# CS 352
# Flow Control; Congestion Control

Lecture 15

http://www.cs.rutgers.edu/~sn624/352-F22

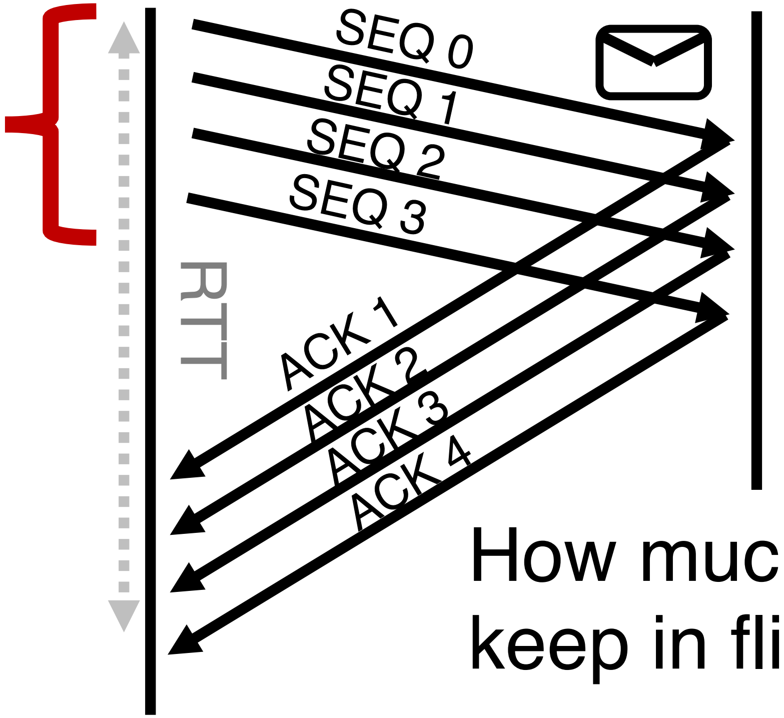
Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

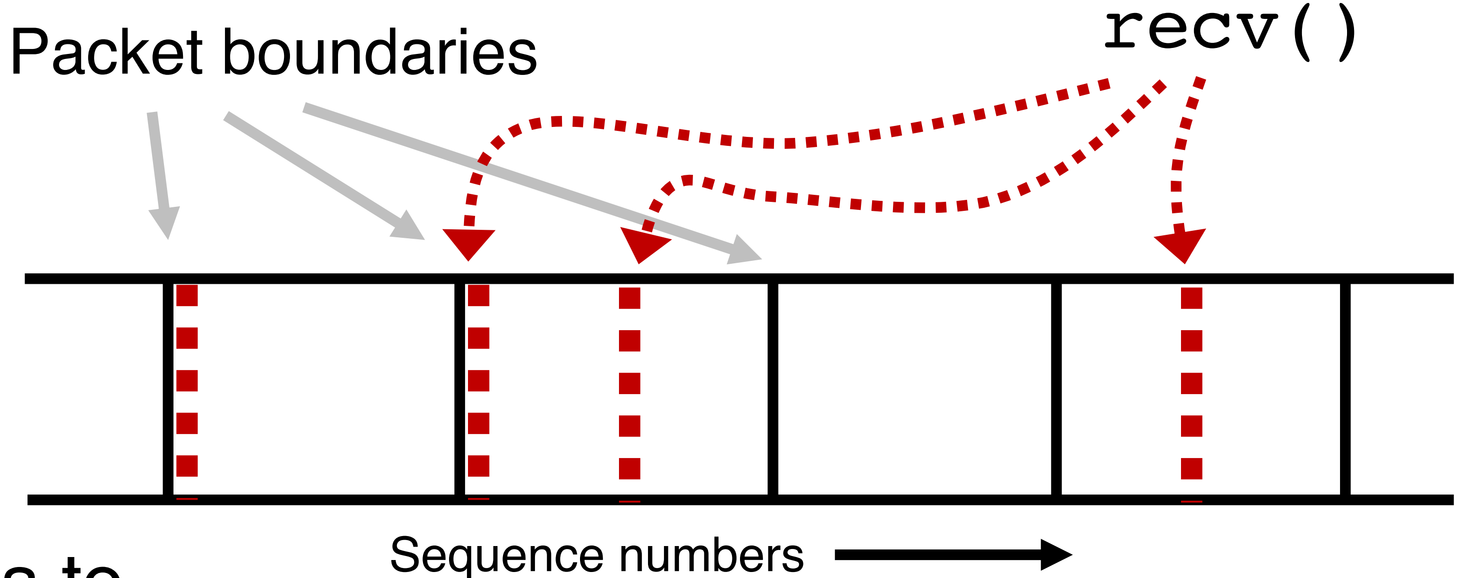
# Quick recap of concepts


TCP:
Reliability
Ordering

Reordering degrades connection throughput. Apps can't `recv()`. Packets may even be dropped due to insufficient buffering.




How much data to keep in flight?



