TCP: Reliable Data Transfer

Automatic Repeat reQuest (ARQ):

- **acknowledgement** of received data
- **timeouts** for expected acknowledgements
- **retransmission** of data apparently not received

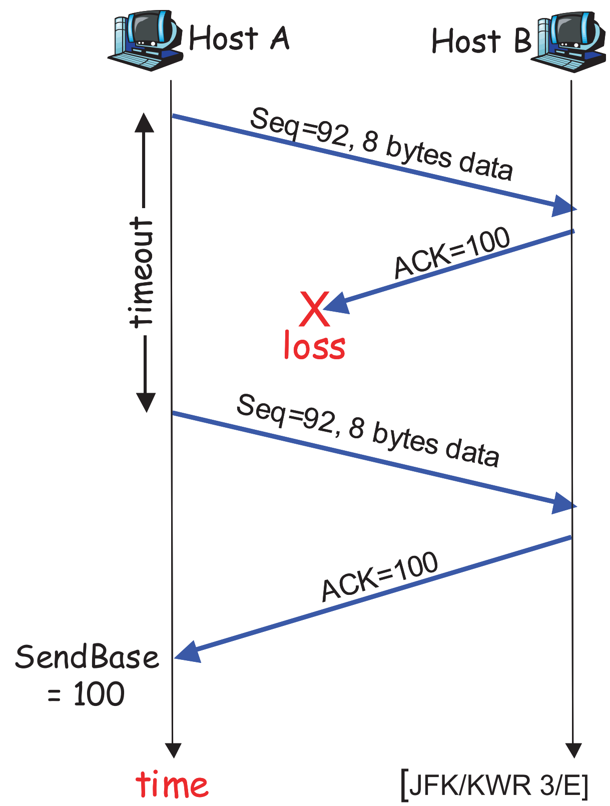
Hybrid of **Go-Back-N ARQ** and **Selective Repeat ARQ**

Food for Thought

Why might a TCP sender **fail to receive**, within a timeout interval, an **acknowledgement** for data transmitted?

To use network resources **efficiently**, should TCP **behave differently** depending upon **reason** for **failure** to receive an acknowledgement and why?

- A Lost ACK scenario



TCP: Efficient Use of Network Resources

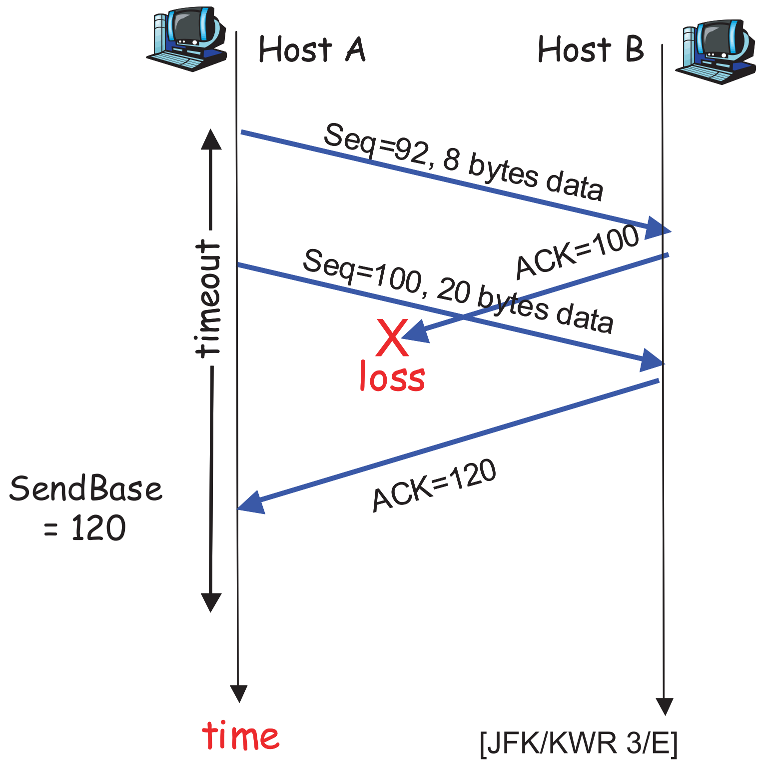
Reduce number and length of **idle intervals** by:

