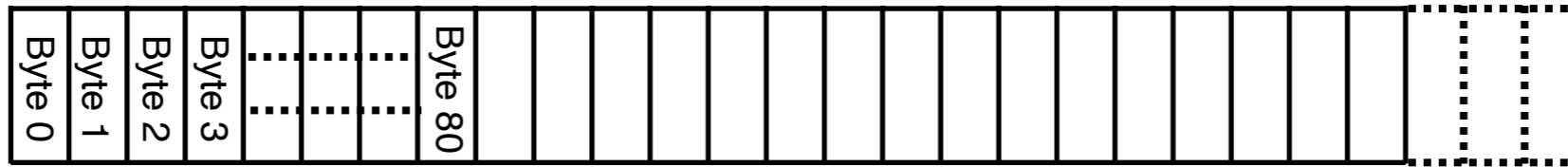
Basic Components of TCP

- **Connections:** Explicit set-up and tear-down of TCP sessions/connections
- **Segments, Sequence numbers, ACKs**
 - TCP uses byte sequence numbers to identify payloads
 - ACKs referred to sequence numbers
 - Window sizes expressed in terms of # of bytes
- **Retransmissions**
 - Can't be correct without retransmitting lost/corrupted data
 - TCP retransmits based on timeouts and duplicate ACKs
 - Timeouts based on estimate of RTT
- **Flow Control:** Ensures the sender does not overwhelm the receiver
- **Congestion Control:** Dynamic adaptation to network path's capacity