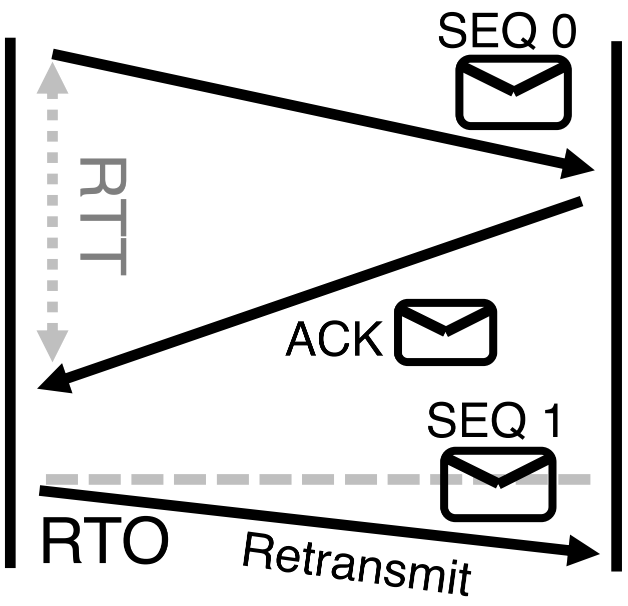

Stream-Oriented Transport

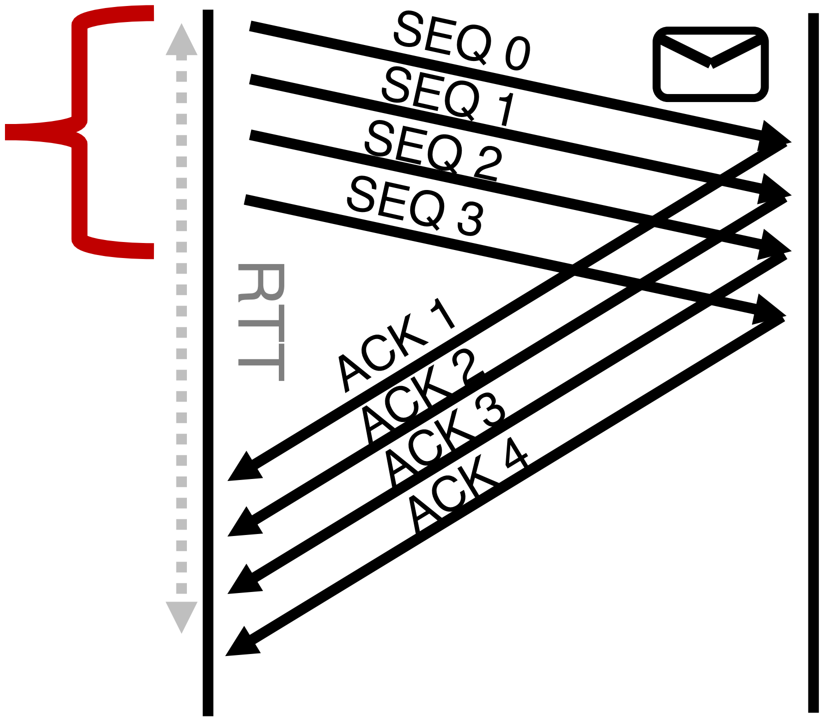
= window size

Proportional to throughput

# How much data to keep in flight?

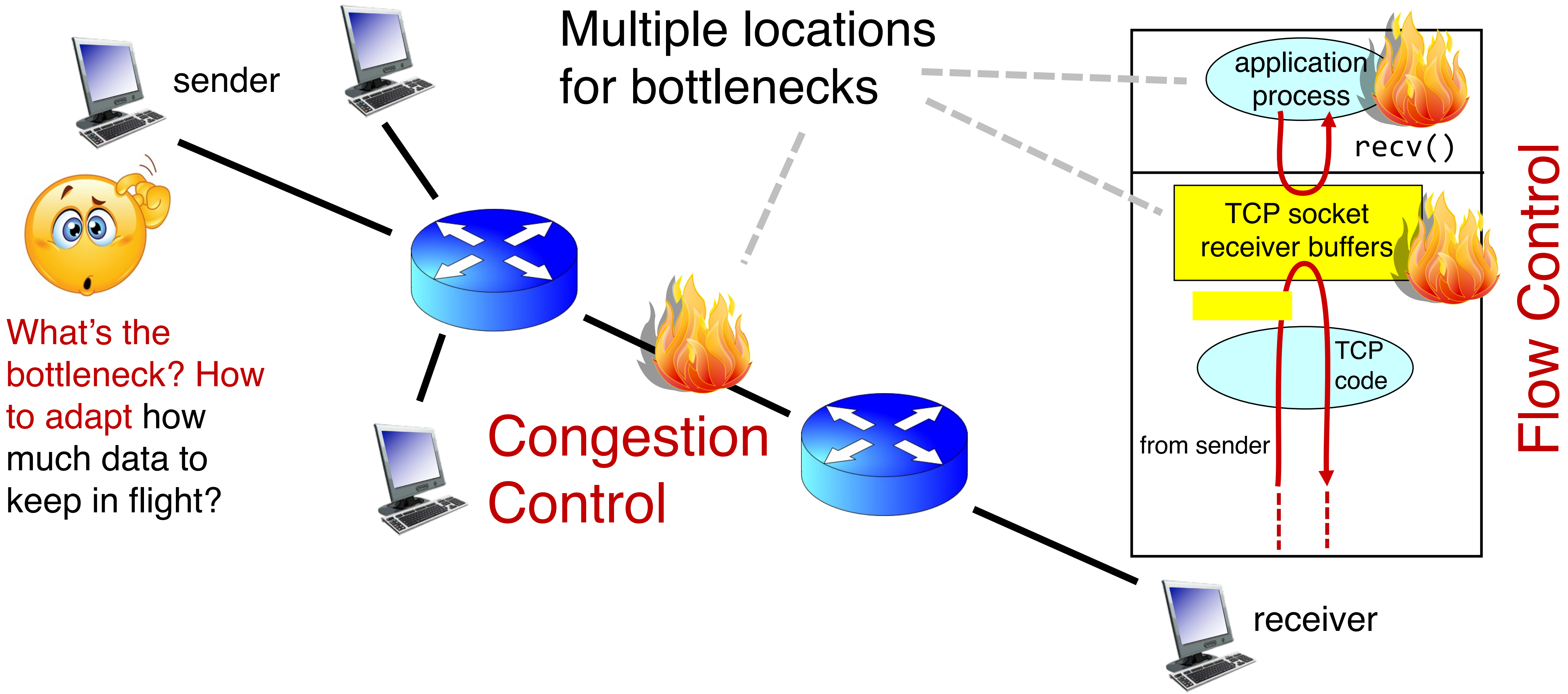
Stop and Wait




Pipelined Reliability

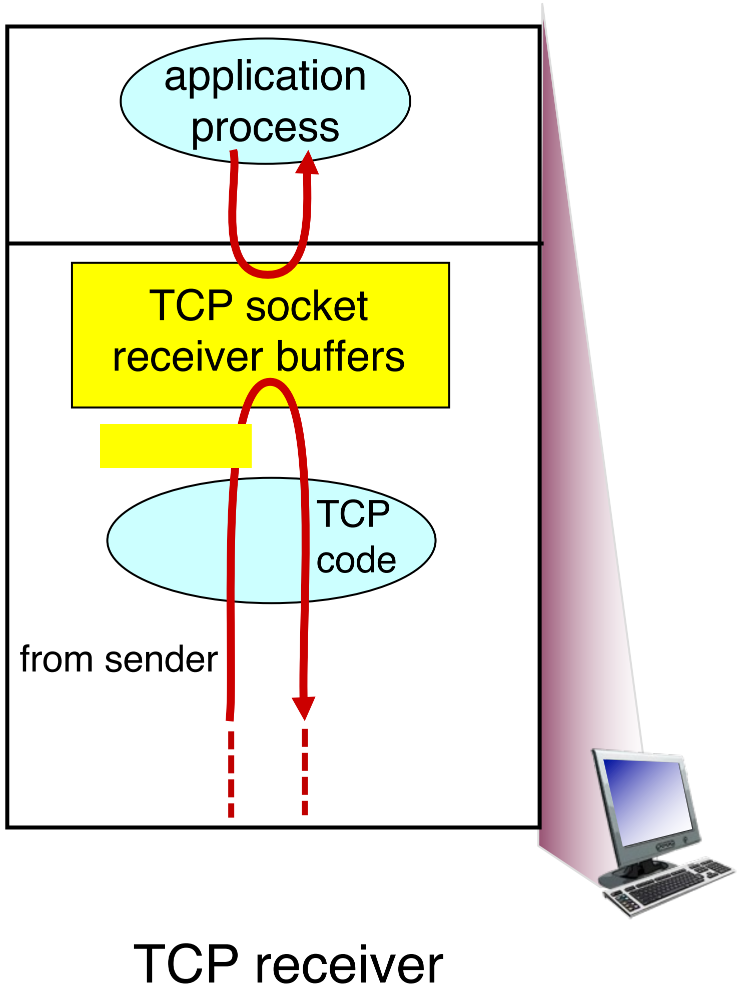
# We want to increase throughput, but …



# Flow Control

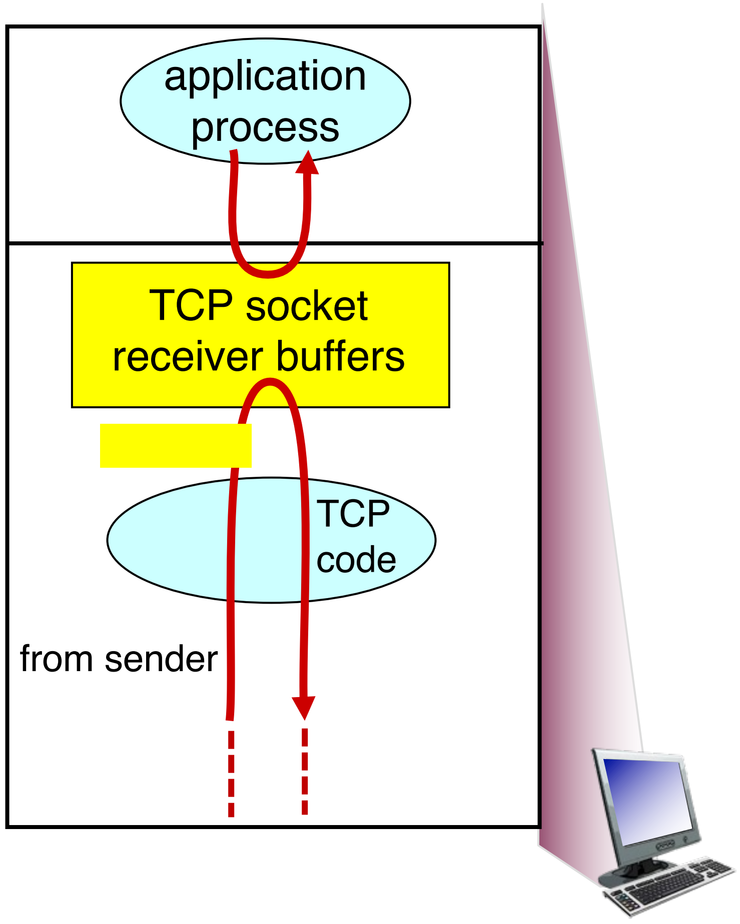
# Socket buffers can become full

- Applications may read data slower than the sender is pushing data in
  - Example: what if an app infrequently or never calls `recv()`?
- There may be too much reordering or packet loss in the network
  - What if the first few bytes of a window are lost or delayed?
- Receivers can only buffer so much before dropping subsequent data



# Goal: avoid drops due to buffer fill

- Have a TCP sender only send as much as the free buffer space available at the receiver.
