End-to-end Communication and Congestion Control

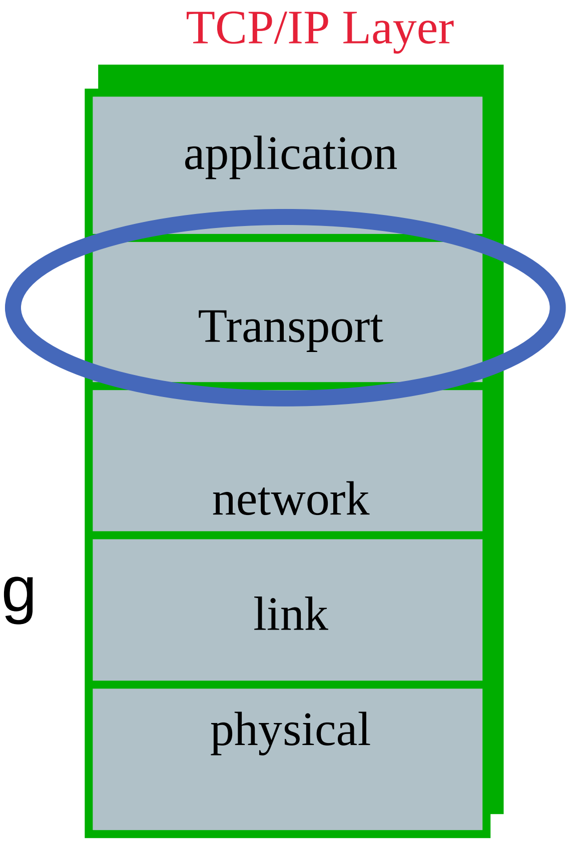
Computer Networks
EDA387/DIT660

Elad Michael Schiller

Q: What does End2End communications mean?

(some context first)
Internet protocol stack

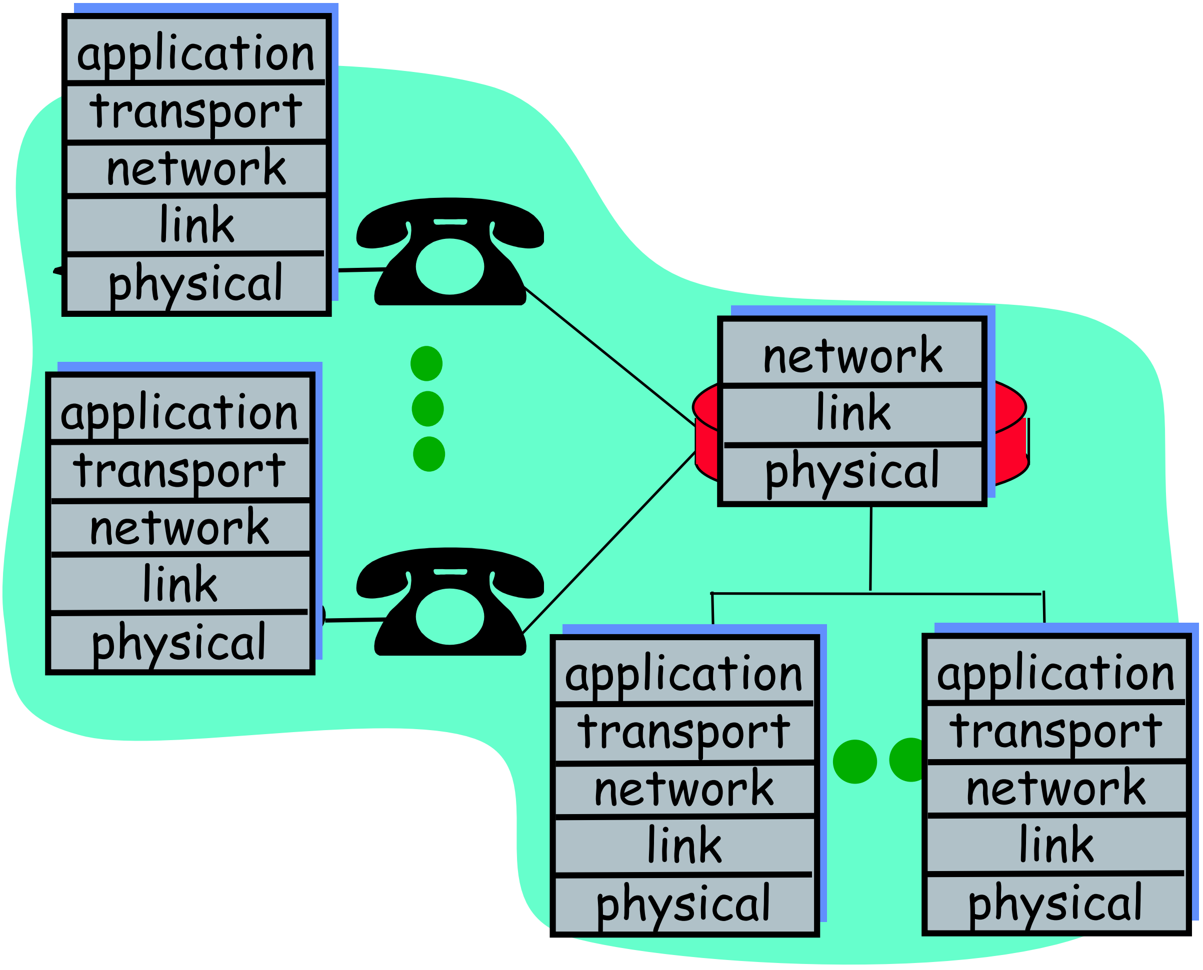
- application: ftp, smtp, http, etc
- transport: tcp, udp, …
- network: routing of datagrams from source to destination
  - ip, routing protocols
- link: data transfer between neighboring network elements
  - ppp, ethernet
- physical: bits “on the wire”



Layering: logical communication

Each layer:

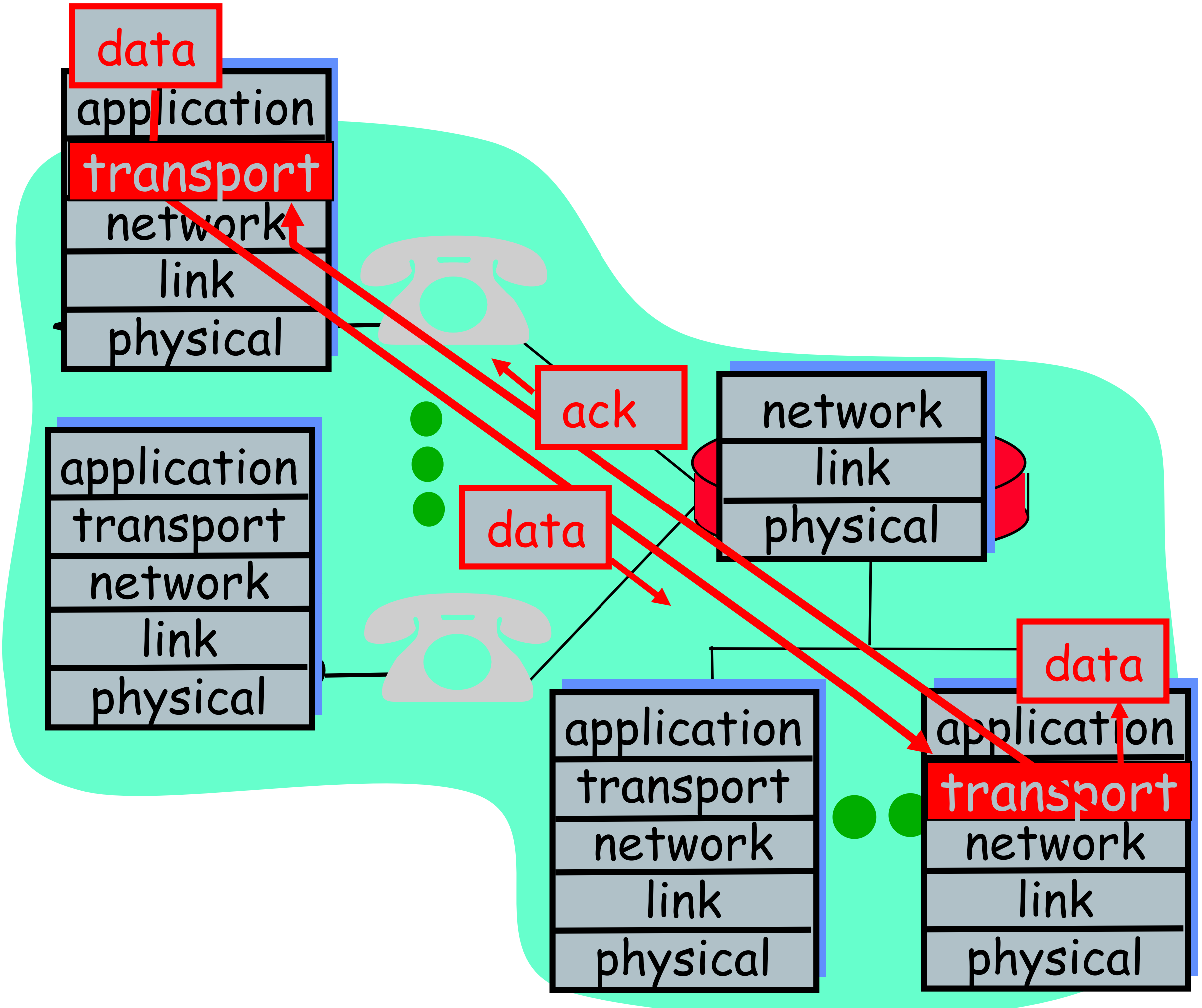
- distributed
- "entities" implement layer functions at each node
- entities perform actions, exchange messages with peers



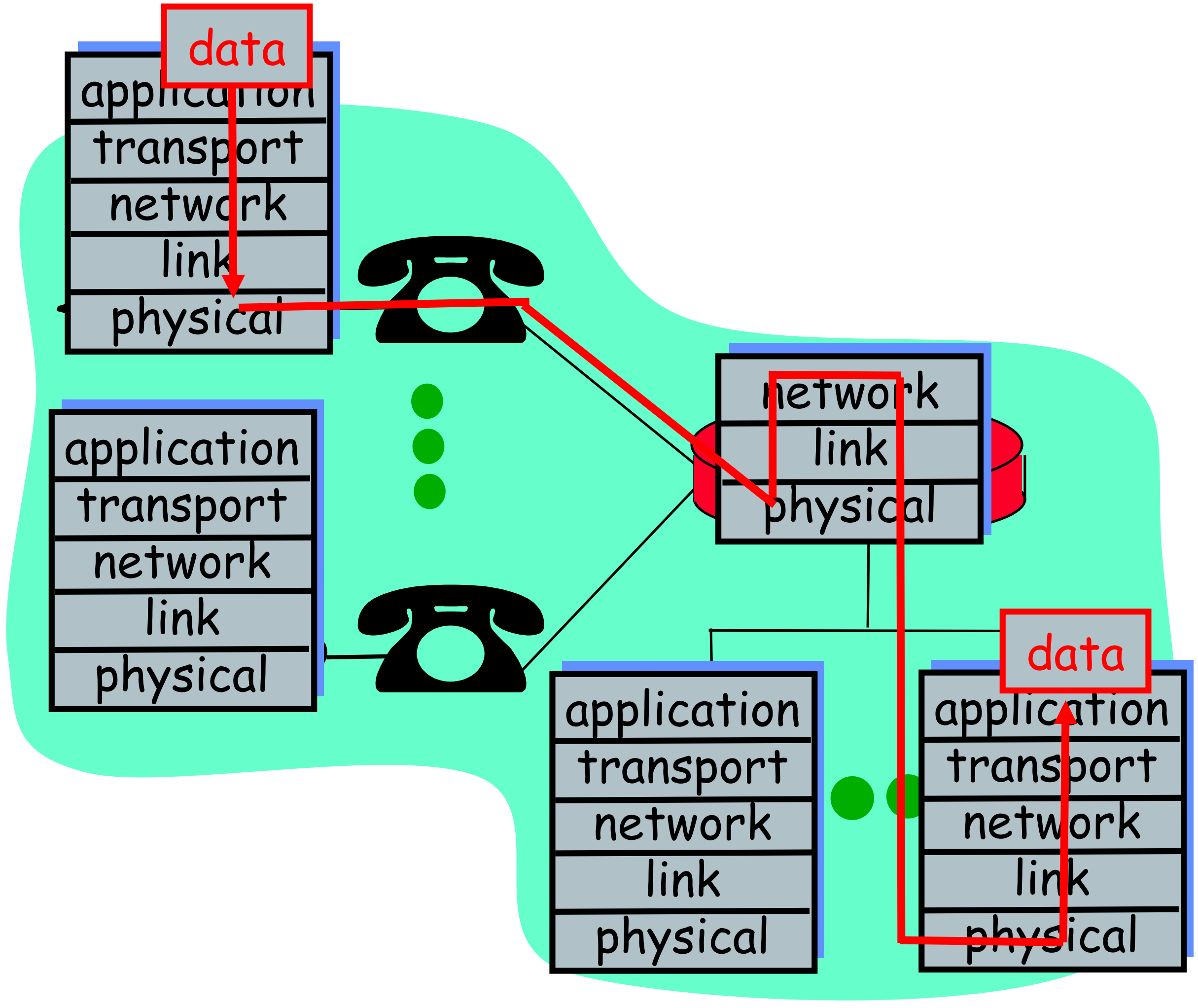
Transport layer: *logical* communication

transport protocols run in end systems:

- take data from app
- add addressing, (maybe reliability info), form "segment"
- send "segment" to peer
- (maybe) wait for peer to ack receipt



(Layering: physical communication)



Q: what does the end users need from the transport layer?

