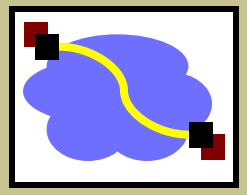
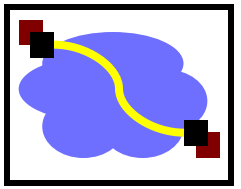


15-441 Computer Networking

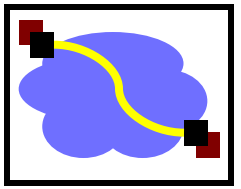
TCP & Congestion Control

Good Ideas So Far…



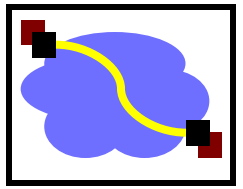
- Flow control
  - Stop & wait
  - Parallel stop & wait
  - Sliding window
- Loss recovery
  - Timeouts
  - Acknowledgement-driven recovery (selective repeat or cumulative acknowledgement)

Outline



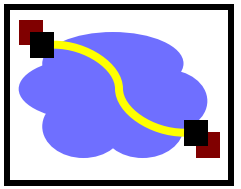
- TCP flow control
- Congestion sources and collapse
- Congestion control basics

Outline (the Halloween Version…)



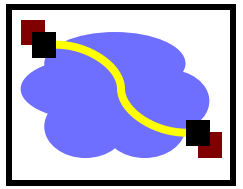
- THE SPOOKY PARTS of Transport Protocols
  - If it doesn't scare you now… it will on the Final!
  - TCP flow control
  - The Candy-exchange Protocol (TCP)
- Congestion sources and collapse
  - The horror of zombie networks
- Congestion control basics
  - Avoiding the death-traps of overloaded routers

Sequence Numbers (reminder)

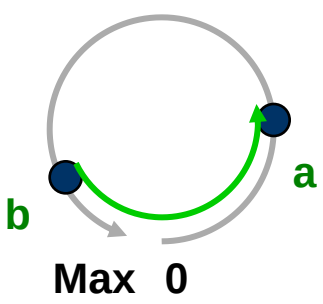


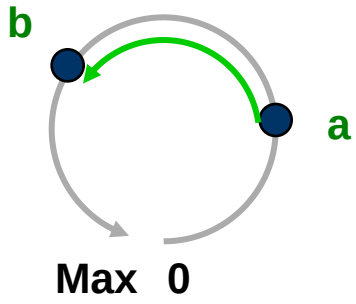
- How large do sequence numbers need to be?
  - Must be able to detect wrap-around
  - Depends on sender/receiver window size
- E.g.
  - Max seq = 7, send win=recv win=7
  - If pkts 0..6 are sent succesfully and all acks lost
    - Receiver expects 7,0..5, sender retransmits old 0..6!!!
- Max sequence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

Sequence Numbers



- 32 Bits, Unsigned → for bytes not packets!
  - Circular Comparison

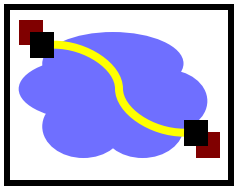




- Why So Big?
  - For sliding window, must have

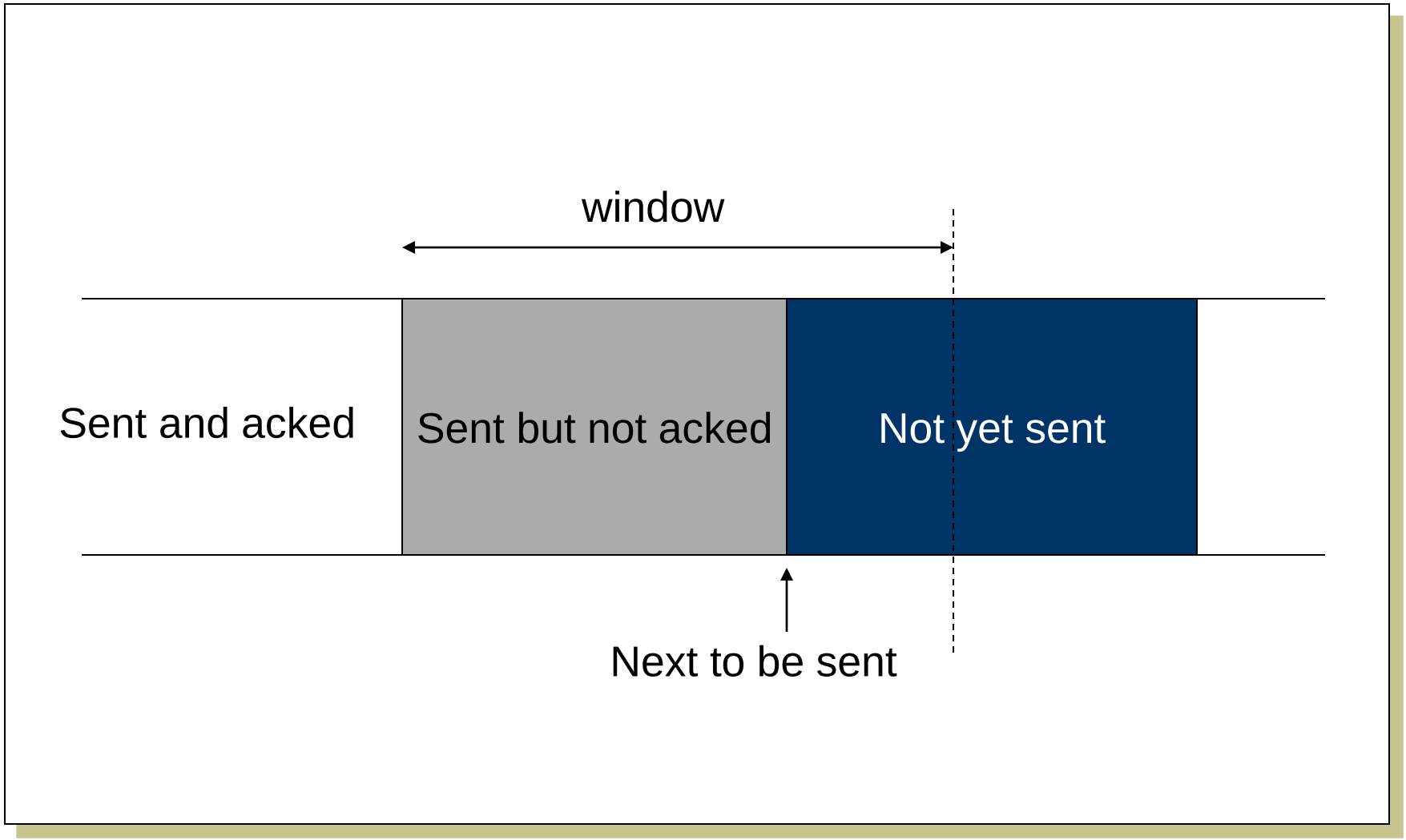
    |Sequence Space| > |Sending Window| + |Receiving Window|
    - No problem
  - Also, want to guard against stray packets
    - With IP, packets have maximum lifetime of 120s
    - Sequence number would wrap around in this time at 286MB/s

TCP Flow Control

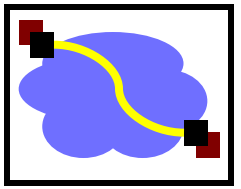


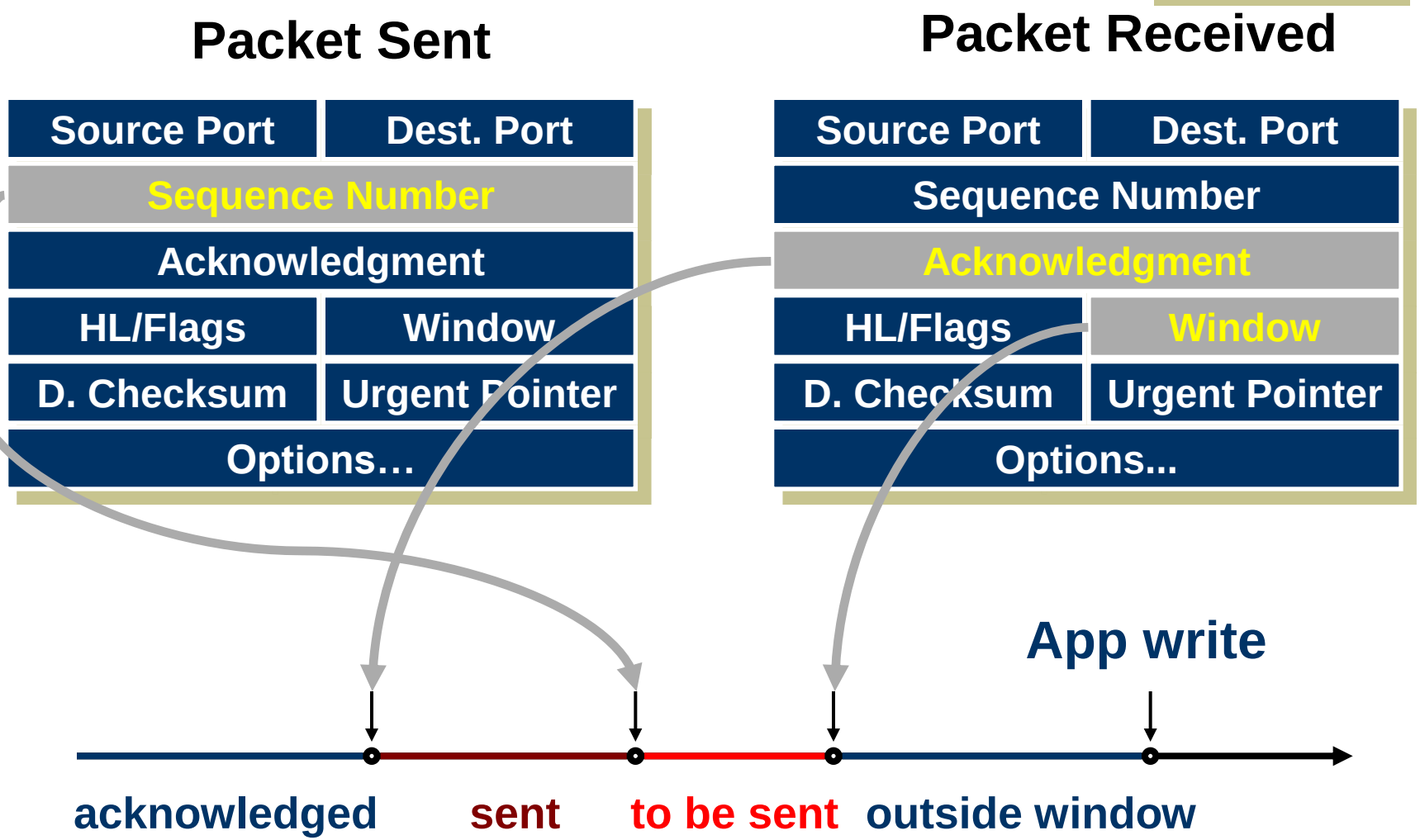
- TCP is a sliding window protocol
  - For window size *n*, can send up to *n* bytes without receiving an acknowledgement
  - When the data is acknowledged then the window slides forward
- Each packet advertises a window size
  - Indicates number of bytes the receiver has space for
- Original TCP always sent entire window
  - Congestion control now limits this

Window Flow Control: Send Side

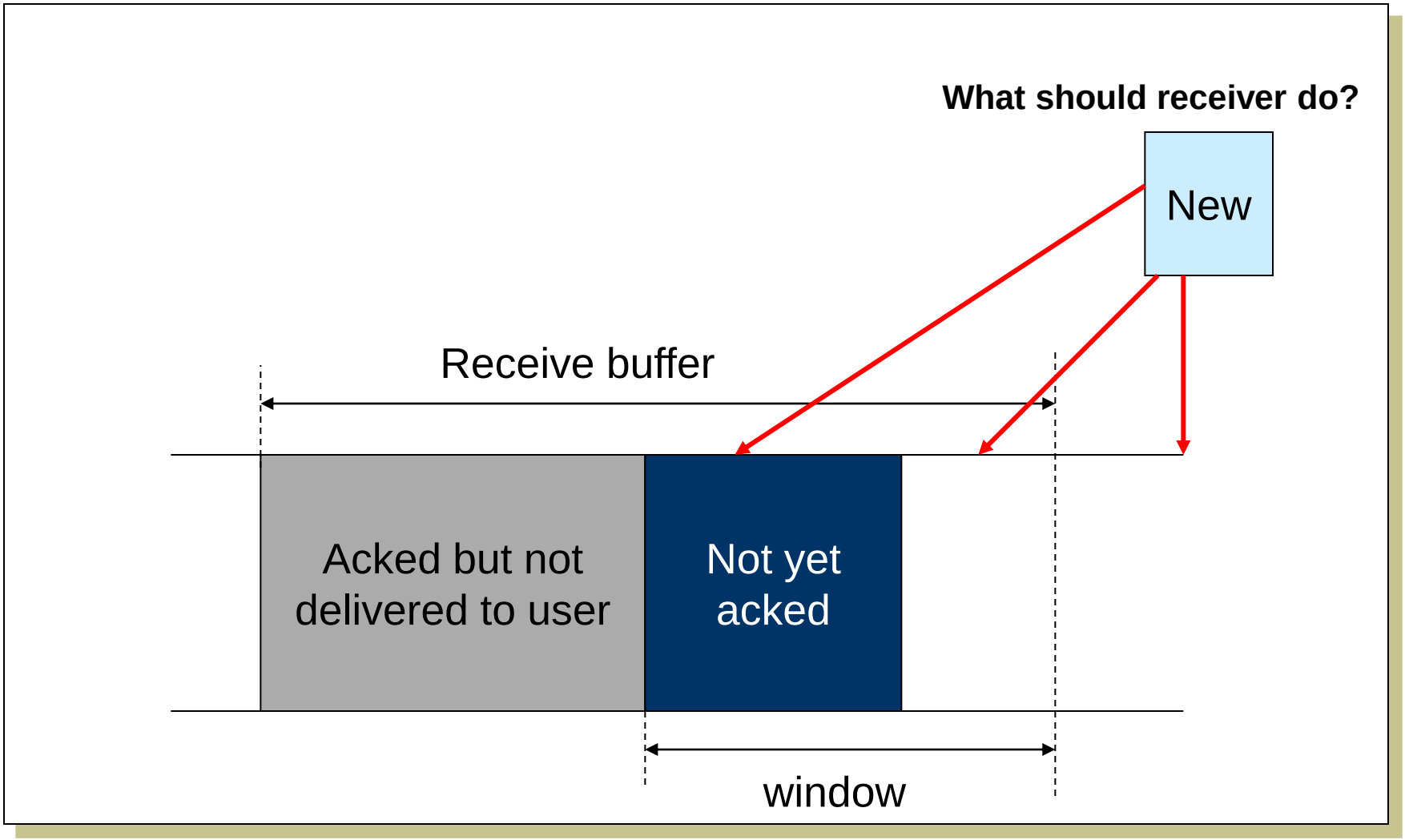


Window Flow Control: Send Side

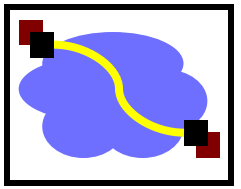




Window Flow Control: Receive Side

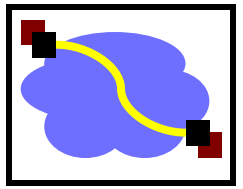


TCP Persist



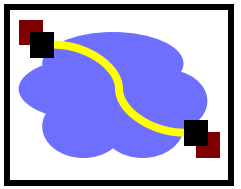
- What happens if window is 0?
  - Receiver updates window when application reads data
  - What if this update is lost?
- TCP Persist state
  - Sender periodically sends 1 byte packets
  - Receiver responds with ACK even if it can't store the packet