- *Amount of free buffer varies over time!*
- TCP implements flow control
- Receiver's ACK contains the amount of data the sender can transmit without running out the receiver's socket buffer
- This number is called the advertised window size




TCP receiver

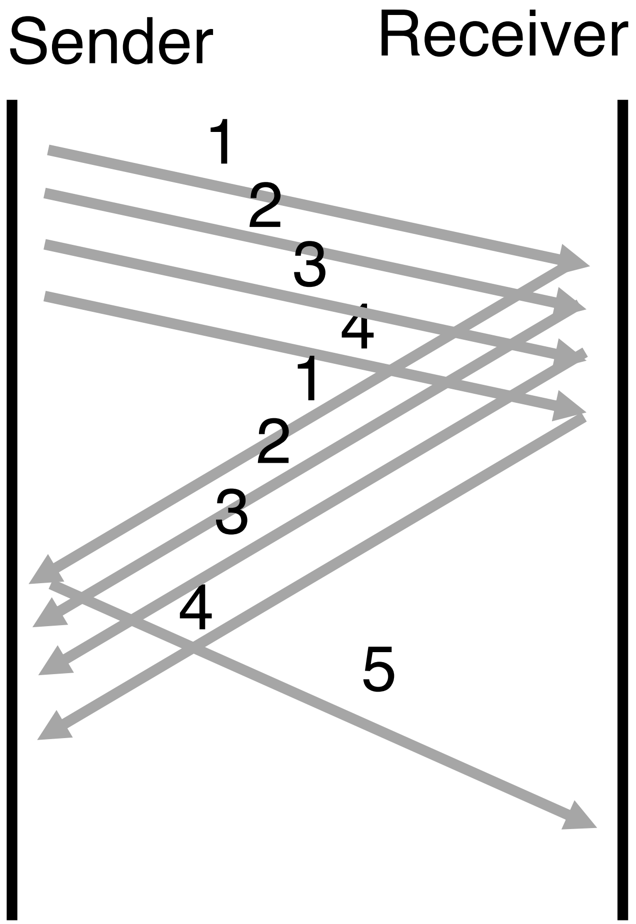
# Flow control in TCP headers

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

                        TCP Header Format

          Note that one tick mark represents one bit position.
```

# TCP flow control

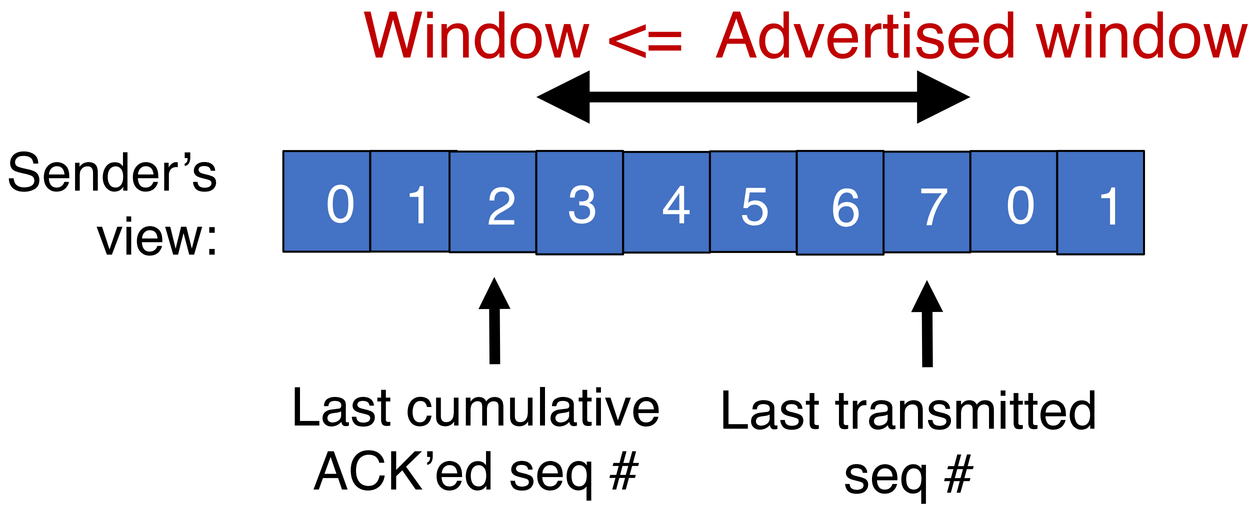
- Receiver advertises to sender (in the ACK) how much free buffer is available

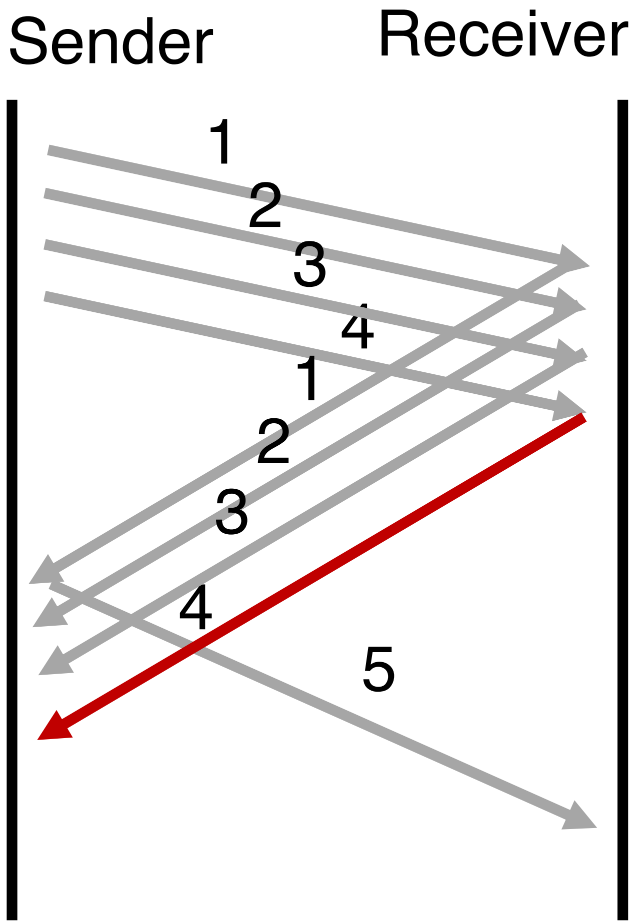
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

                       TCP Header Format

         Note that one tick mark represents one bit position.
```