Care for:

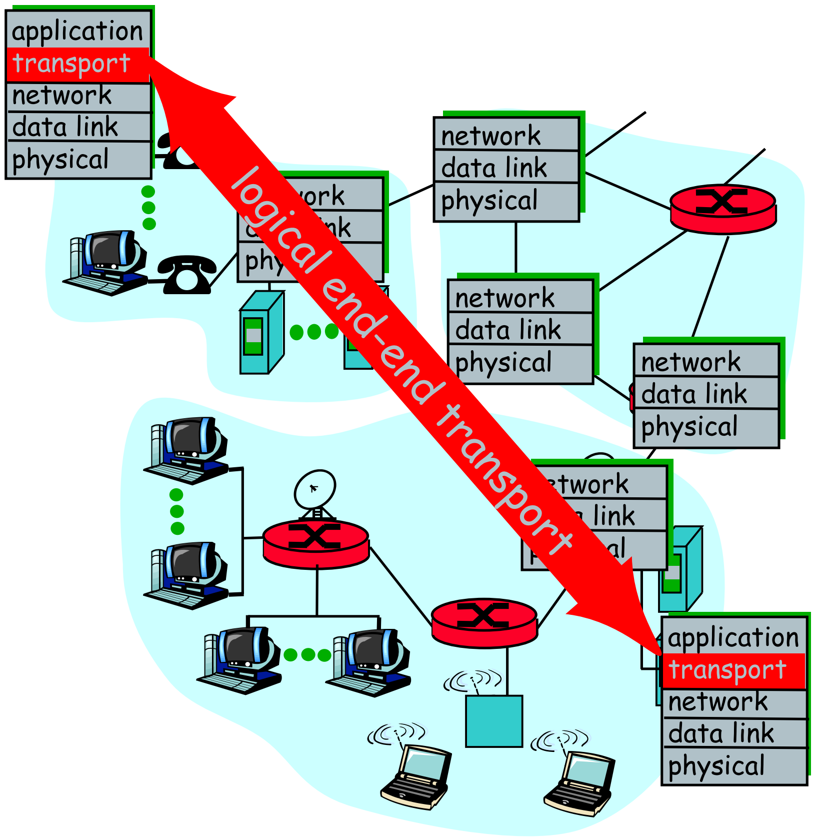
- connection management
- reliability (guaranteed info. arrival)
- timing, e.g., for streamed communication

Q: what services do the Internet transport protocols offer to the applications?

Internet Transport Protocols: Services

TCP protocol service:

- setup (connection-oriented service)
- reliable transport between sender- receiver
- sender won't overwhelm receiver (flow control in service)
- acts when network overloaded: an extra service, both for users and for the network benefit, which is called congestion control
- does not provide: timing, bandwidth guarantees



UDP protocol service:

- Best effort delivery, i.e., put stuff in an envelop and rely on IP for delivery (or not)
  - connectionless service; does not provide: reliability, flow control, congestion control, timing, or bandwidth guarantee

Roadmap

- transport layer services (user, network perspectives)
- User perspective and connection: addressing, multiplexing/demultiplexing
- Reliable data transfer and TCP
  - User perspective
  - Network perspective (how to)
- Flow control
- TCP congestion control
  - Causes and basic goals; end-to-end control; TCP; setting timeouts;
- Broader discussion

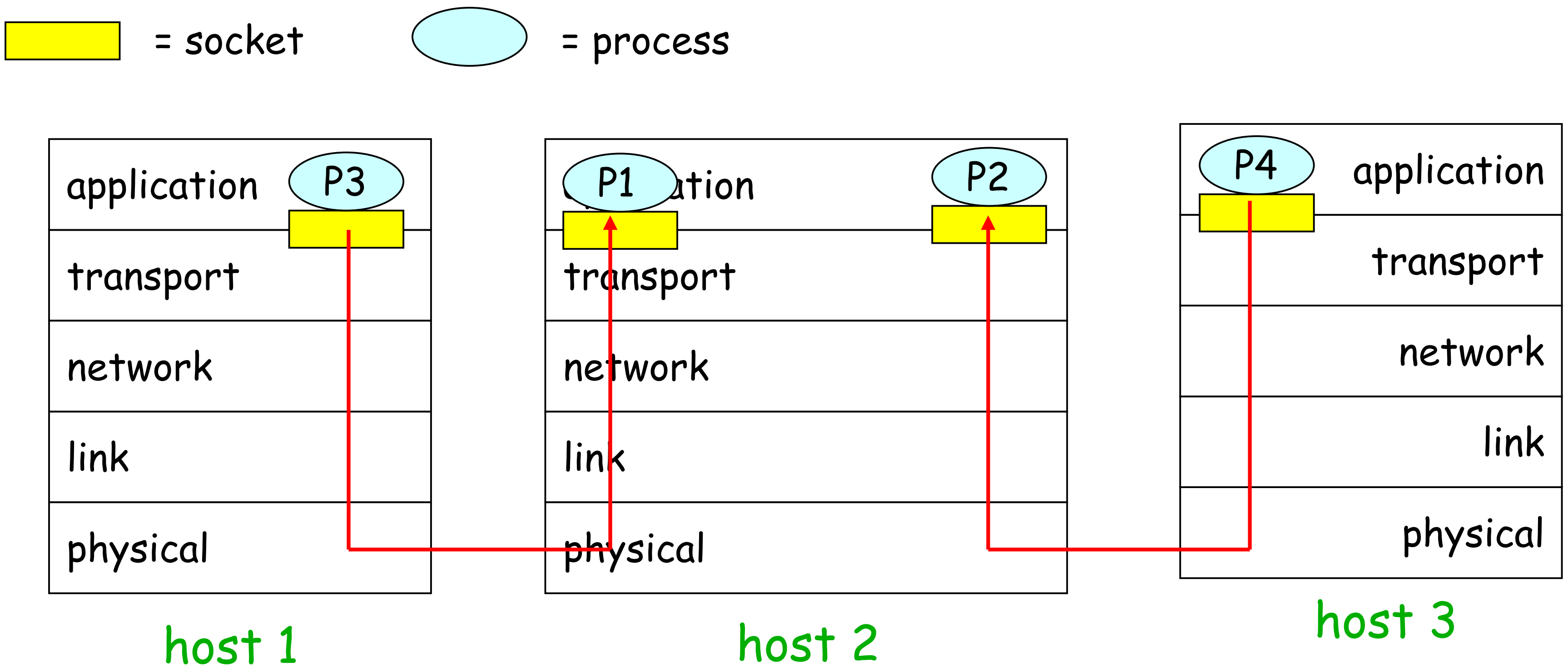
Multiplexing/demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

Multiplexing at send host:

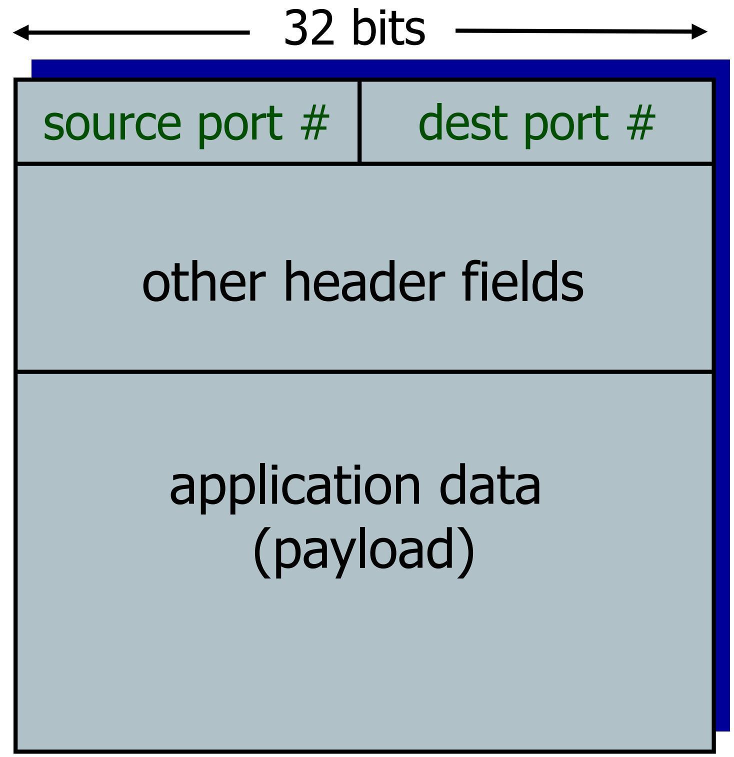
gathering data, enveloping data with header (later used for demultiplexing)



Recall: *segment* - unit of data exchanged between transport layer entities

How addressing — demultiplexing works

- Host uses *IP addresses, port numbers* to direct segment to appropriate socket



TCP/UDP segment format

UDP addressing - demultiplexing

- when creating datagram to send, must specify:
  - destination IP address
  - destination port #

created socket has host-local port #, eg:

```
DatagramSocket mySocket1 = new
 DatagramSocket(12534);
```

- when host receives UDP datagram:
  - checks destination port # in datagram
  - directs UDP datagram to socket with that port #

