

---

# CS 552

## Reliable Data Transfer

R. Martin

Credit slides from I. Stoica, C. Jin, M. Mitzenmacher

# Outline

---

- End-to-End argument
- TCP
  - Structure
  - Historical development
  - Newer Implementations
  - FAST TCP
- Forward Error Correction
  - Digital Fountain Approach

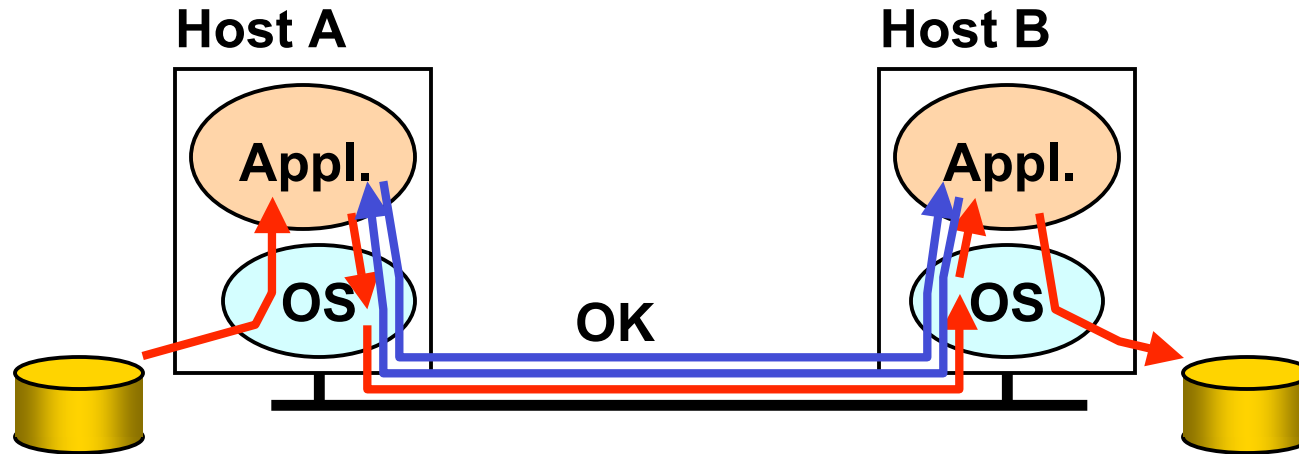
# End-to-End Argument

---

- Think twice before implementing a functionality that you believe that is useful to an application at a lower layer
- If the application can implement a functionality correctly, implement it at a lower layer only as a performance enhancement

# Example: Reliable File Transfer

---



- Solution 1: make each step reliable, and then concatenate them
- Solution 2: end-to-end check and retry

# Traditional Arguments

---

- Solution 1 not complete
  - What happens if the sender or/and receiver misbehave?
- The receiver has to do the check anyway!
- Thus, full functionality can be entirely implemented at application layer; **no** need for reliability from lower layers
- Is there any need to implement reliability at lower layers?

# Discussion

---

- In reality, end-to-end is necessary, but not sufficient
  - E.g. How can you find a bug in your network equipment? Isolate problems?
  - All reliable systems must have MANY intermediate integrity checks
- No networks sacrifices intermediate checks (e.g checksums); for performance
  - Tradeoff is that retry is performed end-to-end, checks only drop packets.

# Trade-offs

---

- Application has more information about the data and the semantic of the service it requires (e.g., can check only at the end of each data unit)
- A lower layer has more information about constraints in data transmission (e.g., packet size, error rate)
  
- Note: these trade-offs are a direct result of layering!

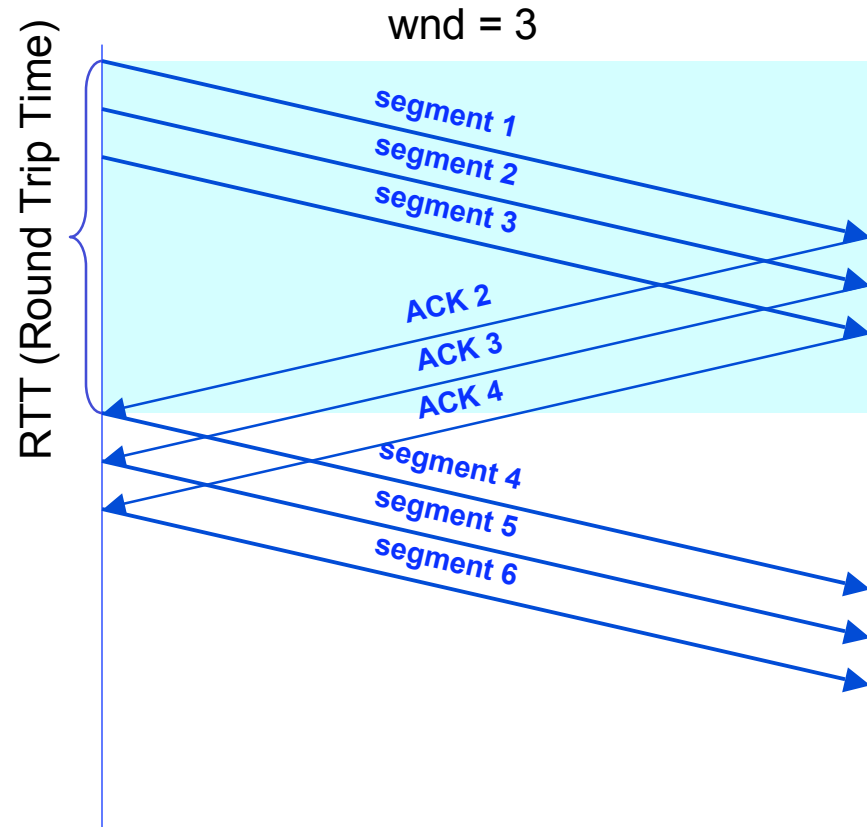
# Reliability/Flow/Congestion relationship

---

- Why are these problems related?
  - Reliability using ARQ: Must retransmit lost packets
  - Flow control: Buffer concerns on receiver
  - Congestion control: When is network overloaded?
- Putting a new packet onto the network affects all 3 issues.
  - Decision to inject a packet must be based on all 3 concerns.

# Flow control: Window Size and Throughput

- Sliding-window based flow control:
  - Higher window  $\rightarrow$  higher throughput
    - $\text{Throughput} = \text{wnd}/\text{RTT}$
  - Need to worry about sequence number wrapping
- Remember: window size control throughput



# Why do You Care About Congestion Control?

---

- Otherwise you get to congestion collapse
- How might this happen?
  - Assume network is congested (a router drops packets)
  - You learn the receiver didn't get the packet
    - either by ACK, NACK, or Timeout
  - What do you do? retransmit packet
  - Still receiver didn't get the packet
  - Retransmit again
  - .... and so on ...
  - And now assume that everyone is doing the same!
- Network will become more and more congested
  - And this with duplicate packets rather than new packets!

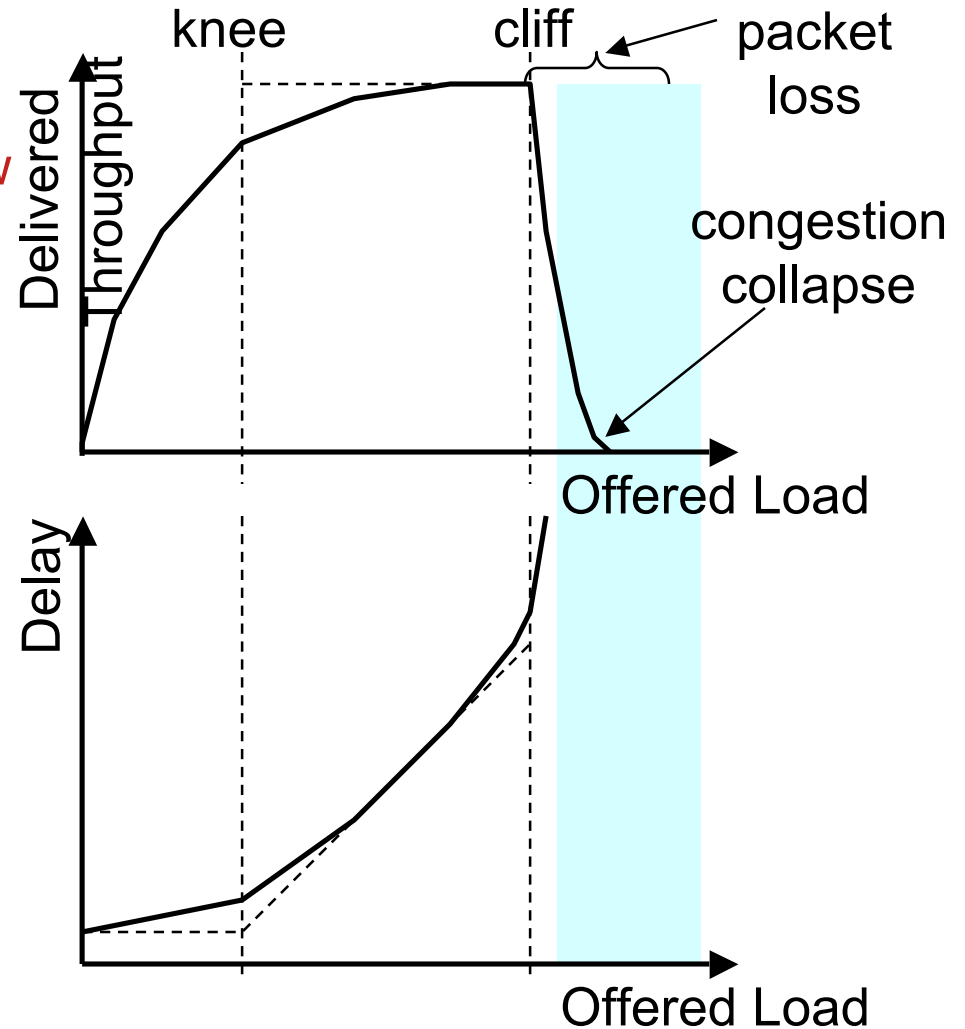
# Solutions?