- **pipelining** of transmissions using **sliding window** of transmitted but unacknowledged data
- **fast retransmit** forced by receipt of **three duplicate acknowledgements**

Reduce number of **distinct acknowledgements** by:

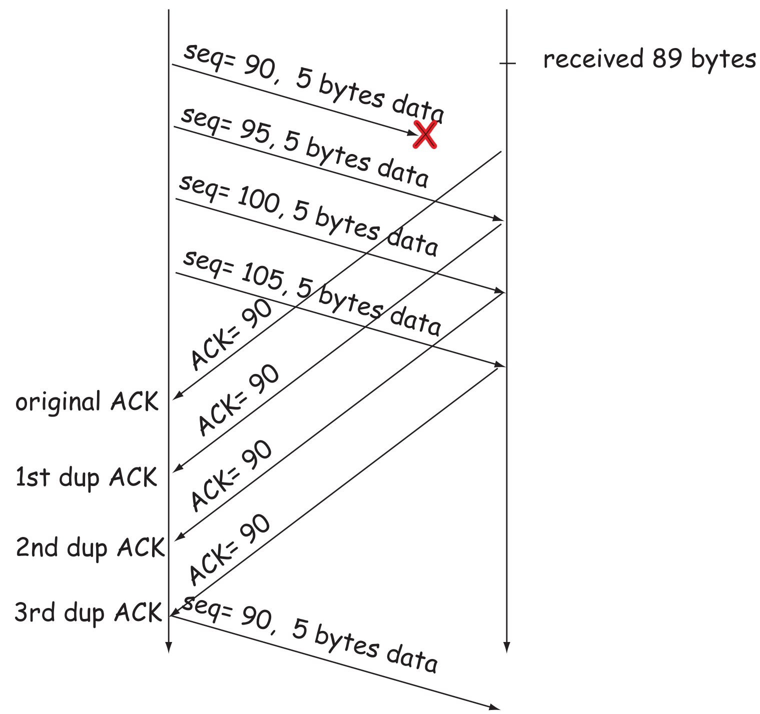
- **piggybacking** acknowledgements on reverse data packets
- **cumulative** acknowledgement
- **delayed** acknowledgement

A Cumulative ACK Scenario



Fast Retransmit:

- time-out period is often relatively long
- a good idea is then to detect lost segments via duplicate ACKs (here we wait for 3 dup. ACKs) and then retransmit an unACK'ed segment



TCP: Efficient Use of Network Resources

Reduce number of **unnecessary retransmissions** by:

- **buffering out-of-order data** at receiver
- **doubling timeout** interval
- **cumulative** acknowledgement
- **selective** acknowledgement

- Delayed ACKs : ACK Generation Recommendation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send duplicate ACK, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

[JFK/KWR 3/E]

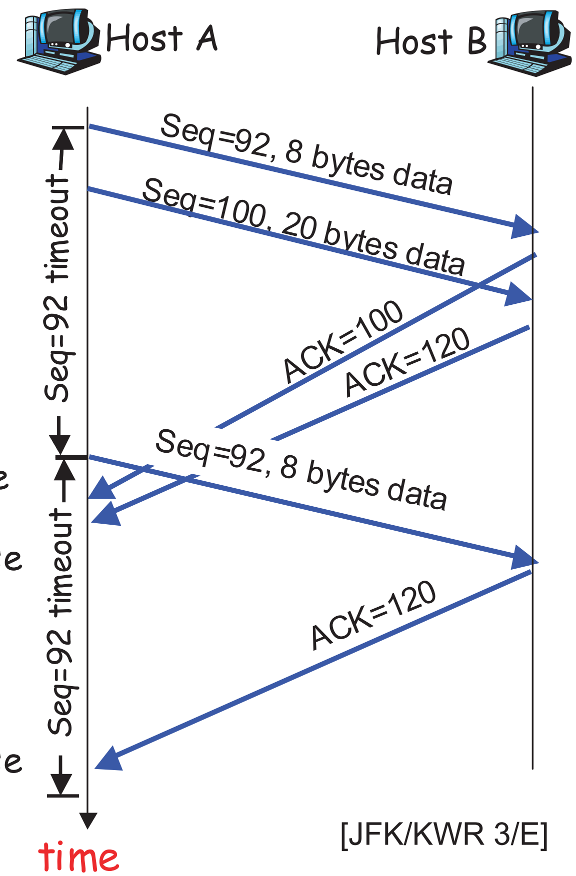
Length of TCP Timeout Interval

What are advantages and disadvantages of a **short timeout interval**?