Performance Considerations



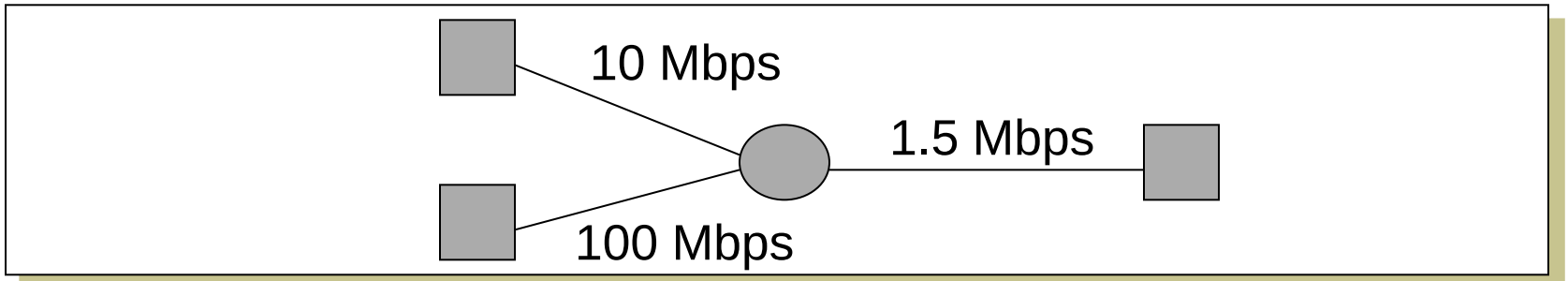
- The window size can be controlled by receiving application
  - Can change the socket buffer size from a default (e.g. 8Kbytes) to a maximum value (e.g. 64 Kbytes)
- The window size field in the TCP header limits the window that the receiver can advertise
  - 16 bits → 64 KBytes
  - 10 msec RTT → 51 Mbit/second
  - 100 msec RTT → 5 Mbit/second
  - TCP options to get around 64KB limit → increases above limit

Outline



- TCP flow control
- Congestion sources and collapse
- Congestion control basics

Congestion

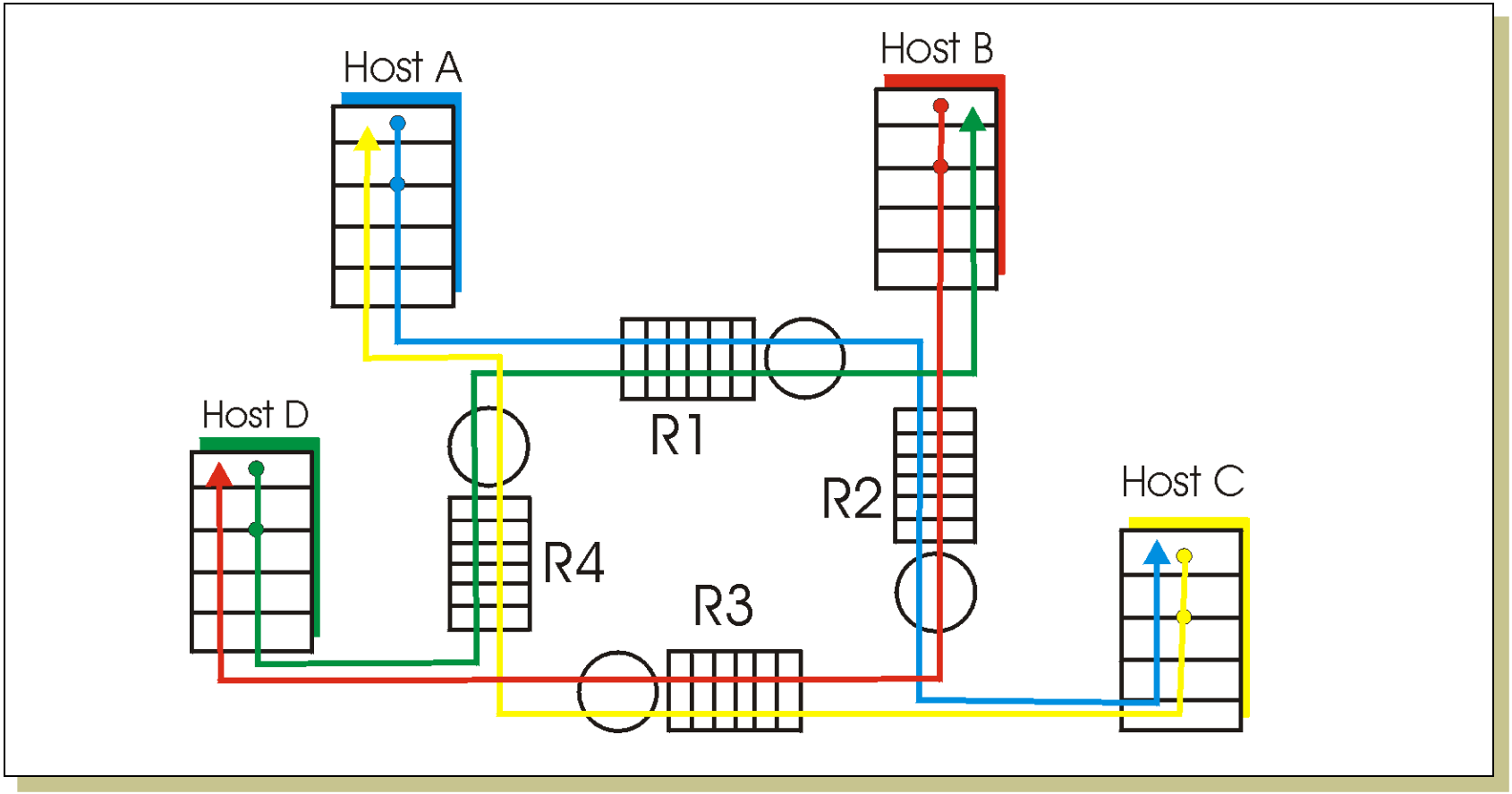


- Different sources compete for resources inside network
- Why is it a problem?
  - Sources are unaware of current state of resource
  - Sources are unaware of each other
- Manifestations:
  - Lost packets (buffer overflow at routers)
  - Long delays (queuing in router buffers)
  - Can result in throughput less than bottleneck link (1.5Mbps for the above topology) → a.k.a. congestion collapse

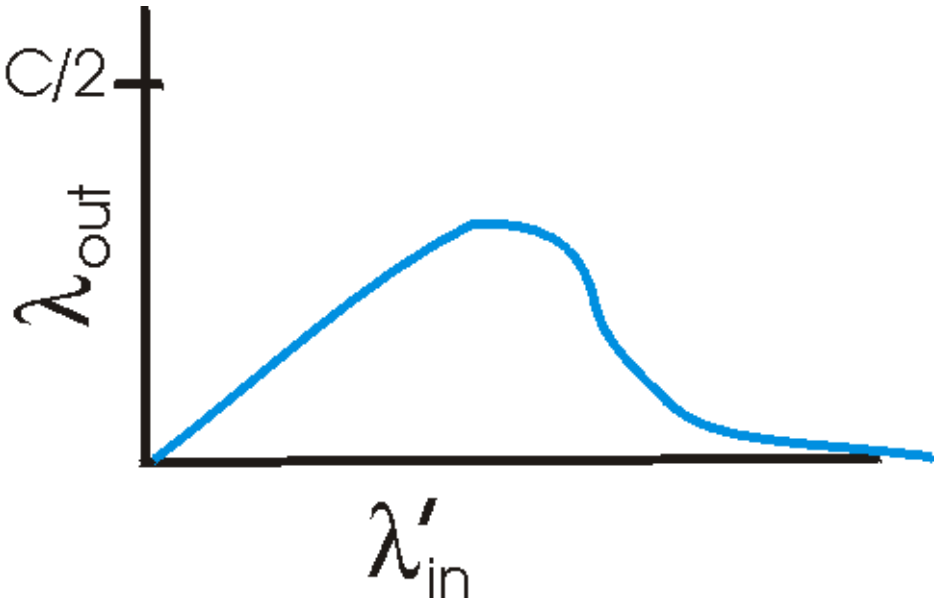
Causes & Costs of Congestion

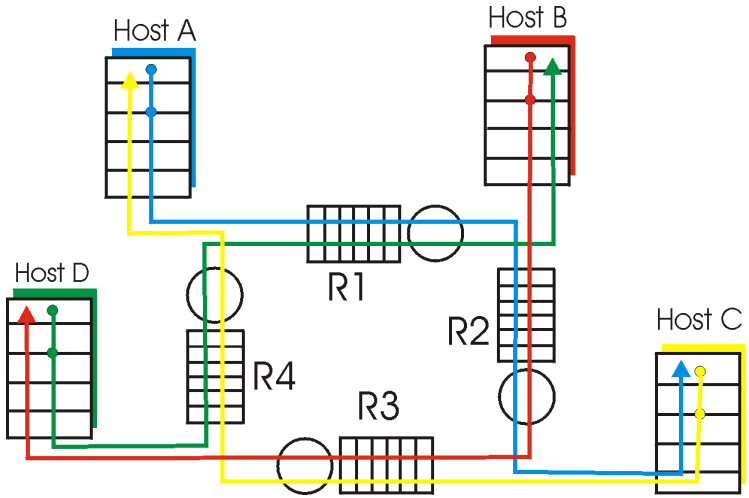
- Four senders – multihop paths
- Timeout/retransmit

Q: What happens as rate increases?



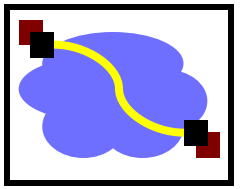
Causes & Costs of Congestion





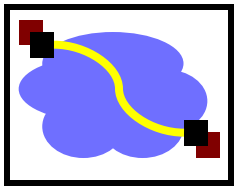
- When packet dropped, any "upstream transmission capacity used for that packet was wasted!

Congestion Collapse



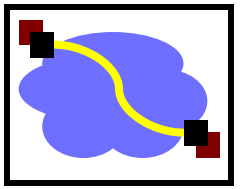
- Definition: *Increase in network load results in decrease of useful work done*
- Many possible causes
  - Spurious retransmissions of packets still in flight
    - Classical congestion collapse
    - How can this happen with packet conservation
    - Solution: better timers and TCP congestion control
  - Undelivered packets
    - Packets consume resources and are dropped elsewhere in network
    - Solution: congestion control for ALL traffic

Congestion Control and Avoidance



- A mechanism which:
  - Uses network resources efficiently
  - Preserves fair network resource allocation
  - Prevents or avoids collapse
- Congestion collapse is not just a theory
  - Has been frequently observed in many networks

Approaches Towards Congestion Control

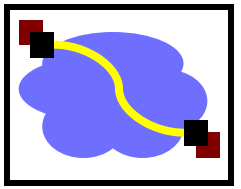


- Two broad approaches towards congestion control:

- End-end congestion control:
  - No explicit feedback from network
  - Congestion inferred from end-system observed loss, delay
  - Approach taken by TCP

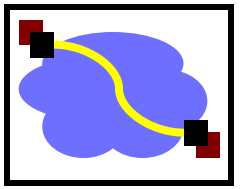
- Network-assisted congestion control:
  - Routers provide feedback to end systems
    - Single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
    - Explicit rate sender should send at
  - Problem: makes routers complicated

Example: TCP Congestion Control



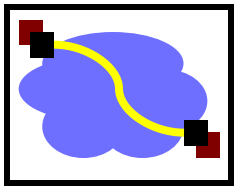
- Very simple mechanisms in network
  - FIFO scheduling with shared buffer pool
  - Feedback through packet drops
- TCP interprets packet drops as signs of congestion and slows down
  - This is an assumption: packet drops are not a sign of congestion in all networks
    - E.g. wireless networks
- Periodically probes the network to check whether more bandwidth has become available.

Outline



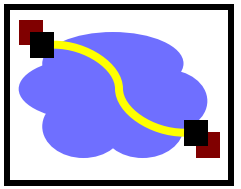
- TCP flow control
- Congestion sources and collapse
- Congestion control basics

Objectives



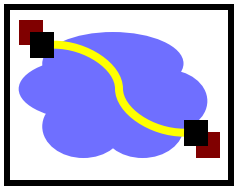
- Simple router behavior
- Distributedness
- Efficiency: $X = \Sigma x_i(t)$
- Fairness: $(\Sigma x_i)^2/n(\Sigma x_i^2)$
  - What are the important properties of this function?
- Convergence: control system must be stable

Basic Control Model



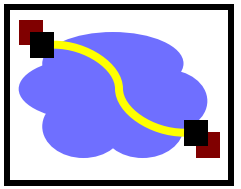
- Reduce speed when congestion is perceived
  - How is congestion signaled?
    - Either mark or drop packets
  - How much to reduce?
- Increase speed otherwise
  - Probe for available bandwidth – how?

