CS 268: Congestion Control and Avoidance

Kevin Lai

Feb 4, 2002

Problem

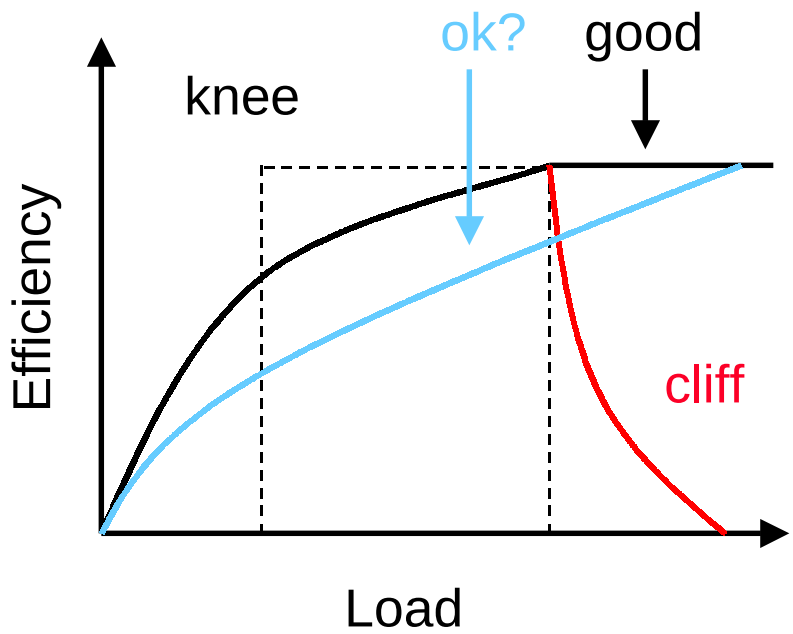
- At what rate do you send data?
  - What is max useful sending rate for different apps?
- two components
  - flow control
    - make sure that the receiver can receive
    - sliding-window based flow control:
      - receiver reports window size to sender
      - higher window → higher throughput
      - throughput = wnd/RTT
  - congestion control
    - make sure that the network can deliver

Goals

- Robust
  - latency: 50us (LAN), 133ms (min, anywhere on Earth, wired), 1s (satellite), 260s (ave Mars)
    - 10^4-10^6 difference
  - bandwidth: 9.6Kb/s (then modem, now cellular), 10 Tb/s
    - 10^9 difference
  - 0-100% packet loss
  - path may change in middle of session (why?)
  - network may/may not support explicit congestion signaling
    - incremental deployment
- Distributed control (survivability)

Non-decreasing Efficiency under Load

- Efficiency = useful_work/time
- critical property of system design
  - network technology, protocol or application
- otherwise, system collapses exactly when most demand for its operation
- trade lower overall efficiency for this?



Congestion Collapse

- Decrease of network efficiency under load
- Waste resources on useless or undelivered data
- All layers
  - load→ increased control traffic (e.g. BGP bug)
- Network layer
  - load→drops, ≥1 fragment dropped
- Transport layer
  - retransmit too many times
  - no congestion control / avoidance
- Application layer
  - load→delay, user uninterested once data arrives
  - load→delay, user aborts uncompleted session

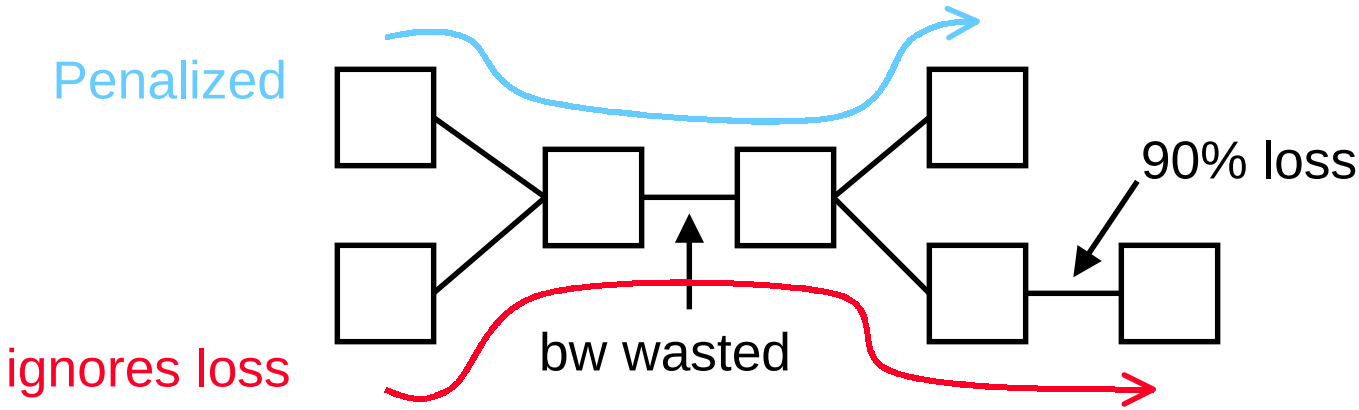
Transport Layer
Congestion Collapse 1

- network is congested (a router drops packets)
- the receiver didn't get a packet
  - know from timeout [JK88], and/or duplicate ack [FF95]
- retransmit packet
- wait, but not long enough (why?)
  - timeout too short, or
  - more acks of following packets
- retransmit again
- rinse, repeat
- assume that everyone is doing the same
- network will become more and more congested
  - with duplicate packets rather than new packets

Transport Layer
Congestion Collapse 1 Solutions

- Fix timeout [JK88]
  - keep mean RTT using low pass filter (why?)
  - keep variance of RTT (actually $(\text{mean\_deviation})^2$, why?)
  - timeout = a + 4v
  - assumes delays have poisson/normal distribution (from queueing theory)
  - still good enough?
  - always use timeout to detect packet loss?