---

- Increase buffer size. Why not?
- Slow down
  - If you know that your packets are not delivered because network congestion, slow down
- Questions:
  - How do you detect network congestion?
  - By how much do you slow down?

# What's Really Happening?

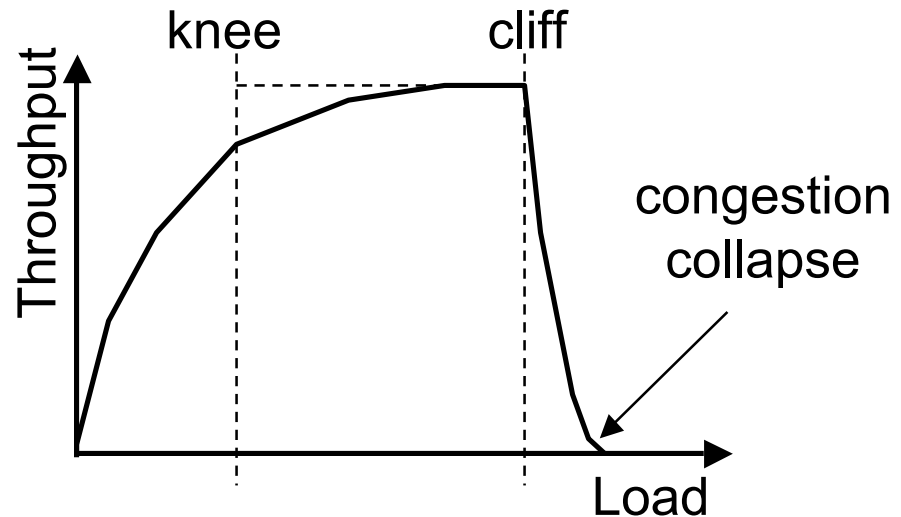
- Knee – point after which
  - Throughput **increases very slow**
  - Delay **increases fast**
- Cliff – point after which
  - Throughput starts to **decrease very fast to zero** (congestion collapse)
  - Delay **approaches infinity**
- Note (in an M/M/1 queue)
  - Delay =  $1/(1 - \text{utilization})$



# Congestion Control vs. Congestion Avoidance

---

- Congestion control goal
  - Stay left of cliff
- Congestion avoidance goal
  - Stay left of knee



# Goals

---

- Operate near the knee point
- Remain in equilibrium
- How to maintain equilibrium?
  - Don't put a packet into network until another packet leaves. How do you do it?
  - Use ACK: send a new packet only after you receive and ACK. Why?
  - Maintain number of packets in network “constant”

# Precise Goals

---

- Efficiency
  - Utilize available bandwidth as much as possible
- Fairness
  - All hosts get equal access to bandwidth
  - Jain's fairness index
- Distributed implementation
  - Only require state at endpoints
- Convergence
  - For constant load, arrive at single solution for using/sharing bandwidth

# How Do You Do It?

---

- Detect when network approaches/reaches knee point
- Stay there
- Questions
  - How do you get there?
  - What if you overshoot (i.e., go over knee point) ?
- Possible solution:
  - Increase window size until you notice congestion
  - Decrease window size if network congested

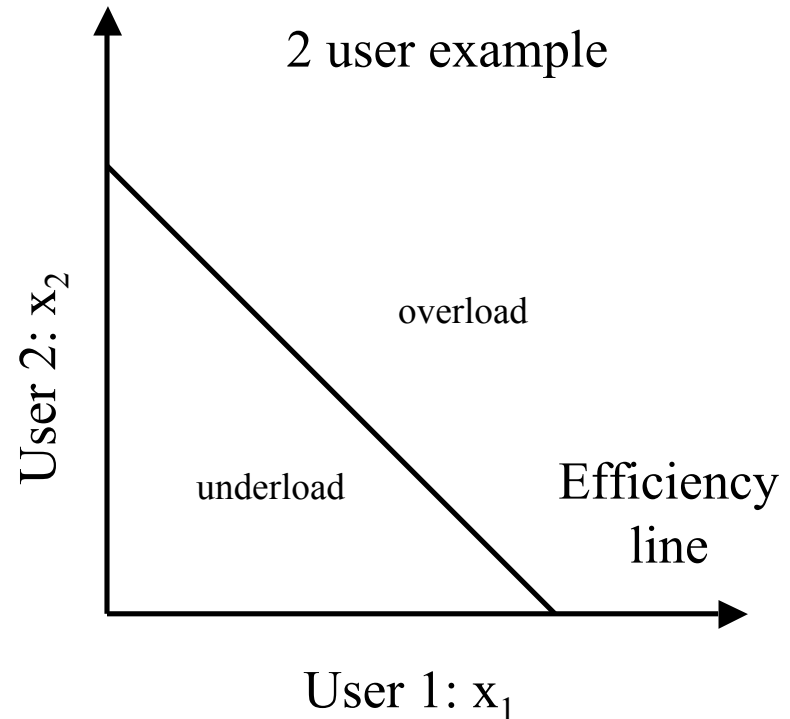
# Detecting Congestion

---

- **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
- **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?

# Efficient Allocation

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

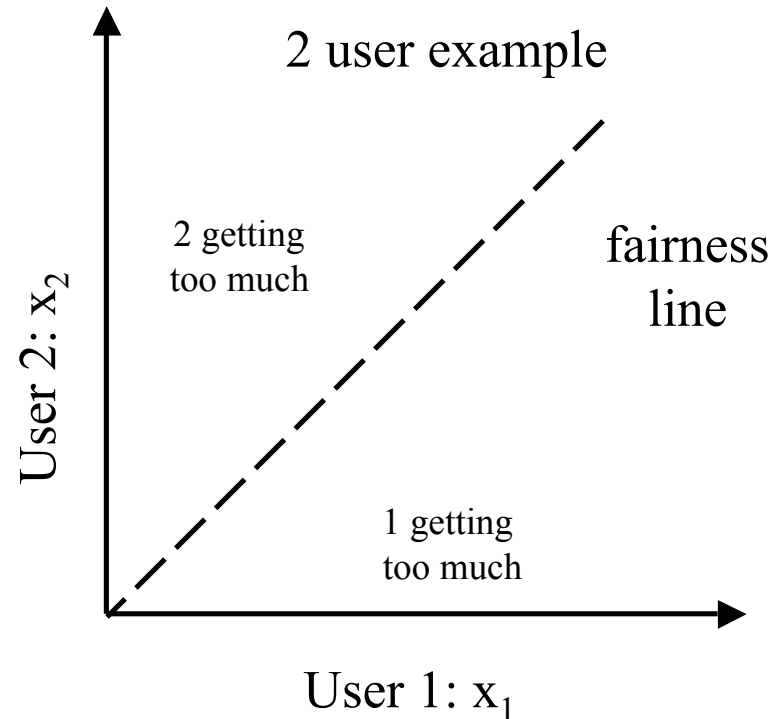


# Fair Allocation

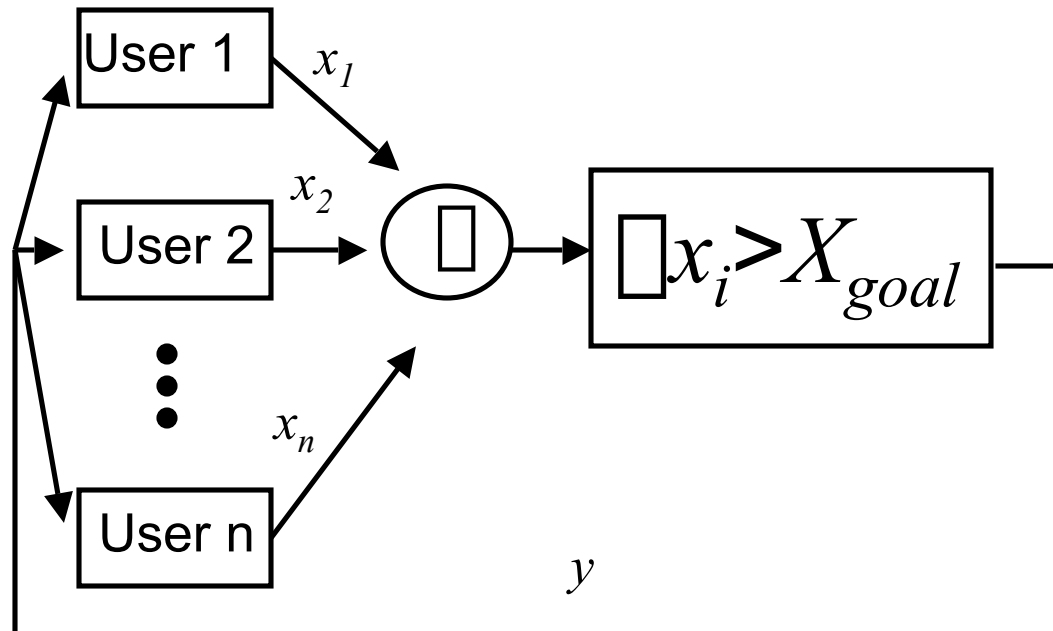
- Maxmin fairness
  - Flows which share the same bottleneck get the same amount of bandwidth

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

- Assumes no knowledge of priorities
- 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