Linear Control

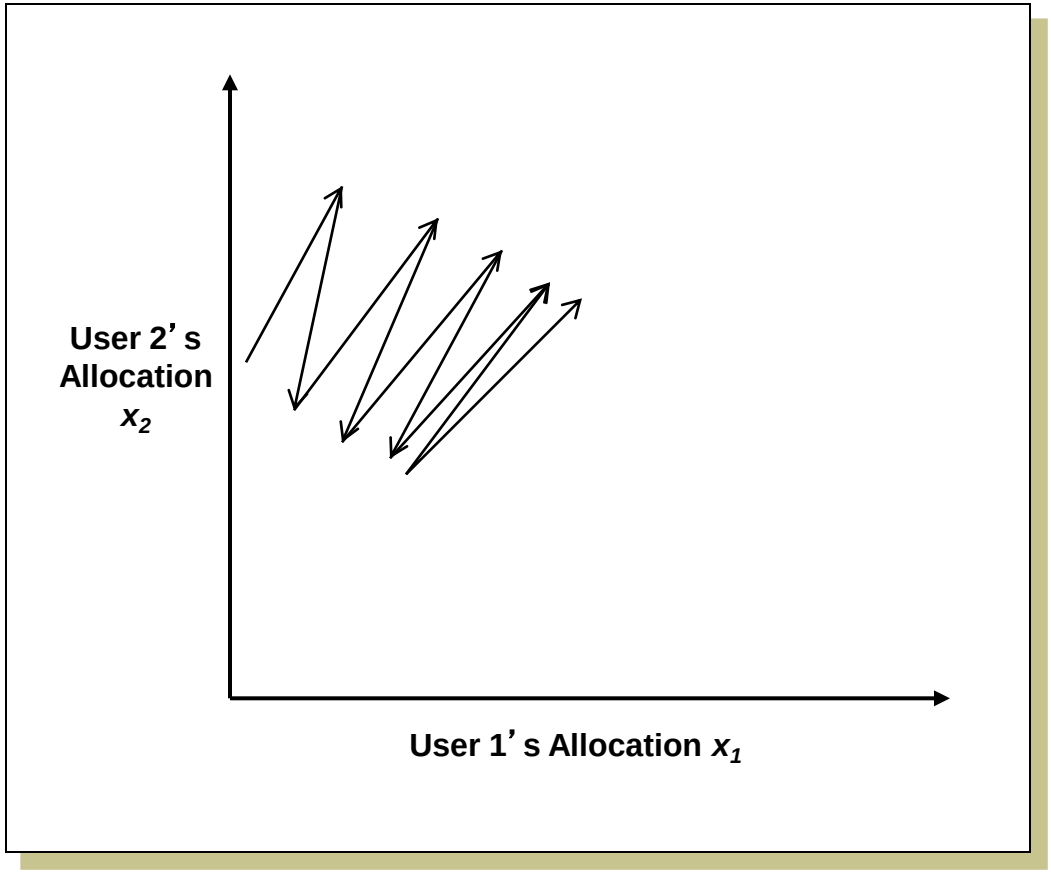


- Many different possibilities for reaction to congestion and probing
  - Examine simple linear controls
    - Window(t + 1) = a + b Window(t)
    - Different a_i/b_i for increase and a_d/b_d for decrease
- Supports various reaction to signals
  - Increase/decrease additively
  - Increased/decrease multiplicatively
  - Which of the four combinations is optimal?

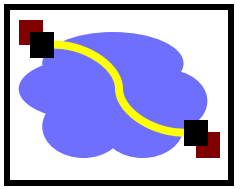
Phase Plots



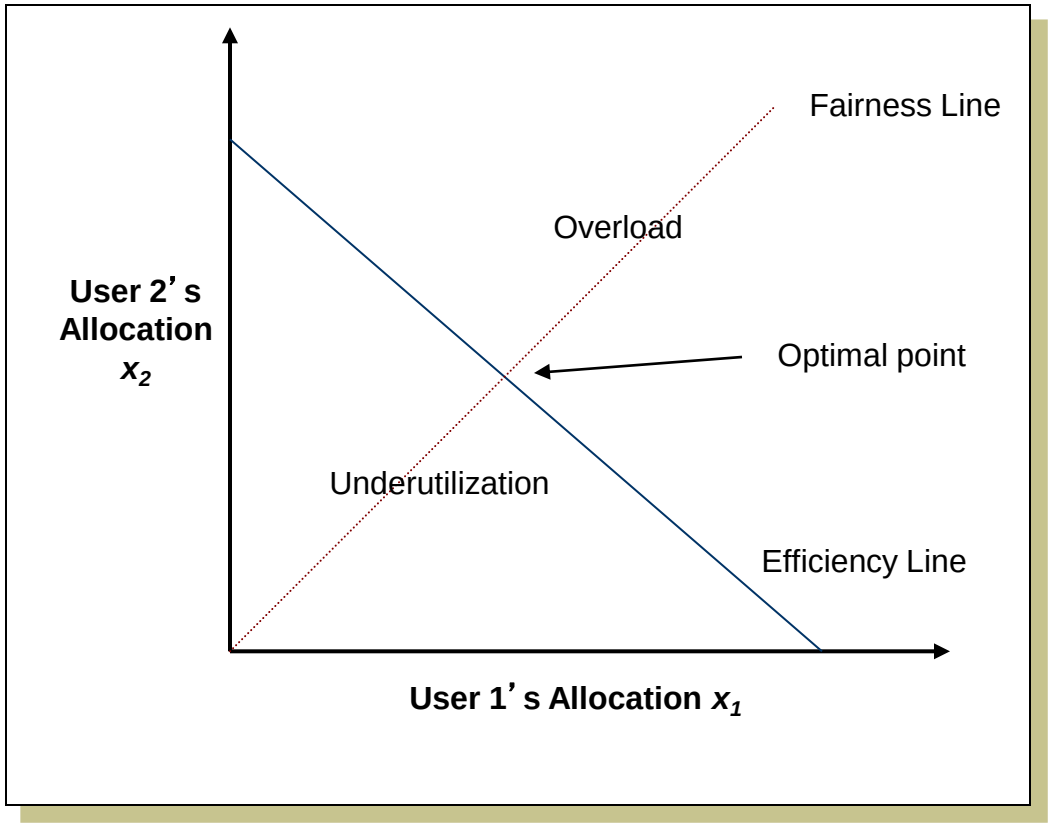
- Simple way to visualize behavior of competing connections over time



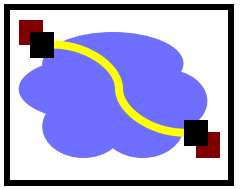
Phase Plots



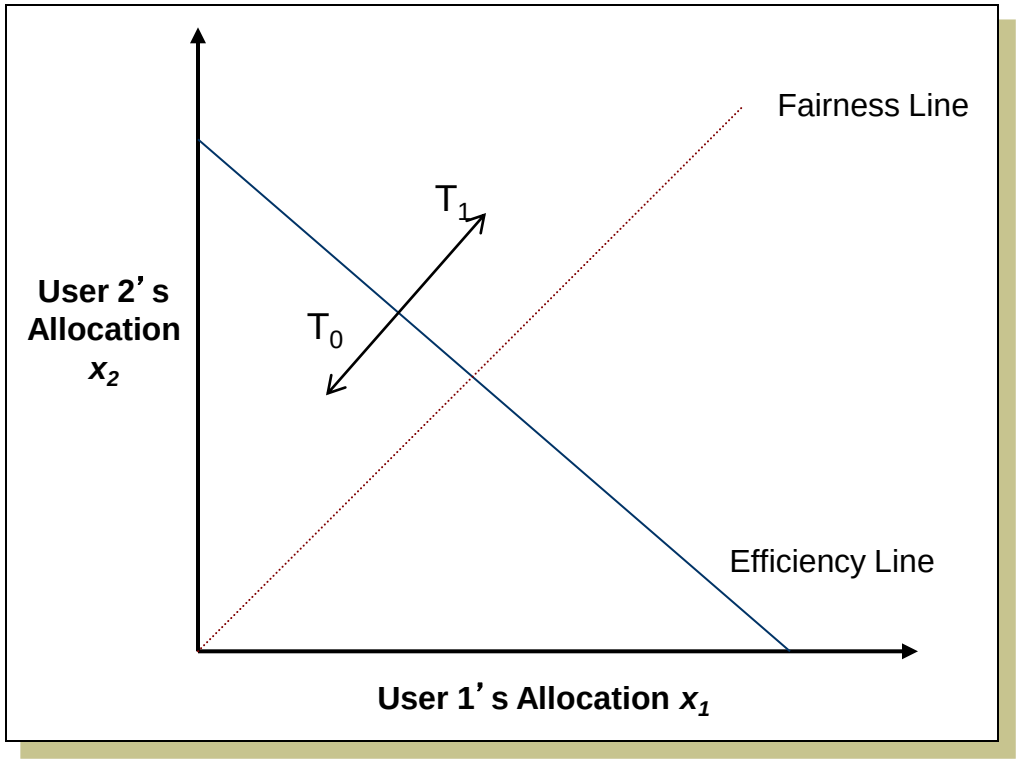
- What are desirable properties?
- What if flows are not equal?



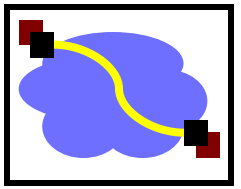
Additive Increase/Decrease



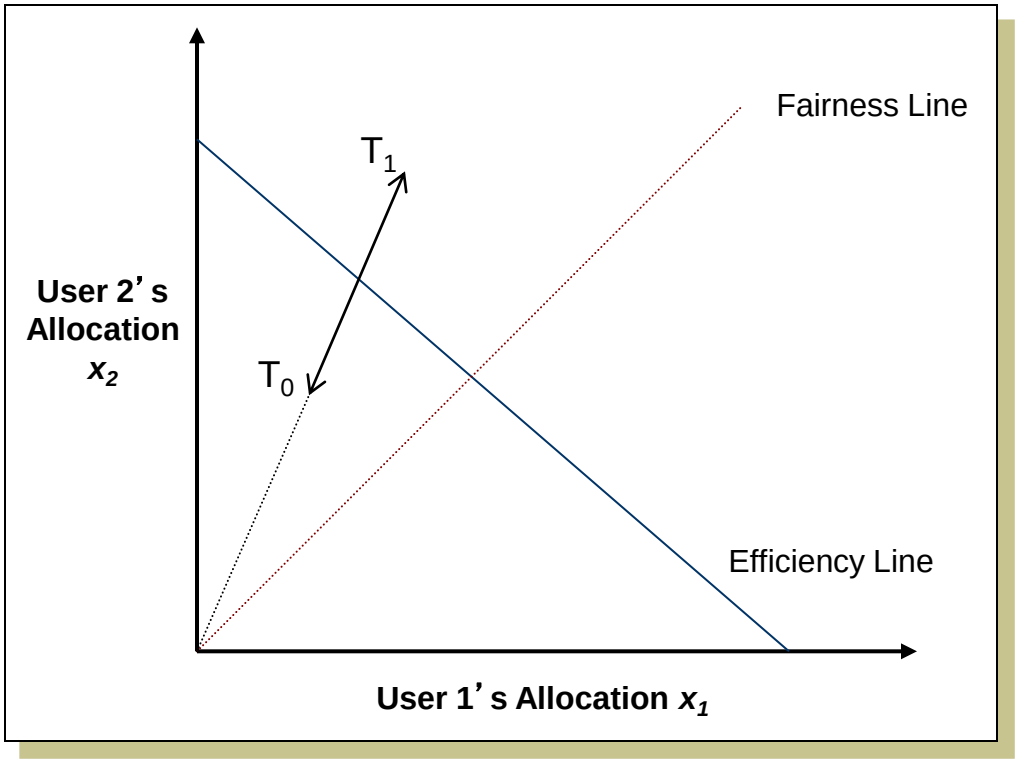
- Both X_1 and X_2 increase/ decrease by the same amount over time
  - Additive increase improves fairness and additive decrease reduces fairness



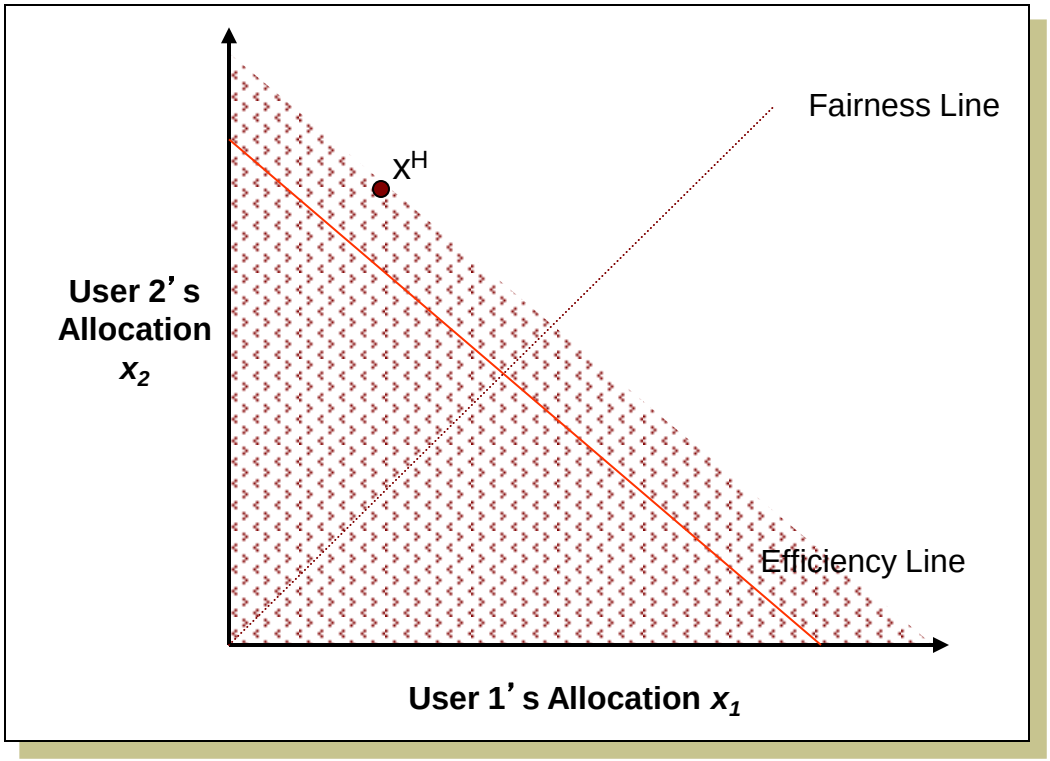
Muliplicative Increase/Decrease



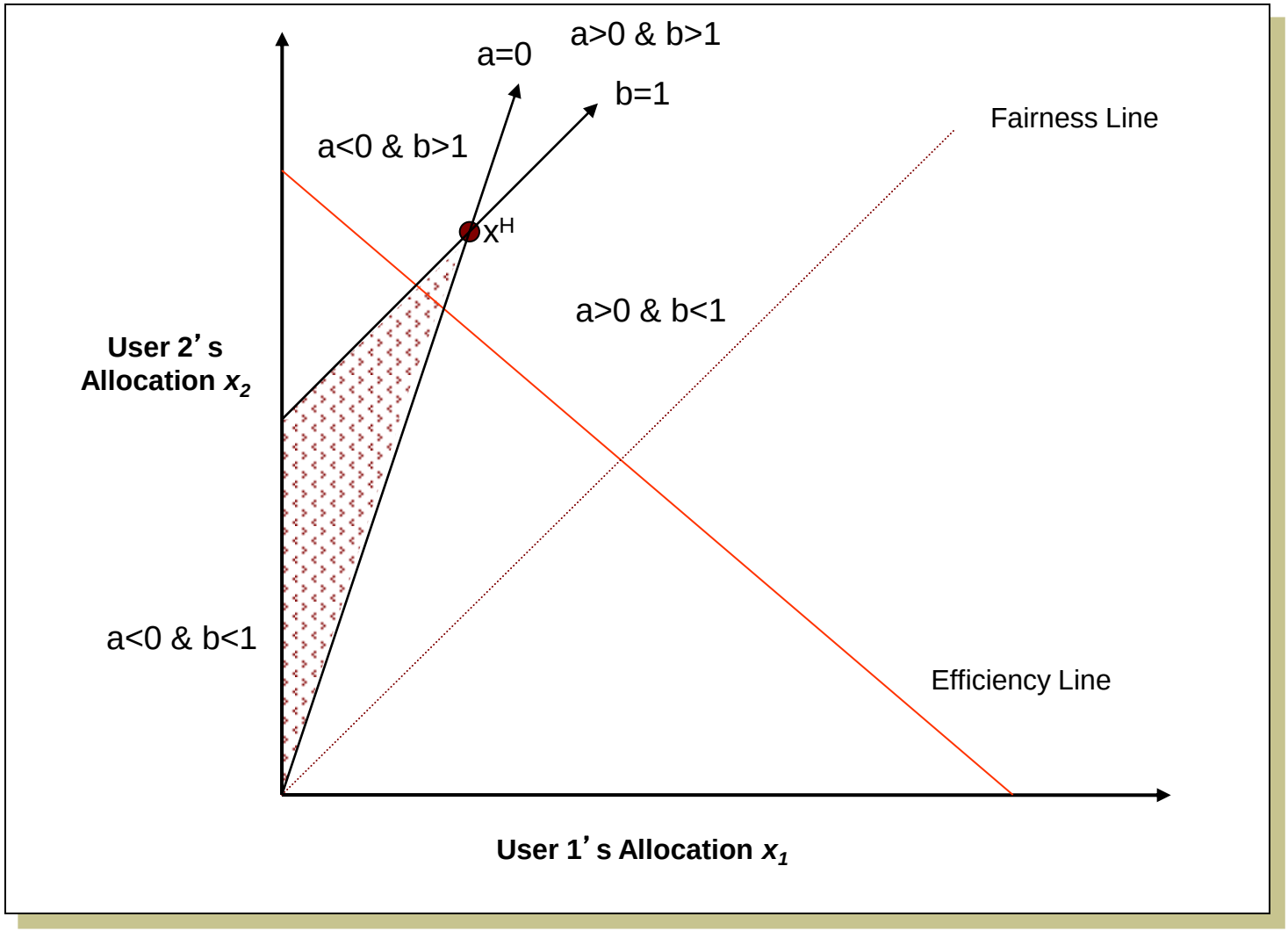
- Both X_1 and X_2 increase by the same factor over time
  - Extension from origin – constant fairness