Segments and Sequence Numbers

TCP "Stream of Bytes" Service

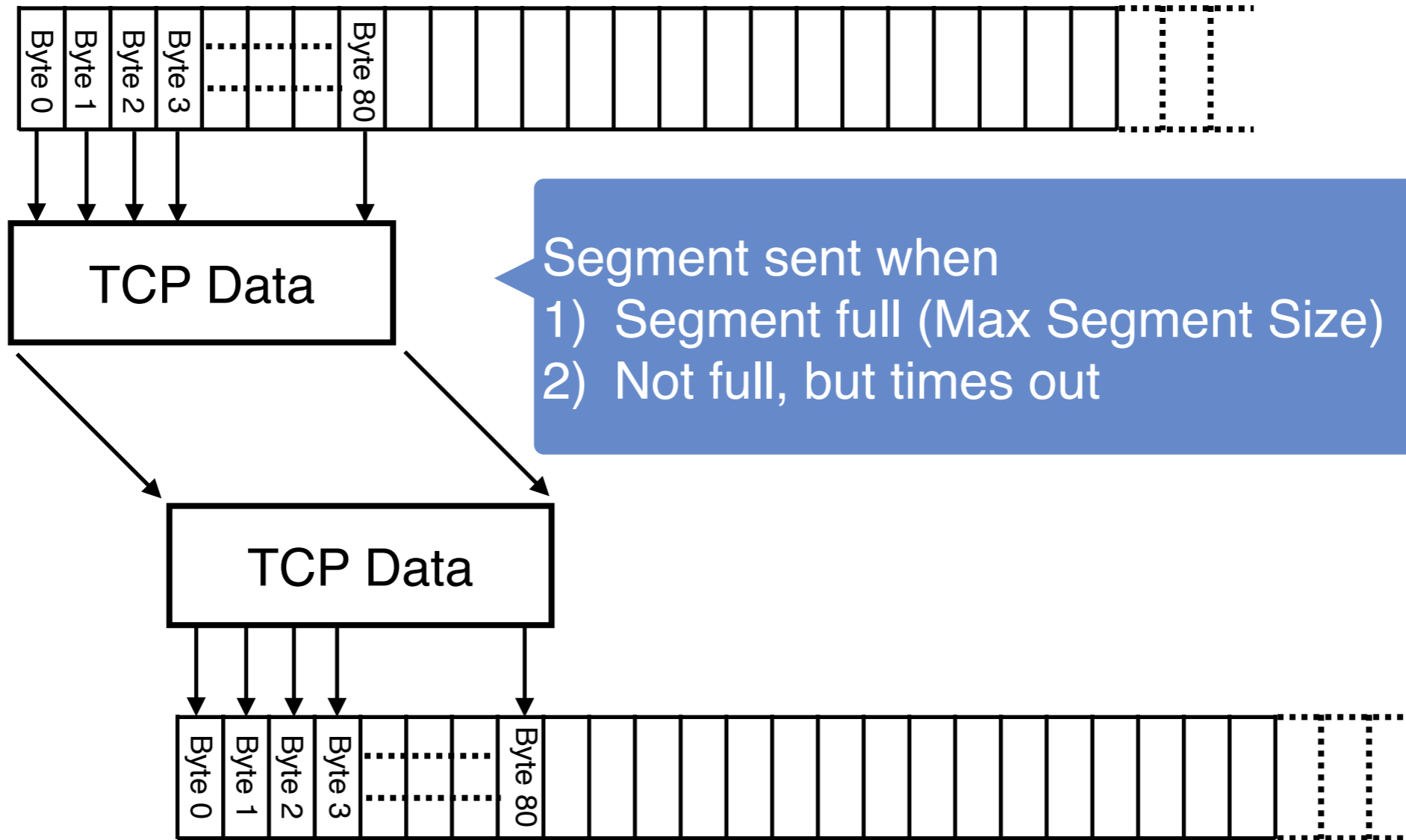
Application @ Host A



Application @ Host B

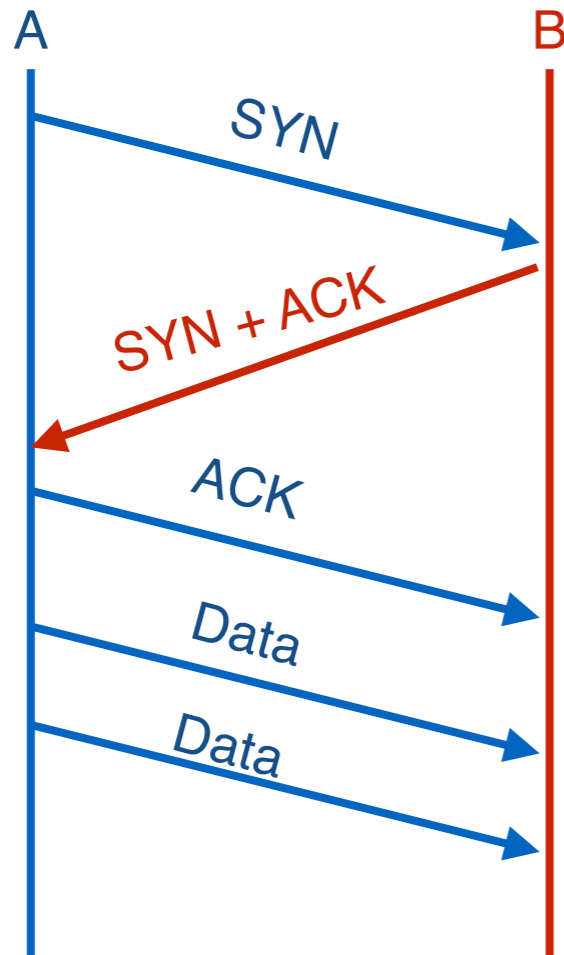
TCP "Stream of Bytes" Service

Application @ Host A



Application @ Host B

Establishing a TCP Connection



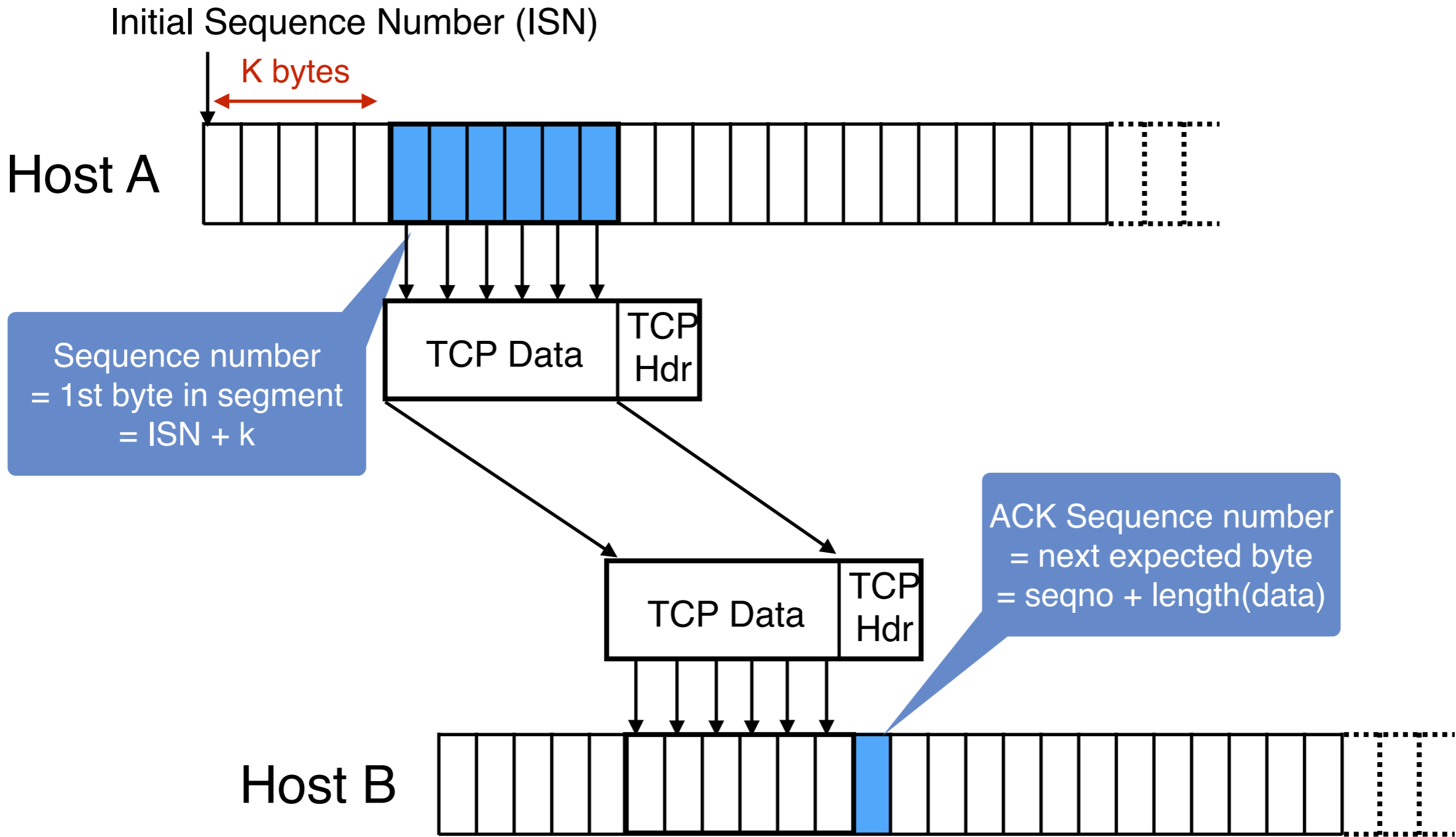
Each host tells its ISN to the other host.

- Three-way handshake to establish connection
 - Host A sends a **SYN** (open; “synchronize sequence numbers”) to host B
 - Host B returns a SYN acknowledgement (**SYN ACK**)
 - Host sends an **ACK** to acknowledge the SYN ACK

Initial Sequence Number (ISN)

- Sequence number for the very first byte
 - E.g., Why not just use ISN = 0?
- Practical issue
 - IP addresses and port #s uniquely identify a connection
 - Eventually, though, these port #s do get **used again**
 - ... small chance an old packet is **still in flight**
- TCP therefore requires changing ISN
 - Set from 32-bit clock that ticks every 4 microseconds
 - ... only wraps around once every 4.55 hours
- To establish a connection, hosts exchange ISNs
 - How does this help?