---

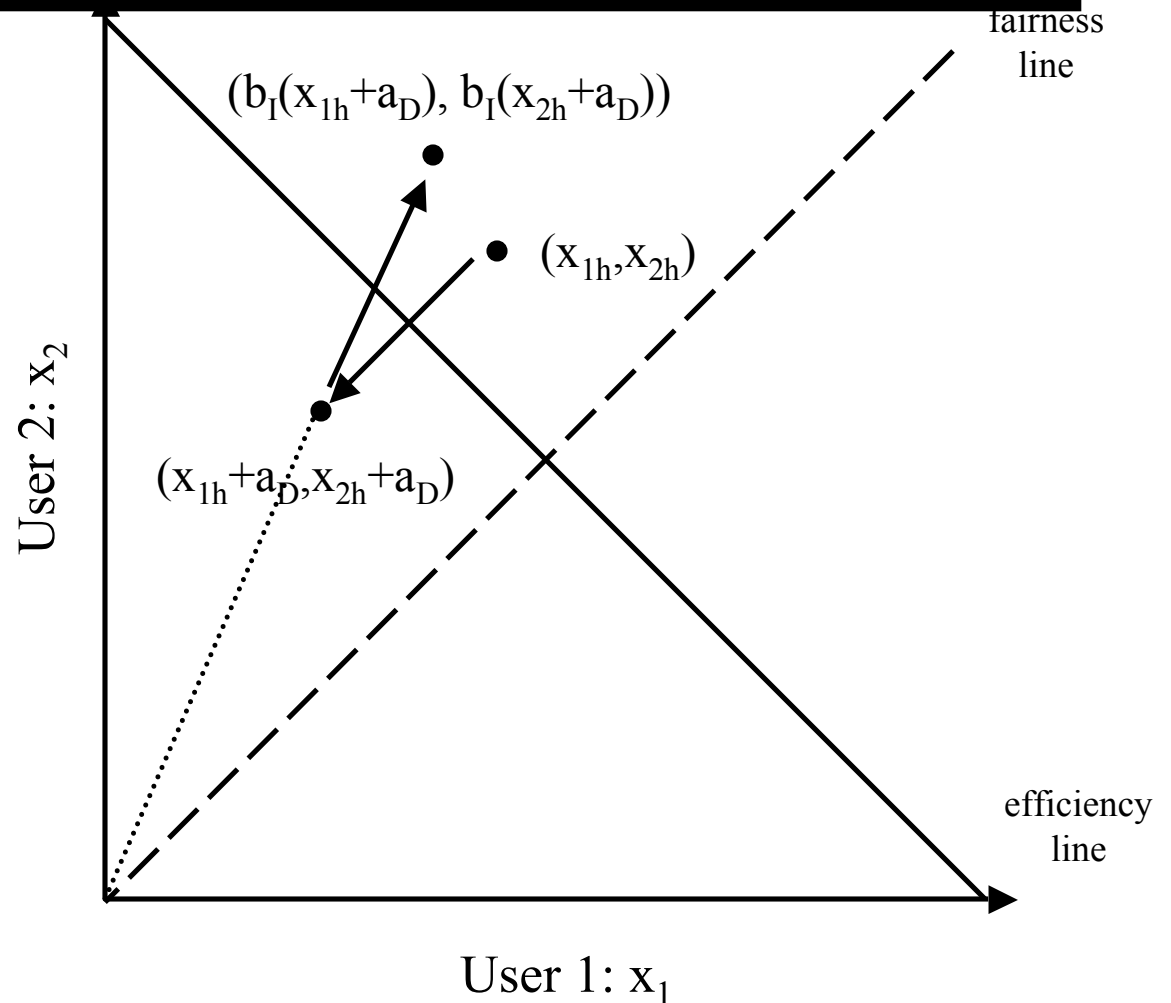
$$x_i(t+1) = \begin{cases} a_I + b_I x_i(t) & \text{increase} \\ a_D + b_D x_i(t) & \text{decrease} \end{cases}$$

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

# Multiplicative Increase, Additive Decrease

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

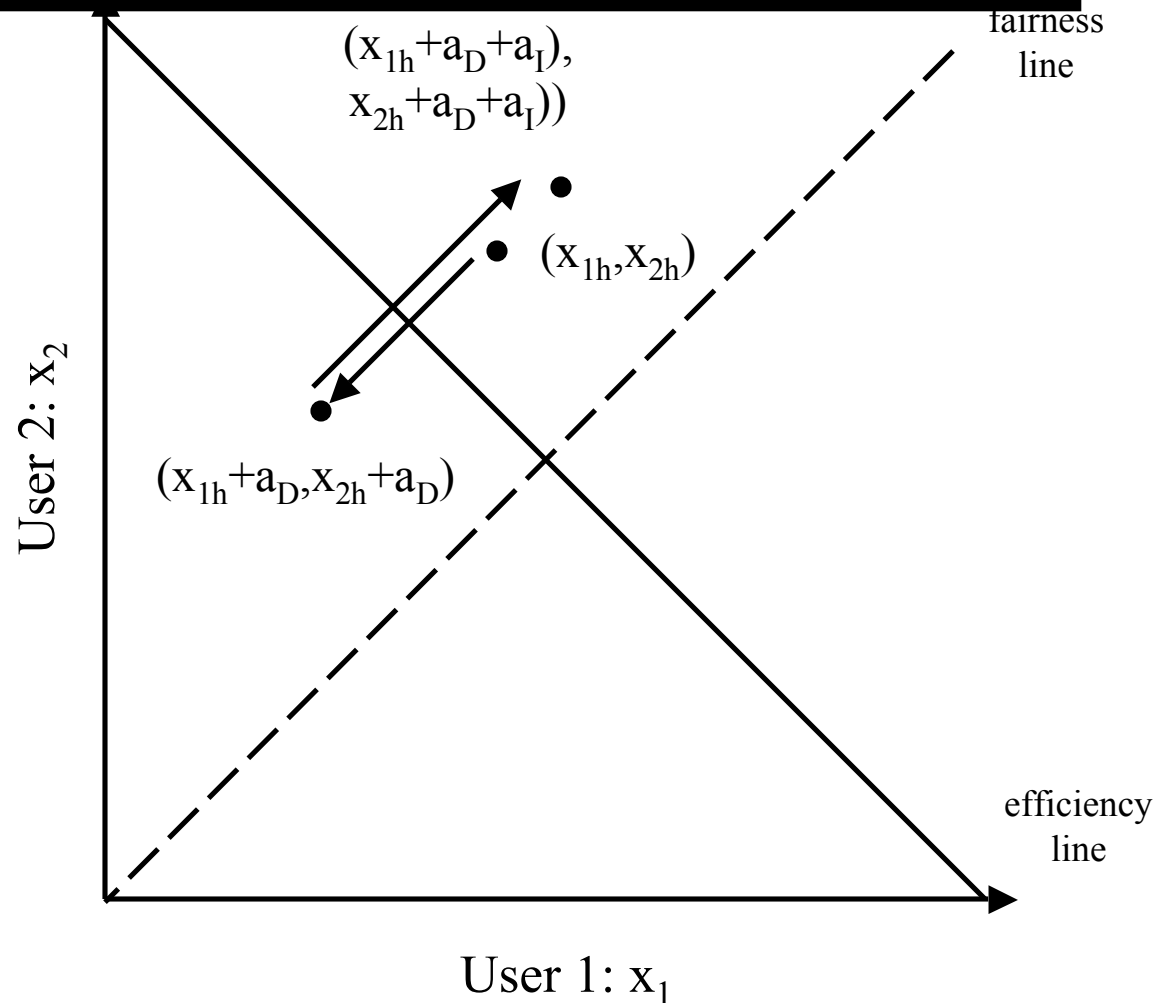
$$x_{1h} = x_{2h} = \frac{b_I a_D}{1 \square b_I}$$



# Additive Increase, Additive Decrease

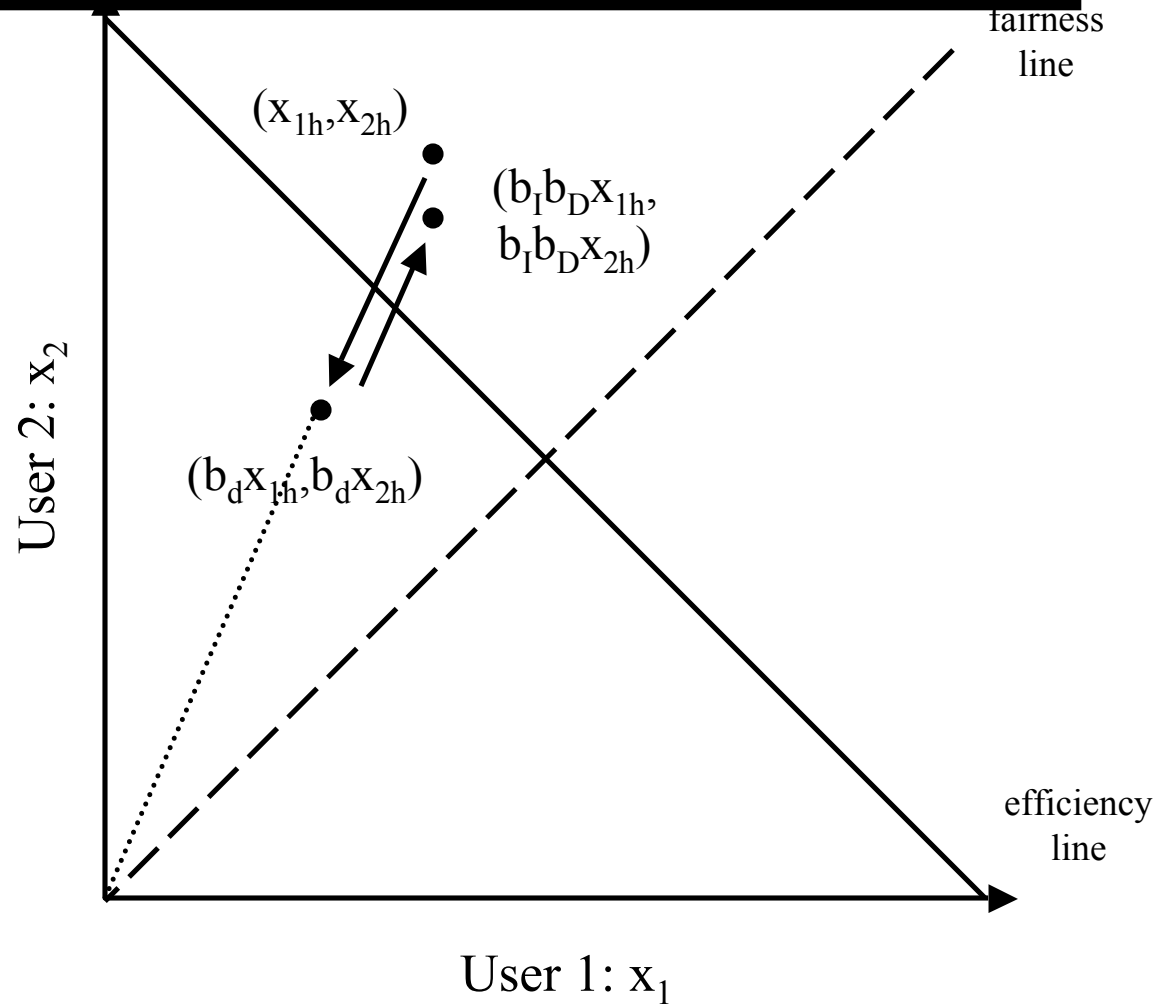
- Does not converge to fairness
  - Stable
- Does not converge to efficiency
  - Stable iff