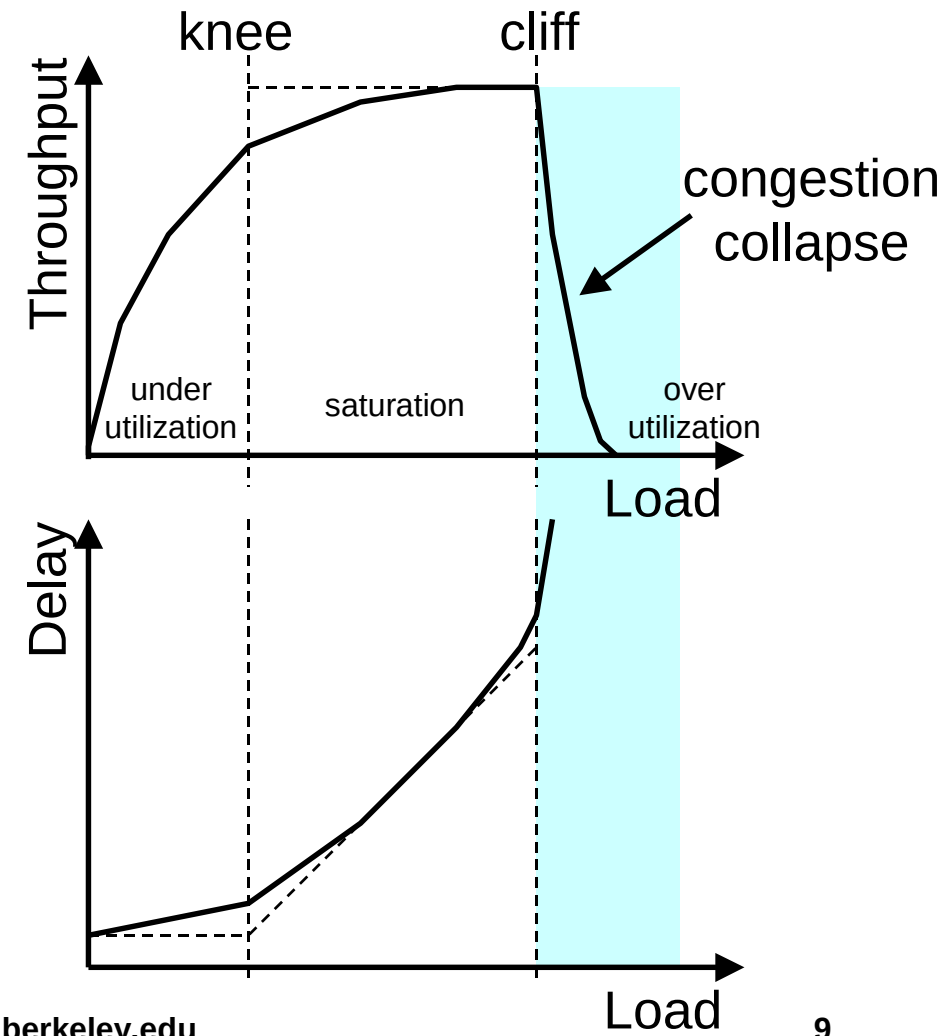
Transport Layer
Congestion Collapse 2



- Waste resources on undelivered data
- A flow sends data at a high rate despite loss
- Its packets consume bandwidth at earlier links, only to be dropped at a later link

Congestion Collapse and Efficiency

- knee – point after which
  - throughput increases slowly
  - delay increases quickly
- cliff – point after which
  - throughput decreases quickly to zero (congestion collapse)
  - delay goes to infinity
- Congestion avoidance
  - stay at knee
- Congestion control
  - stay left of (but usually close to) cliff
- Note (in an M/M/1 queue)
  - delay = 1/(1 – utilization)



Transport Layer
Congestion Collapse 2 Solutions

- Reduce loss by increasing buffer size. Why not?
- if congestion, then send slower
  else if sending at lower than fair rate, then send faster
  - congestion control and avoidance (finally)
  - how to detect network congestion?
  - how to communicate allocation to sources?
  - how to determine efficient allocation?
  - how to determine fair allocation?

Metrics

- Efficiency
  - ratio of aggregate throughput to capacity
- Fairness
  - degree to which everyone is getting equal share
- Convergence time (responsiveness)
  - How long to get to fairness, efficiency
- Size of oscillation (smoothness)
  - dynamic system→oscillations around optimal point

Detecting Congestion

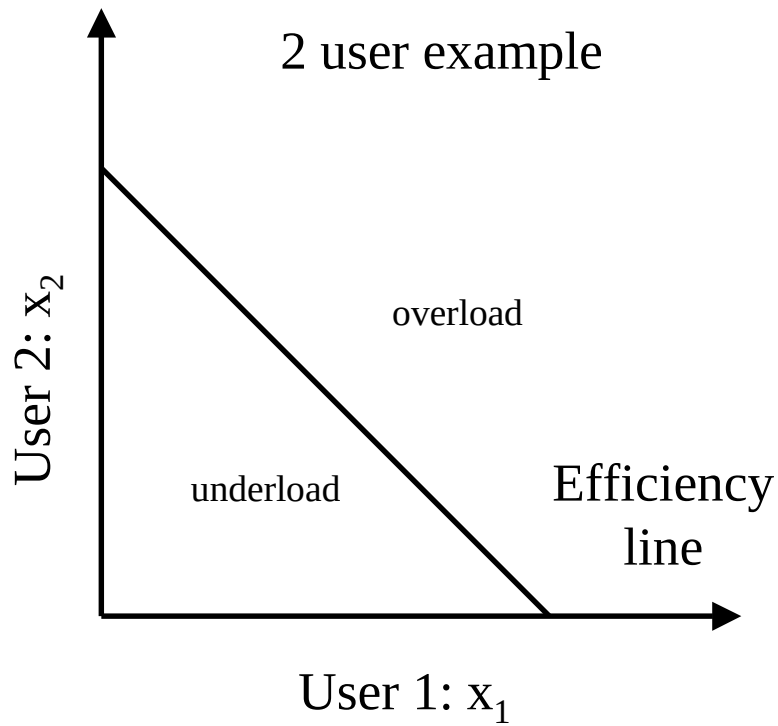
- Explicit network signal
  - Send packet back to source (e.g. ICMP Source Quench)
    - control traffic congestion collapse
  - Set bit in header (e.g. DEC DNA/OSI Layer 4[CJ89], ECN)
    - can be subverted by selfish receiver [SEW01]
  - Unless on every router, still need end-to-end signal
  - Could be be robust, if deployed (DoS?)
- Implicit network signal
  - Loss (e.g. TCP Tahoe, Reno, New Reno, SACK)
    - +relatively robust, -no avoidance
  - Delay (e.g. TCP Vegas)
    - +avoidance, -difficult to make robust
  - Easily deployable
  - Robust enough? Wireless?

Communicating Allocation to Sources

- Explicit
  - Send packet back to source or set in packet header
    - control traffic congestion collapse
    - trust receiver
  - Need to keep per flow state (anti-Internet architecture)
    - what happens if router fails, route changes, mobility
  - Unless on every router, still need end-to-end signal
  - Efficient, fair, responsive, smooth
- Implicit: Chiu and Jain 1988
  - Can converge to efficiency and fairness without explicit signal of fair rate
  - Easily deployable
  - Good enough?