Sequence Numbers



ACKing and Sequence Numbers

- Sender sends segments (byte stream)
 - Data starts with sequence number X
 - Packet contains B bytes
 - $X, X+1, X+2, \dots, X+B-1$
- Upon receipt of a segment, receiver sends an ACK
 - If all data prior to X already received:
 - ACK acknowledges $X+B$ (because that is next expected byte)
 - If highest contiguous byte received is smaller value Y
 - ACK acknowledges $Y+1$
 - Even if this has been ACKed before

Basic Components of TCP

- **Connections:** Explicit set-up and tear-down of TCP sessions/connections
- **Segments, Sequence numbers, ACKs**
 - TCP uses byte sequence numbers to identify payloads
 - ACKs referred to sequence numbers
 - Window sizes expressed in terms of # of bytes
- **Retransmissions**
 - Can't be correct without retransmitting lost/corrupted data
 - TCP retransmits based on timeouts and duplicate ACKs
 - Timeouts based on estimate of RTT
- **Flow Control:** Ensures the sender does not overwhelm the receiver
- **Congestion Control:** Dynamic adaptation to network path's capacity

TCP Retransmission

Two Mechanisms for Retransmissions

- Duplicate ACKs
- Timeouts

Loss with Cumulative ACKs

- Sender sends packets with 100B and seqnos
 - 100, 200, 300, 400, 500, 600, 700, 800, 900
- Assume 5th packet (seqno 500) is lost, but no others
- Stream of ACKs will be
 - 200, 300, 400, 500, 500, 500, 500, 500