# TCP flow control

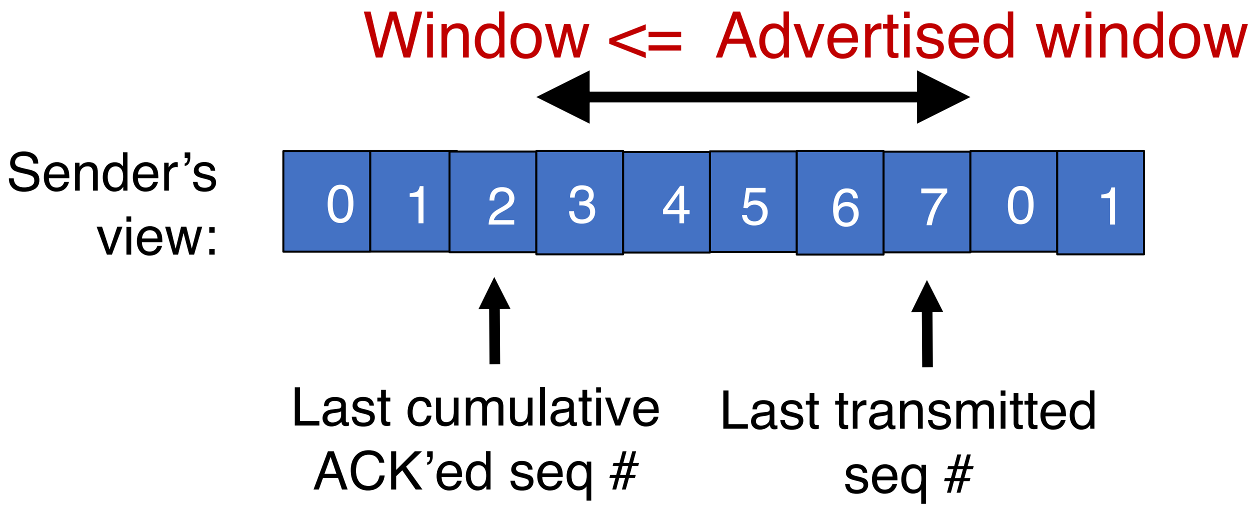
- Subsequently, the sender's sliding window cannot be larger than this value
- Restriction on new sequence numbers that can be transmitted
- == restriction on sending rate!