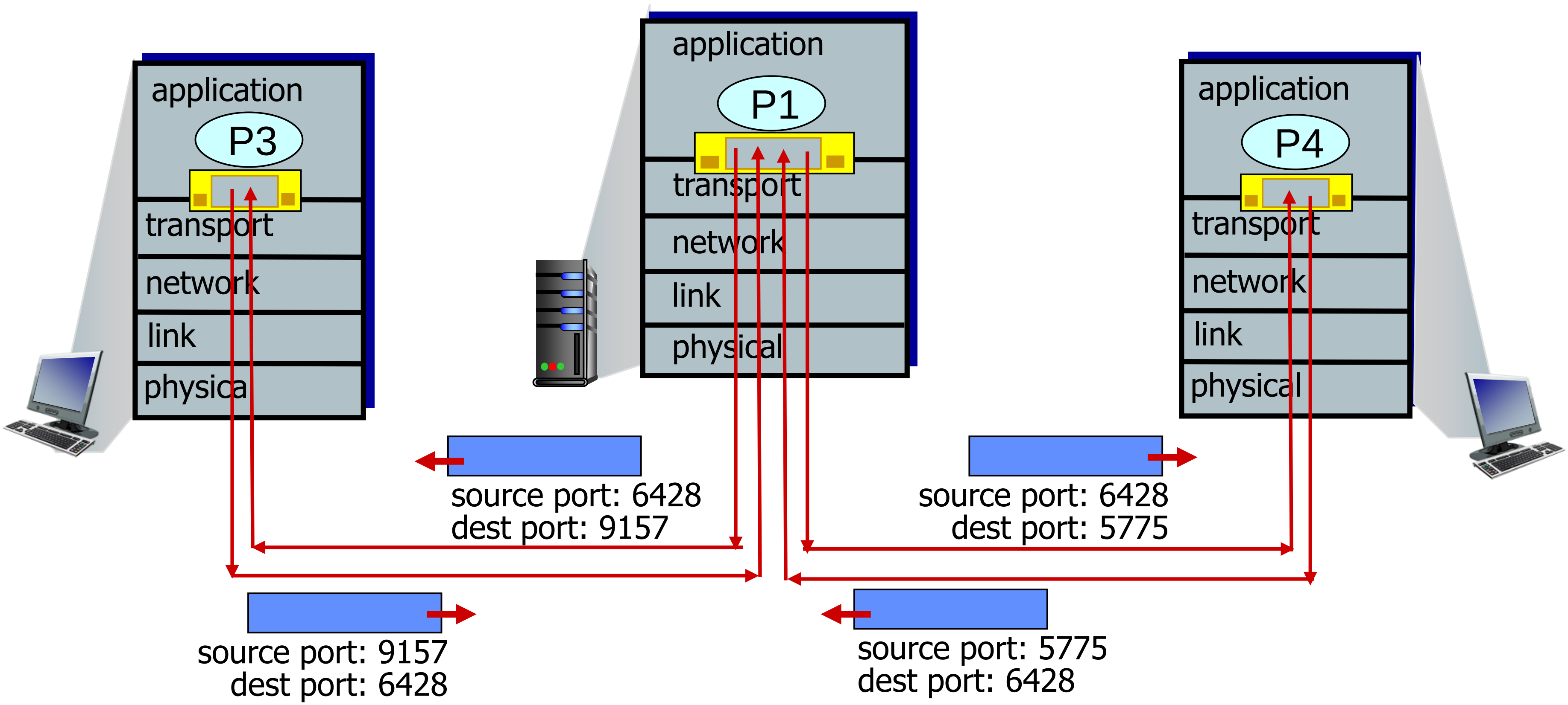
IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed *to the same socket*

UDP demux: example

```
DatagramSocket
 mySocket2 = new
 DatagramSocket(9157);
```

```
DatagramSocket
 serverSocket = new
DatagramSocket(6428);
```

```
DatagramSocket
 mySocket1 = new
 DatagramSocket(5775);
```

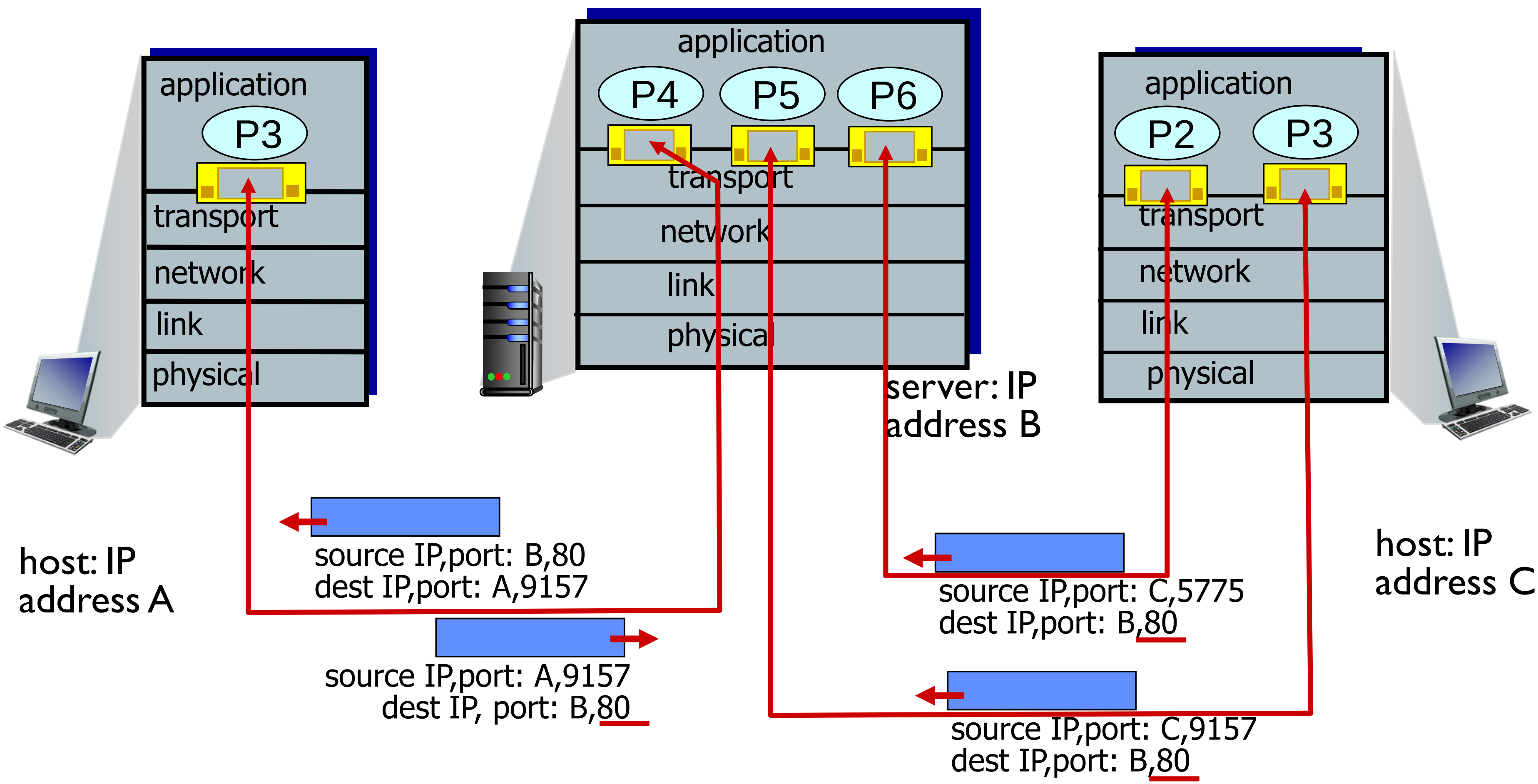


Connection-oriented (TCP) demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket

- server host may support many simultaneous TCP sockets:
  - one socket per connection
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will even have different sockets for each request

TCP demux: example



three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

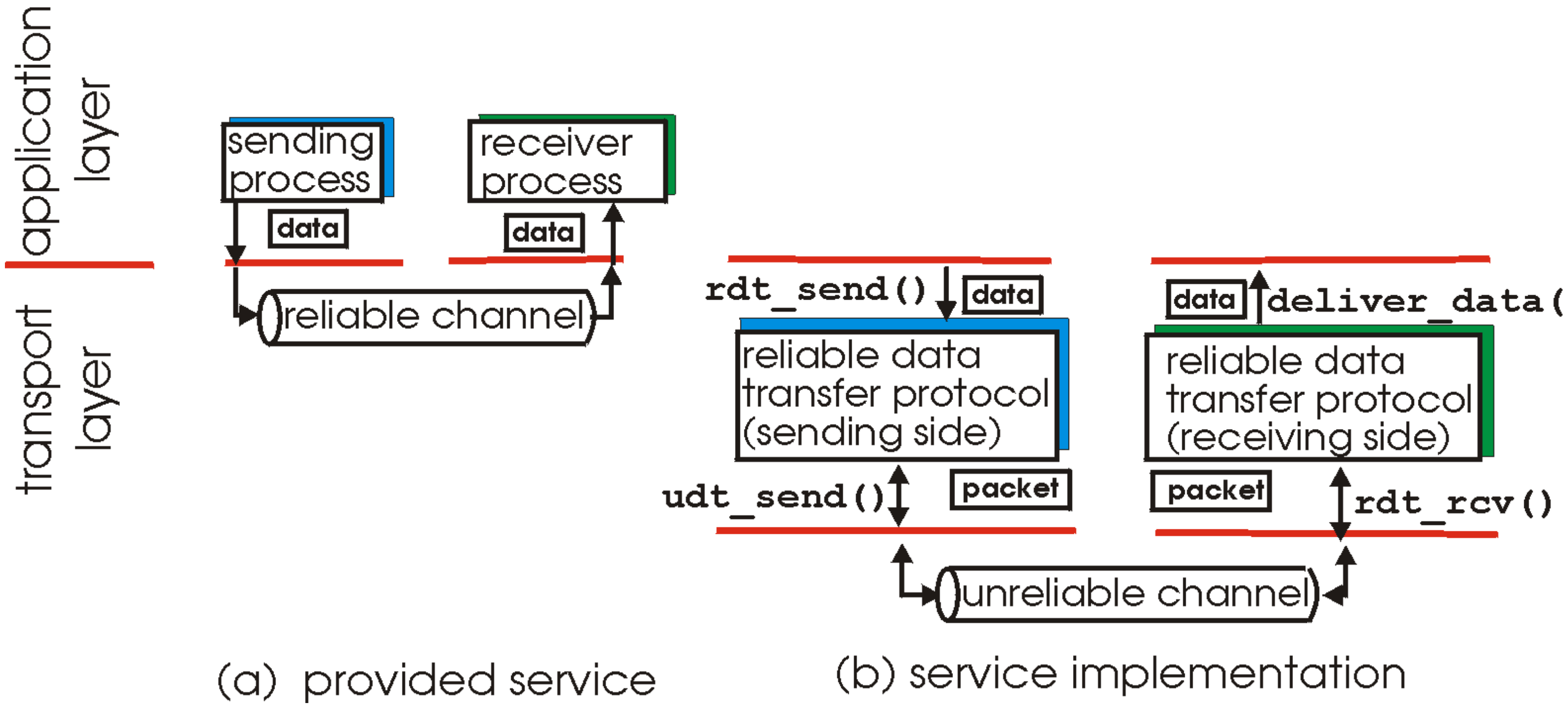
TCP next station:
it provides reliability (error control, in-order delivery)

Roadmap

- transport layer services (user, network perspectives)
- User perspective &connection: addressing, multiplexing/demultiplexing
- Reliable data transfer & TCP
  - User perspective
  - Network perspective (how to)
- Flow control
- TCP congestion control
  - Causes and basic goals; end-to-end control; TCP; setting timeouts;
- Summary and review questions

Principles of Reliable data transfer

- important in (app.,) transport, link layers
- in top-10 list of important networking topics!