$$a_D = a_I$$



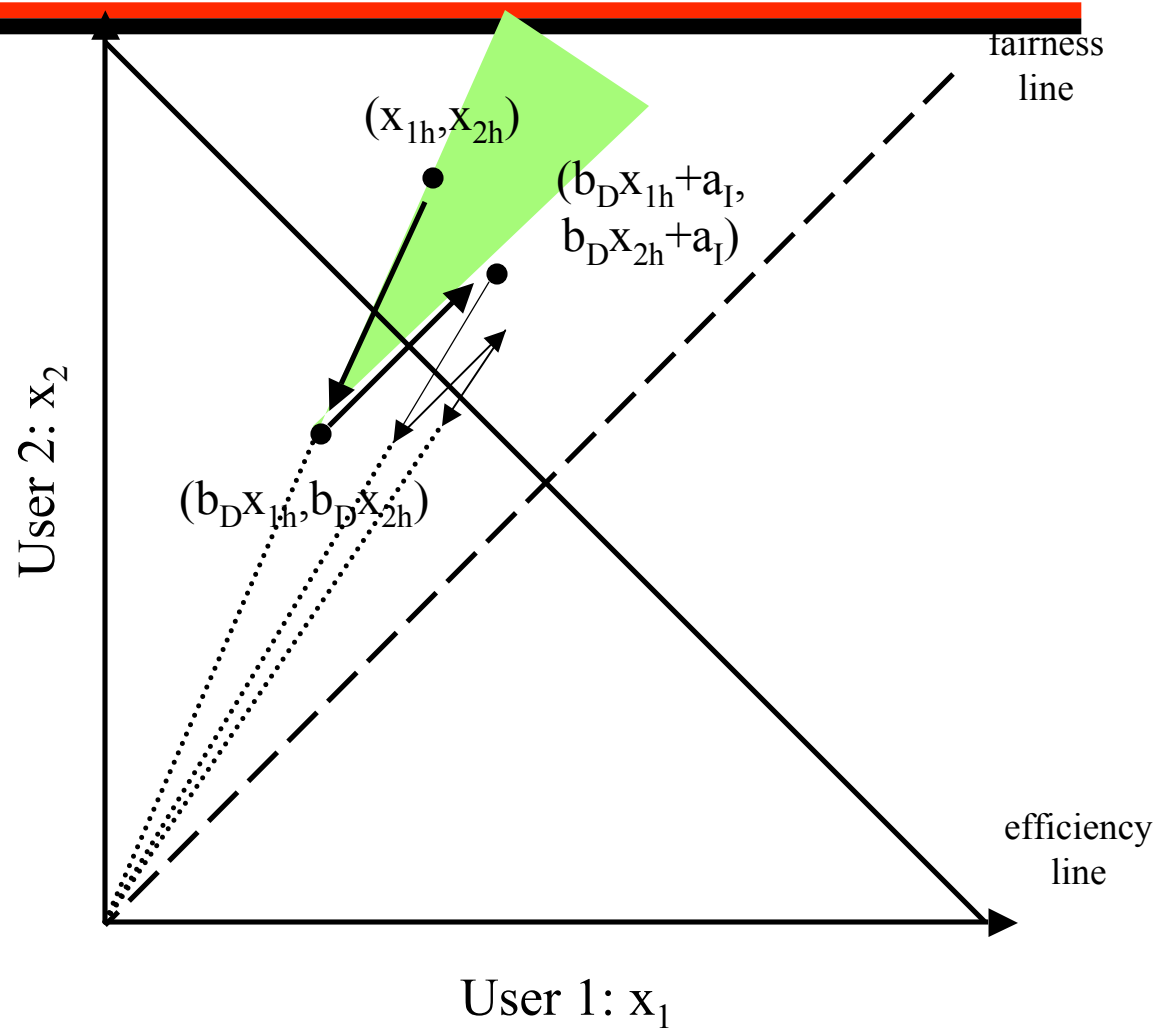
# Multiplicative Increase, Multiplicative Decrease

- Does not converge to fairness
  - Stable
- Converges to efficiency  
iff  $b_I \geq 1$   
 $0 \leq b_D < 1$



# Additive Increase, Multiplicative Decrease

- Converges to fairness
- Converges to efficiency
- Increments smaller as fairness increases
  - effect on metrics?



# 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 after 15 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  useless
  - Unrealistic, simple model  can teach something about best case, worst case, etc.

# TCP Congestion Control

---

- [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:  $\text{win} = \mathbf{\min}(\text{flow\_win}, \text{cwnd})$