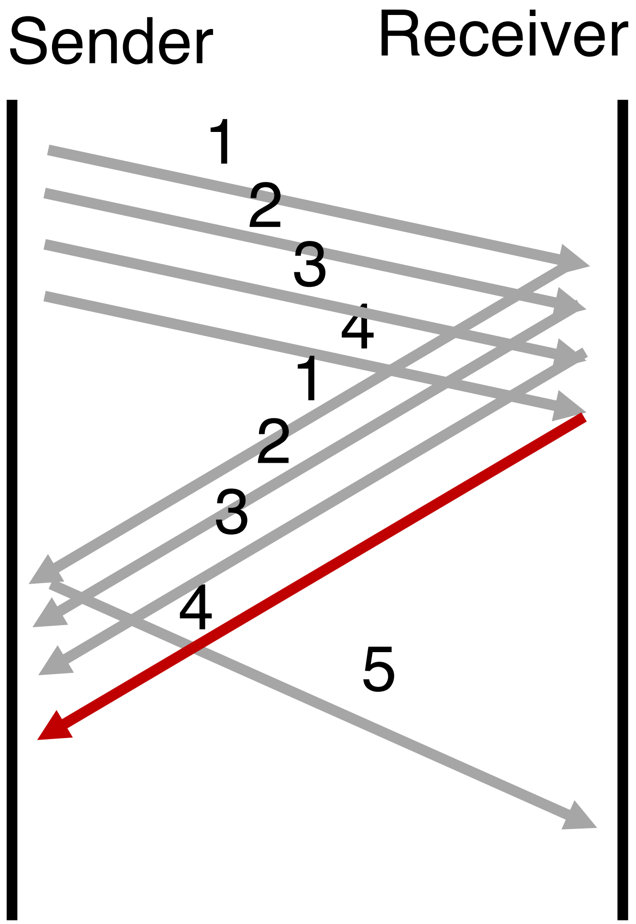


# TCP flow control

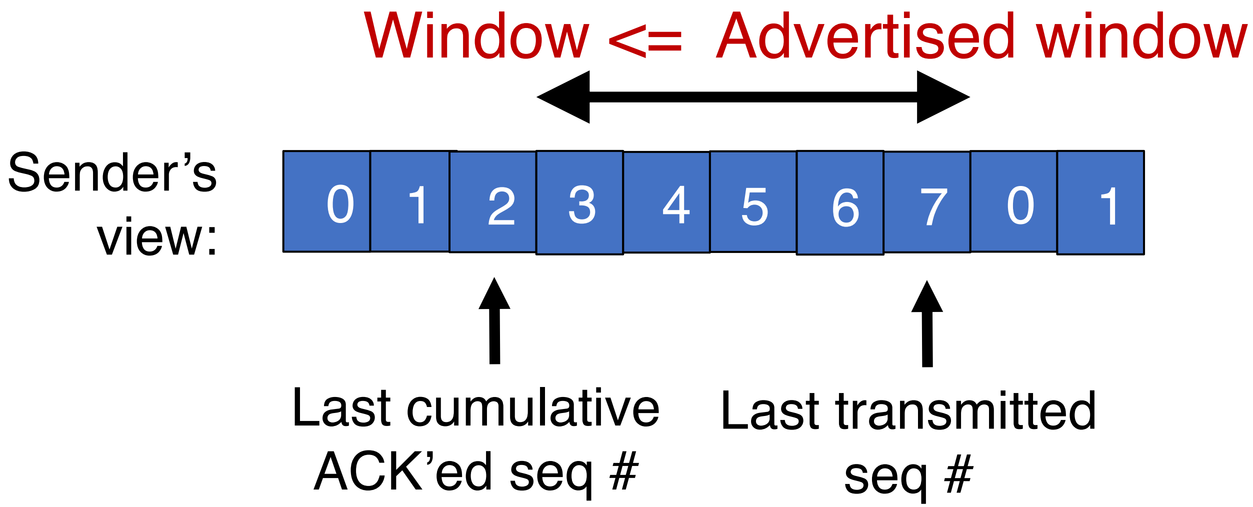
- If receiver app is too slow reading data:
  - receiver socket buffer fills up
  - So, advertised window shrinks
  - So, sender's window shrinks
  - So, sender's sending rate reduces

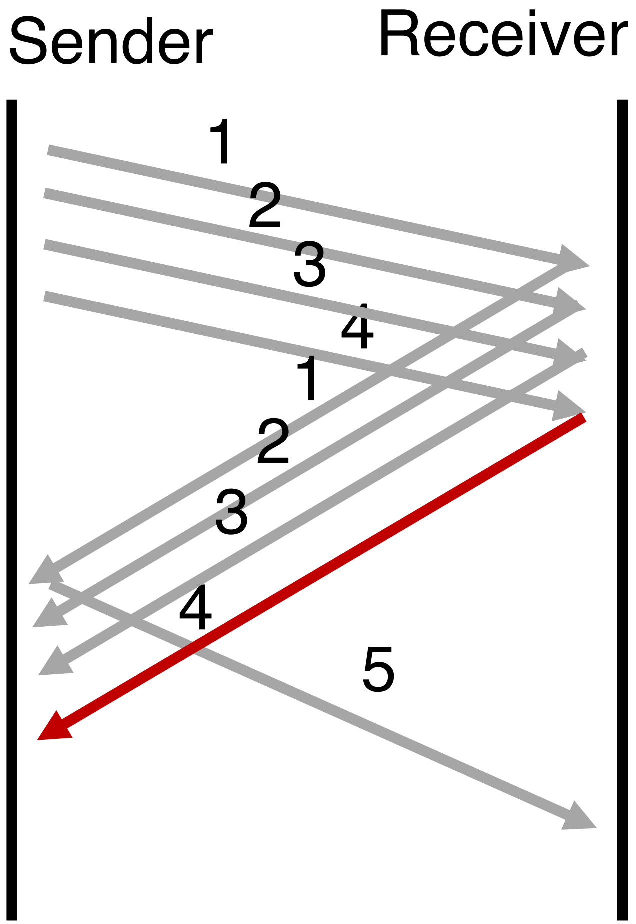


# TCP flow control

Flow control matches the sender's write speed to the receiver's read speed.



# Sizing the receiver’s socket buffer

- Operating systems have a default receiver socket buffer size
  - Listed among `sysctl -a | grep net.inet.tcp` on MAC
  - Listed among `sysctl -a | grep net.ipv4.tcp` on Linux

- If socket buffer is too small, sender can’t keep too many packets in flight ➔ lower throughput

- If socket buffer is too large, too much memory consumed per socket

- How big should the receiver socket buffer be?

# Q: how large a receiver socket buffer?

- Case 1: Suppose the receiving app is reading data too slowly:
  - no amount of receiver buffer can prevent low sender throughput if the connection is long-lived!

# Q: how large a receiver socket buffer?

- Case 2: Suppose the receiving app reads sufficiently fast *on average* to match the sender's writing speed.
  - Assume the sender has a window of size W.
  - The receiver must use a buffer of size at least W. Why?

- Captures two cases:
- (1) When the first sequence #s in the window are dropped
  - *Selective repeat*: data in window buffered until the ACKs of delivered data (within window) reach sender. Adv. win reduces sender's window
- (2) When the sender sends a burst of data of size W
  - Receiver may not match the *instantaneous* rate of the sender

# Summary of flow control

- Keep memory buffers available at the receiver whenever the sender transmits data
- Buffers needed to hold for selective repeat and reassemble data in order
- Inform the sender on an on-going basis (each ACK)
- Function: match sender speed to receiver speed

- Correct socket buffer sizing is important for TCP throughput

# Info on (tuning) TCP stack parameters

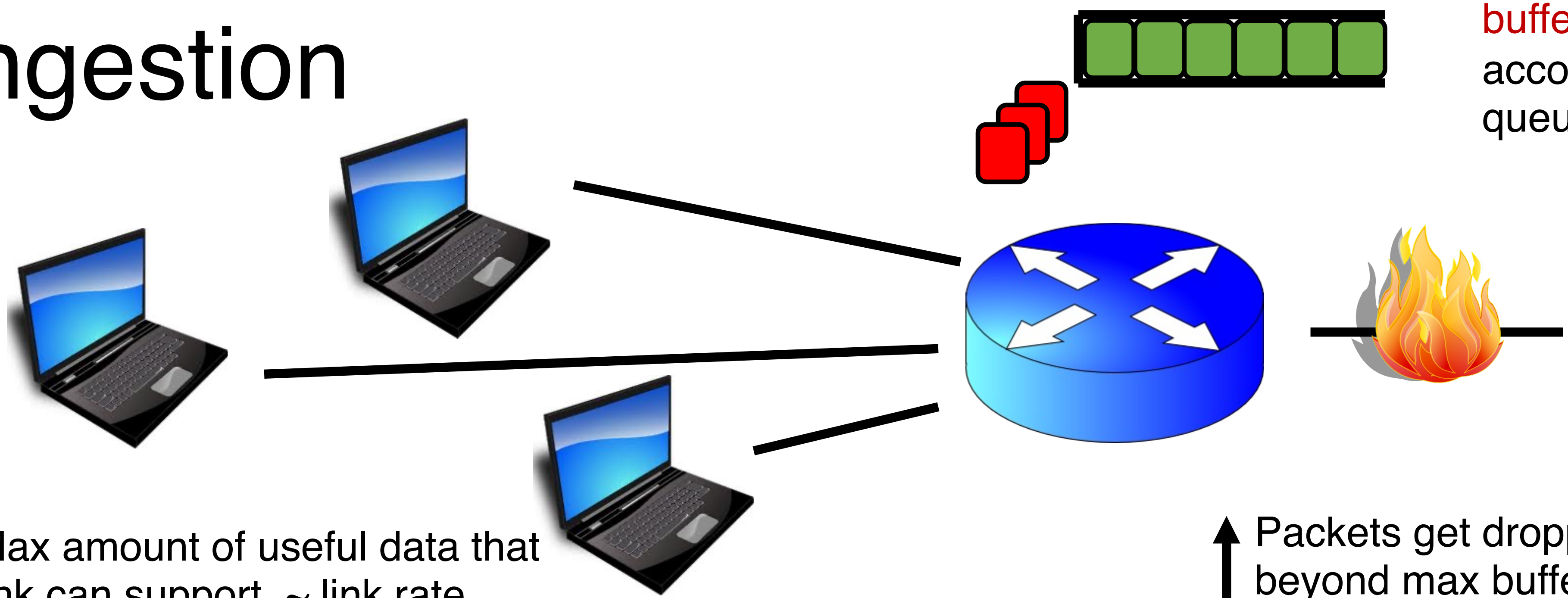
- https://www.ibm.com/support/knowledgecenter/linuxonibm/liaag/wkvm/wkvm_c_tune_tcpip.htm