What are advantages and disadvantages of a **long timeout interval**?

How should **timeout interval** be set for **efficient** use of network resources?

A Premature Timeout Scenario



Note: the second segment will not be retransmitted.

TCP: Determining the Timeout Interval

Estimate* **round-trip time**:

$$\begin{aligned} EstimatedRTT(t+1) = \; & EstimatedRTT(t) + \\ & \alpha(SampleRTT(t+1) - EstimatedRTT(t)) \end{aligned}$$

Estimate* **deviation** in round-trip time:

$$\begin{aligned} DevRTT(t+1) = \; & DevRTT(t) + \\ & \beta((SampleRTT(t+1) - EstimatedRTT(t)) - DevRTT(t)) \end{aligned}$$

Compute **timeout interval**:

$$TimeoutInterval(t) = EstimatedRTT(t) + 4DevRTT(t)$$

*Estimates are derived as **exponential, recency-weighted averages** with $0 < \alpha, \beta \leq 1$

Hypothetical TCP

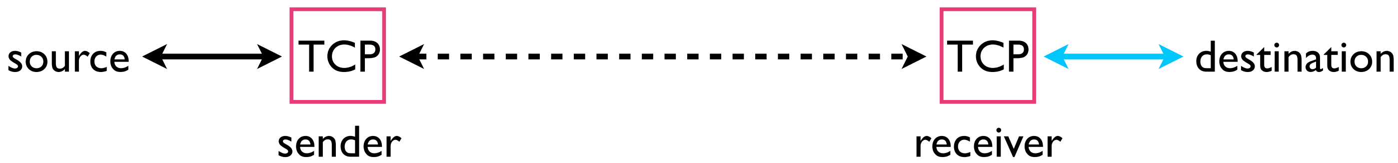
Suppose at a TCP sender a **single window** of **multiple segments** were **replaced** with a set of **parallel stop-and-wait** single-segment **windows**.

What are **implications**?

Are there **advantages** to this approach?

Are there **disadvantages** to this approach?

TCP: Flow Control

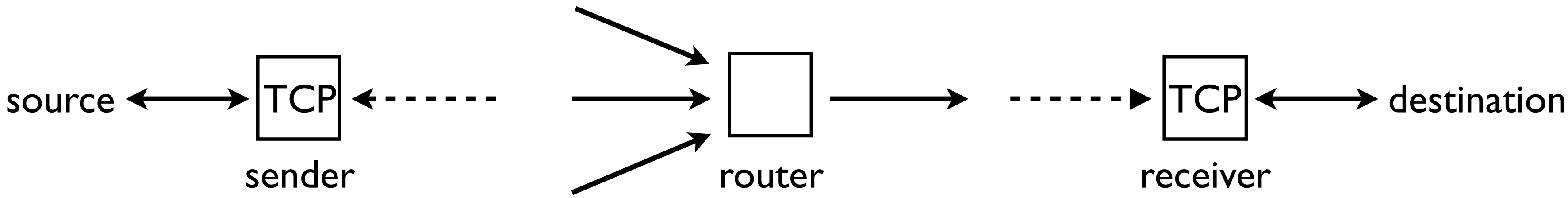


receive window at receiver **controls** amount of **data queued** awaiting delivery to **destination** application

receiver tells sender current amount of **room** in receive **window**, and hence receive window at sender also **controls** amount of **data transmitted** to destination and awaiting acknowledgement

when receive **window** is **full**, **sender probes receiver** with one-byte data messages to **determine** when **room** becomes **available** in receive window