Efficient Allocation

- Too slow
  - fail to take advantage of available bandwidth → underload
- Too fast
  - overshoot knee → overload, high delay, loss
- Everyone's doing it
  - may all under/over shoot → large oscillations
- Optimal:
  - $\Sigma x_i = X_{goal}$
- Efficiency = 1 - distance from efficiency line

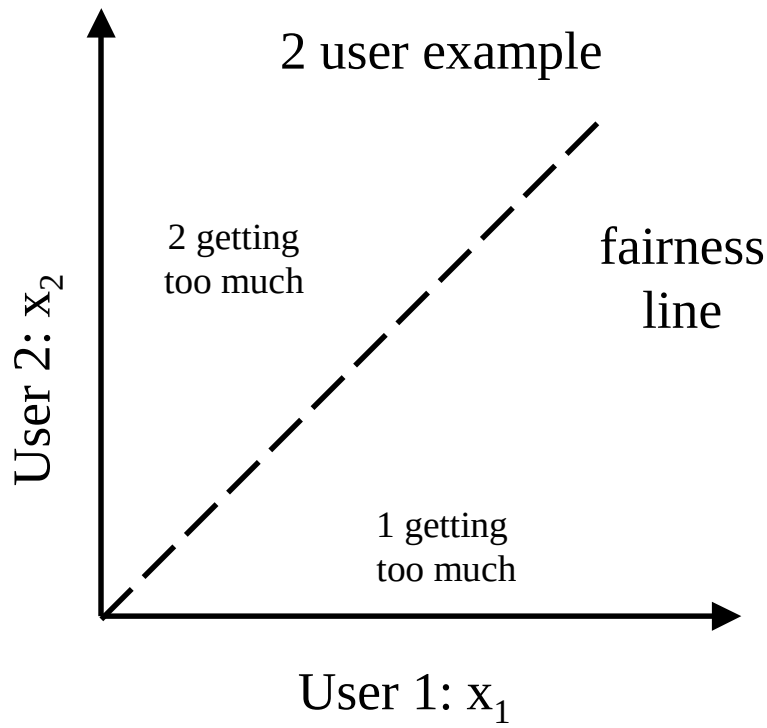


Fair Allocation

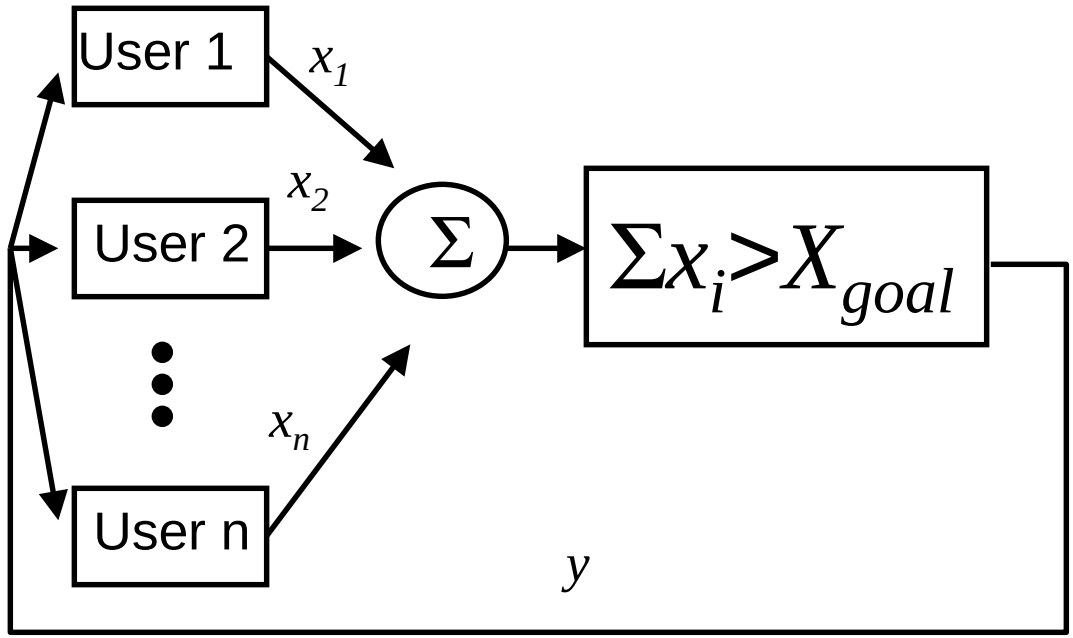
- Maxmin fairness
  - flows which share the same bottleneck get the same amount of bandwidth

$$F(x) = \frac{\left(\sum x_i\right)^2}{n\left(\sum x_i^2\right)}$$

- Assumes no knowledge of priorites
- Fairness = 1 - distance from fairness line



Control System Model [CJ89]



- Simple, yet powerful model
- Explicit binary signal of congestion
  - why explicit (TCP uses implicit)?
- Implicit allocation of bandwidth

Possible Choices

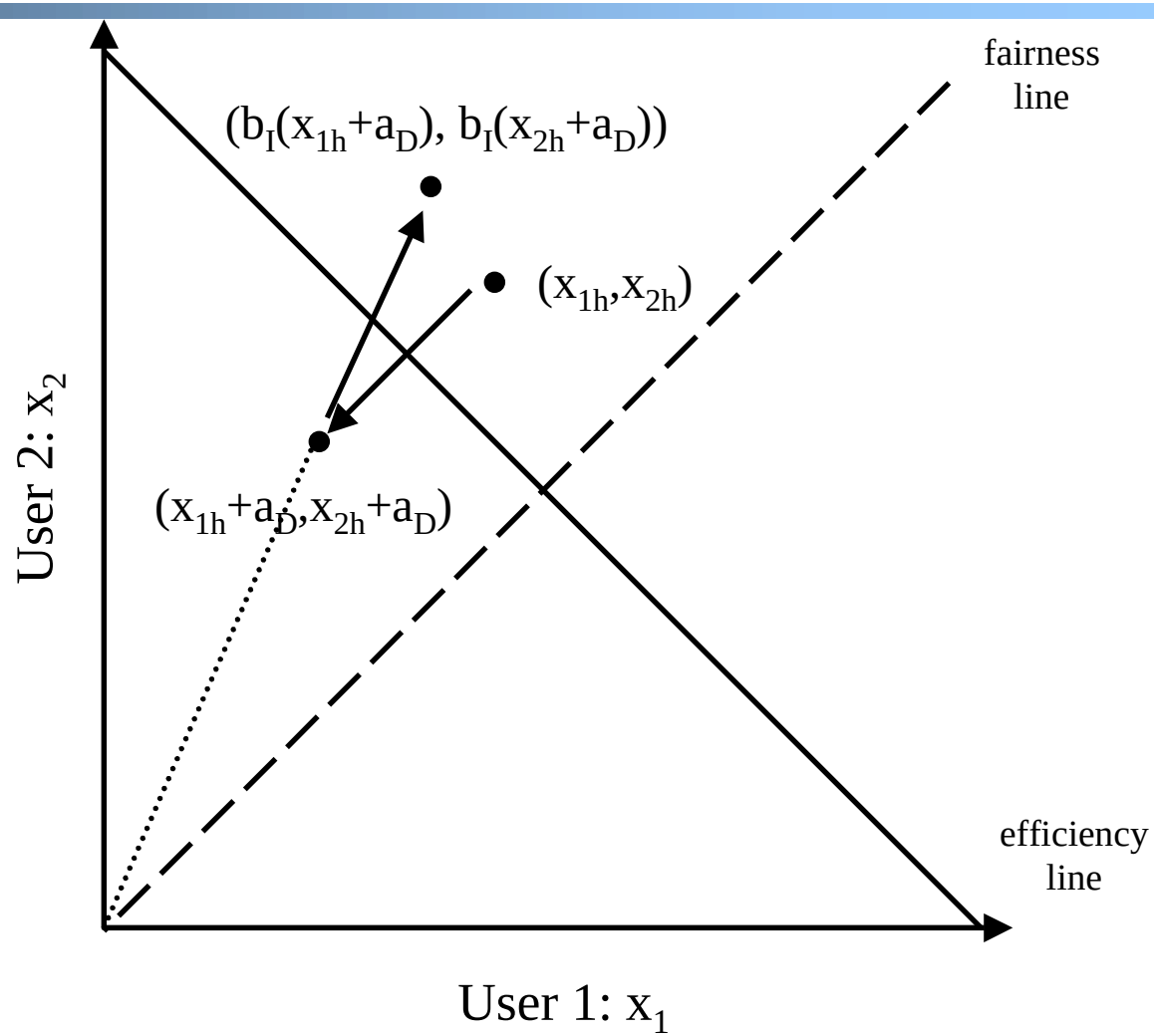
$$x_i(t+1) = \begin{cases} a_I + b_I x_i(t) & increase \\ a_D + b_D x_i(t) & decrease \end{cases}$$