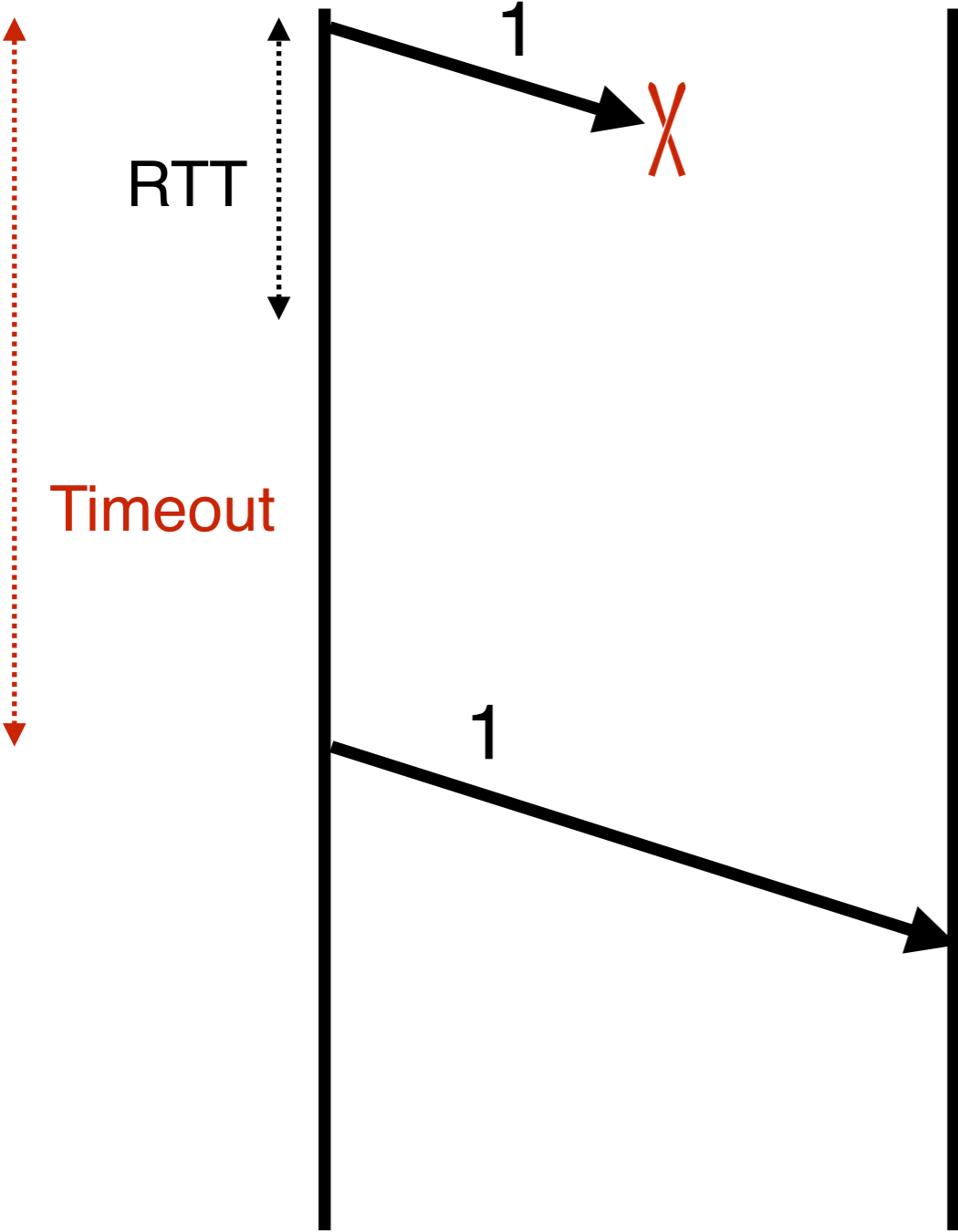
Loss with Cumulative ACKs

- Duplicate ACKs are a sign of an isolated loss
 - The lack of ACK progress means 500 hasn't been delivered
 - Stream of ACKs means some packets are being delivered
- Therefore, could trigger resend upon receiving k duplicate ACKs
 - TCP uses $k = 3$
- We will revisit this in congestion control