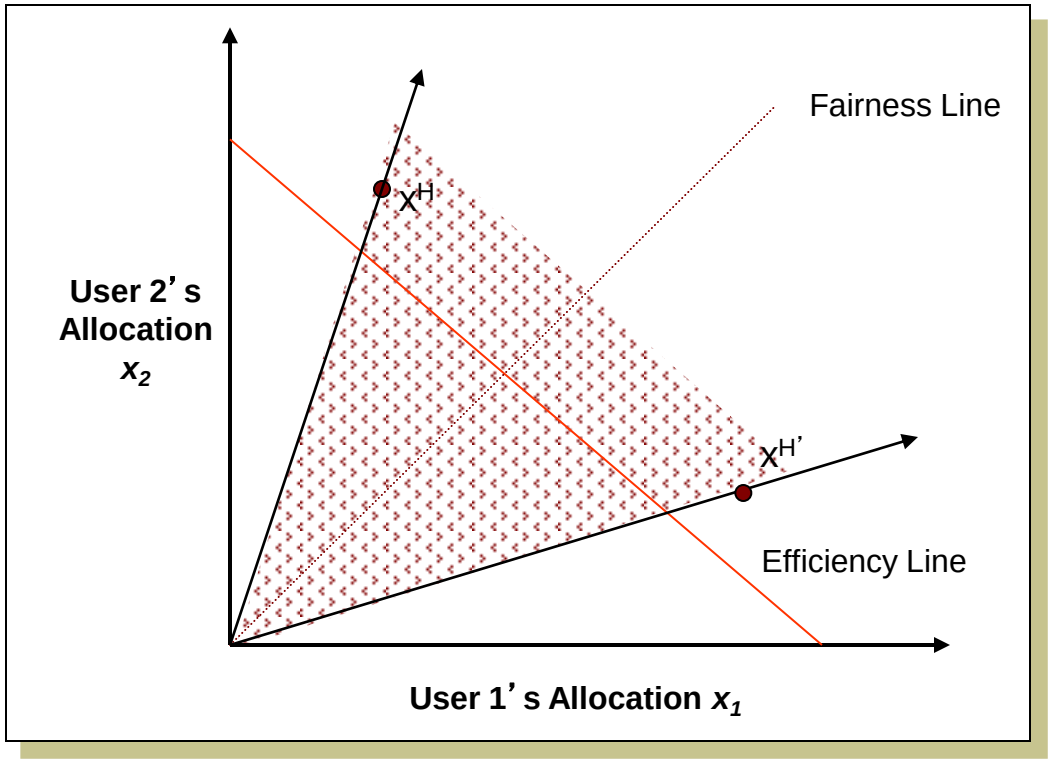
Convergence to Efficiency



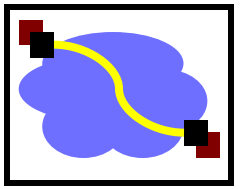
Distributed Convergence to Efficiency



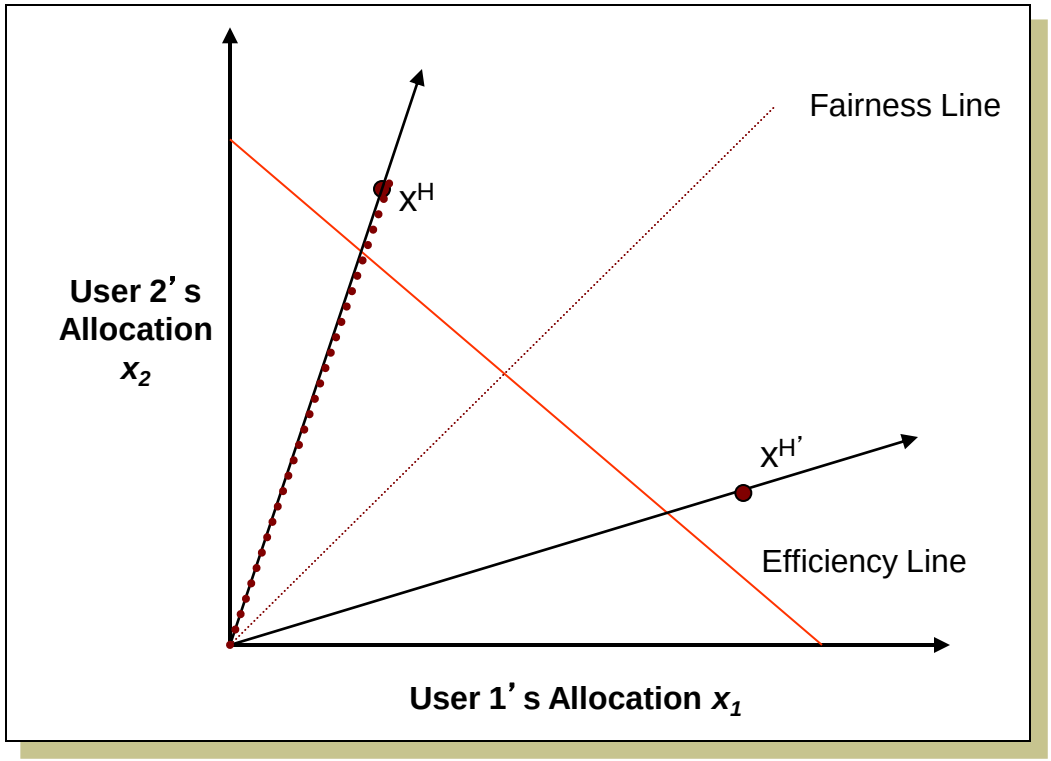
Convergence to Fairness



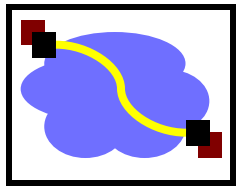
Convergence to Efficiency & Fairness



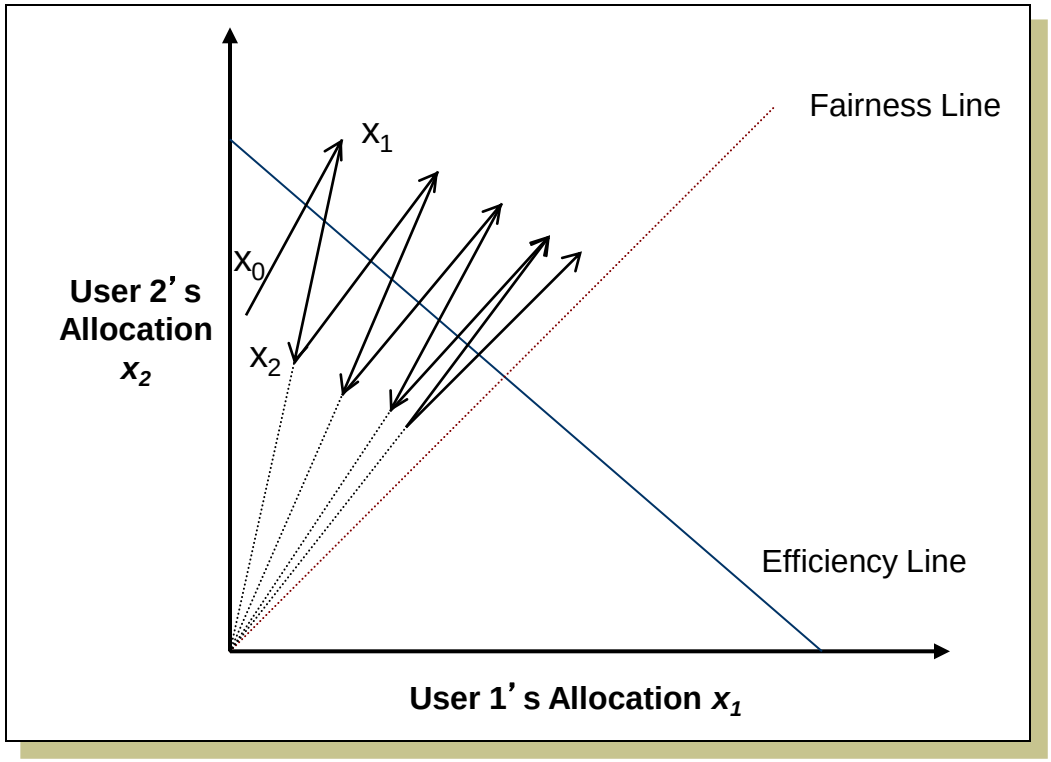
- Intersection of valid regions
- For decrease: a=0 & b < 1



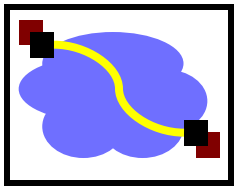
What is the Right Choice?



- Constraints limit us to AIMD
  - Can have multiplicative term in increase (MAIMD)
  - AIMD moves towards optimal point

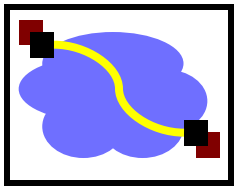


Important Lessons



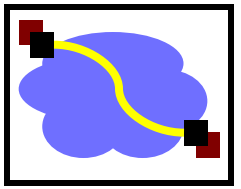
- Transport service
  - UDP → mostly just IP service
  - TCP → congestion controlled, reliable, byte stream
- Types of ARQ protocols
  - Stop-and-wait → slow, simple
  - Go-back-n → can keep link utilized (except w/ losses)
  - Selective repeat → efficient loss recovery
- Sliding window flow control
- TCP flow control
  - Sliding window → mapping to packet headers
  - 32bit sequence numbers (bytes)

Important Lessons



- Why is congestion control needed?
- How to evaluate congestion control algorithms?
  - Why is AIMD the right choice for congestion control?
- TCP flow control
  - Sliding window → mapping to packet headers
  - 32bit sequence numbers (bytes)

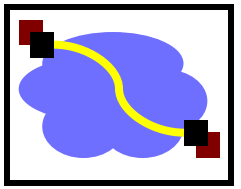
Good Ideas So Far…



- Flow control
  - Stop & wait
  - Parallel stop & wait
  - Sliding window (e.g., advertised windows)
- Loss recovery
  - Timeouts
  - Acknowledgement-driven recovery (selective repeat or cumulative acknowledgement)
- Congestion control
  - AIMD → fairness and efficiency

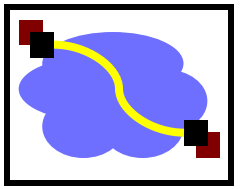
- Next Lecture: How does TCP actually implement these?

Outline

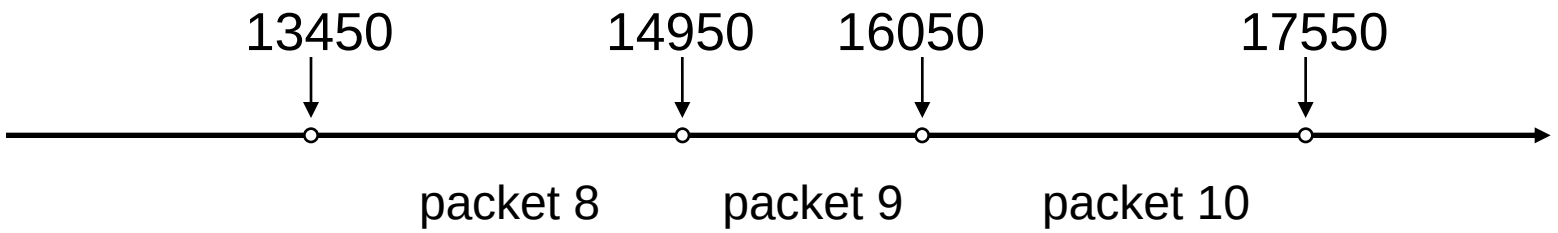


- TCP connection setup/data transfer
- TCP reliability
- TCP congestion avoidance