- multiplicative increase, additive decrease
  - $a_I=0$, $b_I>1$, $a_D<0$, $b_D=0$
- additive increase, additive decrease
  - $a_I>0$, $b_I=0$, $a_D<0$, $b_D=0$
- multiplicative increase, multiplicative decrease
  - $a_I=0$, $b_I>1$, $a_D=0$, $0<b_D<1$
- additive increase, multiplicative decrease
  - $a_I>0$, $b_I=0$, $a_D=0$, $0<b_D<1$
- Which one?

Multiplicative Increase, Additive Decrease

- Does not converge to fairness
  - Not stable at all
- Does not converges to efficiency
  - stable iff

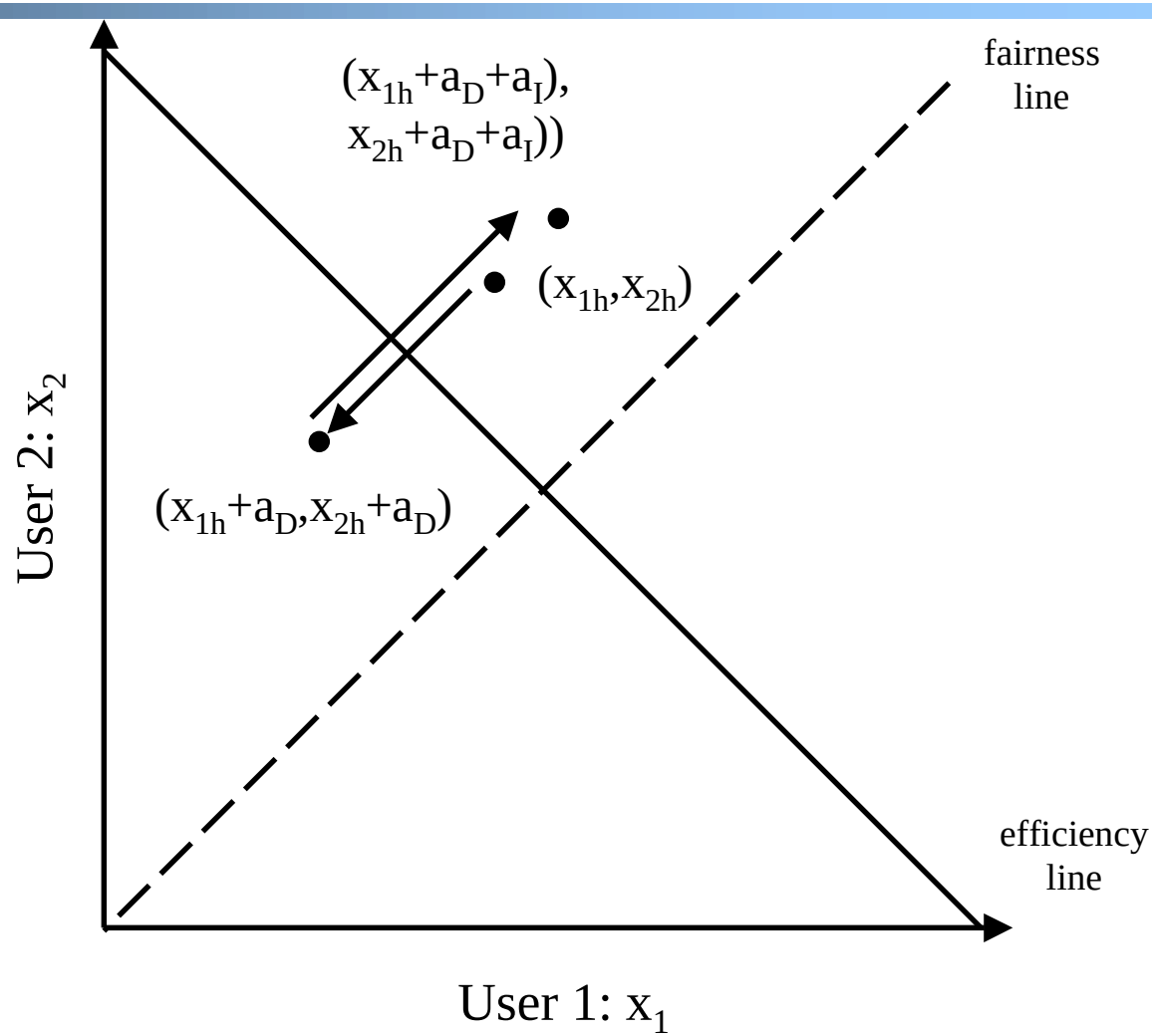
$$x_{1h} = x_{2h} = \frac{b_I a_D}{1 - b_I}$$



Additive Increase, Additive Decrease

- Does not converge to fairness
  - stable
- Does not converge to efficiency
  - stable iff