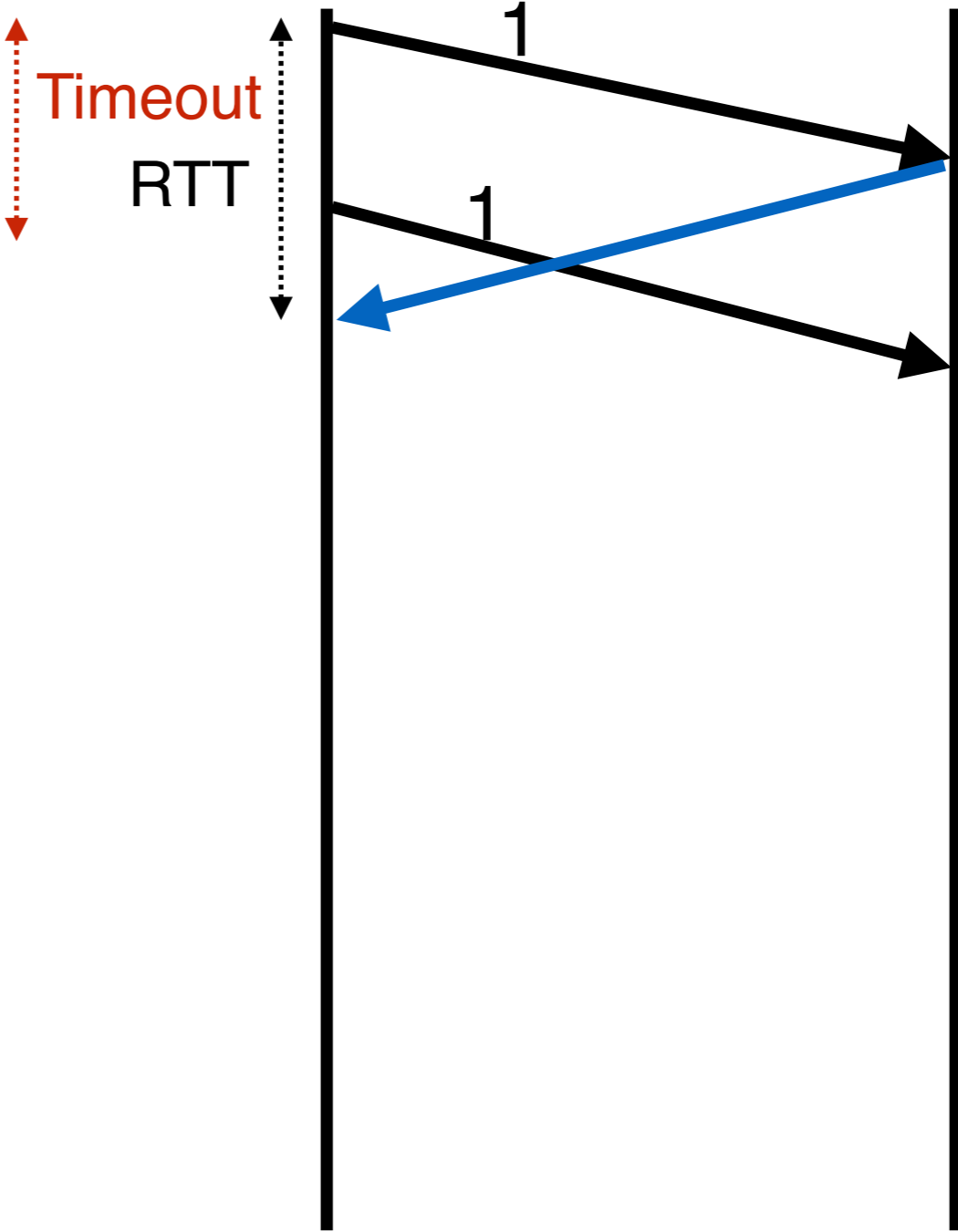
Timeouts and Retransmissions

- Reliability requires retransmitting lost data
- Involves setting timers and retransmitting on timeouts
- TCP only has a single timer
- TCP resets timer whenever new data is ACKed
- Retransmit packet containing “next byte” when timer expires
- RTO (Retransmit Time Out) is the basic timeout value

Setting the Timeout Value (RTO)



Timeout too long -> inefficient



Timeout too short -> duplicate packets

Setting RTO value

- Many ideas
- Implementations often use a coarser-grained timer
 - 500 msec is typical
- **Incurring a timeout is expensive**
- So we rely on duplicate ACKs

Basic Components of TCP

- **Connections:** Explicit set-up and tear-down of TCP sessions/connections
- **Segments, Sequence numbers, ACKs**
 - TCP uses byte sequence numbers to identify payloads
 - ACKs referred to sequence numbers
 - Window sizes expressed in terms of # of bytes
- **Retransmissions**
 - Can't be correct without retransmitting lost/corrupted data
 - TCP retransmits based on timeouts and duplicate ACKs
 - Timeouts based on estimate of RTT
- **Flow Control:** Ensures the sender does not overwhelm the receiver
- **Congestion Control:** Dynamic adaptation to network path's capacity

TCP Flow Control

Flow Control (Sliding Window)

- Advertised Window: W
 - Can send W bytes beyond the next expected byte
- Receiver uses W to prevent sender from overflowing buffer
- Limits number of bytes sender can have in flight

Advertised Window Limits Rate

- Sender can send no faster than W/RTT bytes/sec
- In original TCP, that was the sole protocol mechanism controlling sender's rate

Implementing Sliding Window

- Sender maintains a window
 - Data that has been sent out but not yet ACK'ed
- Left edge of window:
 - Beginning of unacknowledged data
 - Moves when data is ACKed
- Window size = maximum amount of data in flight
- Receiver sets this amount, based on its available buffer space
 - If it has not yet sent data up to the app, this might be small