- https://cloud.google.com/solutions/tcp-optimization-for-network-performance-in-gcp-and-hybrid

# Congestion Control

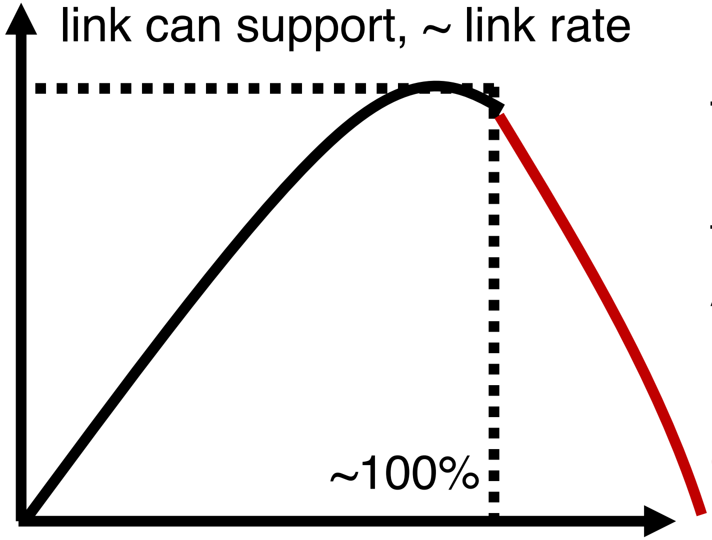
# Congestion

Routers have buffers which accommodate queued packets.



Max amount of useful data that link can support, ~ link rate

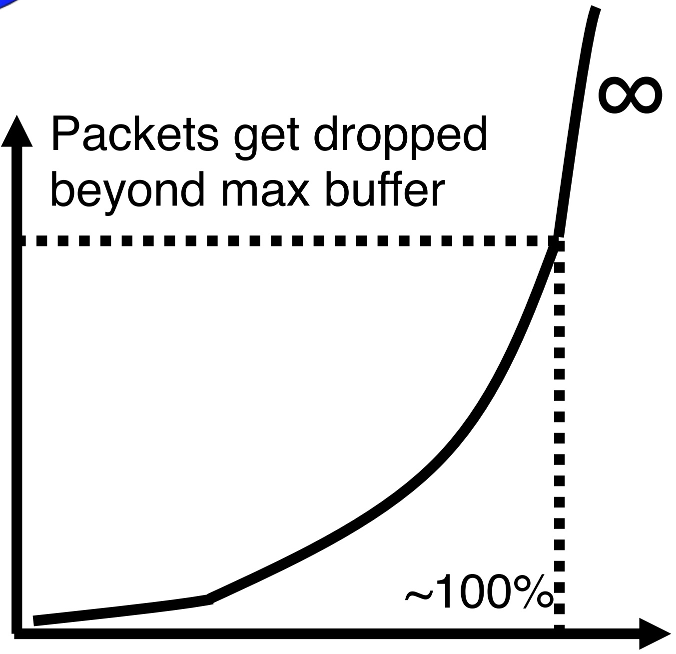
Amount of useful data that gets across to the receiver




Fraction of link used (link load)

Too many retransmissions due to packet drops! Amount of useful (fresh) data plummets.
Congestion collapse

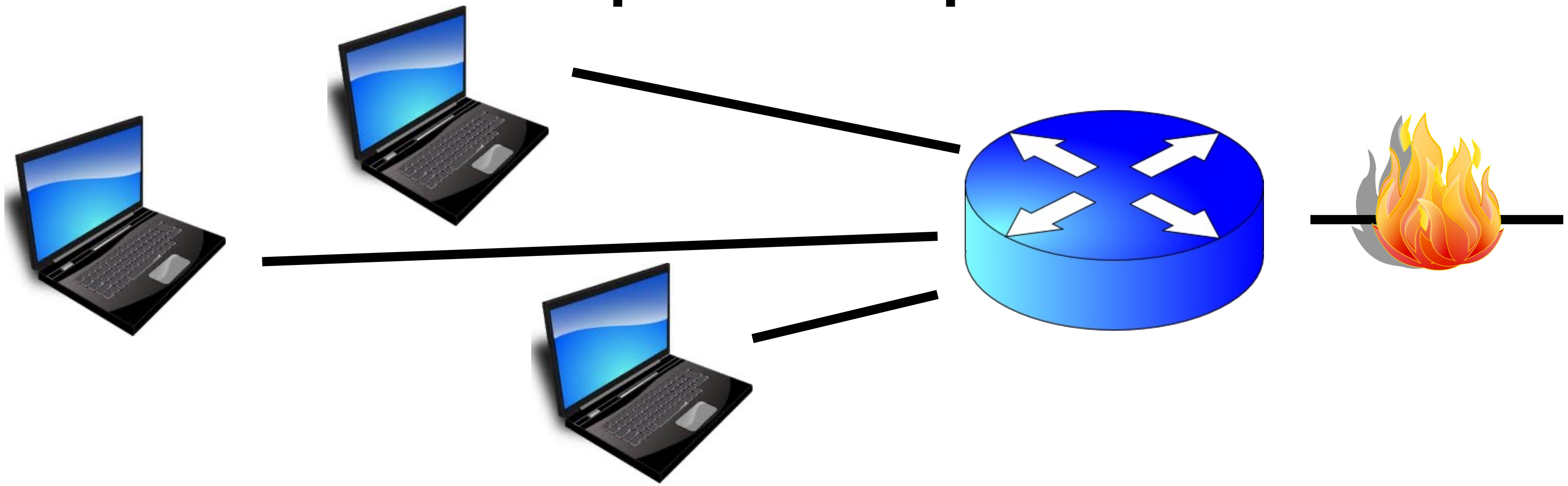
Queueing delay




Link load

https://en.wikipedia.org/wiki/Network_congestion#Congestive_collapse

# How should multiple endpoints share net?



- It is difficult to know where the bottleneck link is
- It is difficult to know how many other endpoints are using that link
- Endpoints may join and leave at any time
- Network paths may change over time, leading to different bottleneck links (with different link rates) over time

The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.