$$a_D = a_I$$

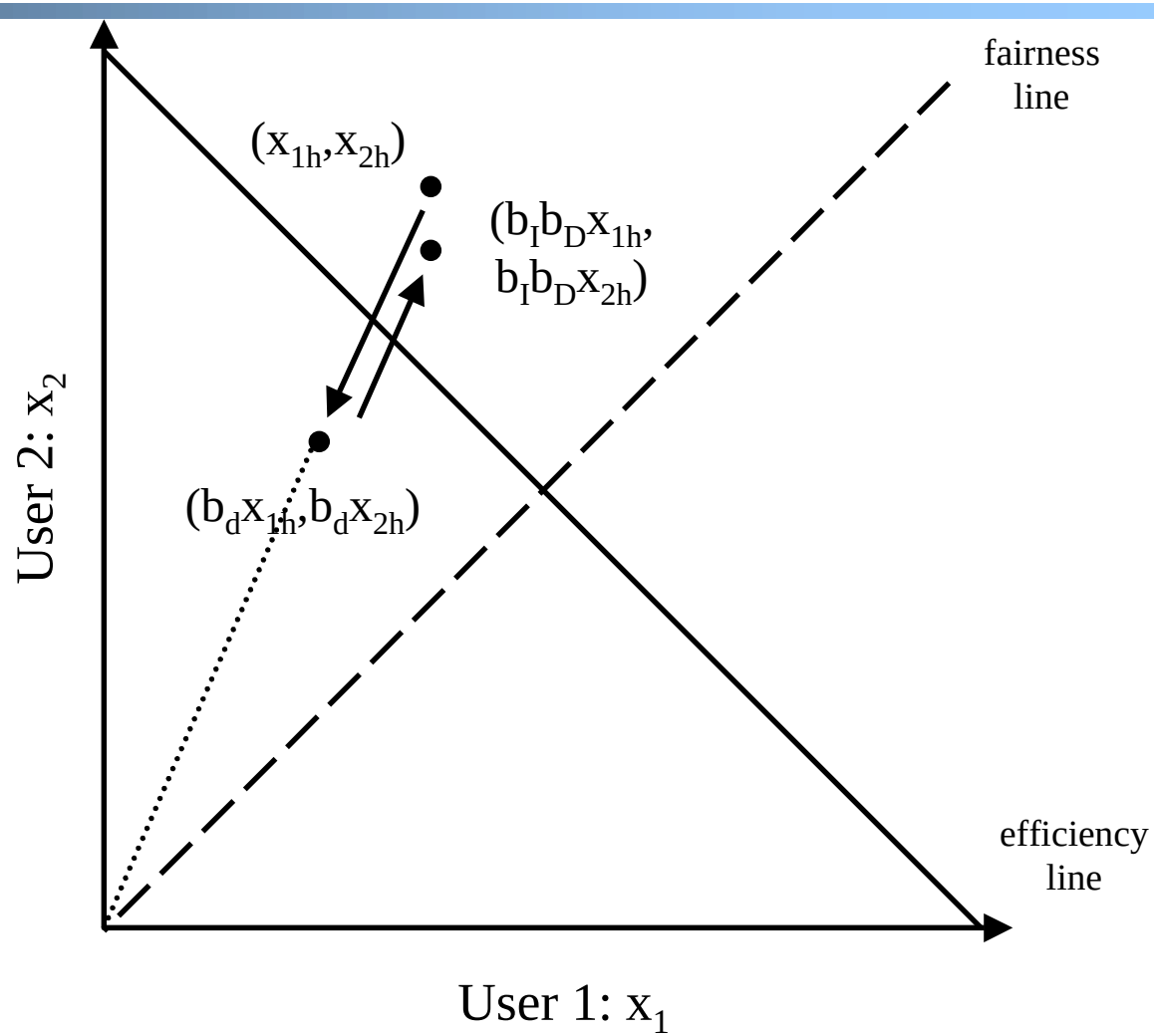


Multiplicative Increase, Multiplicative Decrease

- Does not converge to fairness
  - stable
- Converges to efficiency iff

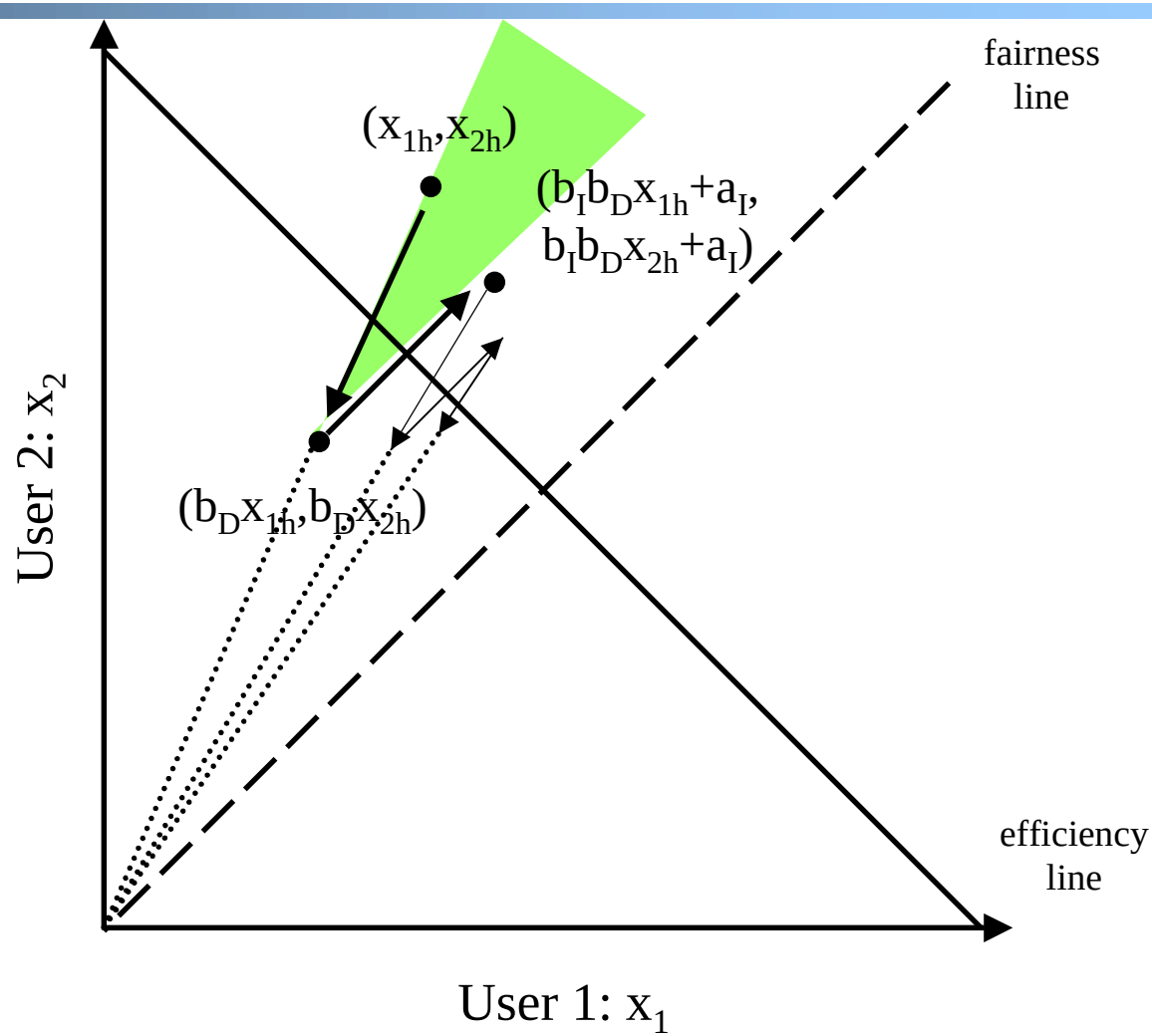
$$b_I \geq 1$$

$$0 \leq b_D < 1$$



Additive and Multiplicative Increase, Multiplicative Decrease

- Converges to fairness
- Converges to efficiency iff
  - $b_I >= 1$
- Increments smaller as fairness increases
  - effect on metrics?
- Additive Increase is better
  - why?