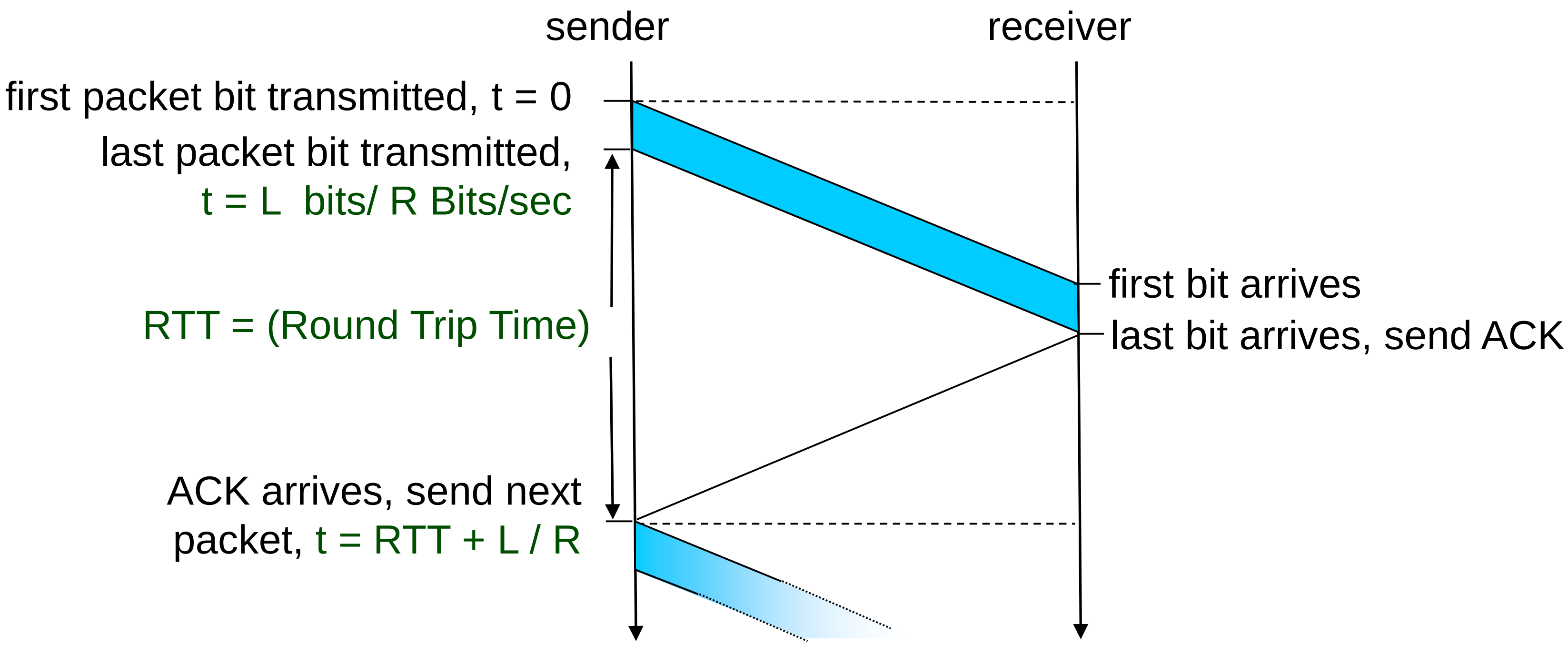
- characteristics of unreliable channel will determine complexity of reliable data transfer (RDT) protocol

Providing Reliability

- Traditional technique: Positive Acknowledgement with Retransmission (stop&wait)
  - Receiver sends *acknowledgement* when data arrives
  - Sender starts timer whenever transmitting
  - Sender retransmits if timer expires before acknowledgement arrives

Q: Reliability vs Efficiency: Any Problem With Simple StopAndWait?

stop-and-wait bandwidth utilization



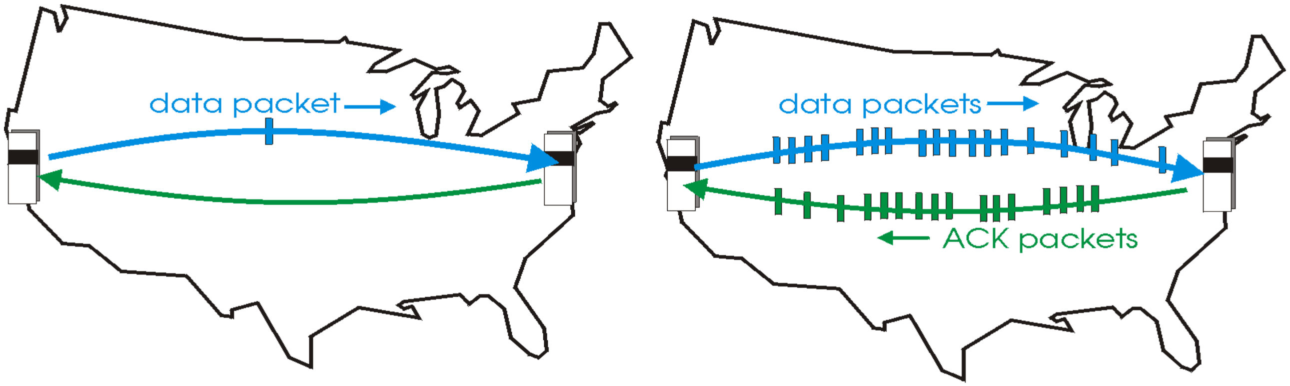
$$U_{sender} = \frac{L / R}{RTT + L / R}$$

Q: Ideas for improving?

Pipelined protocols

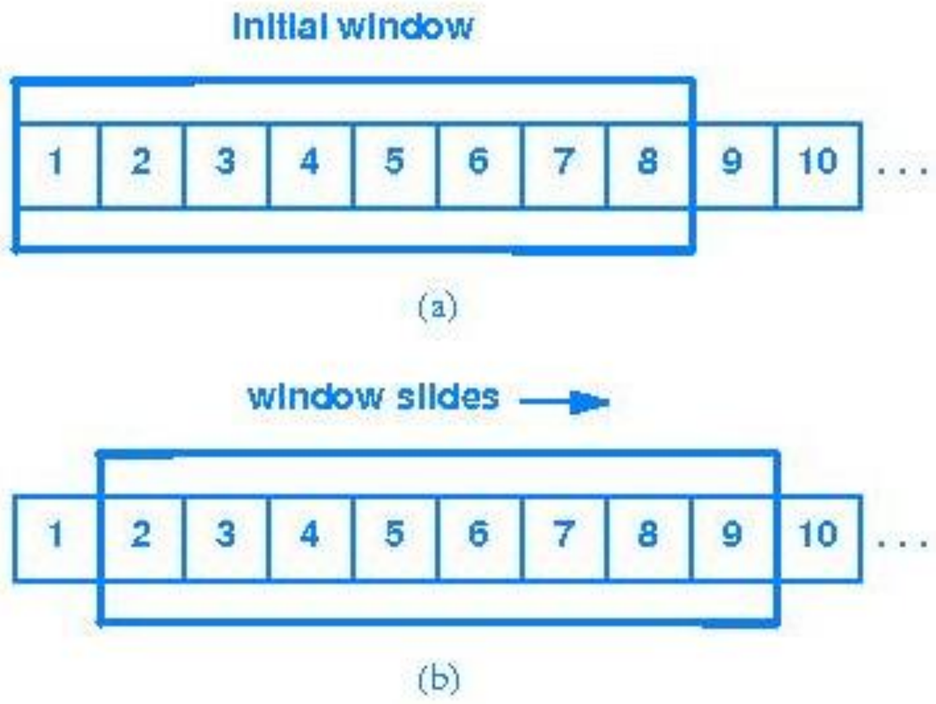
Pipelining: also known as *sliding window*:

- **Solution** to the problem of low utilization of stop-and-wait: sender allows multiple“in-flight”, yet-to-be-ack-ed segments.
  - Choice of N (window size) : optimally, it should allow the sender to continuously transmit during the round-trip transit time
  - Still requires acknowledgements and retransmission:
    - In case of error in sequence, acknowledge the last correctly received segment



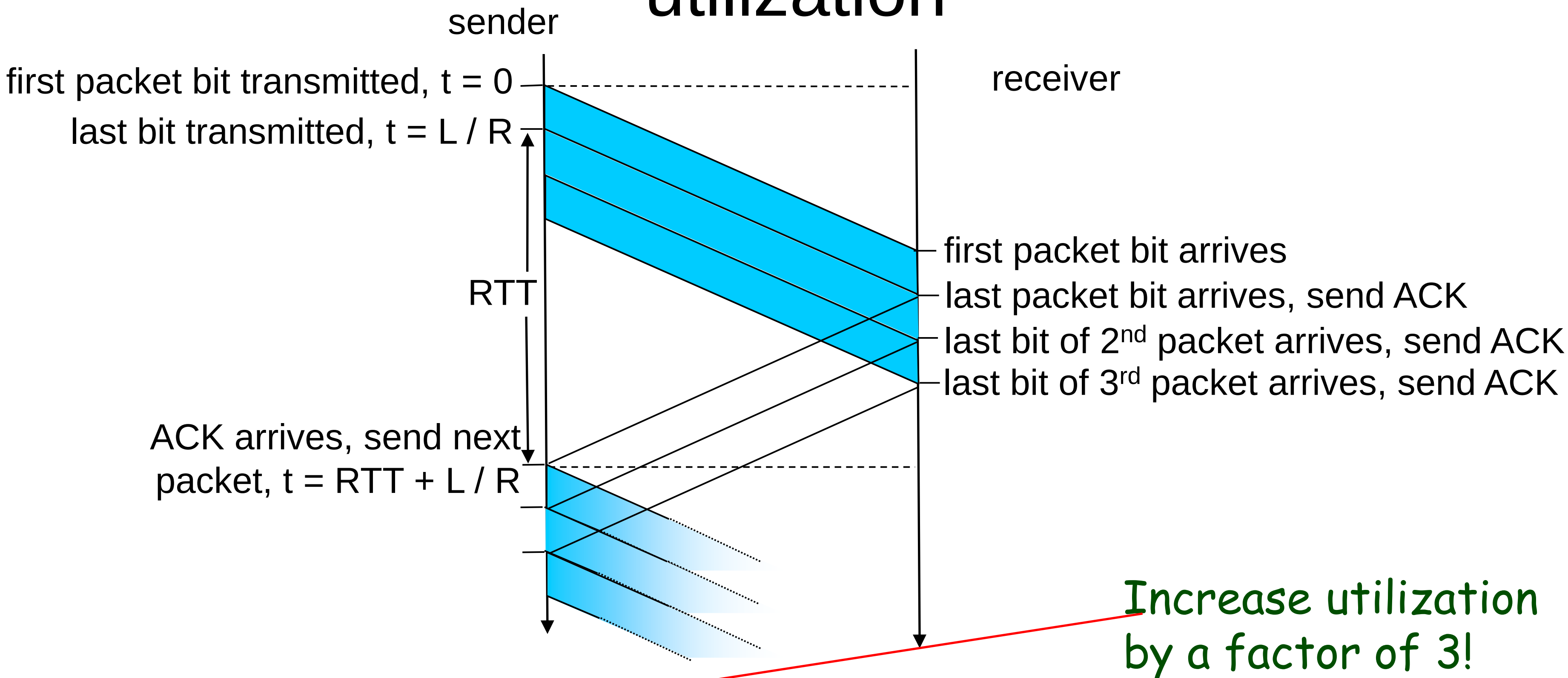
(a) a stop-and-wait protocol in operation   (b) a pipelined protocol in operation

Illustration Of Sliding Window



- As acknowledgement arrives, move the window forward

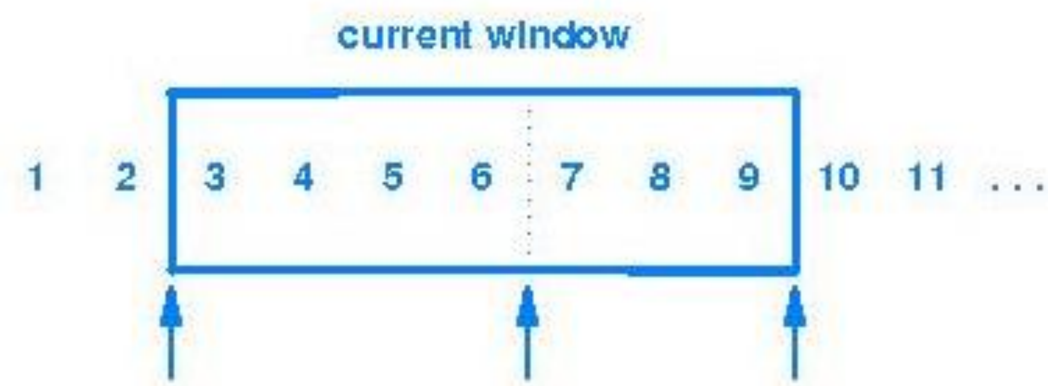
Pipelining (sliding window): increased utilization



$$U_{sender} = \frac{3 * L / R}{RTT + L / R}$$

Sliding Window Used By TCP

- Measured in byte positions



- Bytes through 2 are acknowledged
- Bytes 3 through 6 not yet acknowledged
- Bytes 7 though 9 waiting to be sent
- Bytes above 9 lie outside the window and cannot be sent

Q: when to retransmit?

timeouts?
… and Fast Retransmit in TCP

- Set timeout by adaptive monitoring of RTT (roundtrip time)
  - Some form of averaging of recently measured RTTs, say, exponentially weighted moving average (EWMA).
- Time-out period can be relatively long:
  - long delay before resending lost packet
- But: TCP send duplicate ack's when it misses bytes in sequence
- Detect lost segments via duplicate ACKs.
  - If segment is lost, there will likely be many duplicate ACKs.