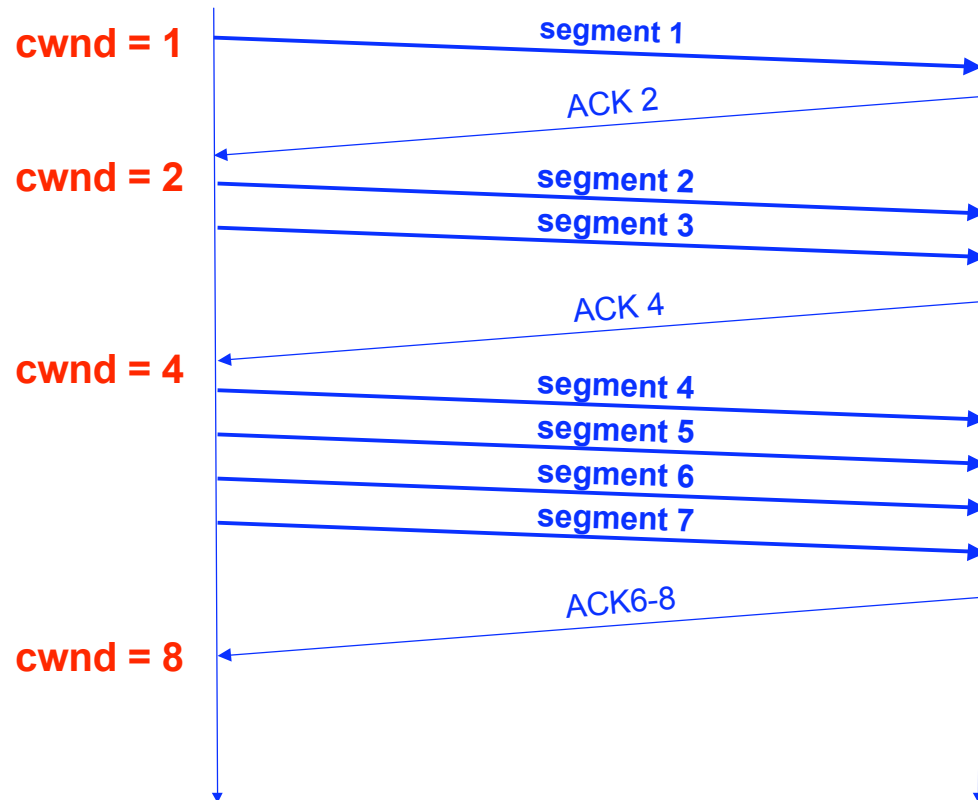
# TCP: Slow Start

---

- 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*  $\geq$  *ssthresh***



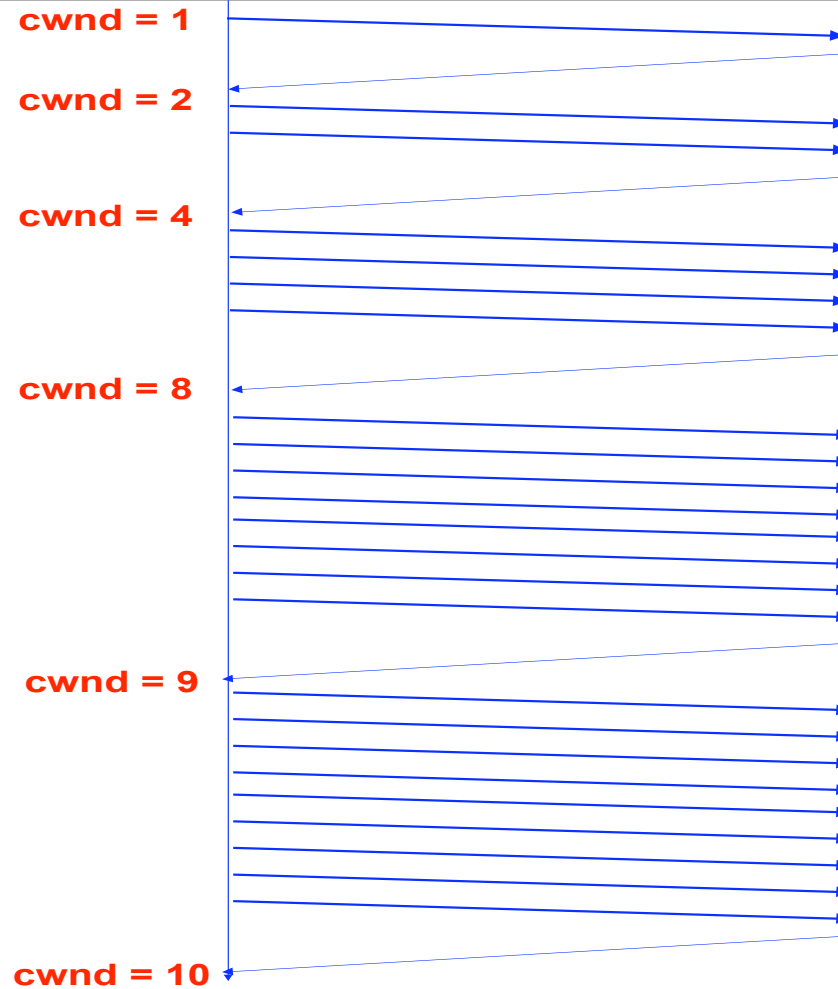
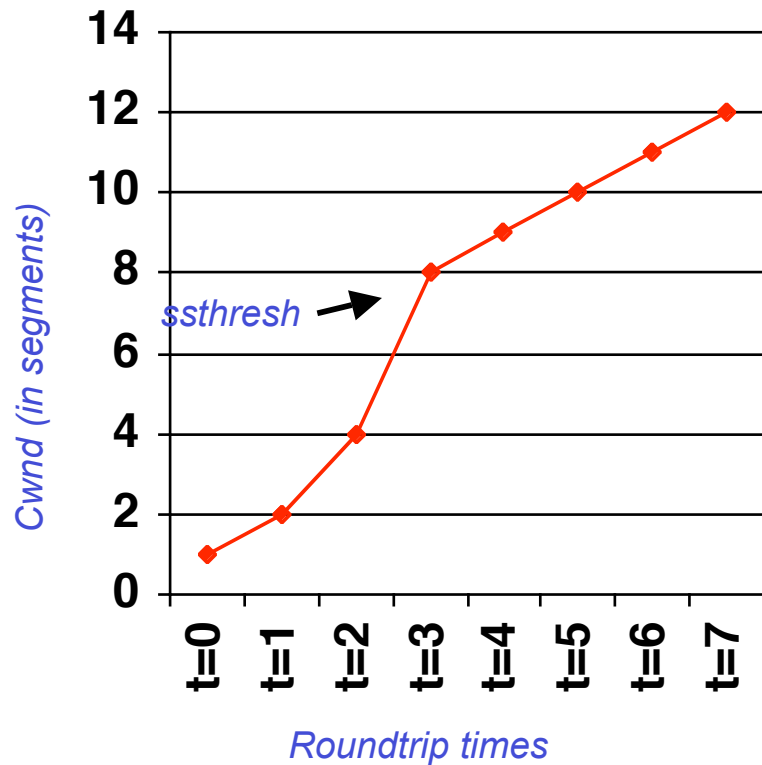
# Congestion Avoidance

---

- 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 acknowledged.
- (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

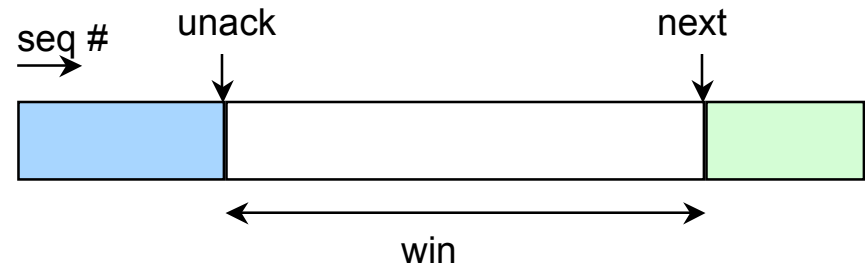
```
  /* Congestion Avoidance */  
  cwnd = cwnd + 1/cwnd;
```

## Timeout:

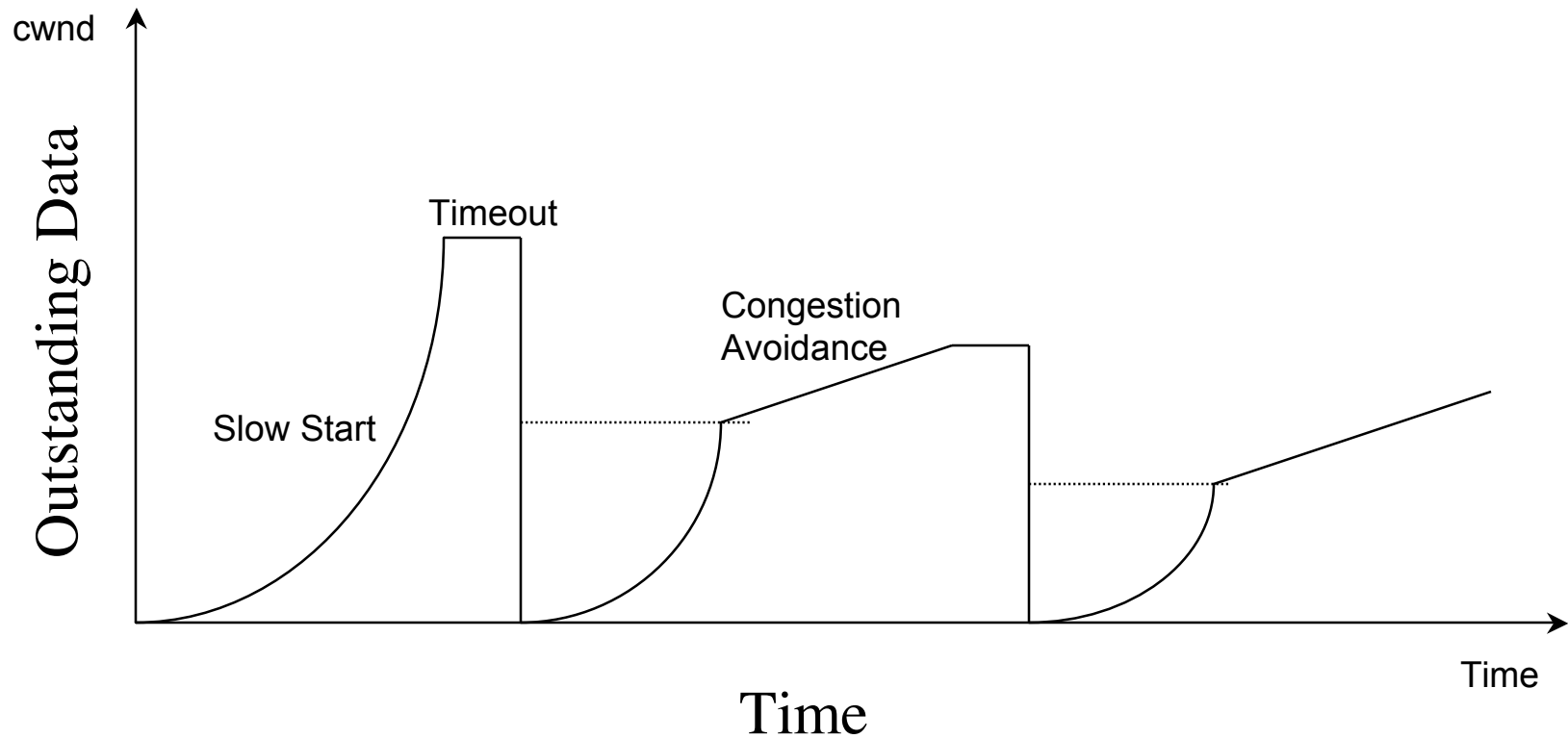
```
  /* Multiplicative decrease */  
  ssthresh = cwnd/2;  
  cwnd = 1;
```

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

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



# The big picture



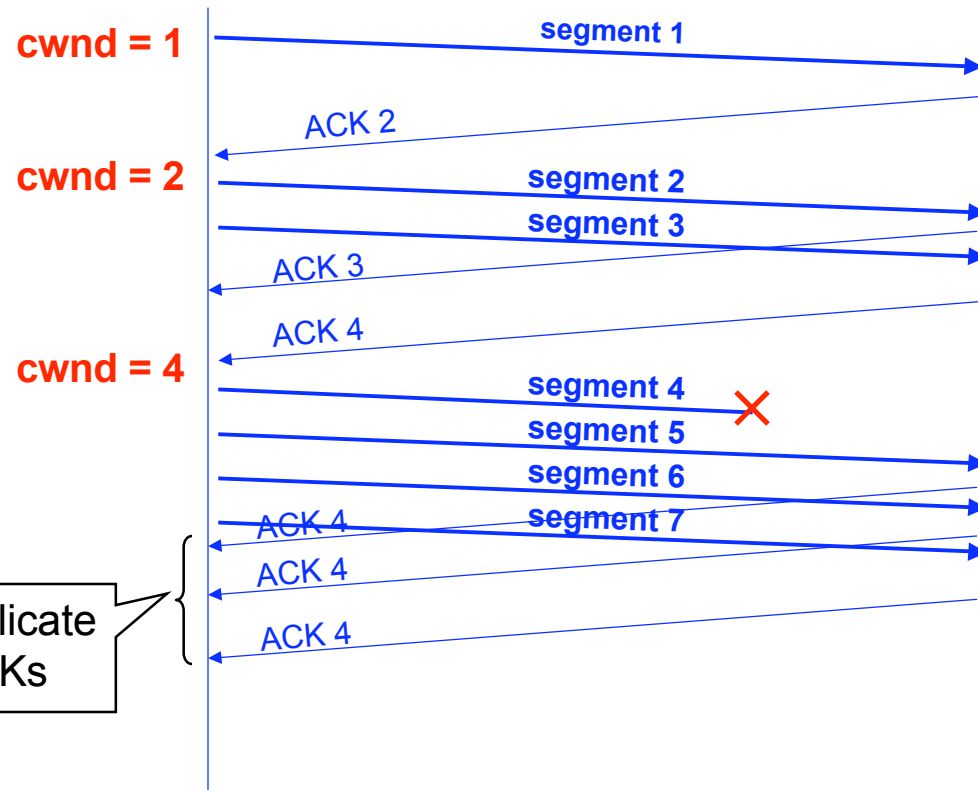
# Fast Retransmit

- 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?
- window may be too small to get duplicate ACKs