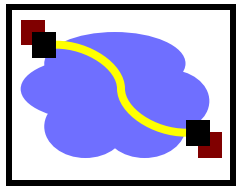
Sequence Number Space



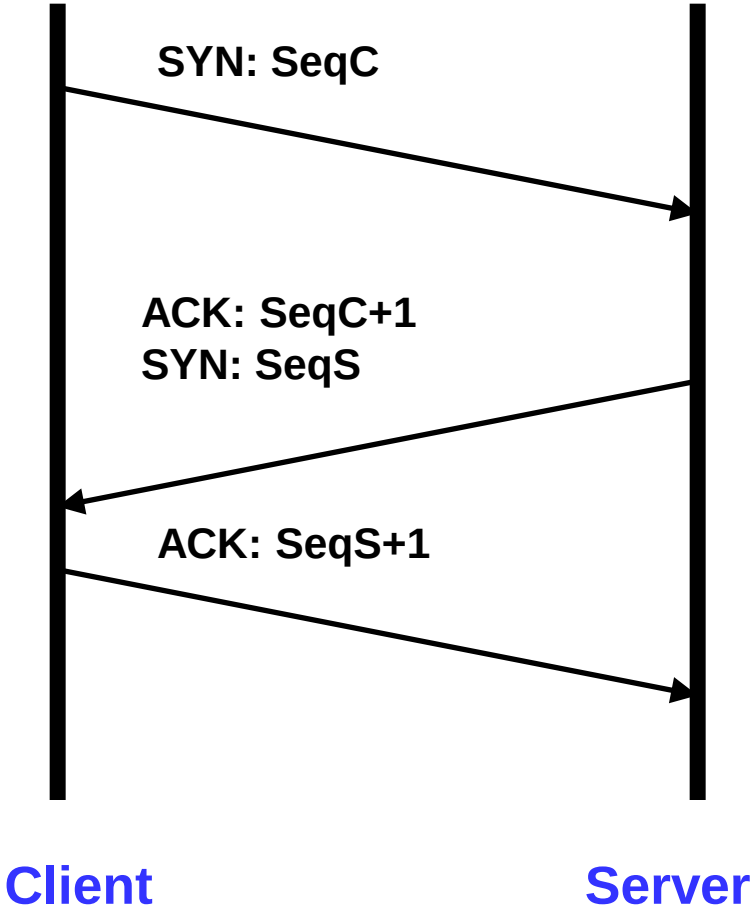
- Each byte in byte stream is numbered.
  - 32 bit value
  - Wraps around
  - Initial values selected at start up time
- TCP breaks up the byte stream into packets.
  - Packet size is limited to the Maximum Segment Size
- Each packet has a sequence number.
  - Indicates where it fits in the byte stream



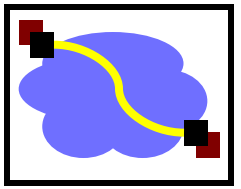
Establishing Connection: Three-Way handshake



- Each side notifies other of starting sequence number it will use for sending
  - Why not simply chose 0?
    - Must avoid overlap with earlier incarnation
    - Security issues
- Each side acknowledges other's sequence number
  - SYN-ACK: Acknowledge sequence number + 1
- Can combine second SYN with first ACK



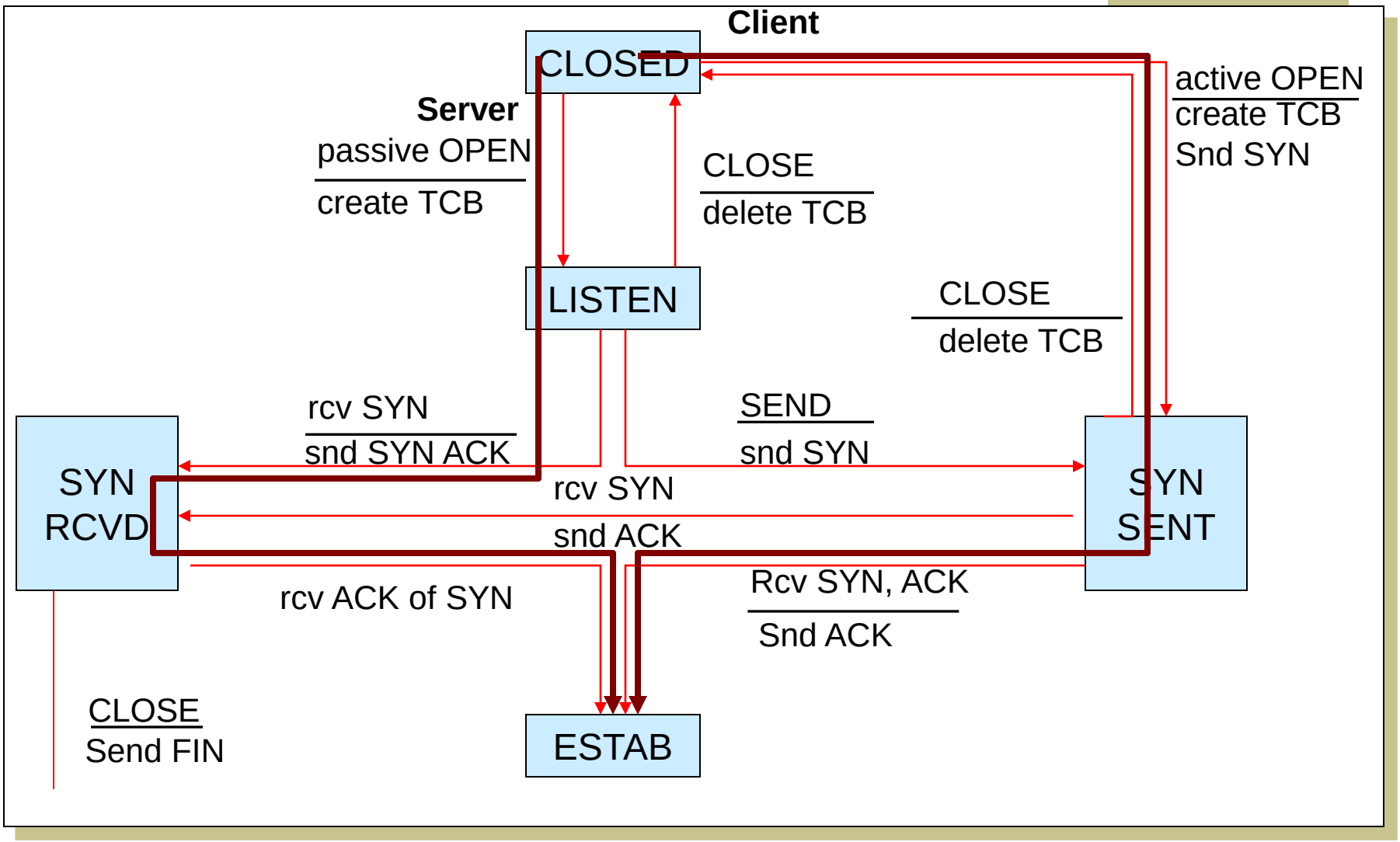
TCP Connection Setup Example



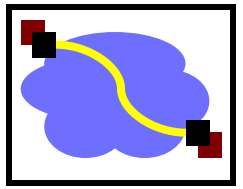
```
09:23:33.042318 IP 128.2.222.198.3123 > 192.216.219.96.80:
S 4019802004:4019802004(0) win 65535
<mss 1260,nop,nop,sackOK> (DF)

09:23:33.118329 IP 192.216.219.96.80 > 128.2.222.198.3123:
S 3428951569:3428951569(0) ack 4019802005 win 5840
<mss 1460,nop,nop,sackOK> (DF)

09:23:33.118405 IP 128.2.222.198.3123 > 192.216.219.96.80:
. ack 3428951570 win 65535 (DF)
```

- Client SYN
  - SeqC: Seq. #4019802004, window 65535, max. seg. 1260
- Server SYN-ACK+SYN
  - Receive: #4019802005 (= SeqC+1)
  - SeqS: Seq. #3428951569, window 5840, max. seg. 1460
- Client SYN-ACK
  - Receive: #3428951570 (= SeqS+1)

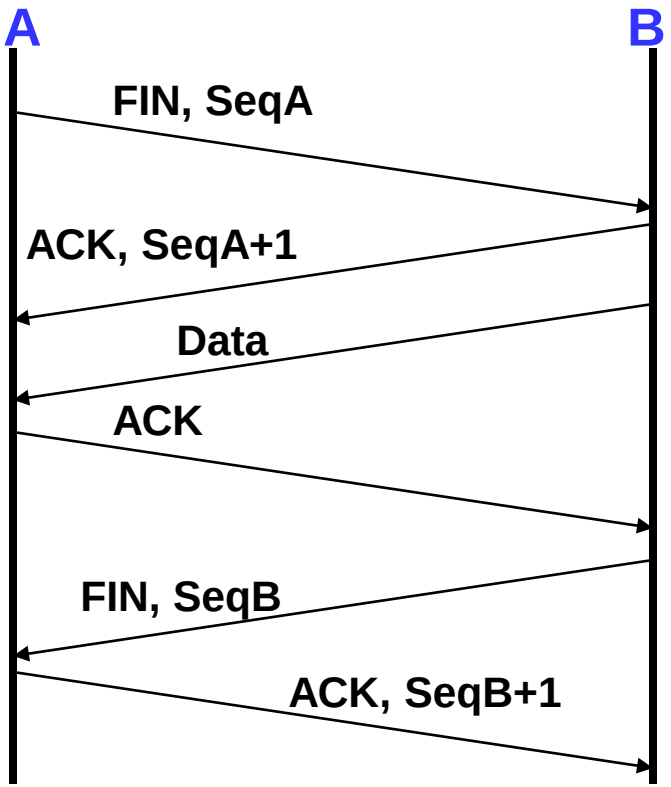
TCP State Diagram: Connection Setup



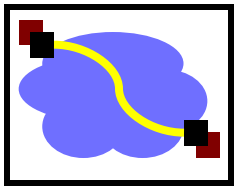
Tearing Down Connection



- Either side can initiate tear down
  - Send FIN signal
  - “I’m not going to send any more data”
- Other side can continue sending data
  - Half open connection
  - Must continue to acknowledge
- Acknowledging FIN
  - Acknowledge last sequence number + 1



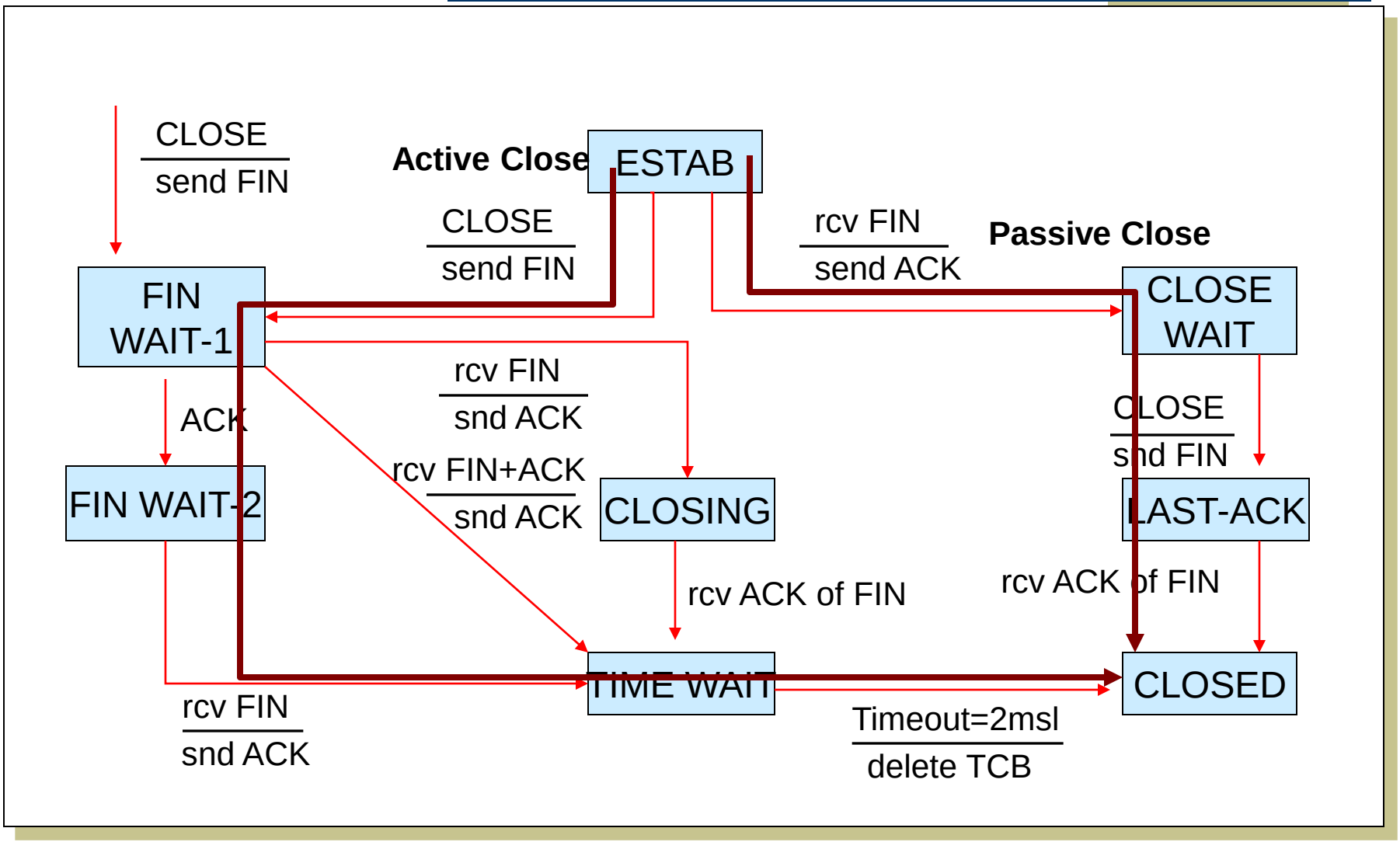
TCP Connection Teardown Example



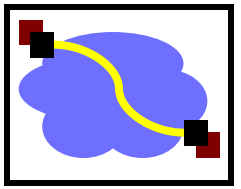
```
09:54:17.585396 IP 128.2.222.198.4474 > 128.2.210.194.6616:
F 1489294581:1489294581(0) ack 1909787689 win 65434 (DF)

09:54:17.585732 IP 128.2.210.194.6616 > 128.2.222.198.4474:
F 1909787689:1909787689(0) ack 1489294582 win 5840 (DF)

09:54:17.585764 IP 128.2.222.198.4474 > 128.2.210.194.6616:
. ack 1909787690 win 65434 (DF)
```

- Session
  - Echo client on 128.2.222.198, server on 128.2.210.194
- Client FIN
  - SeqC: 1489294581
- Server ACK + FIN
  - Ack: 1489294582 (= SeqC+1)
  - SeqS: 1909787689
- Client ACK
  - Ack: 1909787690 (= SeqS+1)

State Diagram: Connection Tear-down

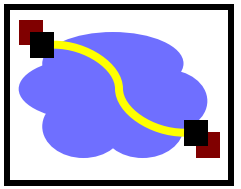


Outline



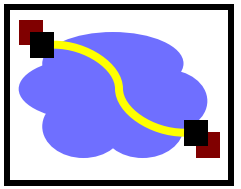
- TCP connection setup/data transfer
- TCP reliability
- TCP congestion avoidance

Reliability Challenges



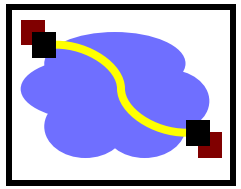
- Congestion related losses
- Variable packet delays
  - What should the timeout be?
- Reordering of packets
  - How to tell the difference between a delayed packet and a lost one?

TCP = Go-Back-N Variant



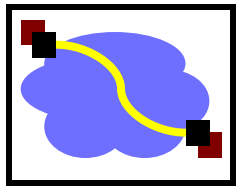
- Sliding window with cumulative acks
  - Receiver can only return a single “ack” sequence number to the sender.
  - Acknowledges all bytes with a lower sequence number
  - Starting point for retransmission
  - Duplicate acks sent when out-of-order packet received
- But: sender only retransmits a single packet.
  - Reason???
    - Only one that it knows is lost
    - Network is congested → shouldn’t overload it
- Error control is based on byte sequences, not packets.
  - Retransmitted packet can be different from the original lost packet – Why?