If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:

- fast retransmit: resend segment before timer expires

Q: What do Ack's achieve besides reliability?

- **Flow control**: receiver can ack its receiving capacity (receiver's buffer, aka receivers window),
- i.e. ***avoid swamping the receiver***

Roadmap

- transport layer services (user, network perspectives)
- User perspective &connection: addressing, multiplexing/demultiplexing
- Reliable data transfer & TCP
  - User perspective
  - Network perspective (howto)
- Flow control
- TCP congestion control
  - Causes and basic goals; End to end control; TCP; setting timeouts;
- Broader discussion

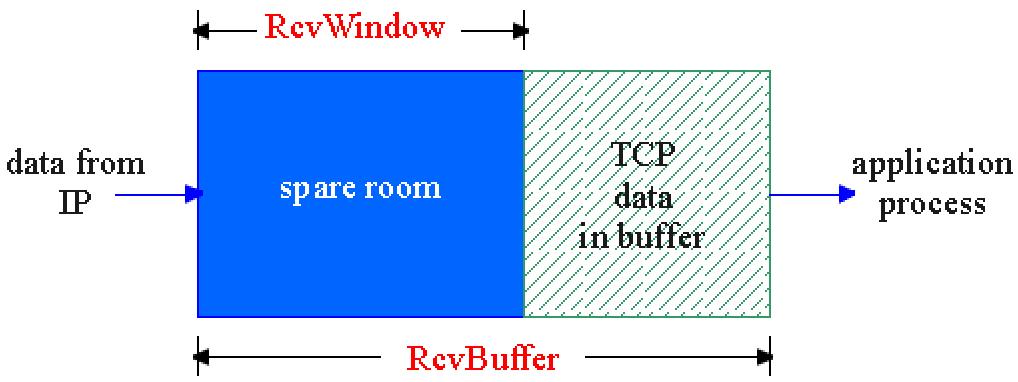
TCP Flow Control: Dynamic sliding windows

flow control

sender won't overrun receiver's buffers by transmitting too much, too fast

`RcvBuffer` = size or TCP Receive Buffer

`RcvWindow` = amount of spare room in Buffer



receiver buffering

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space

- **`RcvWindow` field** in TCP segment

sender: keeps the amount of transmitted, unACKed data less than most recently received **`RcvWindow`**

Roadmap

- transport layer services (user, network perspectives)
- User perspective &connection: addressing, multiplexing/demultiplexing
- Reliable data transfer & TCP
  - User perspective
  - Network perspective (howto)
- Flow control
- TCP congestion control
  - Causes and basic goals; end-to-end control; TCP; setting timeouts;
- Summary and review questions

Congestion Control

- Causes and basic goals
- End to end control
- TCP: Congestion Control
- TCP: setting timeouts

Q: Is congestion control the same as flow control?

- No!
- Congestion control =
  - ***Avoid congesting the network***
  - i.e. network core issue (in contrast to flow-control, which is end-host, i.e., sender-receiver issue)

Principles of Congestion Control

Congestion:

- informally: “too many sources sending too much data too fast for *network* to handle”
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queuing in router buffers)
- a highly important problem!

Goals of congestion control

- Throughput:
  - Maximize goodput
  - the total number of bits end-end
- Fairness:
  - Give different sessions “equal” share.
  - Max-min fairness
    - Maximize the minimum rate session.
  - Single link:
    - Capacity: R
    - Sessions: m
    - Each sessions: R/m

Max-min fairness

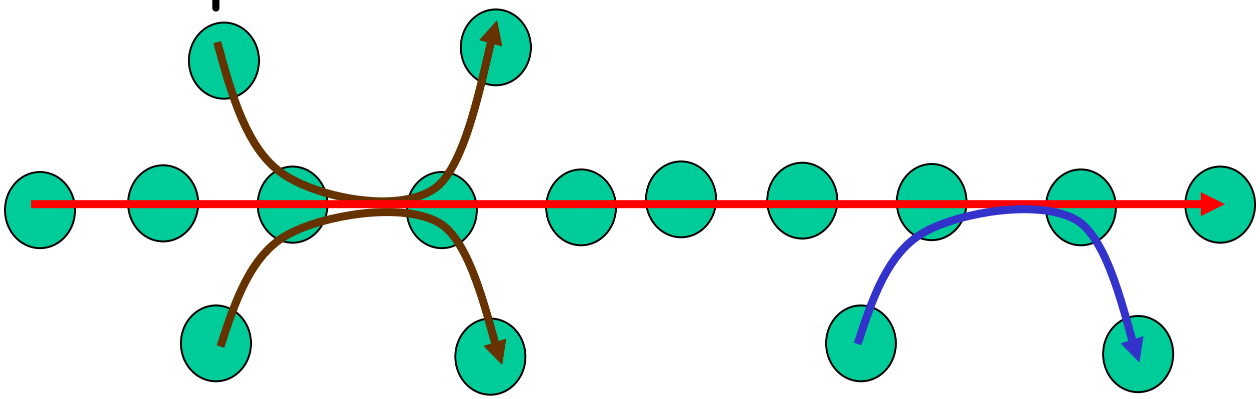
- Model: Graph G(V,e) and sessions $s_1 \ldots s_m$
- For each session s_i a rate r_i is selected.
- The rates are a Max-Min fair allocation:
  - The allocation is maximal
    - No r_i can be simply increased
  - Increasing allocation r_i requires reducing
    - Some session j
    - $r_j \leq r_i$
- Maximize minimum rate session.

Max-min fairness: Algorithm

- Model: Graph G(V,e) and sessions $s_1 \ldots s_m$
- Algorithmic view:
  - For each link compute its fair share f(e).
    - Capacity / # session
  - select minimal fair share link.
  - Each session passing on it, allocate f(e).
  - Subtract the capacities and delete sessions
  - continue recessively.

Max-min fairness

- Example



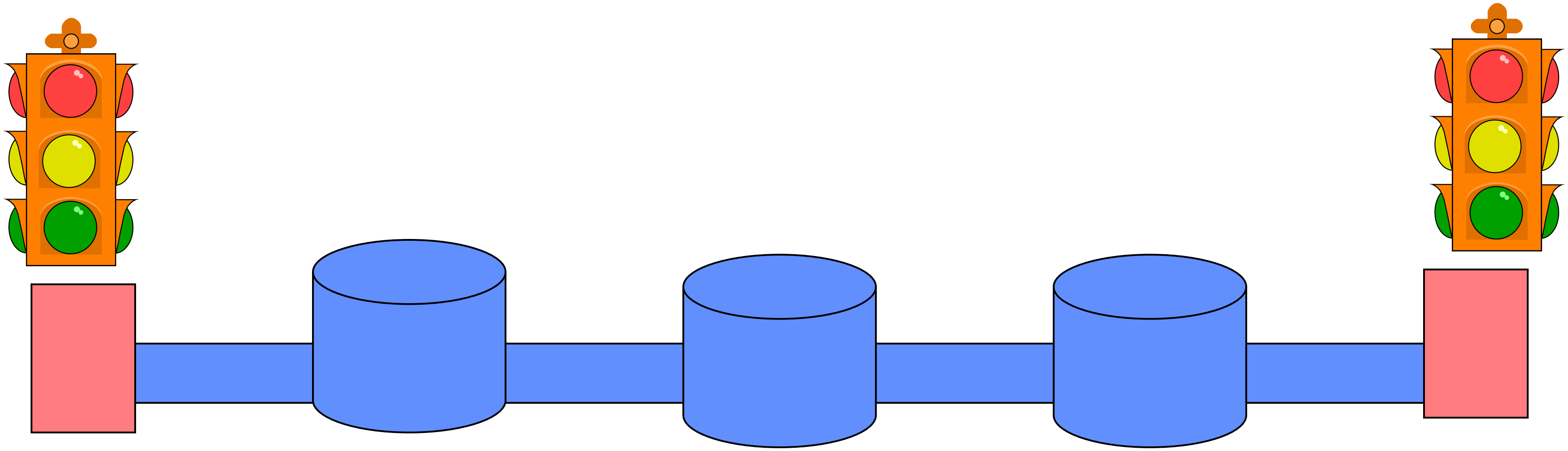
- Throughput versus fairness

Demands: {2,2.6,4,5}; Capacity: 10; f(e): 2.5

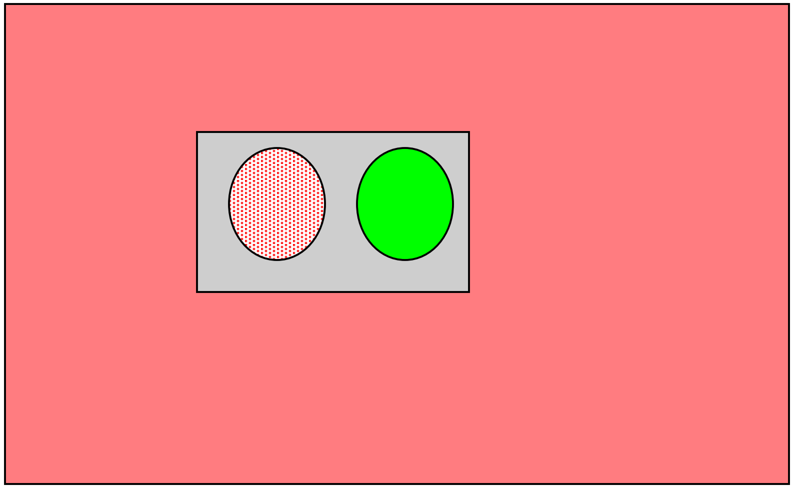
Allocation: {~~2.5~~ 2, ~~2.67~~, 2.6, 2.67, 2.67}

End to end feedback

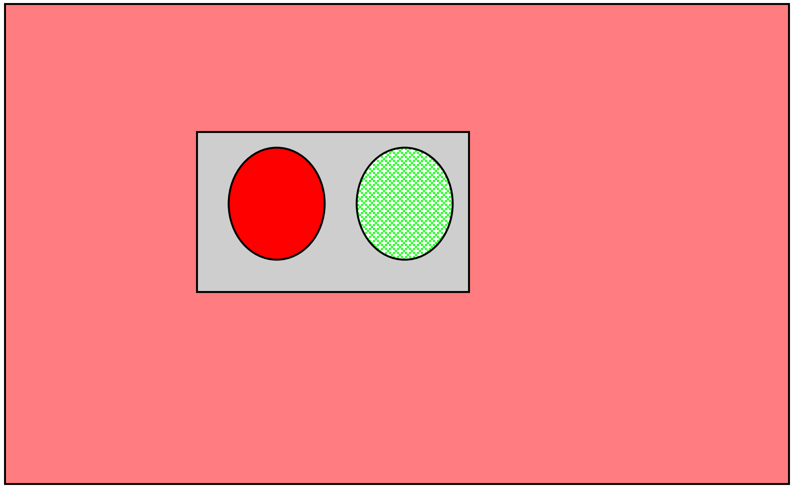
- Abstraction:
  - Alarm flag.
  - observable at the end stations



Simple Abstraction



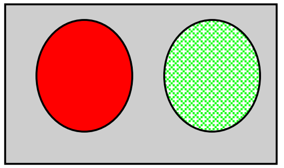
Simple Abstraction



Simple feedback model

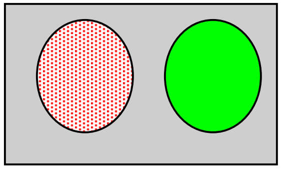
- Every RTT receive feedback
  - High Congestion

    Decrease rate



  - Low congestion

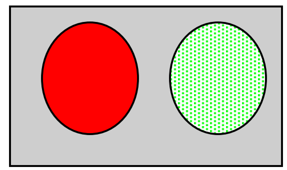
    Increase rate



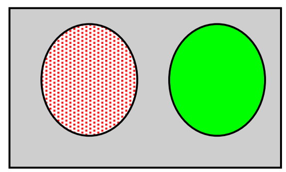
- Variable rate controls the sending rate