Significance

- Characteristics
  - converges to efficiency, fairness
  - easily deployable
  - fully distributed
  - no need to know full state of system (e.g. number of users, bandwidth of links) (why good?)
- Theory that enabled the Internet to grow beyond 1989
  - key milestone in Internet development
  - fully distributed network architecture requires fully distributed congestion control
  - basis for TCP

Modeling

- Critical to understanding complex systems
  - [CJ89] model relevant for 13 years, 10^6 increase of bandwidth, 1000x increase in number of users
- Criteria for good models
  - realistic
  - simple
    - easy to work with
    - easy for others to understand
  - realistic, complex model $\rightarrow$ useless
  - unrealistic, simple model $\rightarrow$ can teach something about best case, worst case, etc.

TCP Congestion Contol

- [CJ89] provides theoretical basis
  - still many issues to be resolved
- How to start?
- Implicit congestion signal
  - loss
  - need to send packets to detect congestion
  - must reconcile with AIMD
- How to maintain equilibrium?
  - use ACK: send a new packet only after you receive and ACK. Why?
  - maintain number of packets in network “constant”

TCP Congestion Control

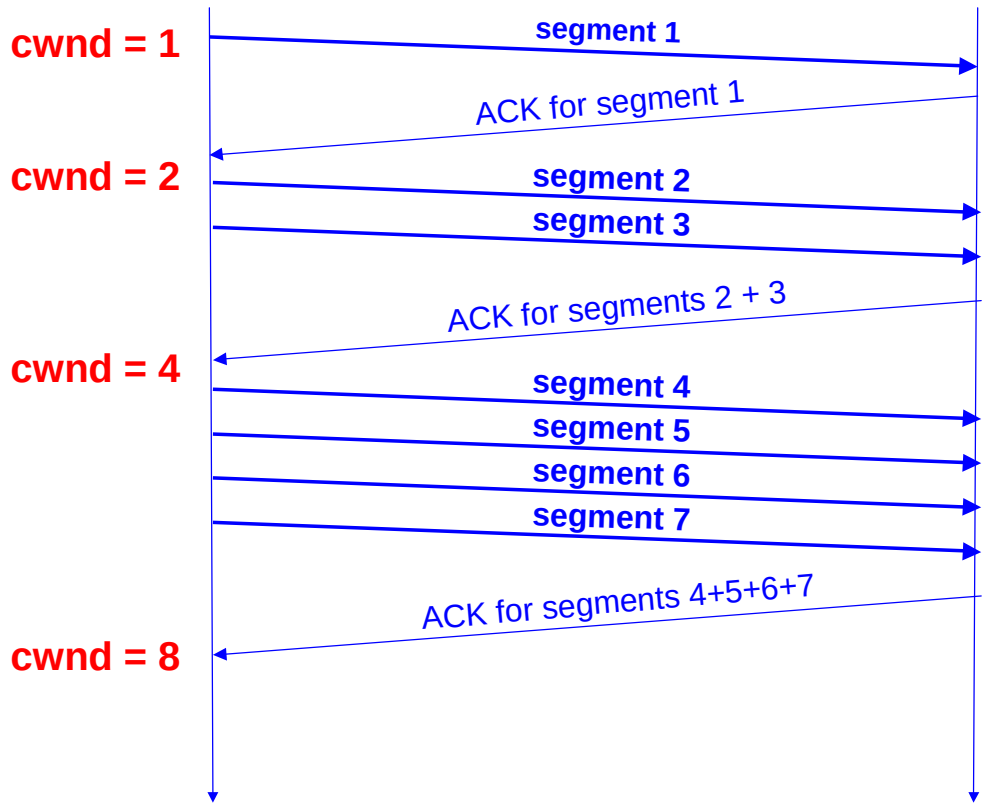
- Maintains three variables:
  - cwnd – congestion window
  - flow_win – flow window; receiver advertised window
  - ssthresh – threshold size (used to update cwnd)
- For sending use: win = **min**(flow_win, cwnd)

TCP: Slow Start

- Goal: discover congestion quickly
- How?
  - quickly increase *cwnd* until network congested → get a rough estimate of the optimal of *cwnd*
  - Whenever starting traffic on a new connection, or whenever increasing traffic after congestion was experienced:
    - Set *cwnd* =1
    - Each time a segment is acknowledged increment *cwnd* by one (*cwnd*++).
- Slow Start is not actually slow
  - *cwnd* increases exponentially

Slow Start Example

- The congestion window size grows very rapidly
- TCP slows down the increase of *cwnd* when ***cwnd >= ssthresh***