Congestion

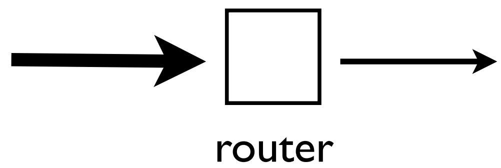


occurs when **demand exceeds capacity** of resource

queues form behind saturated resource

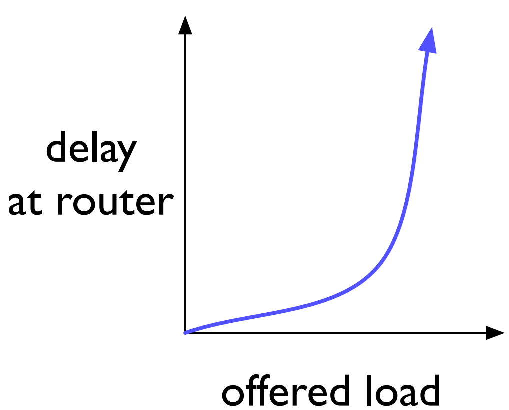
What happens to **delay** and **throughput** during congestion?

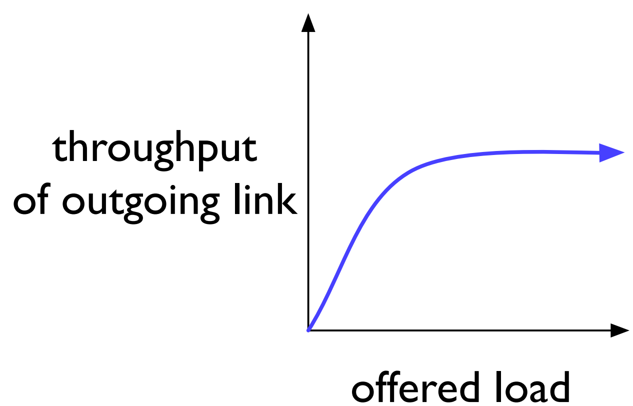
Case 1: No Retransmission, Infinite Buffering



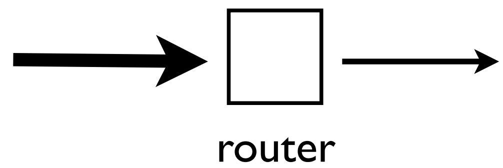
no data are lost

all data transmitted on outgoing link are new



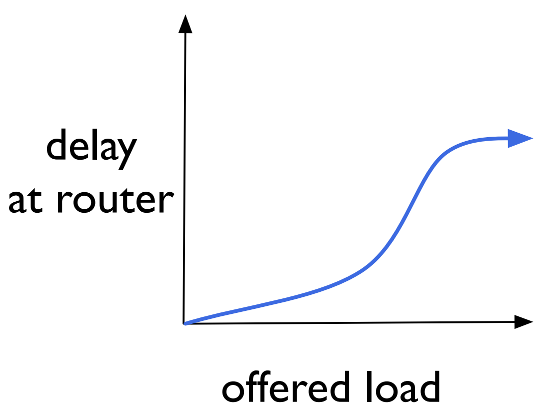


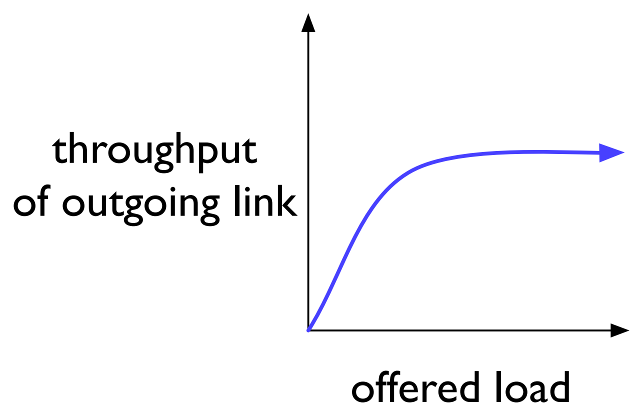
Case 2: No Retransmission, Finite Buffering