The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.

No one can centrally view or control all the endpoints and bottlenecks in the Internet.

Every endpoint must try to reach a globally good outcome by itself: i.e., in a distributed fashion.

This also puts a lot of trust in endpoints.

The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.

If there is spare capacity in the bottleneck link, the endpoints should use it.

The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.

If there are N endpoints sharing a bottleneck link, they should be able to get equitable shares of the link's capacity.

For example: 1/N'th of the link capacity.

# Flow Control vs. Congestion Control

**Flow Control**

- Avoid overwhelming the receiving application
- Sender is managing the receiver's socket buffer

**Congestion Control**

- Avoid overwhelming the bottleneck network link
- Sender is managing the bottleneck link capacity and bottleneck router buffers

The approach that the Internet takes is to use a distributed algorithm to converge to an efficient and fair outcome.


How to achieve this?

Approach: sense and react

Example: taking a shower

Use a feedback loop with signals and knobs

# Signals and Knobs in Congestion Control

- Signals
  - Packets being ACK’ed
  - Packets being dropped (e.g. RTO fires)
  - Packets being delayed (RTT)
  - Rate of incoming ACKs

Implicit feedback signals measured directly at sender. (There are also explicit signals that the network might provide.)

- Knobs
  - What can you change to “probe” the available bottleneck capacity?
  - Suppose receiver buffer is unbounded:
  - Increase window/sending rate: e.g., add x or multiply by a factor of x
  - Decrease window/sending rate: e.g., subtract x or reduce by a factor of x

# Sense and react, sure…but how?

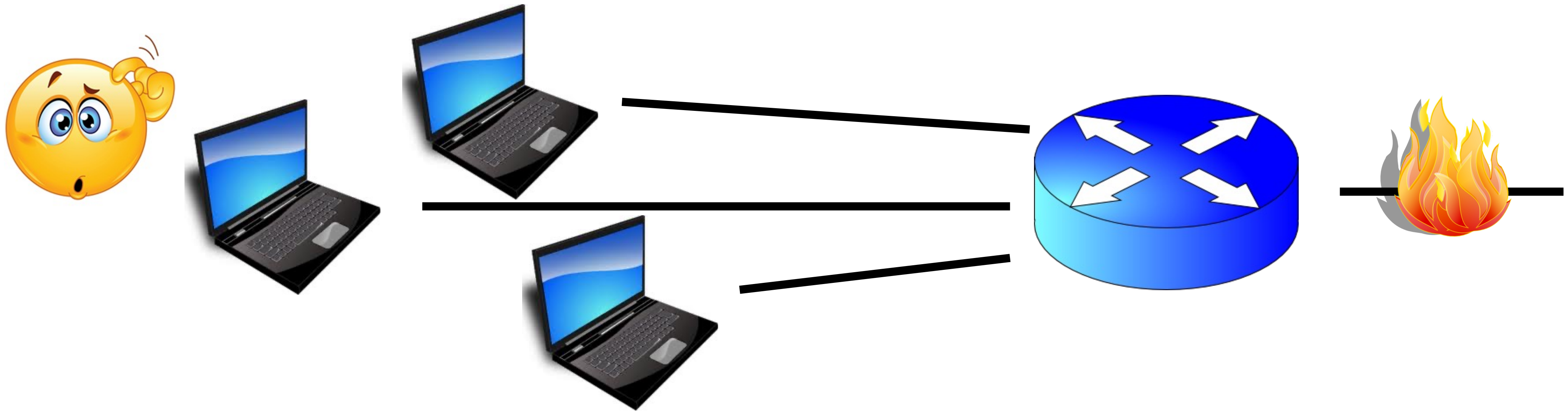
- Where do you want to be?
  - The steady state
- How do you get there?
  - Congestion control algorithms
- Sense accurately
- React proportionately

# The Steady State

## Efficiency of a single TCP conversation

# What does efficiency look like?

- Suppose we want to achieve an efficient outcome for one TCP conversation by observing network signals from the endpoint



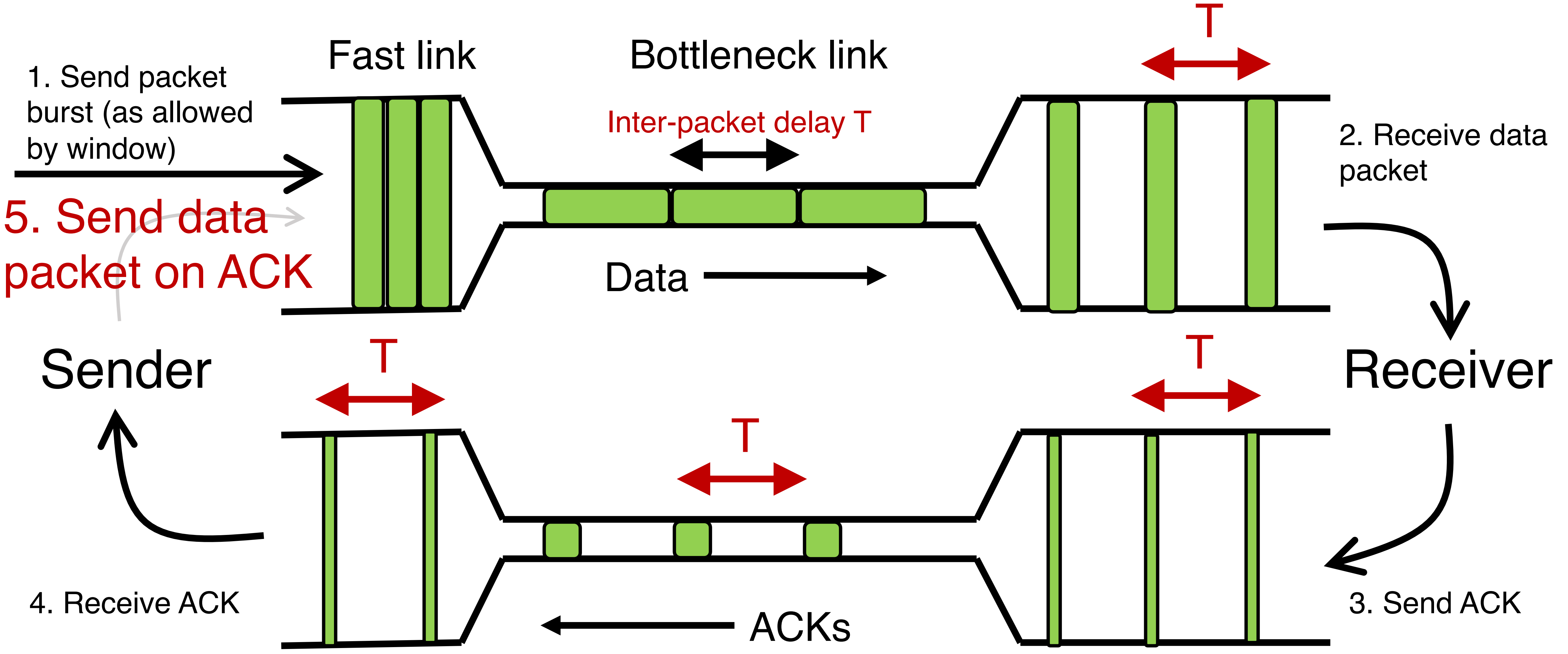
- Q: How should the endpoint behave at steady state?
- Challenge: bottleneck link is remotely located

# Steady state: Ideal goal

- High sending rate: Use the full capacity of the bottleneck link
- Low delay: Minimize the overall delay of packets to get to the receiver
  - Overall delay = propagation + queueing + transmission
  - Assume propagation and transmission components fixed
- “Low delay” reduces to low queueing delay
- i.e., don’t push so much data into the network that packets have to wait in queues

- Key question: When to send the next packet?