Round-trip Time Estimation

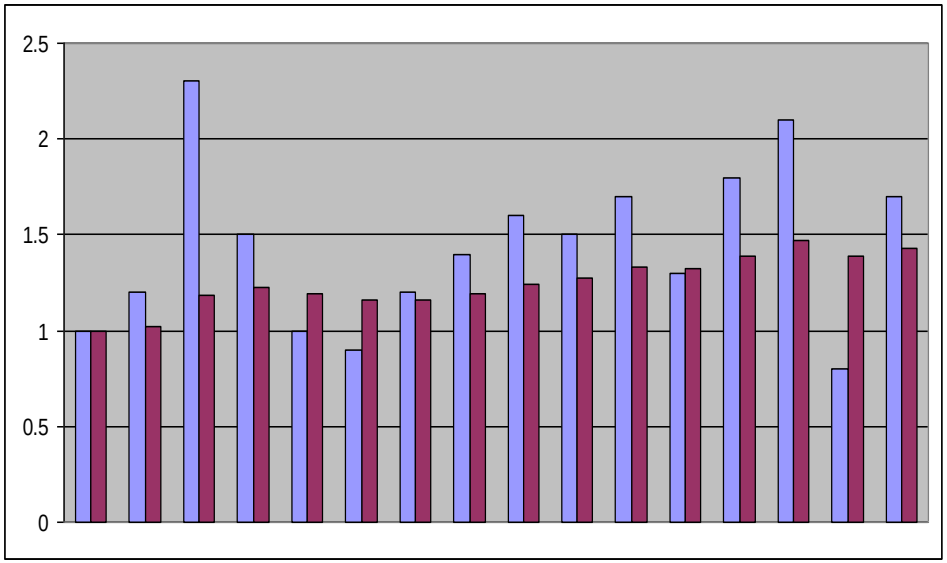


- Wait at least one RTT before retransmitting
- Importance of accurate RTT estimators:
  - Low RTT estimate
    - unneeded retransmissions
  - High RTT estimate
    - poor throughput
- RTT estimator must adapt to change in RTT
  - But not too fast, or too slow!
- Spurious timeouts
  - “Conservation of packets” principle – never more than a window worth of packets in flight

Original TCP Round-trip Estimator

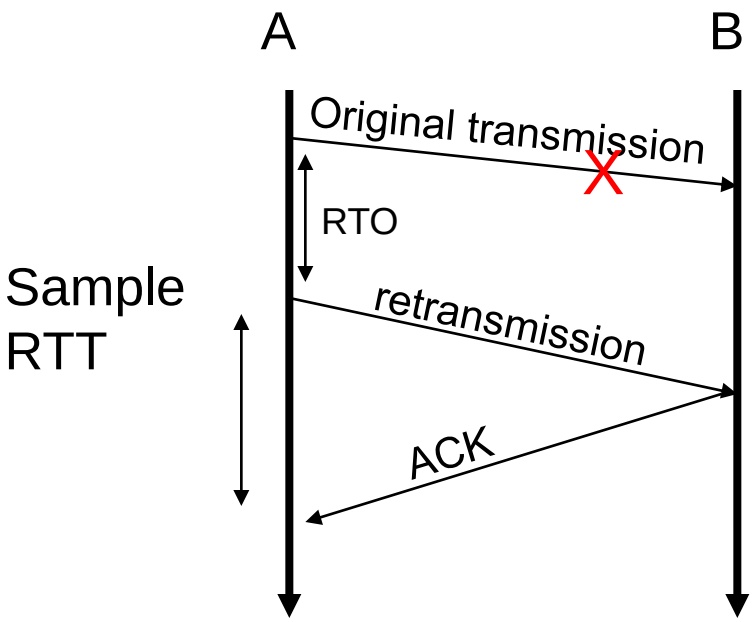


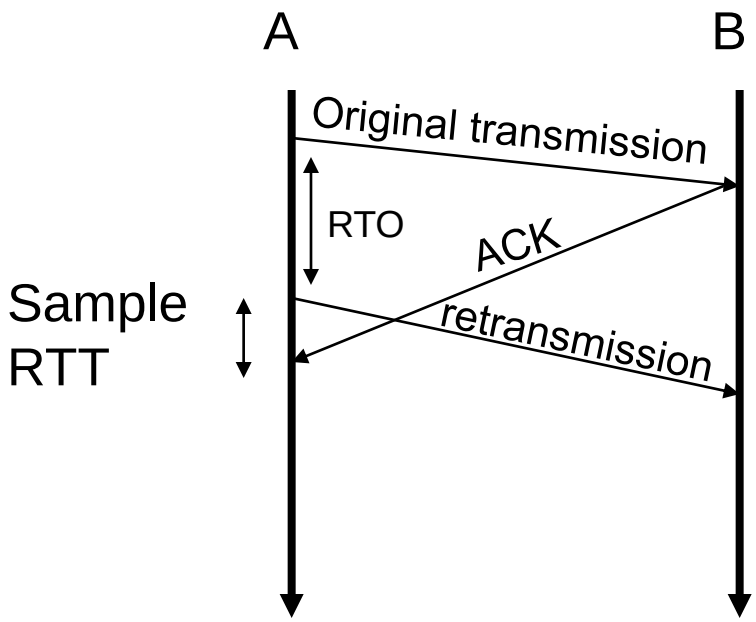
- Round trip times exponentially averaged:
  - New RTT = α (old RTT) + (1 - α) (new sample)
  - Recommended value for α: 0.8 - 0.9
    - 0.875 for most TCP's



- Retransmit timer set to (b * RTT), where b = 2
  - Every time timer expires, RTO exponentially backed-off
- Not good at preventing spurious timeouts
  - Why?

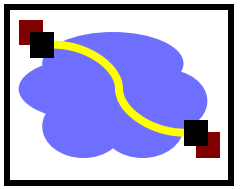
RTT Sample Ambiguity





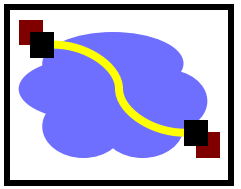
- Karn's RTT Estimator
  - If a segment has been retransmitted:
    - Don't count RTT sample on ACKs for this segment
    - Keep backed off time-out for next packet
    - Reuse RTT estimate only after one successful transmission

Jacobson's Retransmission Timeout



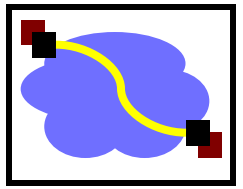
- Key observation:
  - At high loads, round trip variance is high
- Solution:
  - Base RTO on RTT and standard deviation
    - RTO = RTT + 4 * rttvar
  - new_rttvar = β * dev + (1- β) old_rttvar
    - Dev = linear deviation
    - Inappropriately named – actually smoothed linear deviation

Timestamp Extension



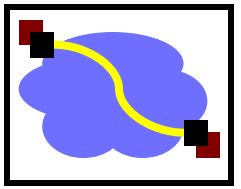
- Used to improve timeout mechanism by more accurate measurement of RTT
- When sending a packet, insert current time into option
  - 4 bytes for time, 4 bytes for echo a received timestamp
- Receiver echoes timestamp in ACK
  - Actually will echo whatever is in timestamp
- Removes retransmission ambiguity
  - Can get RTT sample on any packet

Timer Granularity



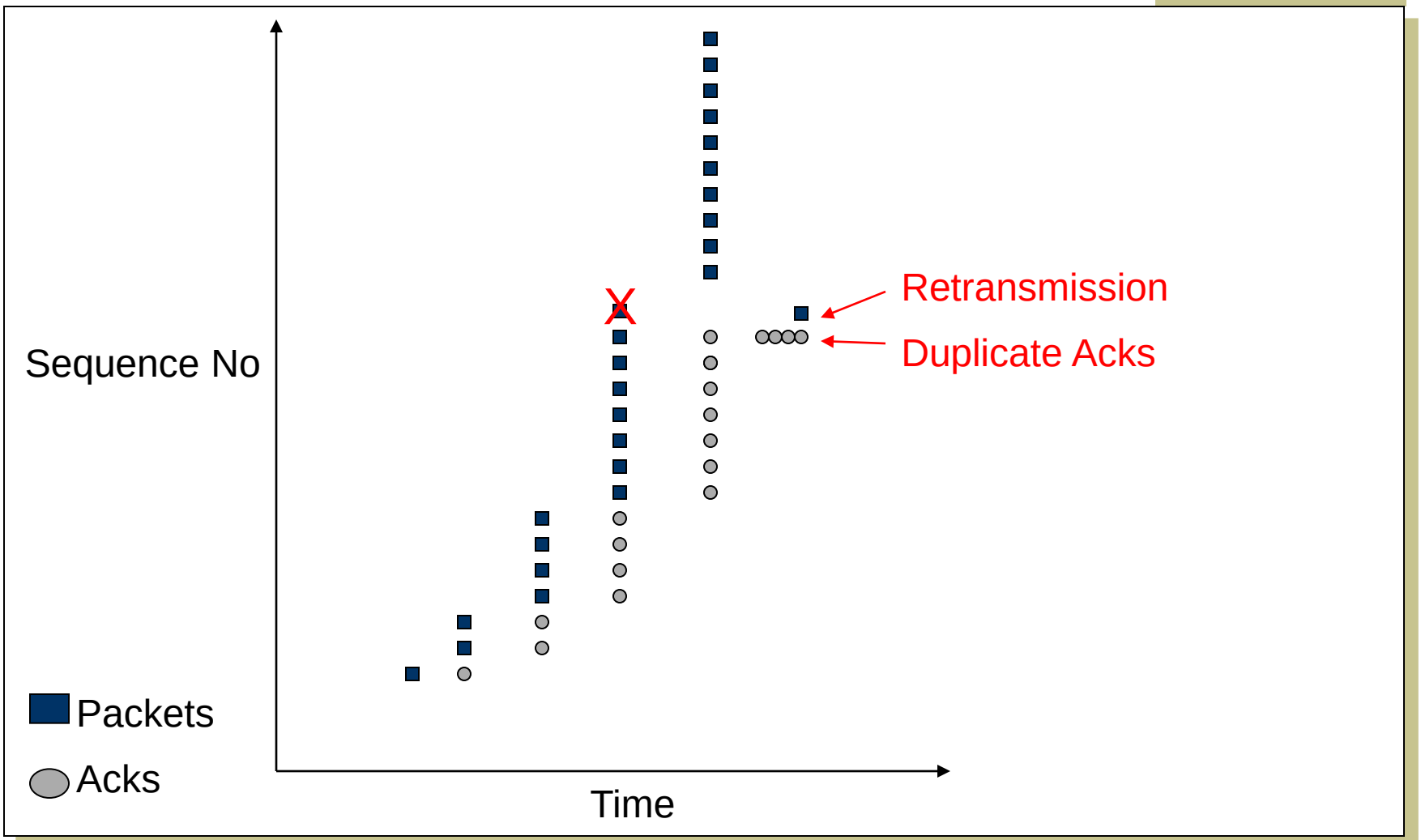
- Many TCP implementations set RTO in multiples of 200,500,1000ms
- Why?
  - Avoid spurious timeouts – RTTs can vary quickly due to cross traffic
  - Make timers interrupts efficient
- What happens for the first couple of packets?
  - Pick a very conservative value (seconds)

Fast Retransmit

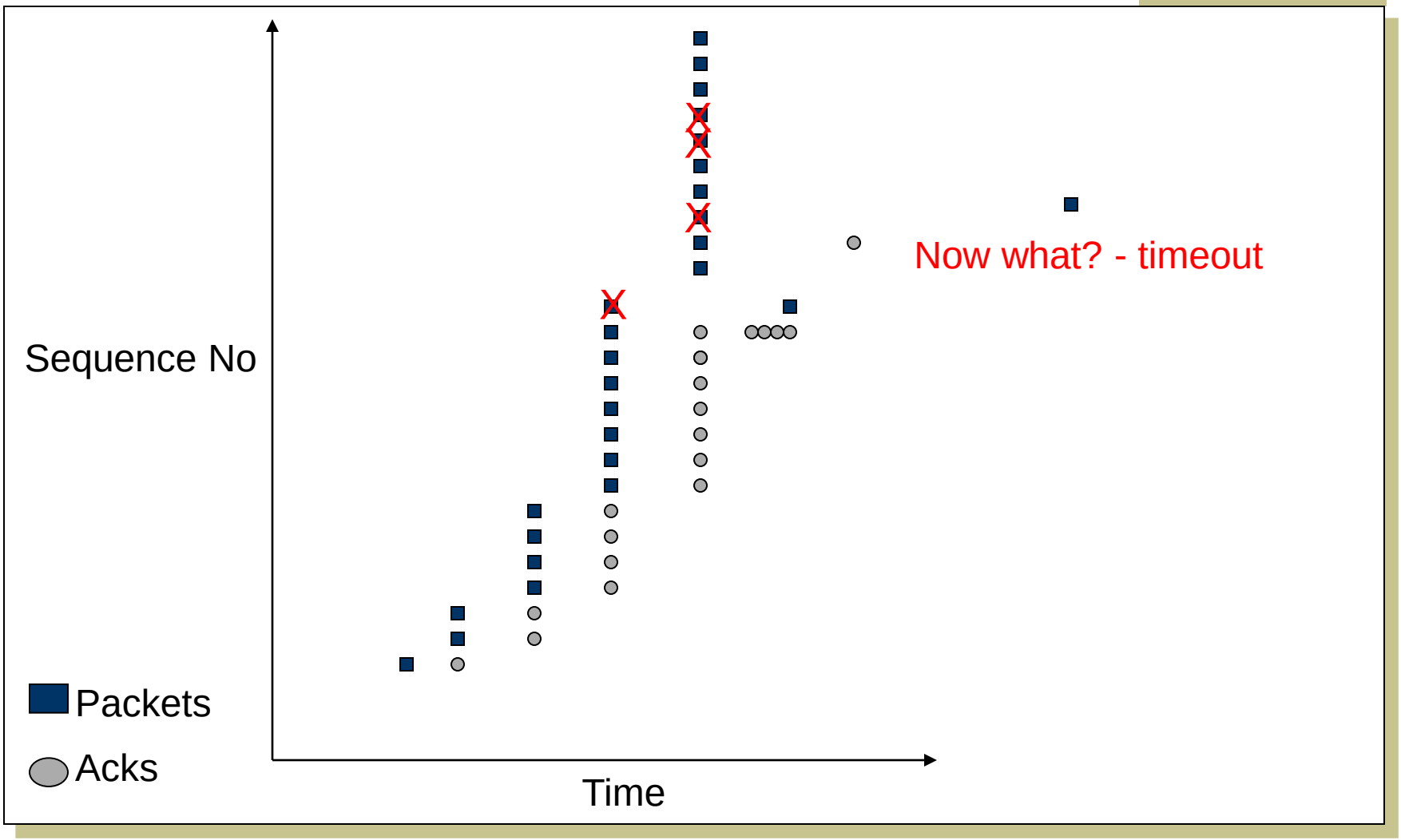


- What are duplicate acks (dupacks)?
  - Repeated acks for the same sequence
- When can duplicate acks occur?
  - Loss
  - Packet re-ordering
  - Window update – advertisement of new flow control window
- Assume re-ordering is infrequent and not of large magnitude
  - Use receipt of 3 or more duplicate acks as indication of loss
  - Don't wait for timeout to retransmit packet