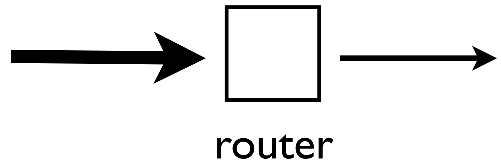
some data are lost

all data transmitted on outgoing link are new





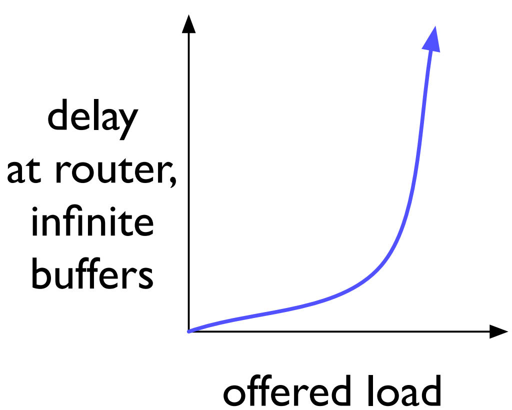
Case 3: Retransmission on Timeout

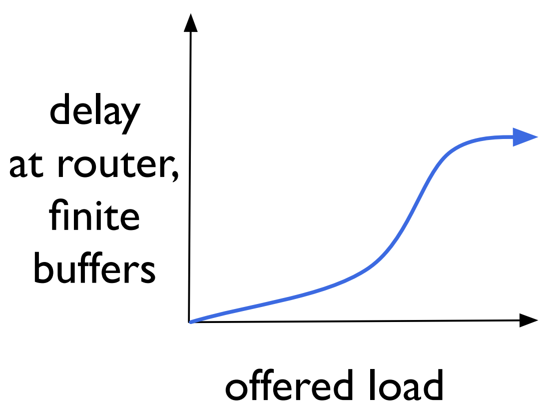


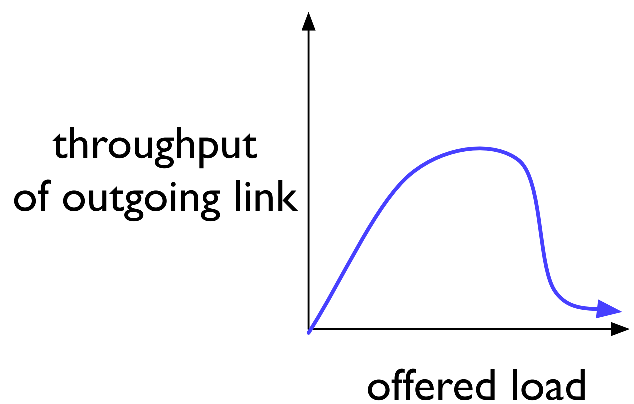
no data are **lost** if **buffers** are **infinite**

some data are **lost** if **buffers** are **finite**

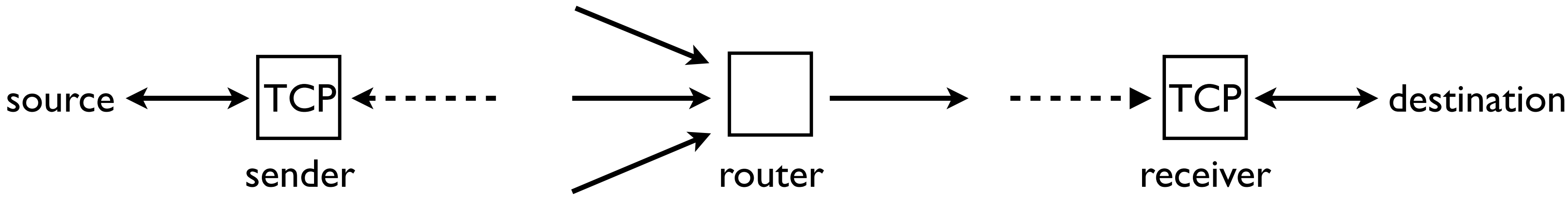
some data transmitted on outgoing link are **retransmissions** of data already sent







Congestion Control



Where and how to **detect** when **congestion** is occurring or likely to occur?

Where and how to **restrict offered load** to congested resource?

Congestion control in the **network**, at **endpoints**, or a **combination** of both?