# When to send the next packet?



# Rationale

- When the sender receives an ACK, that’s a signal that the previous packet has left the bottleneck link (and the rest of the network)

- Hence, it must be safe to send another packet without congesting the bottleneck link

- Such transmissions are said to follow packet conservation

- ACK clocking: “Clock” of ACKs governs packet transmissions

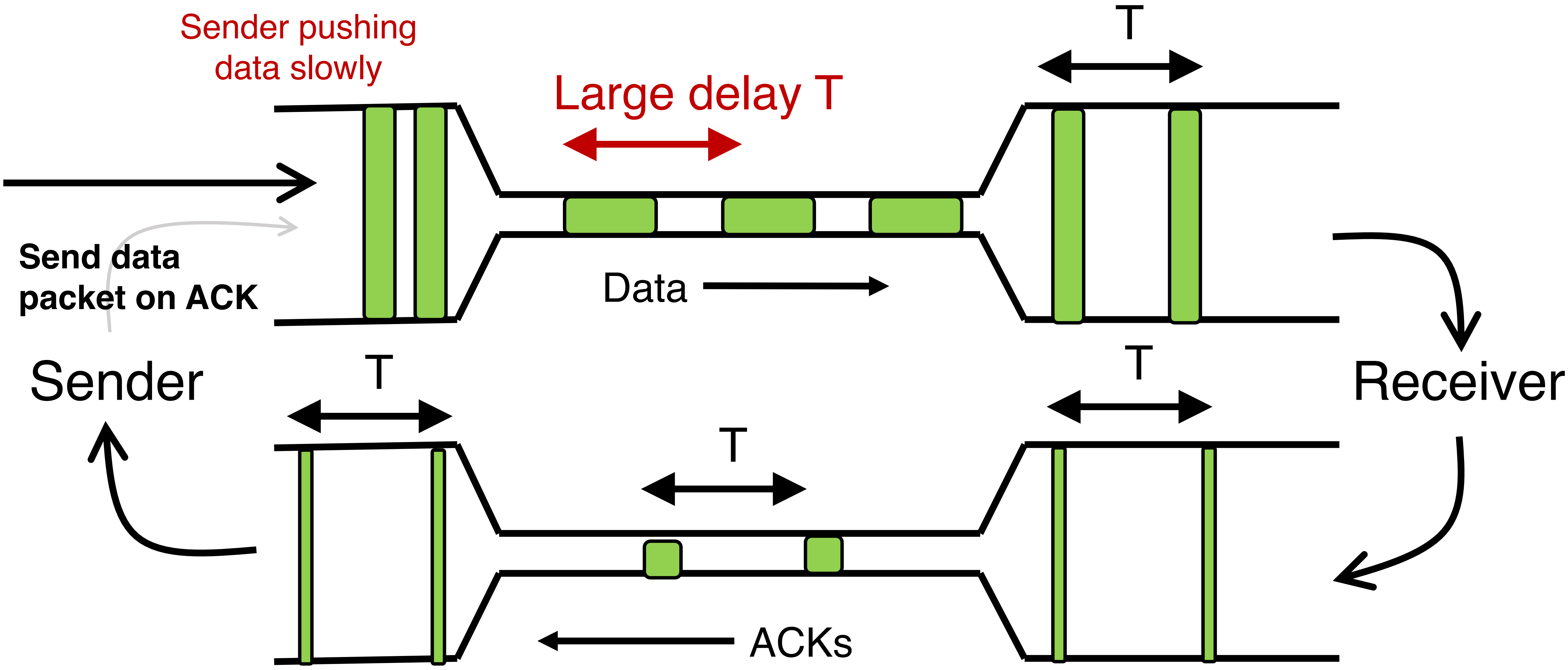
# ACK clocking: analogy

- How to avoid crowding a grocery store?

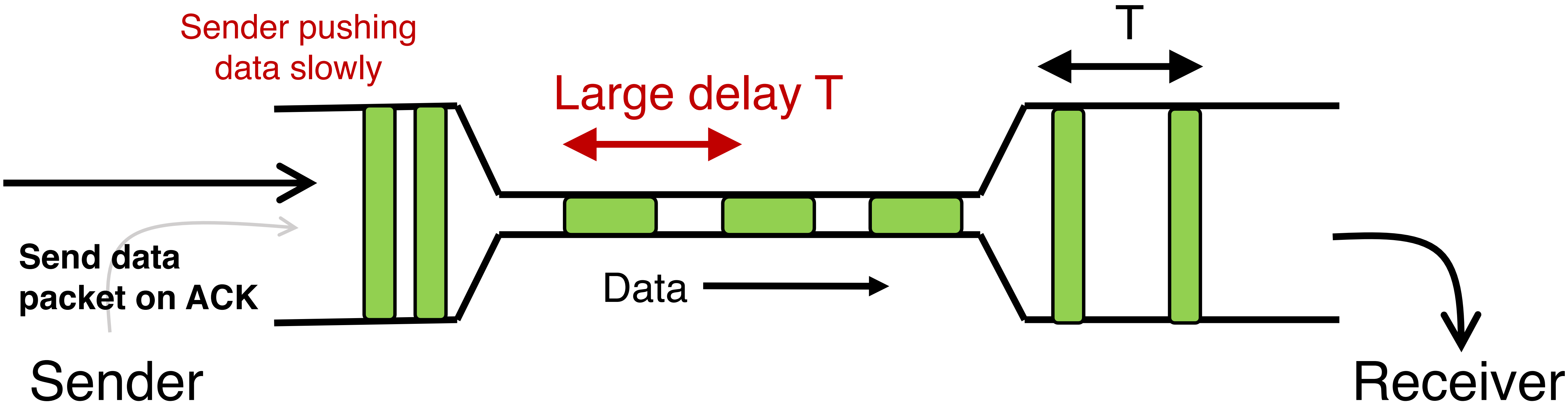
- Strategy: Send the next waiting customer exactly when a customer exits the store

- However, this strategy alone can lead to inefficient use of resources…

# ACK clocking alone can be inefficient



# ACK clocking alone can be inefficient




The sending rate should be high enough to keep the “pipe” full

Analogy: a grocery store with only 1 customer in entire store

If the store isn’t “full”, you’re using store space inefficiently

# Steady State of Congestion Control

- Send at the highest rate possible (to keep the pipe full)
- while being ACK-clocked (to avoid congesting the pipe)

- So, how to get to steady state?