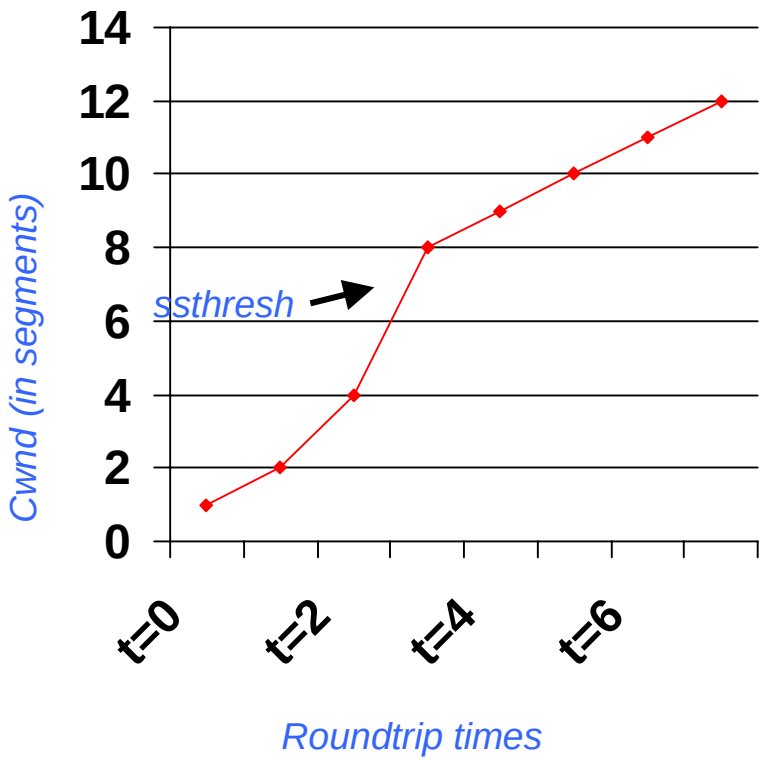


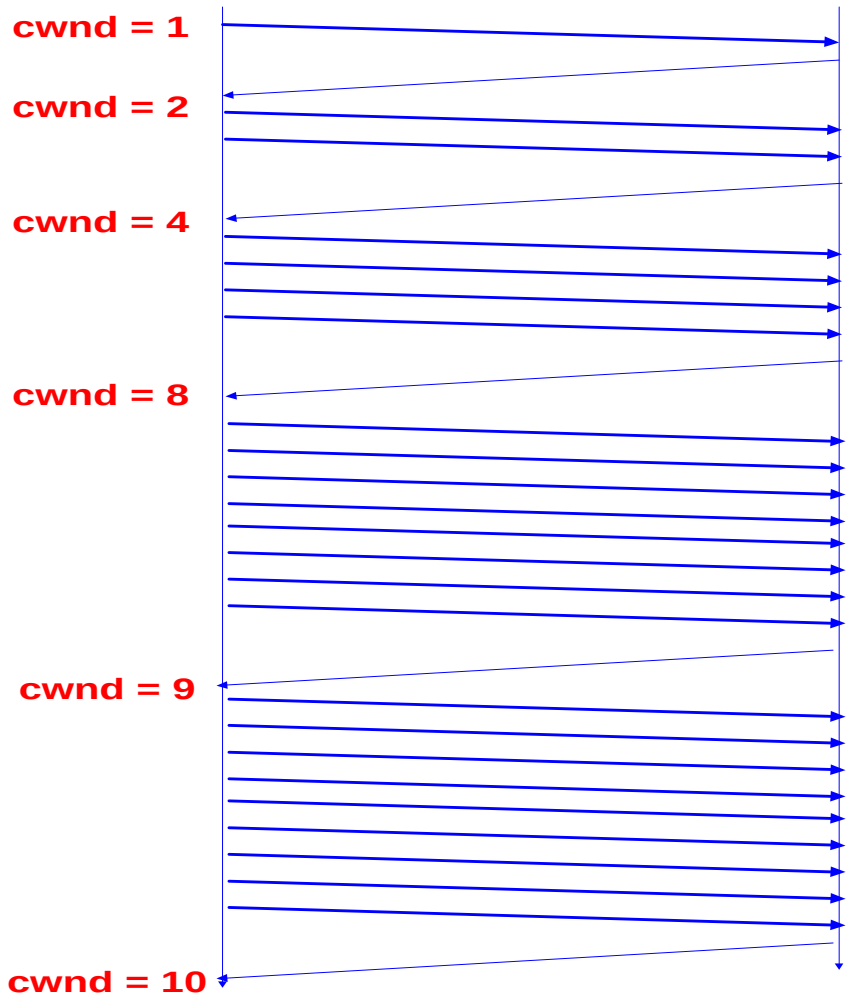
Congestion Avoidance

- Slow down “Slow Start”
- **If** *cwnd* > *ssthresh* **then**
  each time a segment is acknowledged
  increment *cwnd* by *1/cwnd* (*cwnd* += *1/cwnd*).
- So *cwnd* is increased by one only if all segments have been acknowlegded.
- (more about *ssthresh* latter)

Slow Start/Congestion Avoidance Example

- Assume that *ssthresh = 8*



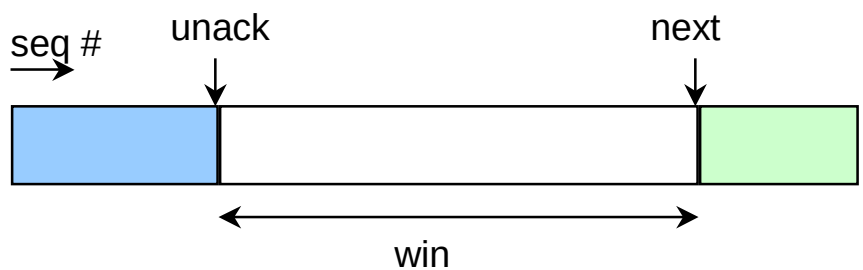


Putting Everything Together: TCP Pseudocode

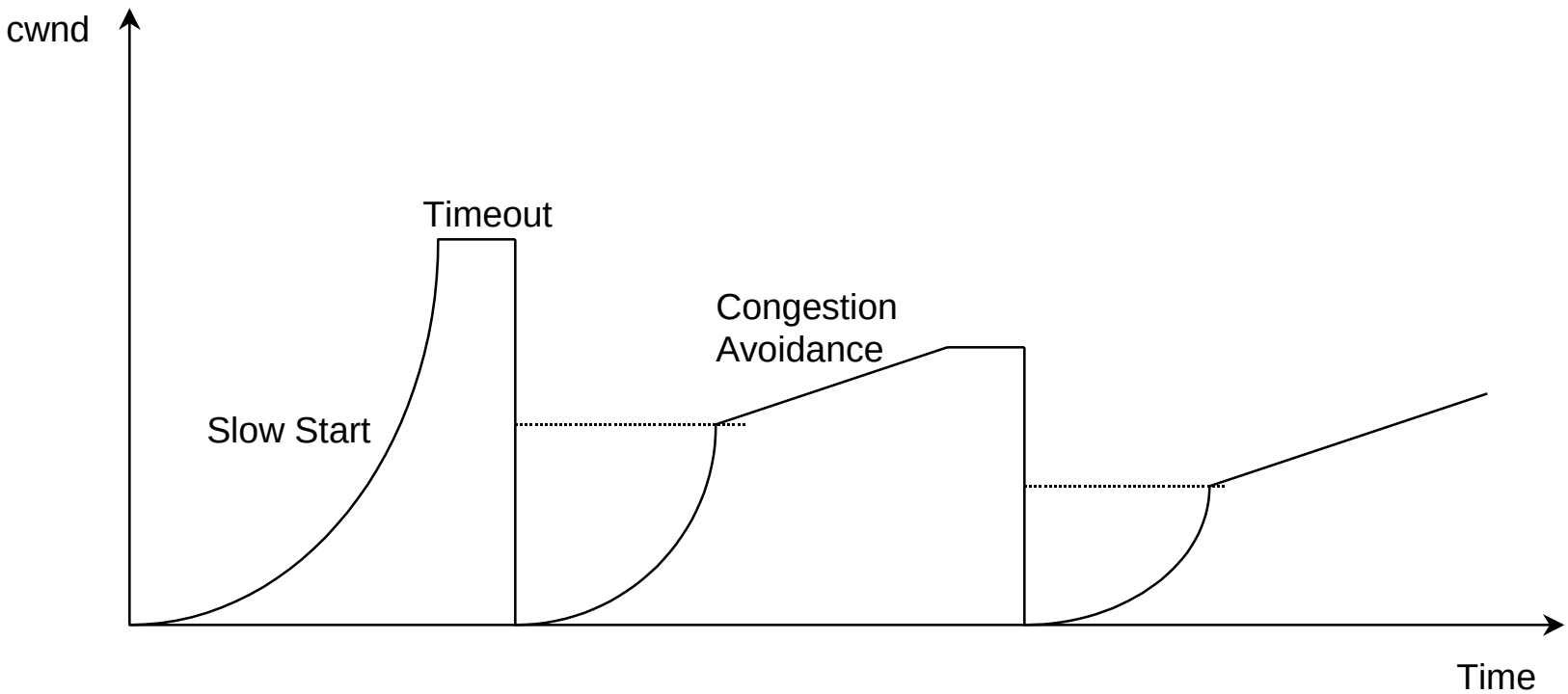
```
Initially:
    cwnd = 1;
    ssthresh = infinite;
New ack received:
    if (cwnd < ssthresh)
        /* Slow Start*/
        cwnd = cwnd + 1;
    else
        /* Congestion Avoidance */
        cwnd = cwnd + 1/cwnd;
Timeout:
    /* Multiplicative decrease */
    ssthresh = win/2;
    cwnd = 1;
```

```
while (next < unack + win)
    transmit next packet;

where   win = min(cwnd,
                  flow_win);
```