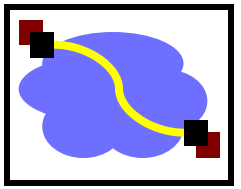
Fast Retransmit



TCP (Reno variant)

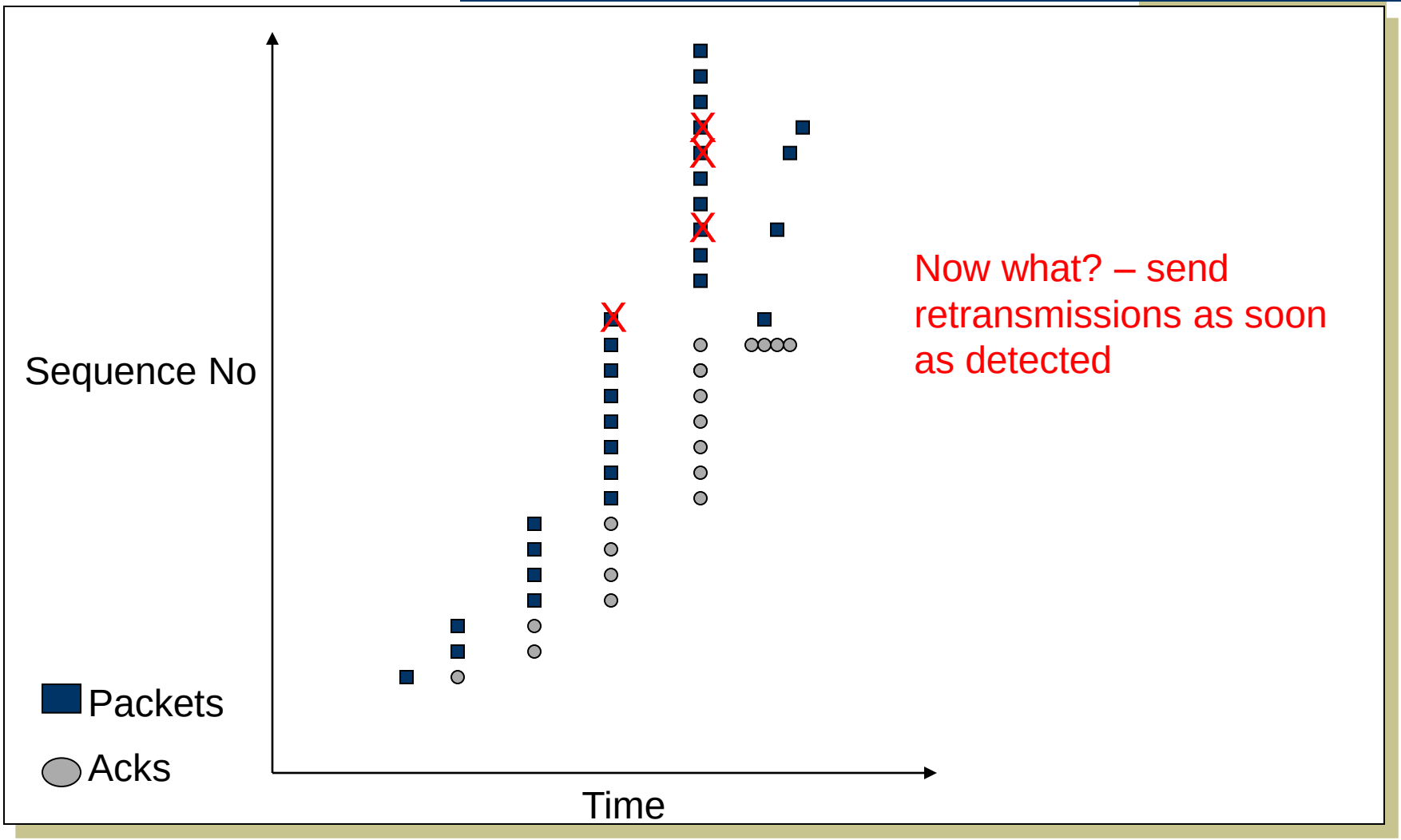


SACK

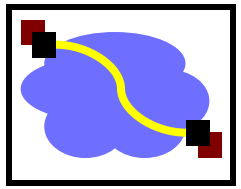


- Basic problem is that cumulative acks provide little information
- Selective acknowledgement (SACK) essentially adds a bitmask of packets received
  - Implemented as a TCP option
  - Encoded as a set of received byte ranges (max of 4 ranges/often max of 3)
- When to retransmit?
  - Still need to deal with reordering → wait for out of order by 3pkts

SACK

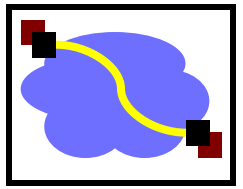

Sequence No
Time
Packets
Acks
Now what? – send retransmissions as soon as detected

Performance Issues



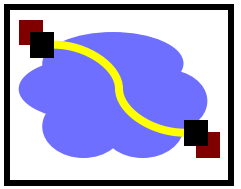
- Timeout >> fast rexmit
- Need 3 dupacks/sacks
- Not great for small transfers
  - Don't have 3 packets outstanding
- What are real loss patterns like?

Outline

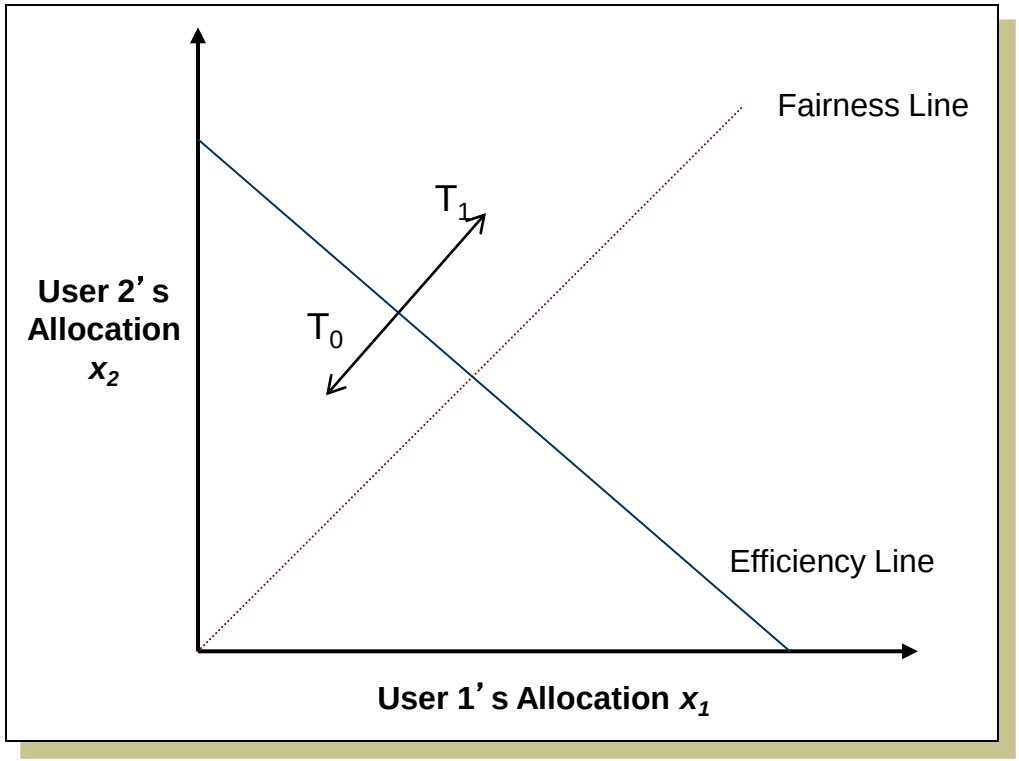


- TCP connection setup/data transfer
- TCP reliability
- TCP congestion avoidance

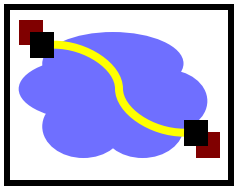
Additive Increase/Decrease



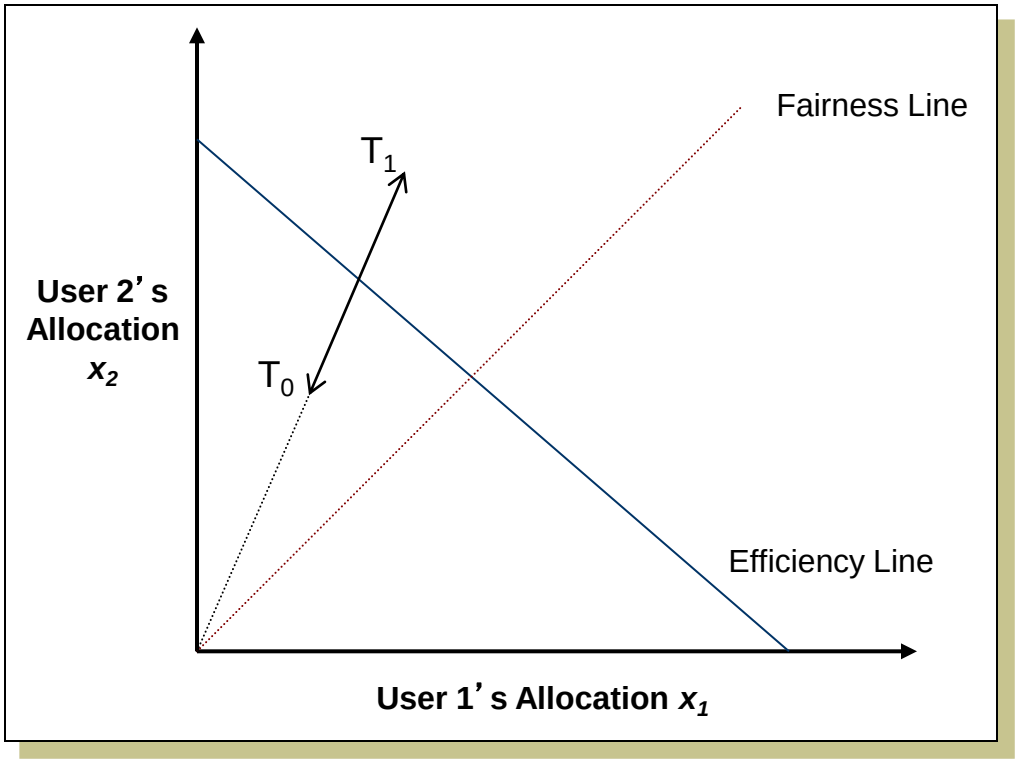
- Both X_1 and X_2 increase/ decrease by the same amount over time
  - Additive increase improves fairness and additive decrease reduces fairness



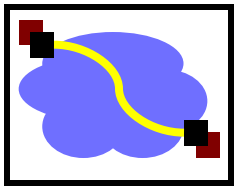
Muliplicative Increase/Decrease



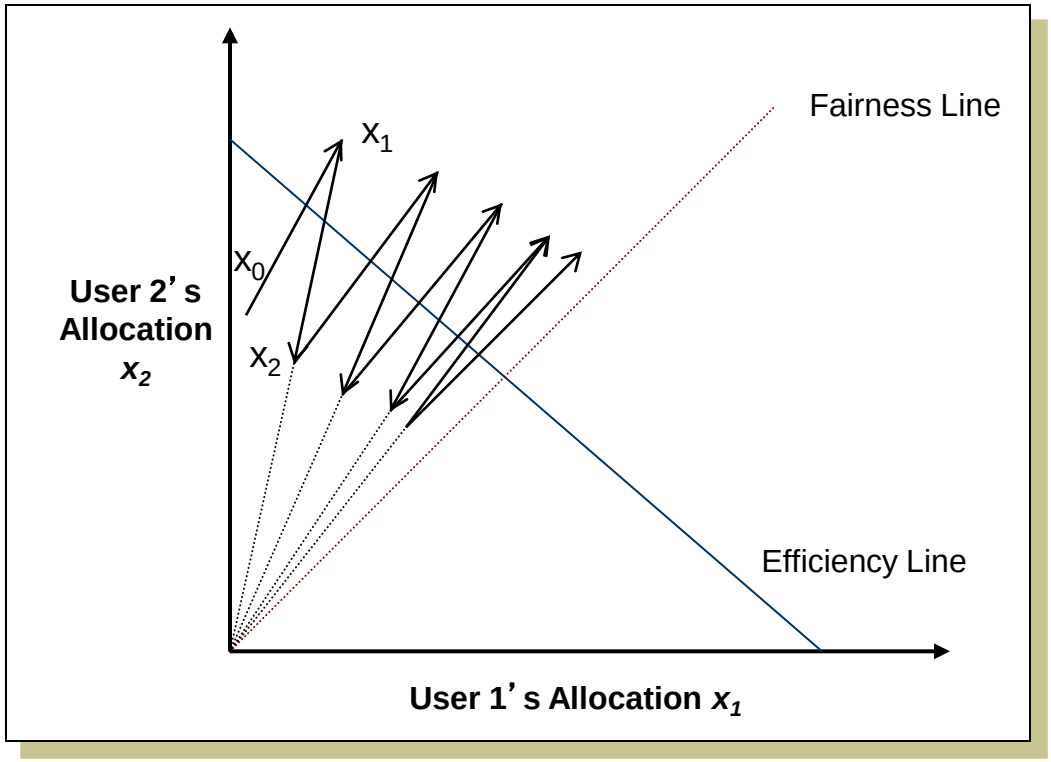
- Both X_1 and X_2 increase by the same factor over time
  - Extension from origin – constant fairness



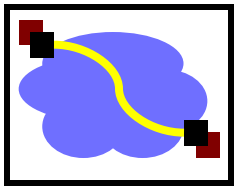
What is the Right Choice?