Multiplicative Update

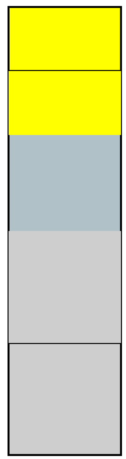
- Congestion:
  - Rate = Rate/2



- No Congestion:
  - Rate= Rate *2

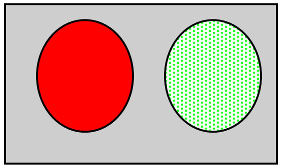


- Performance
  - Fast response
- Fairness:
  - Ratios unchanged (Un-fair)

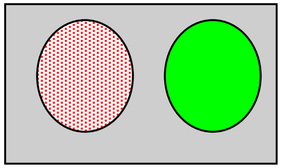


Additive Update

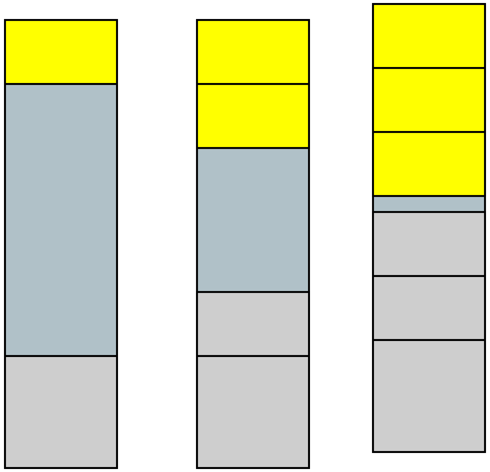
- Congestion:
  - Rate = Rate -1



- No Congestion:
  - Rate= Rate +1



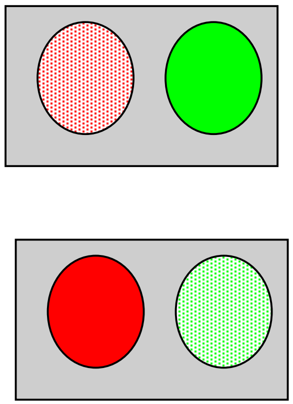
- Performance
  - Slow response
- Fairness:
  - Divides spare bandwidth equally
  - Difference remains unchanged

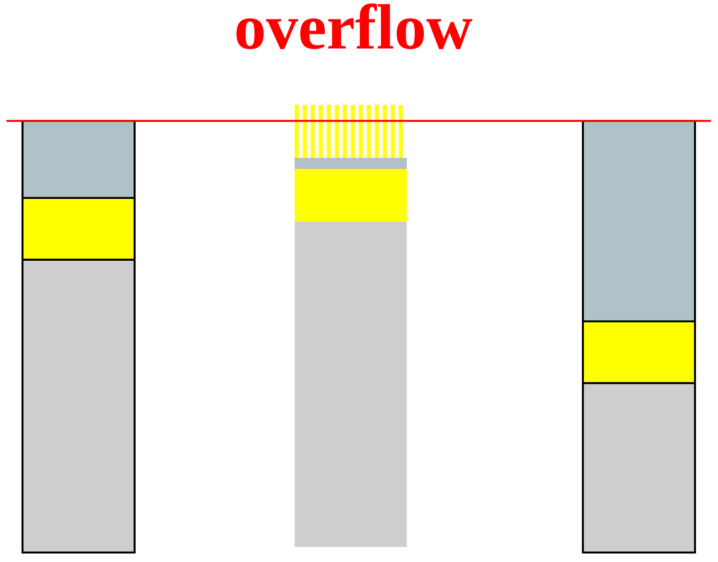


AIMD Scheme

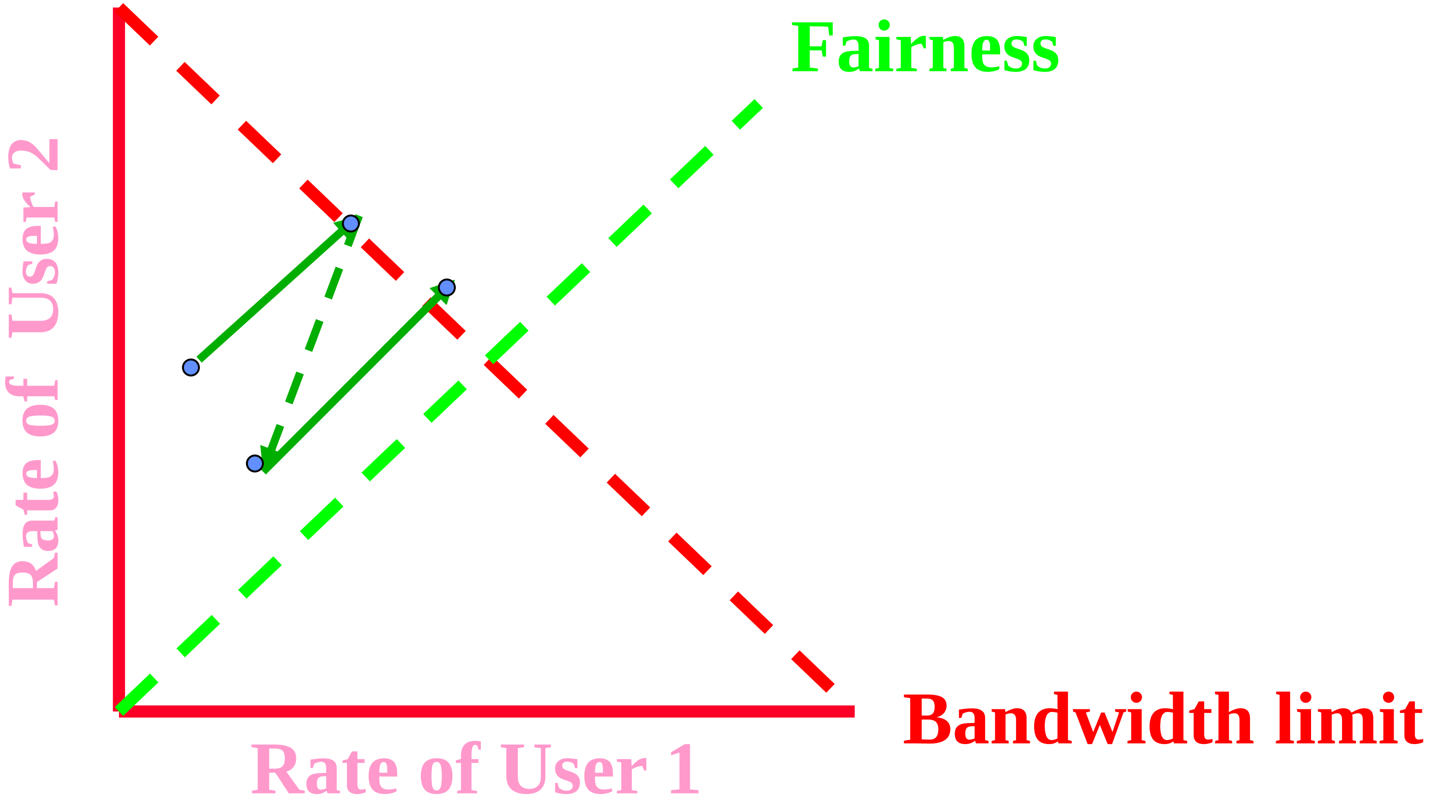
- Additive Increase
  - Fairness: ratios improves
- Multiplicative Decrease
  - Fairness: ratio unchanged
  - Fast response
- Performance:
  - Congestion -

  Fast response
- Fairness?