3 duplicate ACKs

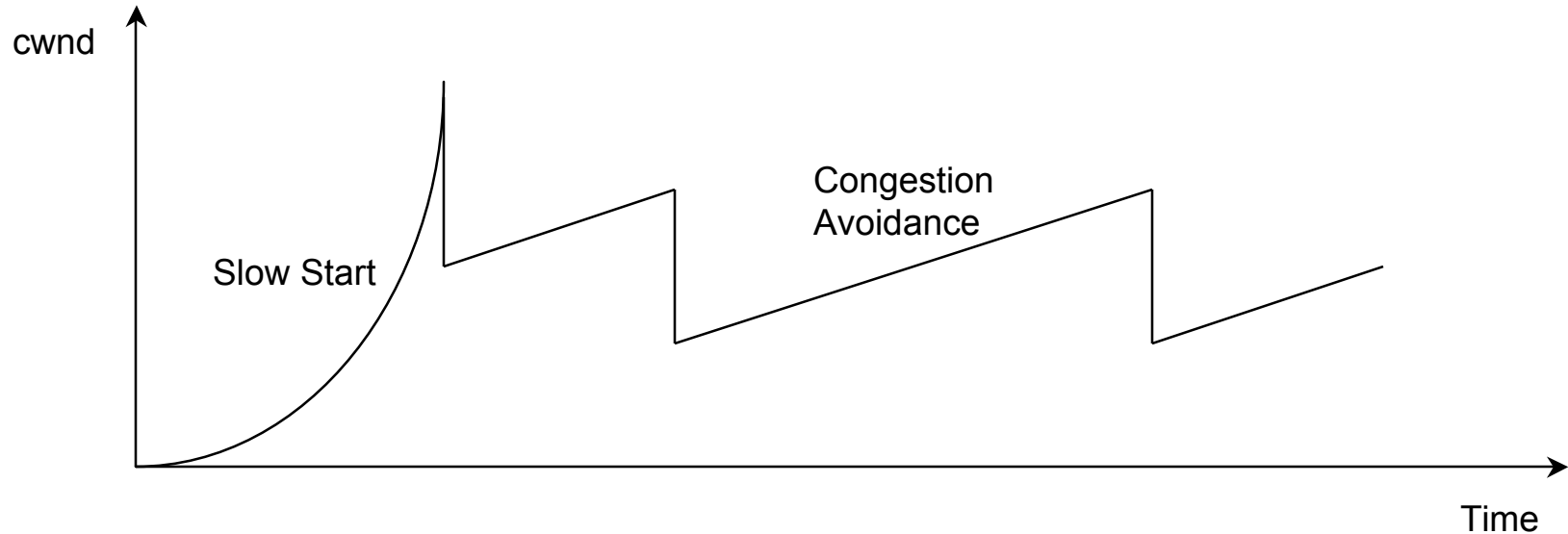


# Fast Recovery

---

- 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.

# Historical TCP's

---

- TCP Tahoe (1983)
  - Slow-Start, Fast Retransmit, Congestion Avoidance
- TCP Reno (1990)
  - Tahoe + Fast Recovery
- TCP New-Reno (1996)
  - Reno + Hoe's partial ACK change that keeps TCP in Fast Recovery
- SACK TCP (1996)
  - Selective acknowledgements
- TCP Vegas (1993)
  - Novel method of congestion avoidance

# Newer TCPs

---

- Multiple TCPs
  - E.g. Used in web-browsers
- HSTCP
- STCP
- FAST TCP

# Problem

---

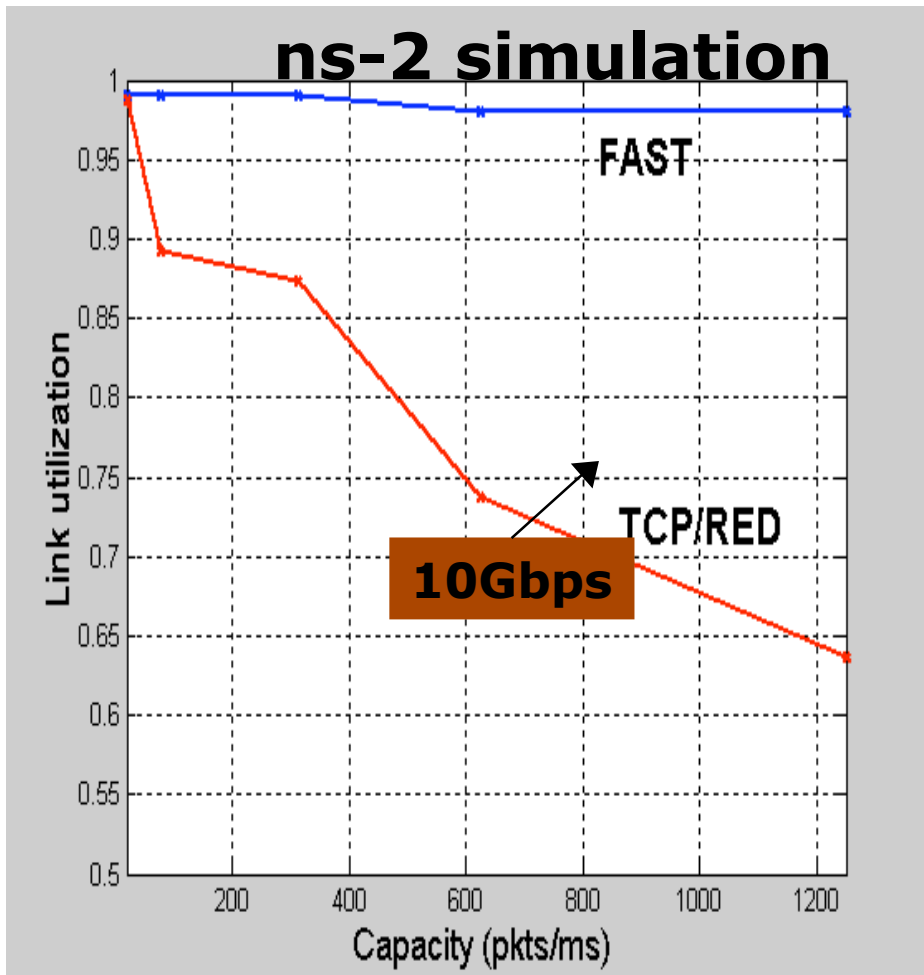
- How much traffic do you send?
- Two components
  - Flow control – make sure that the receiver can receive as fast as you send
  - Congestion control – make sure that the network delivers the packets to the receiver

# Difficulties at large windows

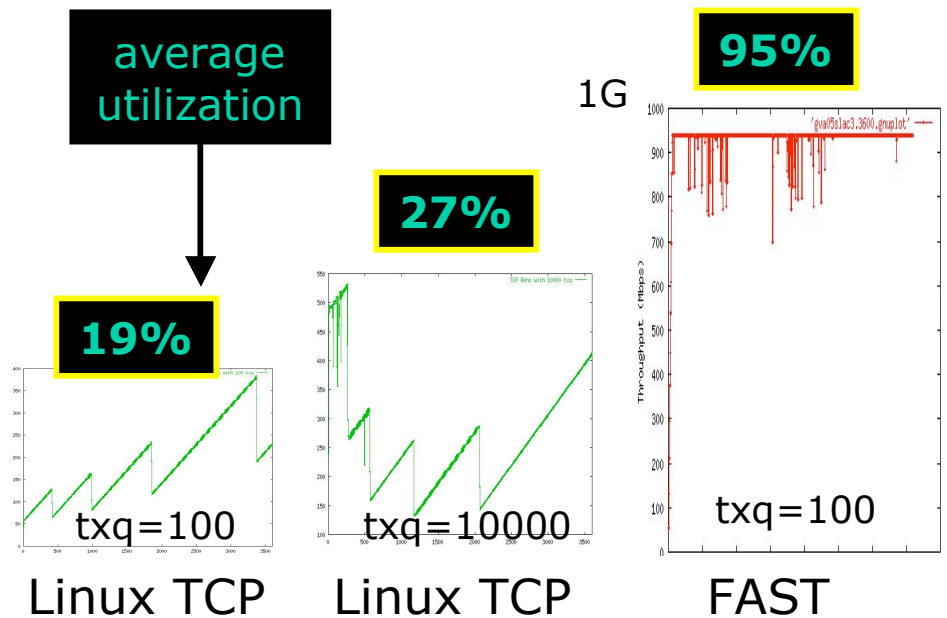
---

- Equilibrium problem
  - Packet level: AI too slow, MD too drastic.
  - Flow level: requires very small loss probability.
- Dynamic problem
  - Packet level: must oscillate on a binary signal.
  - Flow level: unstable at large window.