TCP: Congestion Control

congestion is **inferred** based on **failure** to **receive acknowledgement** for transmitted data

congestion window at sender **controls** amount of **data transmitted** to destination and awaiting acknowledgement

TCP: Slow Start Phase

if **no acknowledgement** received by **timeout**:
set **threshold** to **one-half** of **congestion window** size
and **reduce** congestion **window** size to **1 segment**

increase congestion **window** size by **1 segment** for every **acknowledgement** received up until threshold

exponential increase in size of congestion window:
window size **doubles** every round-trip time

TCP: Congestion Avoidance Phase

if **threshold attained**:
  increase congestion **window** size by **segment-size / congestion-window-size** fraction of a segment for every **acknowledgement** received

additive increase in size of **congestion window**:
  window increases linearly by **1 segment** every round-trip time

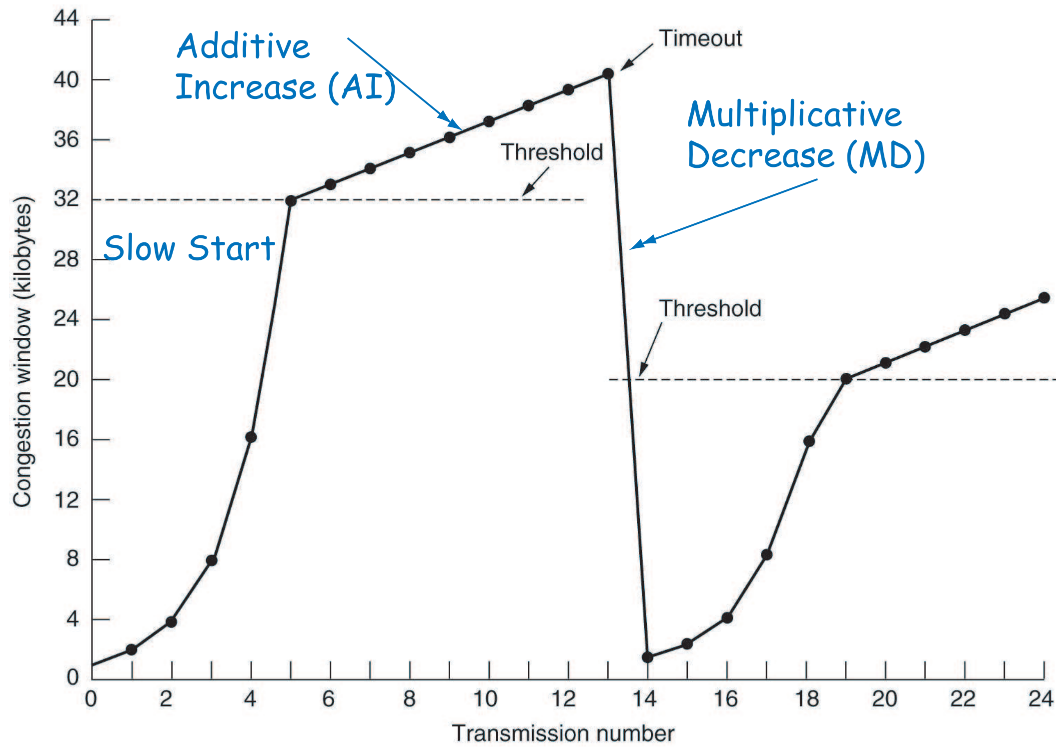
TCP: Fast Recovery Phase

if **three duplicate acknowledgements** received:
set **threshold** to **one-half** of **congestion window** size and **reduce** congestion **window** size by **half**

multiplicative decrease in size of congestion window:
window size **halved**

increase congestion **window** size by **1 segment** for every **duplicate acknowledgement** received

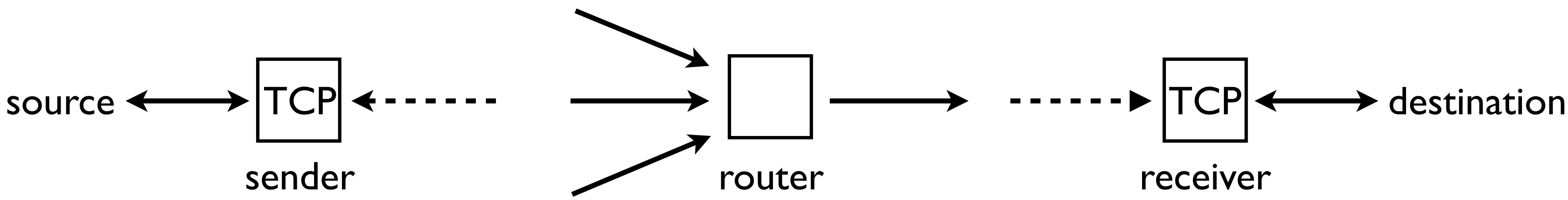
if **acknowledgement** received for **missing segment**:
enter **congestion avoidance** phase