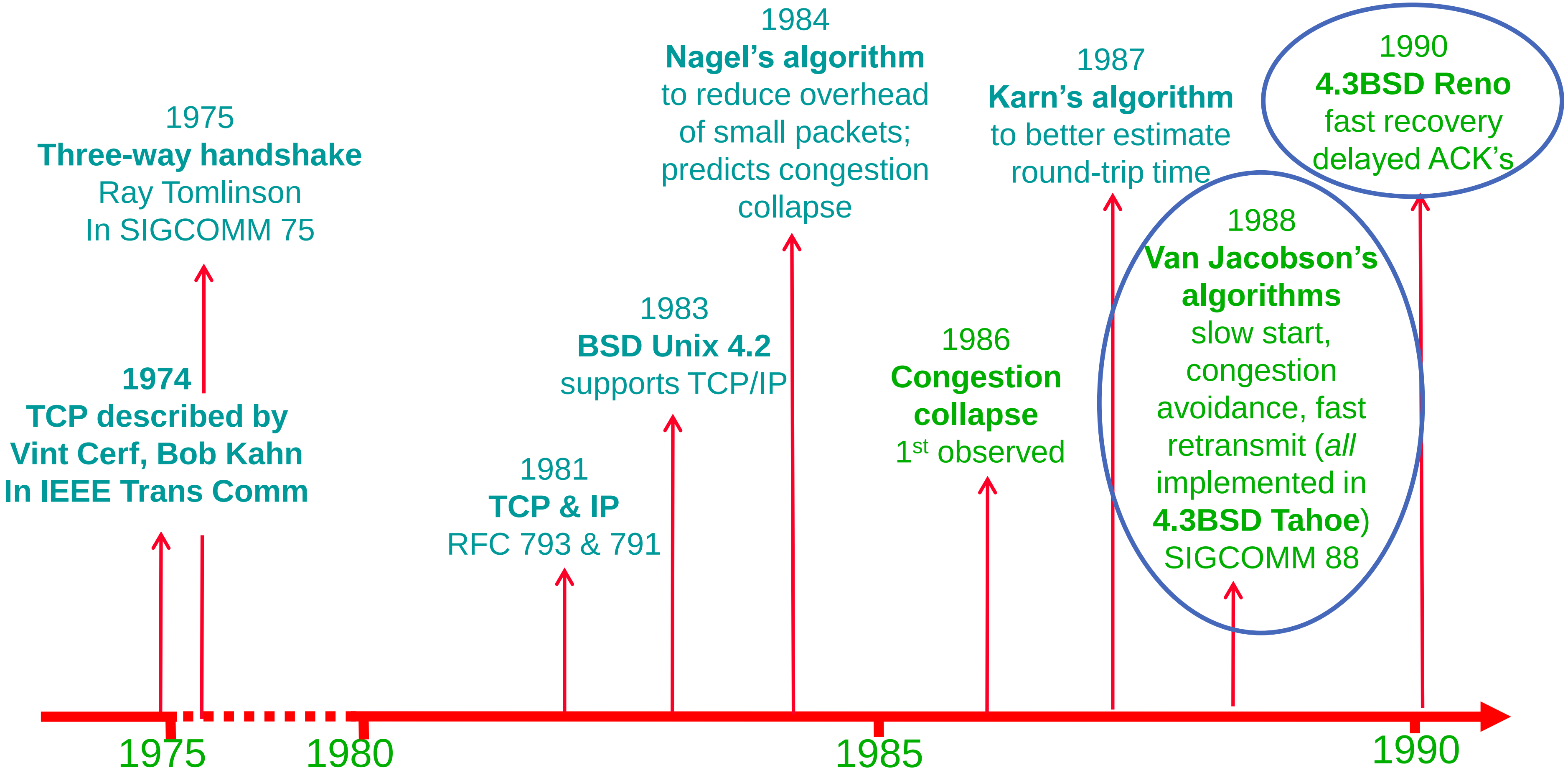


AIMD: Two users, One link

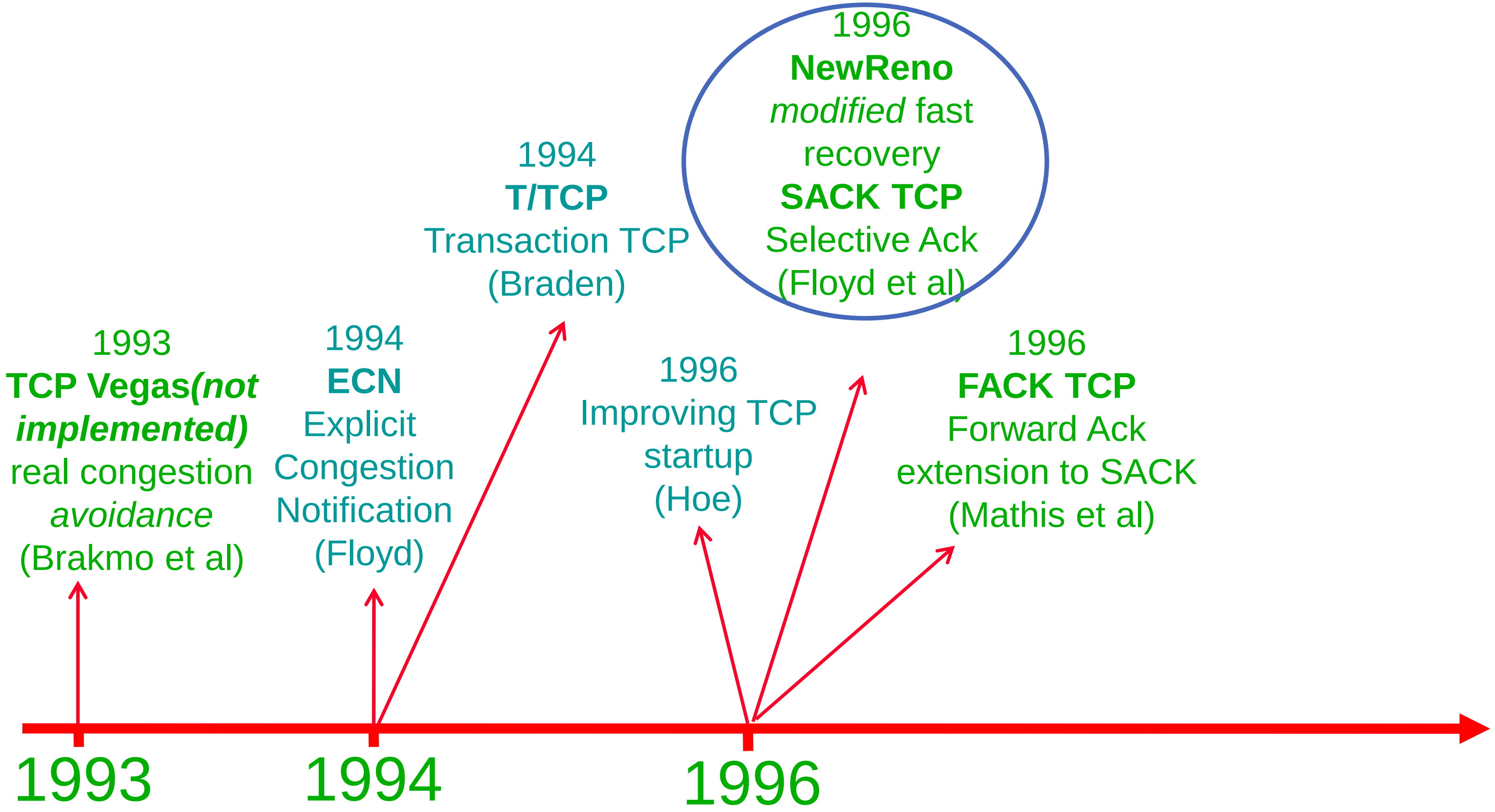


Dah-Ming Chiu, Raj Jain: Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. Computer Networks 17: 1-14 (1989)

Evolution of TCP



Evolution of TCP



TCP Congestion Control

- Closed-loop, end-to-end, window-based congestion control
- Designed by Van Jacobson in late 1980s, based on the AIMD alg. of Dah-Ming Chu and Raj Jain
- Works well so far: the bandwidth of the Internet has increased by more than 200,000 times
- Many versions
  - TCP/Tahoe: this is a less optimized version
  - TCP/Reno: many OSs today implement Reno type congestion control
  - TCP/Vegas: not currently used

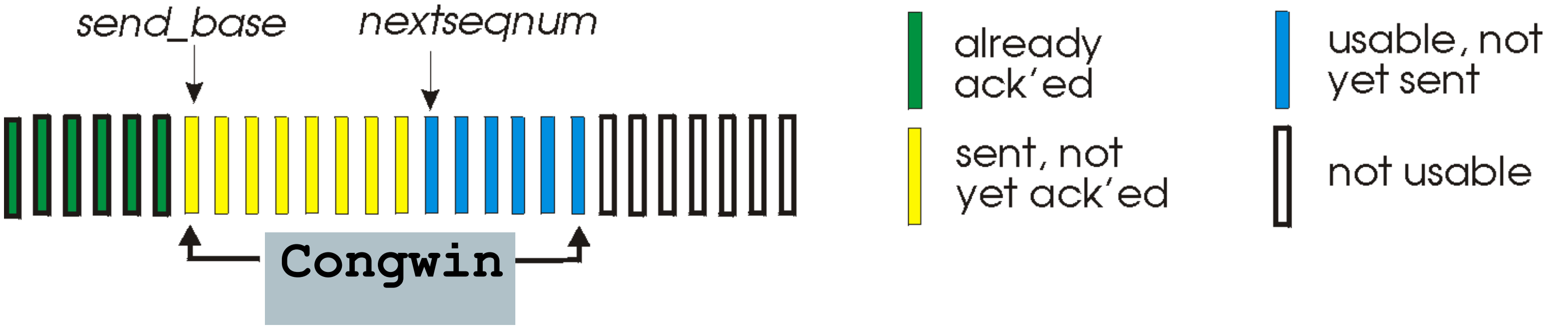
For more details: see TCP/IP illustrated; or read http://lxr.linux.no/source/net/ipv4/tcp_input.c for linux implementation

TCP & AIMD: congestion

- Dynamic window size [Van Jacobson]
  - Initialization: MI
    - Slow start
  - Steady state: AIMD
    - Congestion Avoidance
- Congestion = timeout
  - TCP Taheo
- Congestion = timeout || 3 duplicate ACK
  - TCP Reno & TCP new Reno
- Congestion = higher latency
  - TCP Vegas

TCP Congestion Control

- end-end control (no network assistance)
- transmission rate limited by congestion window size, **`Congwin`**, over segments:



- w segments, each with MSS bytes sent in one RTT:

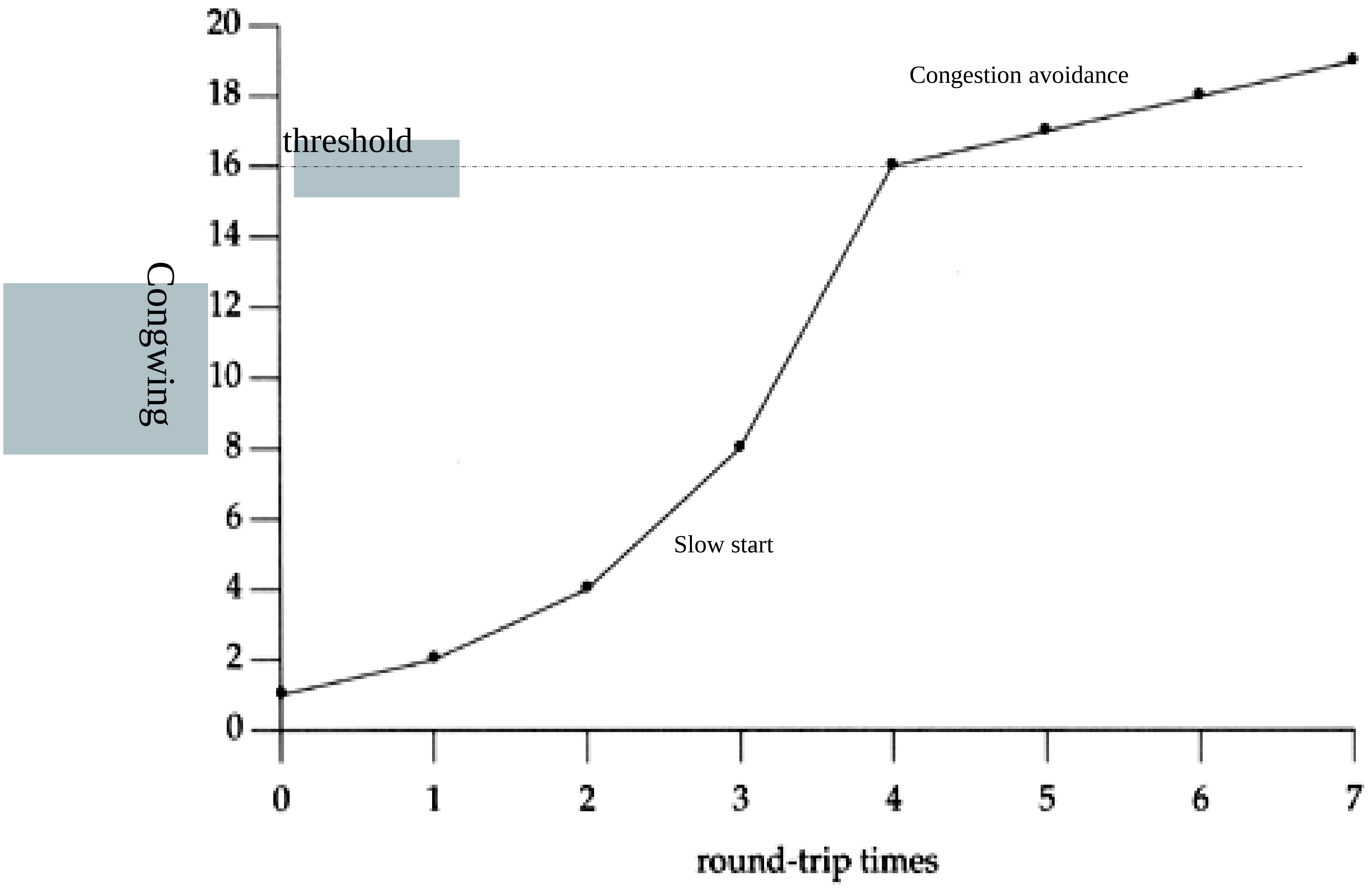
$$\boxed{\text{throughput} = \frac{w * MSS}{RTT} \text{ Bytes/sec}}$$

TCP congestion control:

- "probing" for usable bandwidth:
  - ideally: transmit as fast as possible (`Congwin` as large as possible) without loss
  - *increase* `Congwin` until congestion (loss)
  - Congestion: *decrease* `Congwin`, then begin probing (increasing) again

- Basic structure:
- two "phases"
  - slow start - MI
  - congestion avoidance- AIMD
- important variables:
  - `Congwin:` window size
  - `threshold:` defines threshold between the slow start phase and the congestion avoidance phase

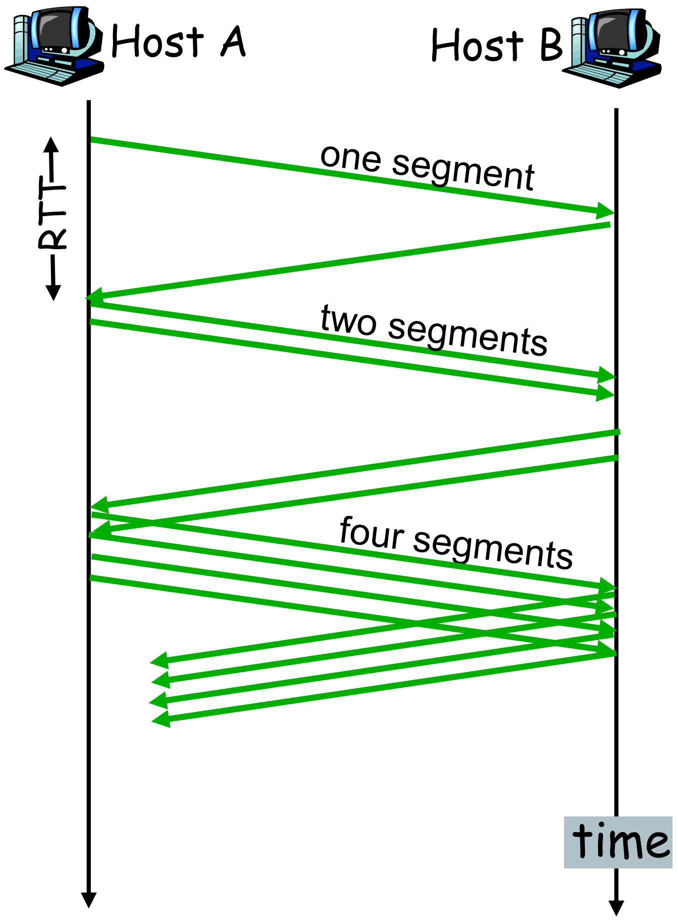
Visualization of the Two Phases



TCP Slowstart: MI

Slowstart algorithm

initialize: Congwin = 1
for (each segment ACKed)
    Congwin++
until (congestion event OR
    CongWin > threshold)