# Performance at large windows



DataTAG Network:  
CERN (Geneva) –  
StarLight (Chicago) –  
SLAC/Level3 (Sunnyvale)



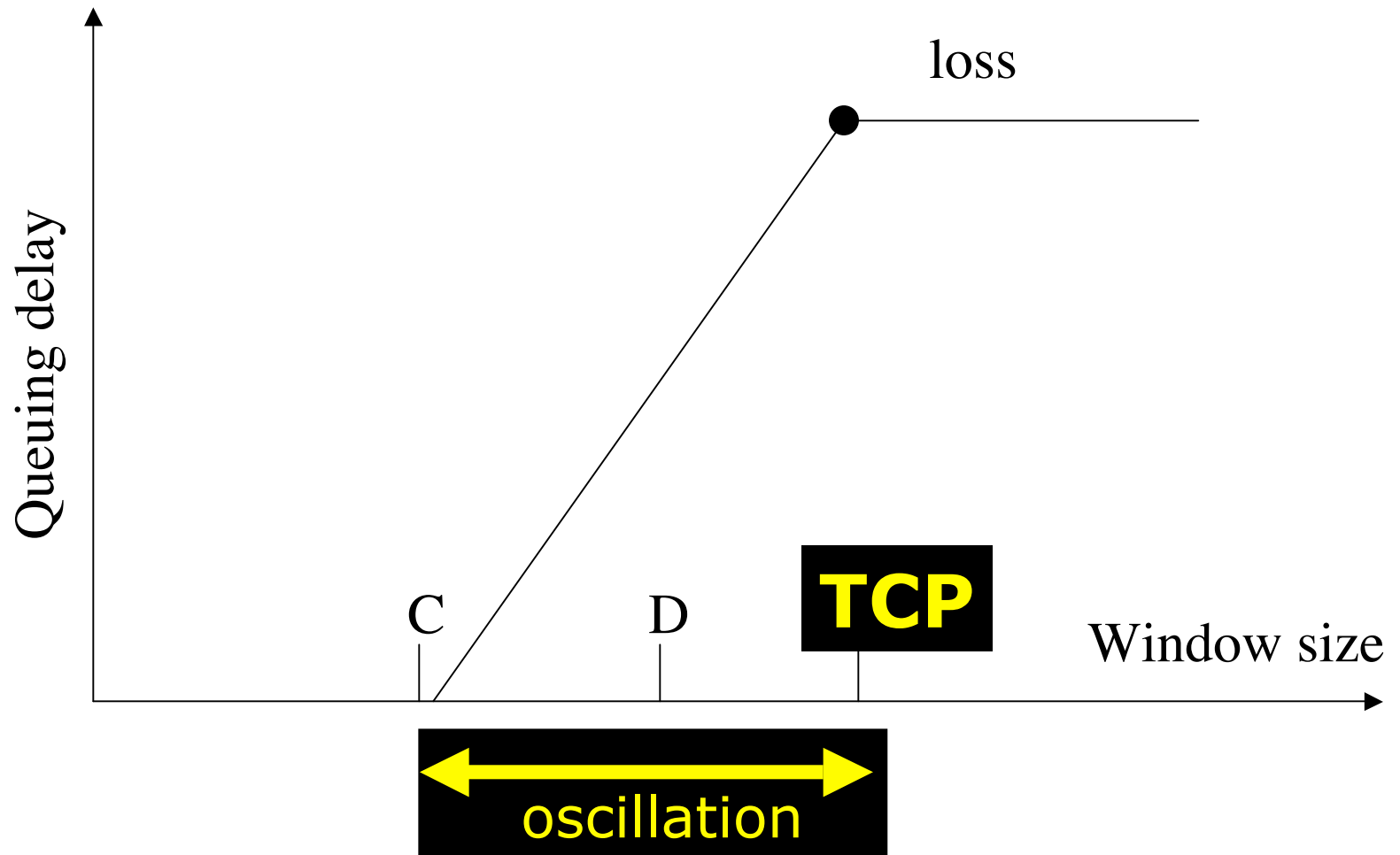
capacity = 1Gbps; 180 ms RTT, 1 flow

# Difficulties at large window

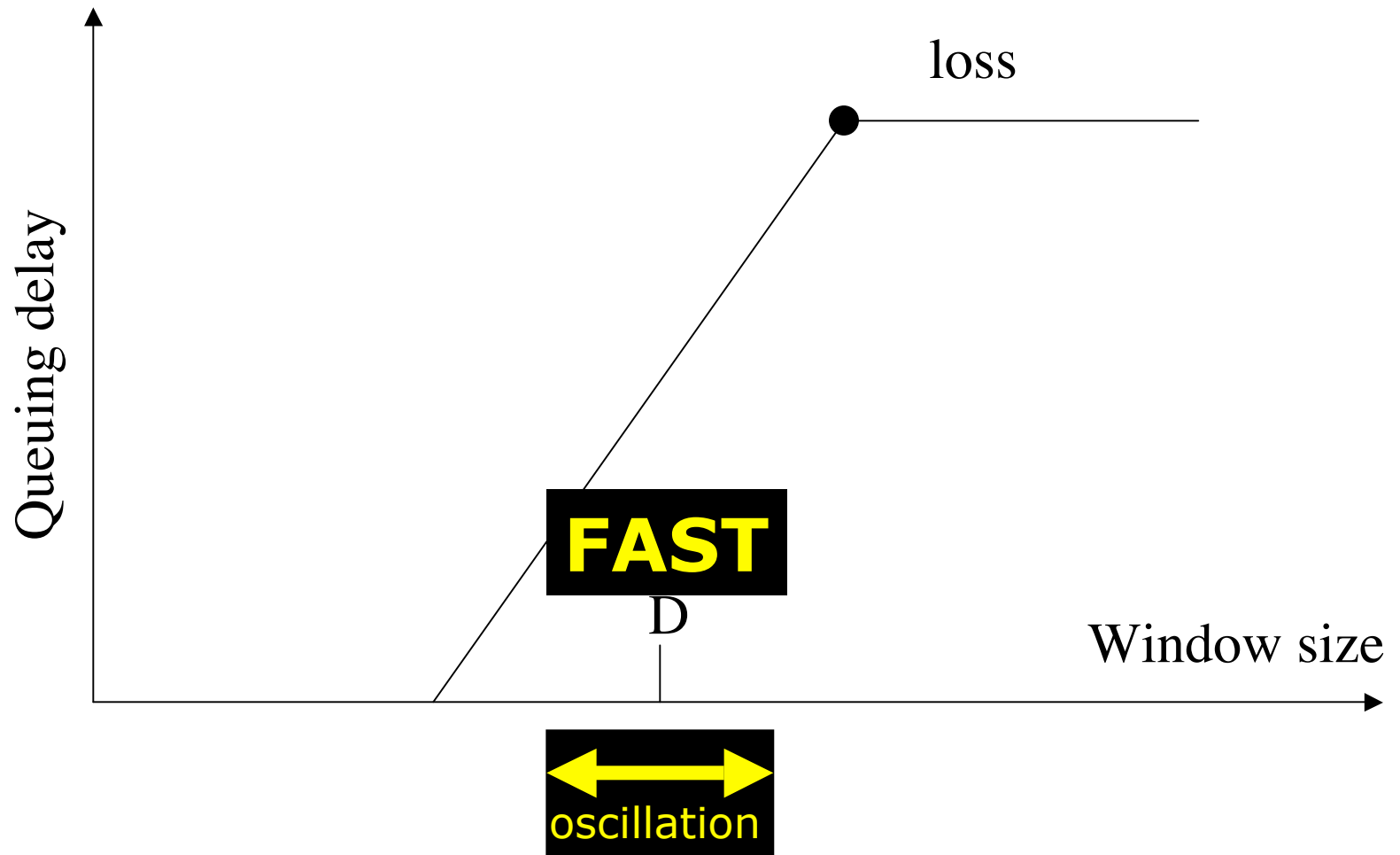
---

- Equilibrium problem
  - Packet level: AI too slow, MD too drastic.
  - Flow level: requires very small loss probability.
- Dynamic problem
  - Packet level: must oscillate on a binary signal.
  - Flow level: unstable at large window.

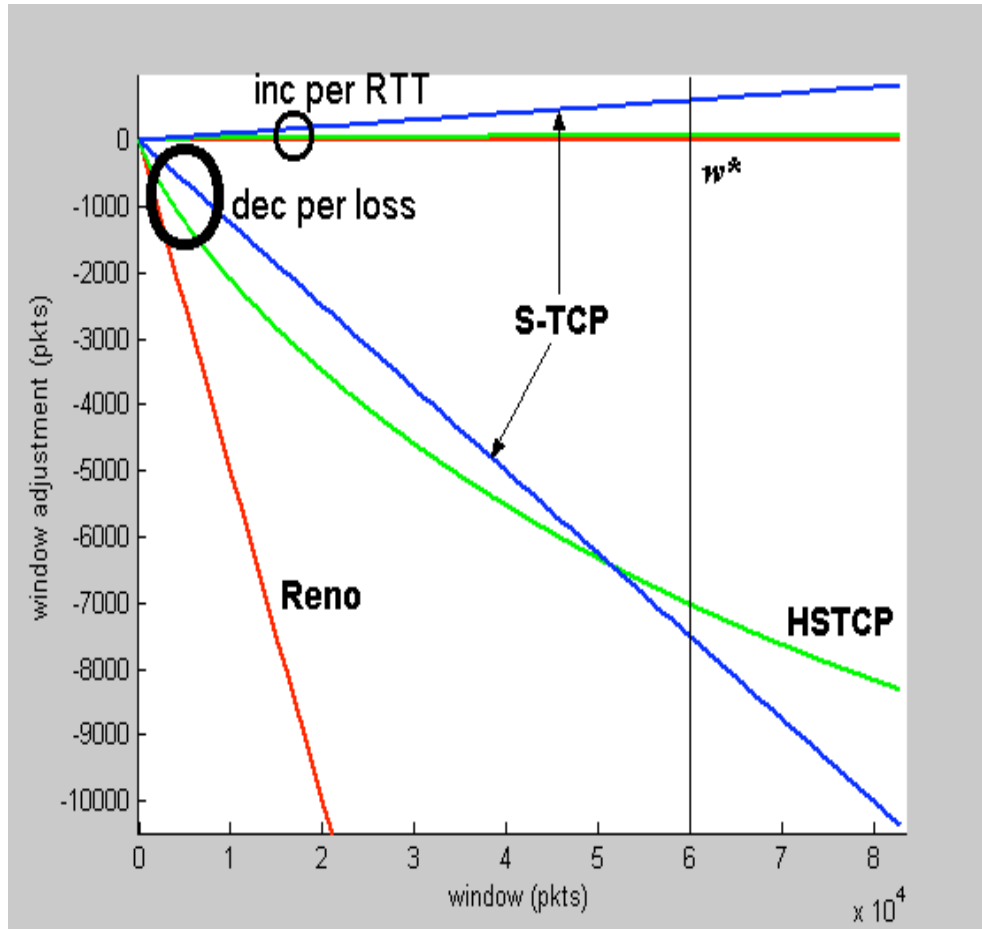
# Problem: binary signal



# Solution: multi-bit signal



# Problem: no target



- Reno: AIMD (1, 0.5)