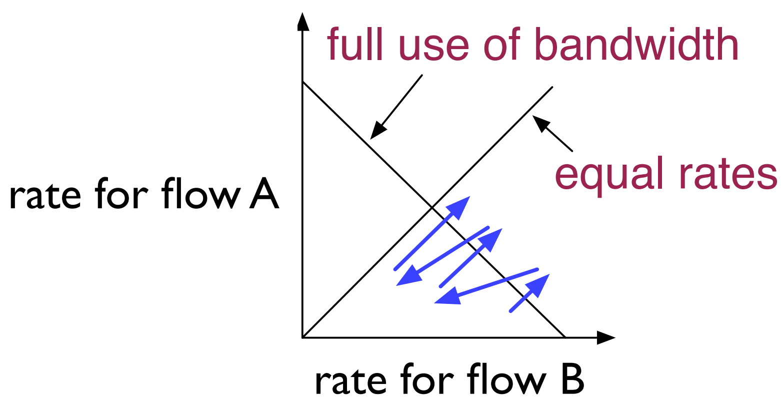
[Tan 4/E]

Note the variable $Threshold$. Initially, $Threshold$ is set to some 'big' value, say 64K Byte (it may change subsequently).

Fairness of TCP Congestion Control



Can greedy TCP flows share a resource equally?



Assumptions: A and B are only two flows using a resource, have same round-trip time, and use same version of TCP,

Arrows parallel to equal rates line represent rate adjustment during congestion avoidance. Arrows pointing toward origin represent rate adjustment during fast recovery after congestion event. Suppose B's initial rate is greater than A's. Eventually, both rates will converge to same value.

Potential Causes of Unfairness Among Flows

non-TCP flows are not subject to TCP congestion control and may monopolize a resource

different versions of TCP may govern different flows, with flows using fast recovery gaining more of a resource

flows with shorter round-trip times can react more quickly to make use of a resource following a congestion event

a single flow may use **multiple TCP connections** enabling it to gain more of a resource than a flow with a single TCP connection

Features of TCP Congestion Control

simple implementation at TCP senders only

no explicit feedback required from network for TCP senders to make rate control decisions

quick reaction by TCP senders to inferred congestion

fairness in resource access for flows with similar properties (TCP version, round-trip time, number of TCP connections per flow)

TCP Congestion Control: Room for Improvement

unnecessary flow rate reduction in environments where errors are likely, because failure to receive acknowledgement for transmitted data is automatically interpreted as loss due to congestion

lack of **fairness** among flows with different **round-trip times** and number of **TCP connections** per flow

flows over **high-speed links** with large amounts of data in flight **cannot tolerate** many **losses** before **throughput degrades** significantly, because of multiplicative decrease of congestion window with each apparent loss of data

Returning to Question about Hypothetical TCP

standard TCP with **one multi-segment window**

versus

hypothetical TCP with **parallel stop-and-wait windows**

Standard TCP

all functions are **window-based**, including reliable data transfer, in-order delivery of data to application, flow control, and congestion control

considered as one **monolithic system** instead of a set of several different network control functions

performing all of these functions with a single sliding window **simplifies implementation** but **restricts choices** of algorithms for realizing these functions

only **one retransmission timer** is needed for ARQ portion of standard TCP

Hypothetical TCP

stop-and-wait provides **reliable data transfer** only

other functions - in-order delivery of data to application, flow control, congestion control - are **independent** of **stop-and-wait** and need not be window-based

stop-and-wait inherently **selective repeat**, hence useful for **rapid retransmission** of **lost data** in error-prone networks

difficulty in successfully **transmitting** one **segment** need **not stall** entire **flow**, as long as flow **rate limits** are **respected** and there is **sufficient buffering** at receiver

one retransmission timer is needed for **each active window**

Hypothetical TCP

separating functions such as reliable data transfer, in-order delivery to application, flow control, and congestion control **simplifies accommodation** of **applications** with **different requirements**

for example, for **applications** requiring **reliable data transfer** but **not in-order data delivery**, TCP receiver need only **reassemble segments** into application data units and can **deliver** an **assembled unit without regard** for its **order** in flow