- Constraints limit us to AIMD
  - Improves or keeps fairness constant at each step
  - AIMD moves towards optimal point

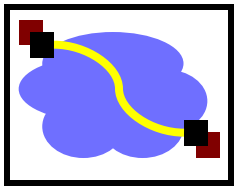


TCP Congestion Control

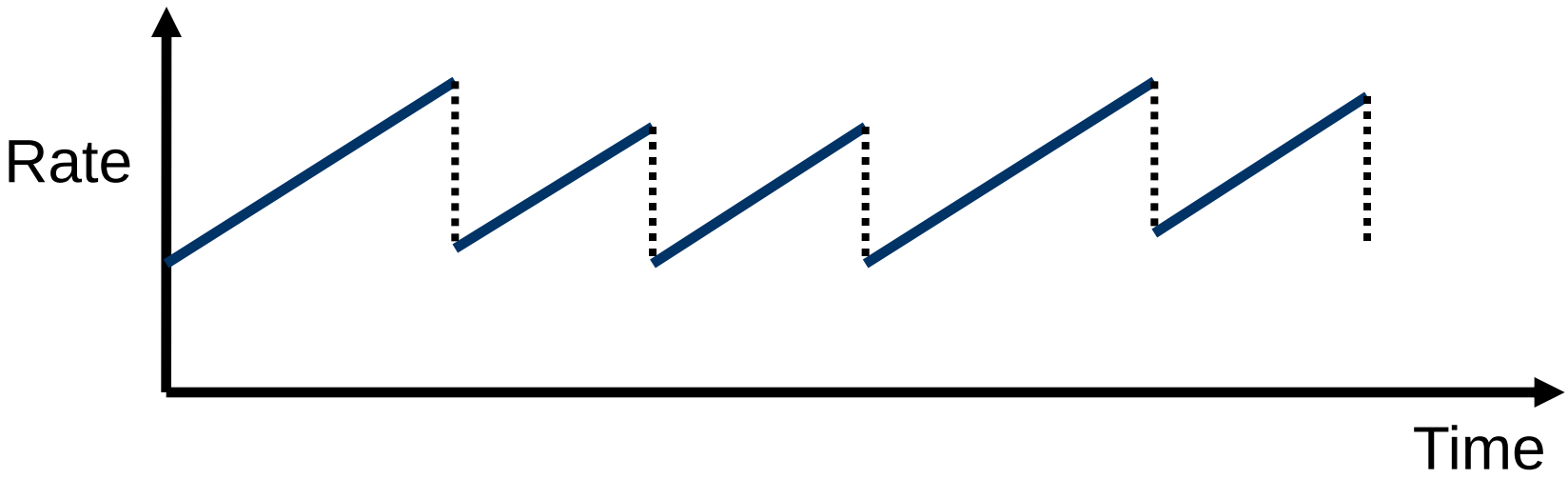


- Changes to TCP motivated by ARPANET congestion collapse
- Basic principles
  - AIMD
  - Packet conservation
  - Reaching steady state quickly
  - ACK clocking

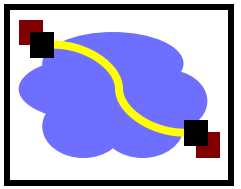
AIMD



- Distributed, fair and efficient
- Packet loss is seen as sign of congestion and results in a multiplicative rate decrease
  - Factor of 2
- TCP periodically probes for available bandwidth by increasing its rate

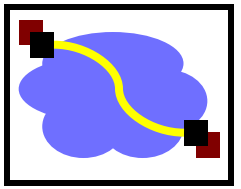


Implementation Issue



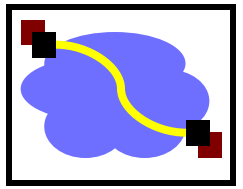
- Operating system timers are very coarse – how to pace packets out smoothly?
- Implemented using a congestion window that limits how much data can be in the network.
  - TCP also keeps track of how much data is in transit
- Data can only be sent when the amount of outstanding data is less than the congestion window.
  - The amount of outstanding data is increased on a "send" and decreased on "ack"
  - (last sent – last acked) < congestion window
- Window limited by both congestion and buffering
  - Sender's maximum window = Min (advertised window, cwnd)

Packet Conservation

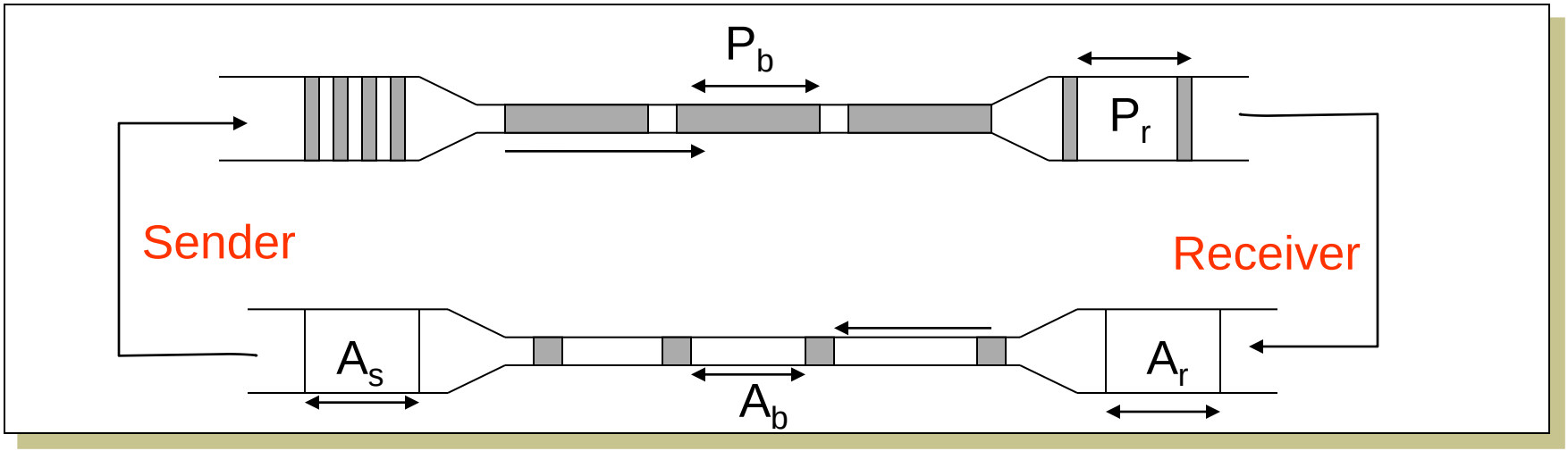


- At equilibrium, inject packet into network only when one is removed
  - Sliding window and not rate controlled
  - But still need to avoid sending burst of packets → would overflow links
    - Need to carefully pace out packets
    - Helps provide stability
- Need to eliminate spurious retransmissions
  - Accurate RTO estimation
  - Better loss recovery techniques (e.g. fast retransmit)

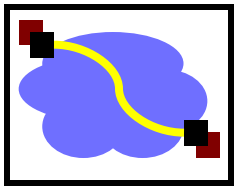
TCP Packet Pacing



- Congestion window helps to “pace” the transmission of data packets
- In steady state, a packet is sent when an ack is received
  - Data transmission remains smooth, once it is smooth
  - Self-clocking behavior

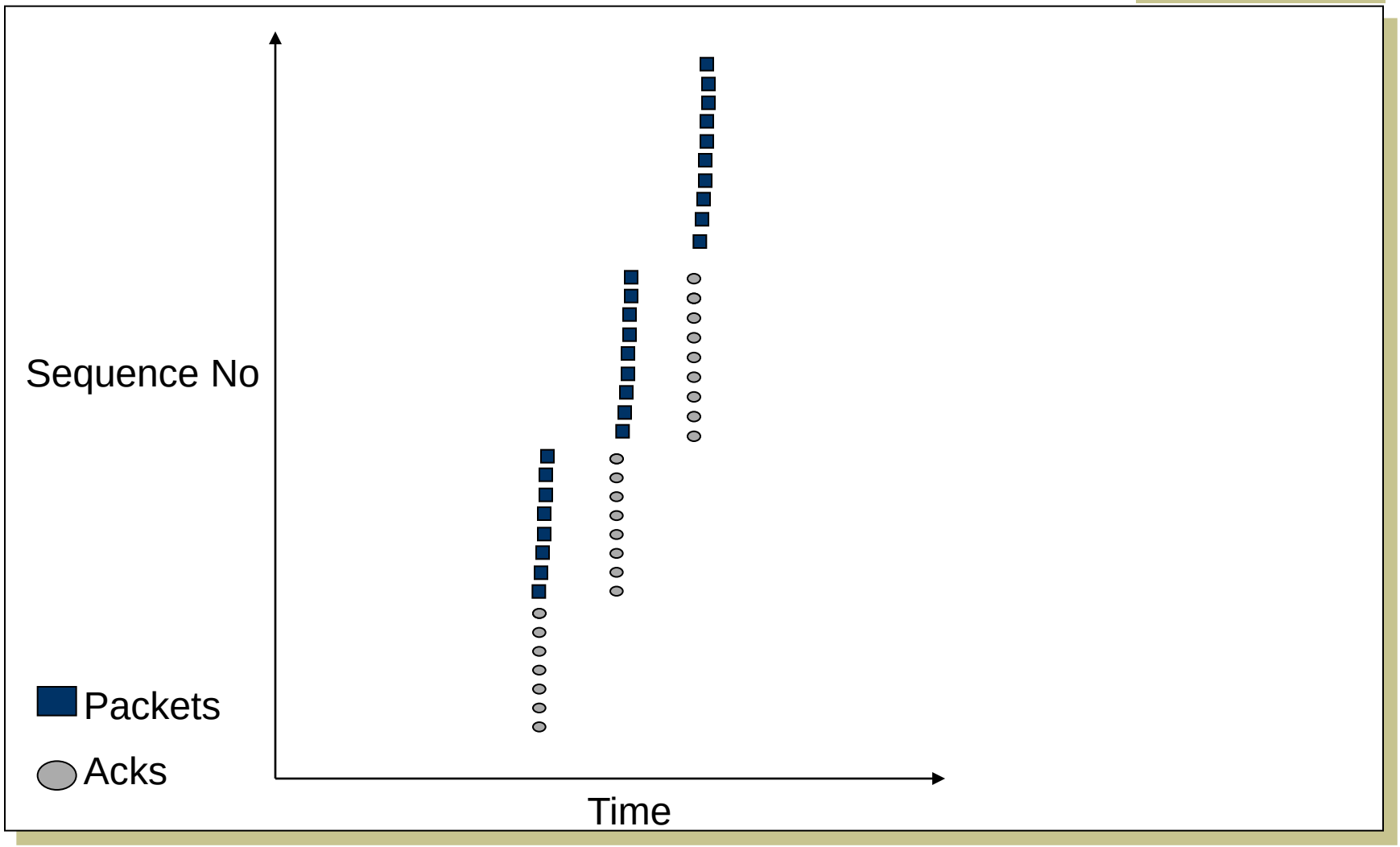


Congestion Avoidance

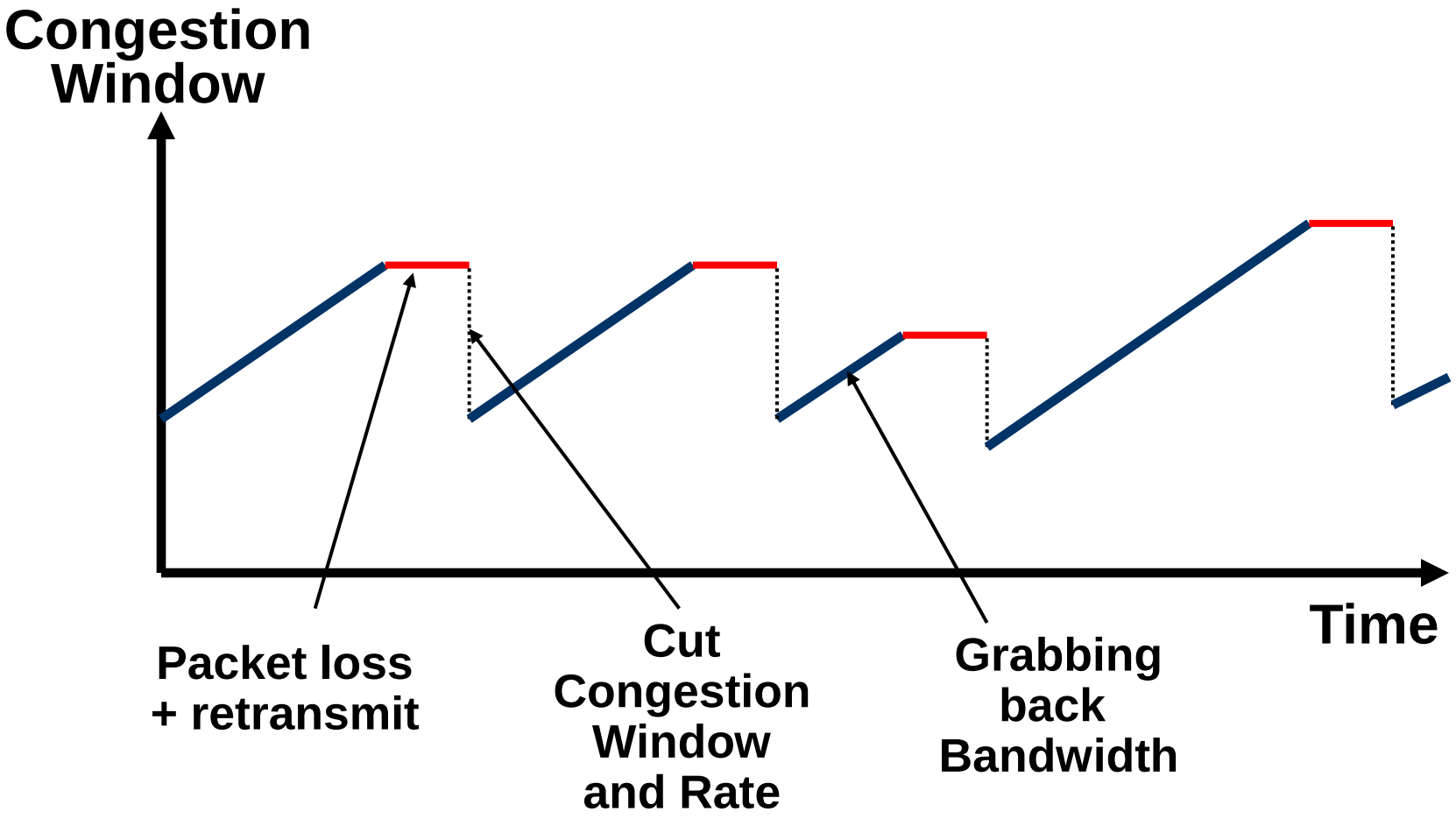


- If loss occurs when cwnd = W
  - Network can handle 0.5W ~ W segments
  - Set cwnd to 0.5W (multiplicative decrease)
- Upon receiving ACK
  - Increase cwnd by (1 packet)/cwnd
    - What is 1 packet? → 1 MSS worth of bytes
    - After cwnd packets have passed by → approximately increase of 1 MSS
- Implements AIMD

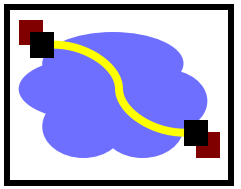
Congestion Avoidance Sequence Plot



Congestion Avoidance Behavior

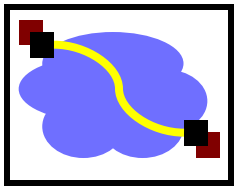


Important Lessons



- 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
- TCP flow control
  - Sliding window → mapping to packet headers
  - 32bit sequence numbers (bytes)

Important Lessons



- TCP state diagram → setup/teardown
- TCP timeout calculation → how is RTT estimated
- Modern TCP loss recovery
  - Why are timeouts bad?
  - How to avoid them? → e.g. fast retransmit