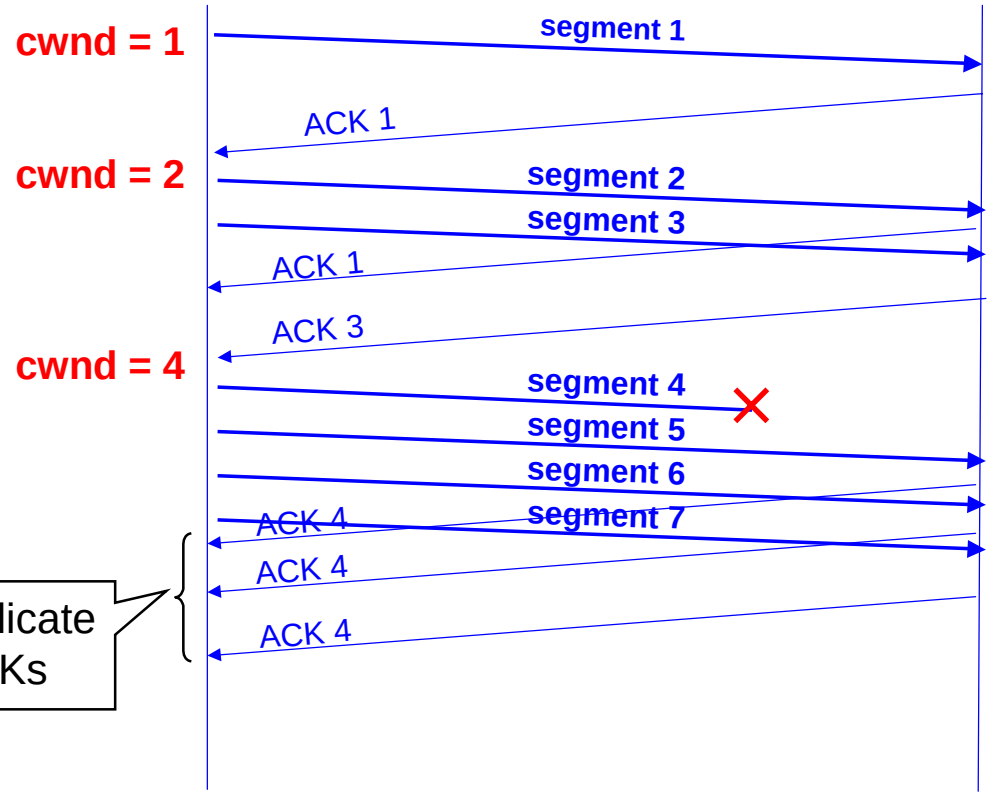


The big picture



Fast Retransmit

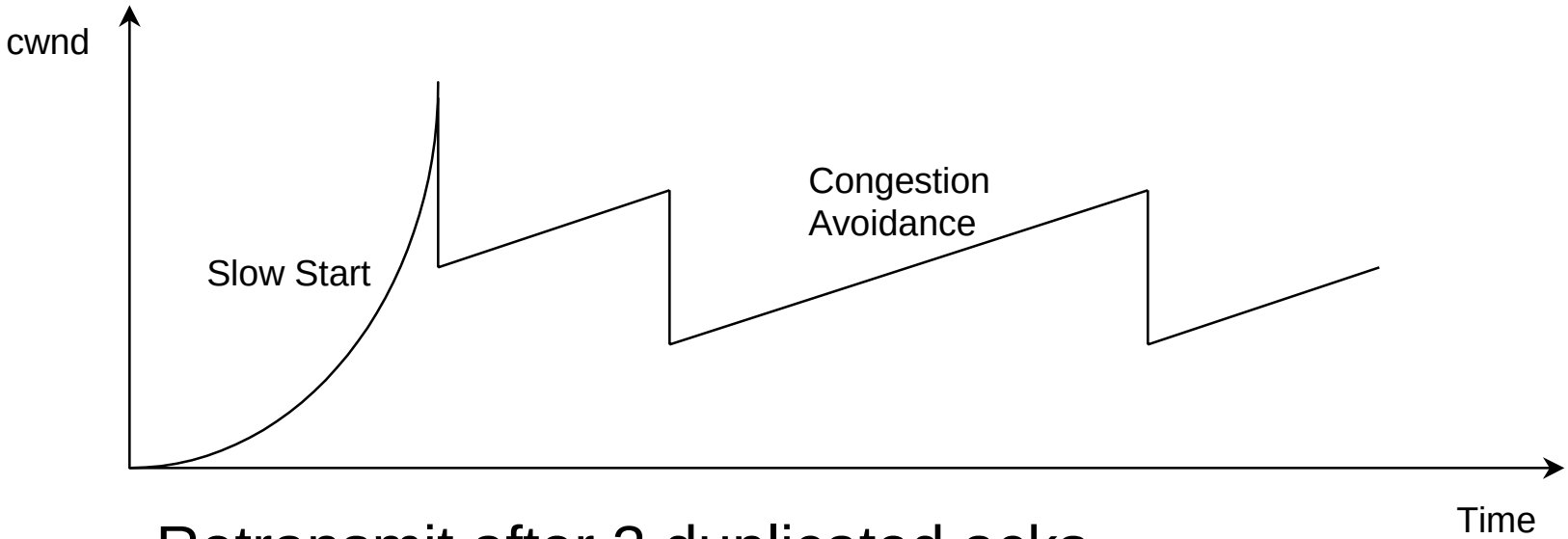
- Don’t wait for window to drain
- Resend a segment after 3 duplicate ACKs
  - remember a duplicate ACK means that an out-of sequence segment was received
- Notes:
  - duplicate ACKs due to packet reordering
    - why reordering?
  - iwindow may be too small to get duplicate ACKs



Fast Recovery

- After a fast-retransmit set *cwnd* to *ssthresh/2*
  - i.e., don't reset *cwnd* to 1
- But when RTO expires still do *cwnd* = 1
- Fast Retransmit and Fast Recovery → implemented by TCP Reno; most widely used version of TCP today

Fast Retransmit and Fast Recovery



- Retransmit after 3 duplicated acks
  - prevent expensive timeouts
- No need to slow start again
- At steady state, *cwnd* oscillates around the optimal window size.

Reflections on TCP

- assumes that all sources cooperate
- assumes that congestion occurs on time scales greater than 1 RTT
- only useful for reliable, in order delivery, non-real time applications
- vulnerable to non-congestion related loss (e.g. wireless)
- can be unfair to long RTT flows