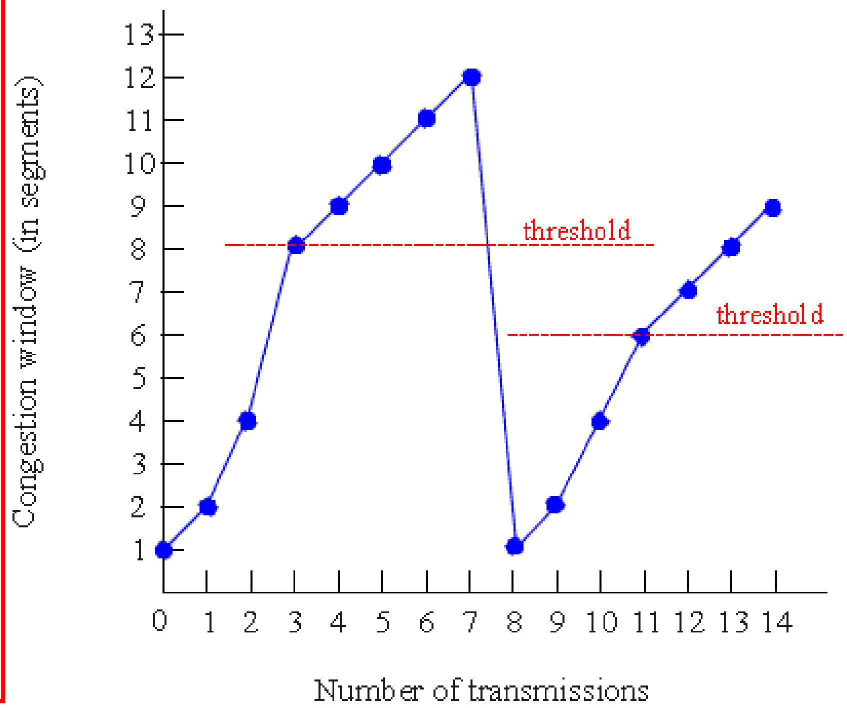


- exponential increase (per RTT) in window size (not so slow!)
- In case of timeout:
  - Threshold=CongWin/2

TCP Taheo Congestion Avoidance

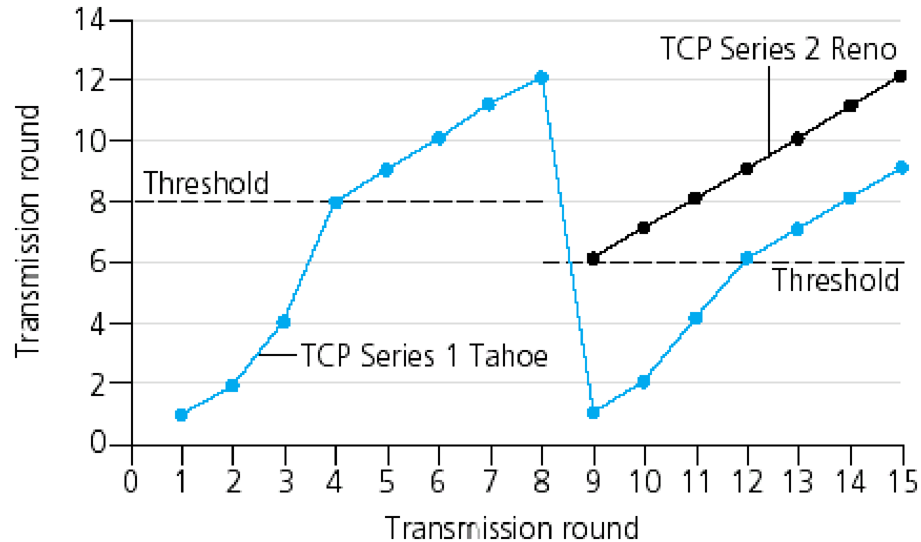
Congestion avoidance

```
/* slowstart is over       */
/* Congwin > threshold */
until (timeout) { /* loss event */
  every ACK:
      Congwin += 1/Congwin
  }
threshold = Congwin/2
Congwin = 1
perform slowstart
```



TCP Taheo

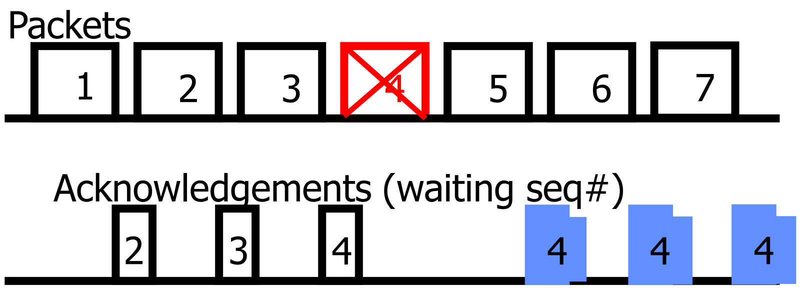
TCP Variants:



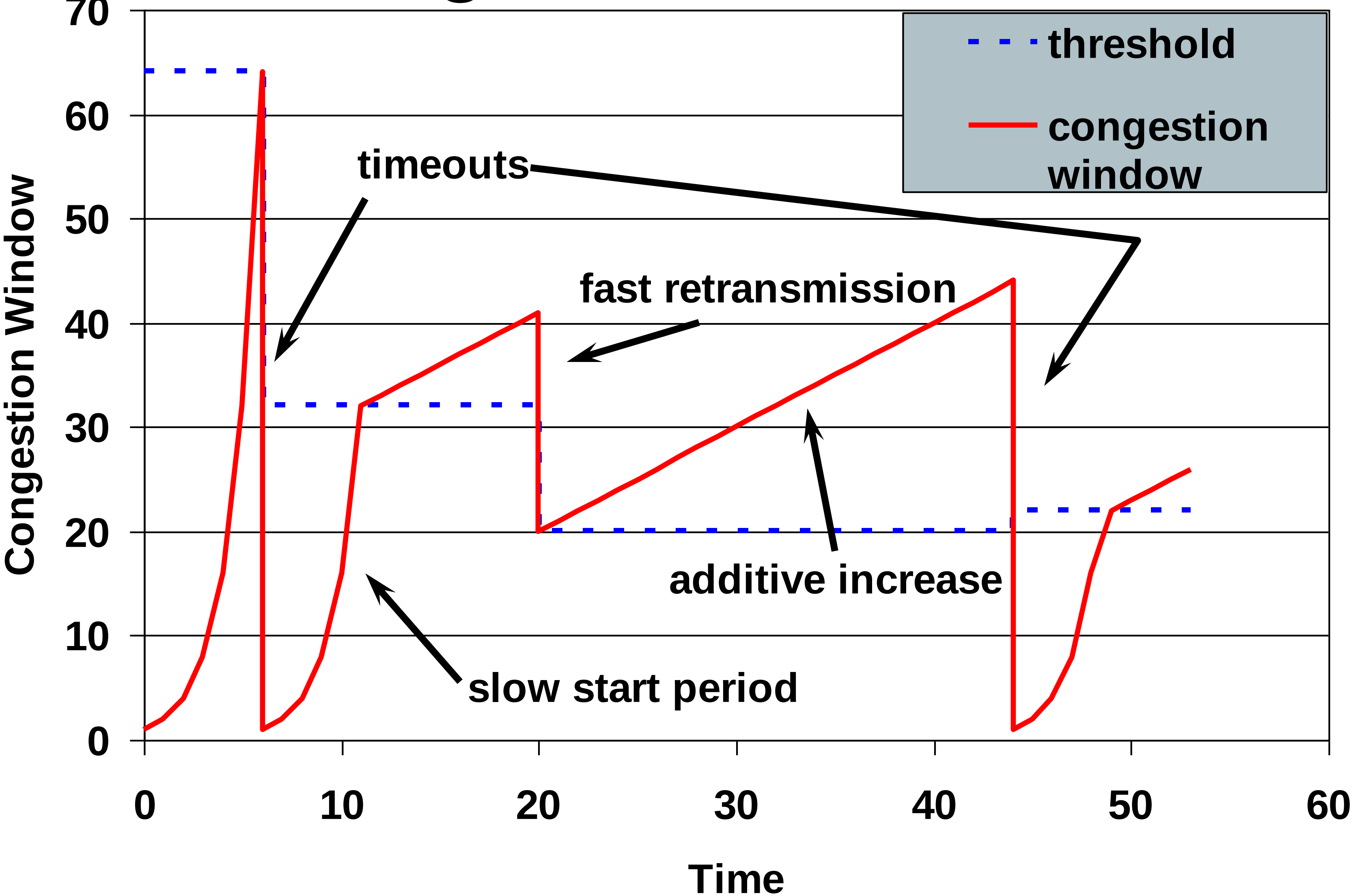
- TCP-Tahoe:
  - implements the slow start, congestion avoidance, and fast retransmit algorithms
- TCP-Reno:
  - implements the slow start, congestion avoidance, fast retransmit, and fast recovery algorithms
- Among other implementations are Vegas, NewReno (the most commonly implemented on webservers today, according to a survey) and SACK TCP.

Fast Retransmit

- Timeout period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - resend segment before timer expires



TCP Congestion Window Trace



Fast Recovery

- Fast recovery:
  - After retransmission do not enter slowstart.
  - Threshold = Congwin/2
  - Congwin = 3 + Congwin/2
  - Each duplicate ACK received Congwin++
  - After new ACK
    - Congwin = Threshold
    - return to congestion avoidance

Summary TCP

- Major transport service in the Internet
- Connection oriented
- Provides end-to-end reliability
- Includes facilities for flow/error control and congestion control
  - The latter can be costly / inappropriate for e.g. streaming applications:
  - Q: what do the latter do to cope with this?
  - A: **application-protocols manage this in contemporary solutions** [cf Kurose, Ross book, chapter multimedia]

Roadmap

- transport layer services (user, network perspectives)
- User perspective &connection: addressing, multiplexing/demultiplexing
- Reliable data transfer & TCP
  - User perspective
  - Network perspective (howto)
- Flow control
- TCP congestion control
  - Causes and basic goals; End to end control; TCP; setting timeouts;
- Summary and review questions

Summary TCP

- Major transport service in the Internet
- Connection oriented
- Provides end-to-end reliability
- Includes facilities for flow/error control and congestion control

Some Review Questions

- TCP uses a finite field to contain stream sequence numbers. Study the protocol specification to find out how it allows an arbitrary length stream to pass from one machine to another.
- Under what conditions of delay, bandwidth, load, and packet loss will TCP retransmit significant volumes of data unnecessarily?
- A lost TCP acknowledgement does not necessarily force retransmission. Explain why.

Review Questions

- Experiment with local machines to determine how TCP handles machine restart. Establish a connection (e.g., a remote login) and leave it idle. Wait for the destination machine to crash and restart, and then force the local machine to send a TCP segment (e.g., by typing characters to the remote login).
- Besides ack and retransmissions, search for other forms for reliable data transfer (e.g. forward-error-control, used by streaming applications)

Review Questions

- Imagine an implementation of TCP that discards segments that arrive out of order, even if they fall in the current window. That is, the imagined version only accepts segments that extend the byte stream it has already received. Does it work? How does it compare to a standard TCP implementation?
- Suppose an implementation of TCP uses initial sequence number 1 when it creates a connection. Explain how a system crash and restart can confuse a remote system into believing that the old connection remained open.

Review Questions

- Assume TCP is sending segments using a maximum window size (64 Kbytes) on a channel that has unbounded bandwidth and an average roundtrip time of 20 milliseconds. What is the maximum throughput? How does throughput change if the roundtrip time increases to 40 milliseconds (while bandwidth remains infinite)? For simplicity consider that the congestion-window is very large and does not impose any limitations.