$$\text{ACK: } W \leftarrow W + 1/W$$

$$\text{Loss: } W \leftarrow W - 0.5W$$

- HSTCP: AIMD ( $a(w)$ ,  $b(w)$ )

$$\text{ACK: } W \leftarrow W + a(w)/W$$

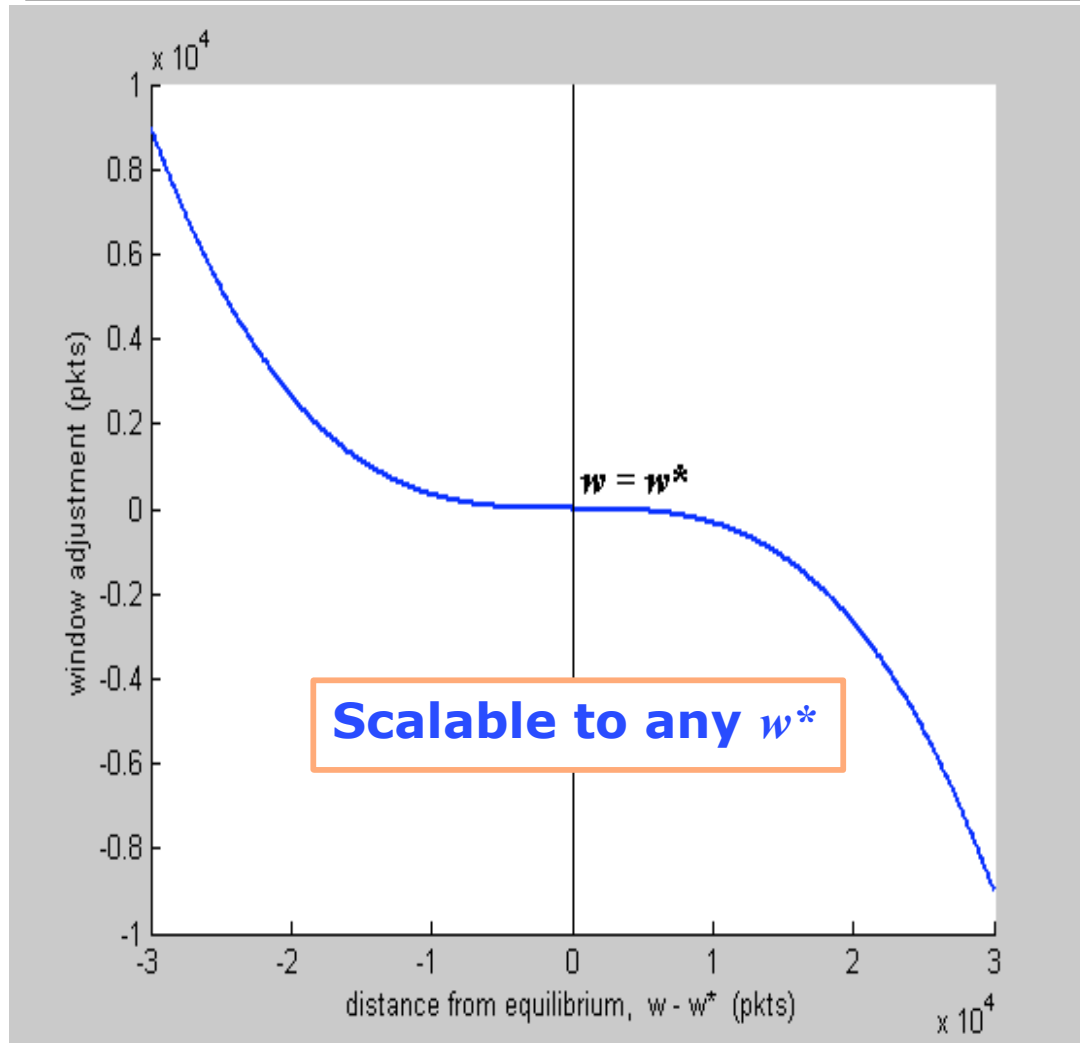
$$\text{Loss: } W \leftarrow W - b(w)W$$

- STCP: MIMD (1/100, 1/8)

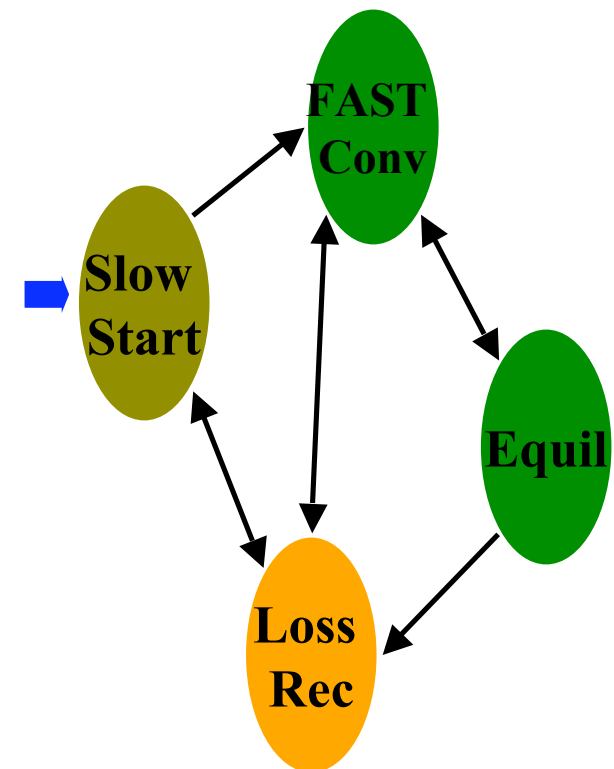
$$\text{ACK: } W \leftarrow W + 0.01$$

$$\text{Loss: } W \leftarrow W - 0.125W$$

# Solution: estimate target



□ FAST

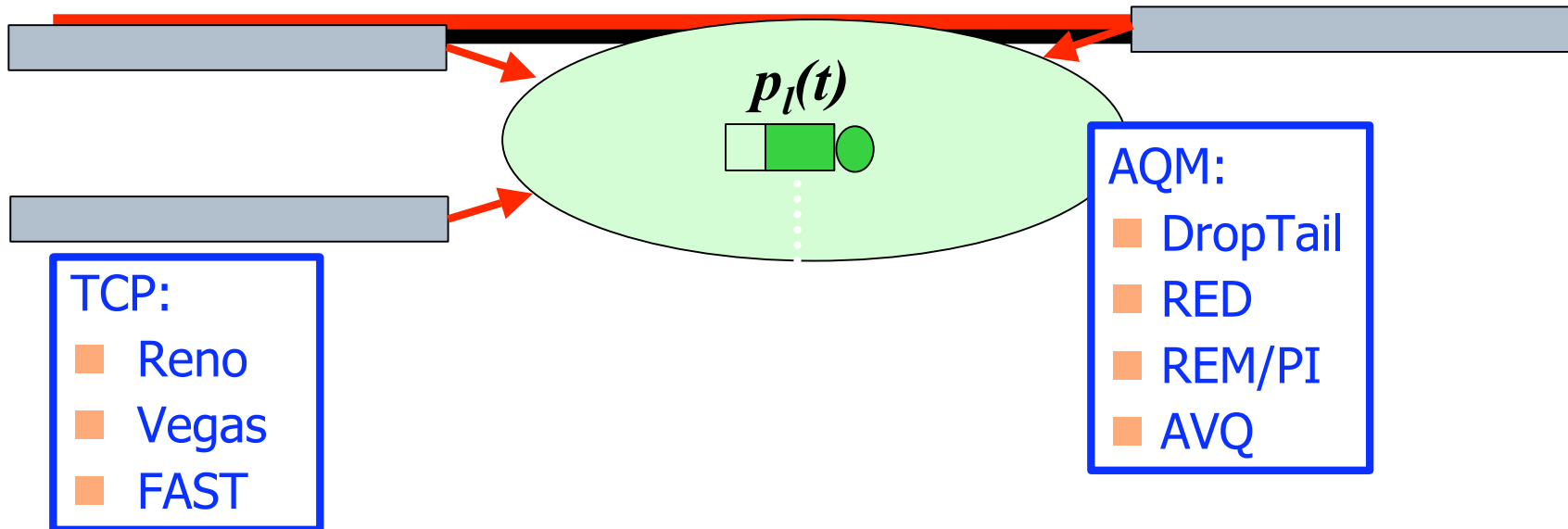


# Brief History of FAST TCP

---

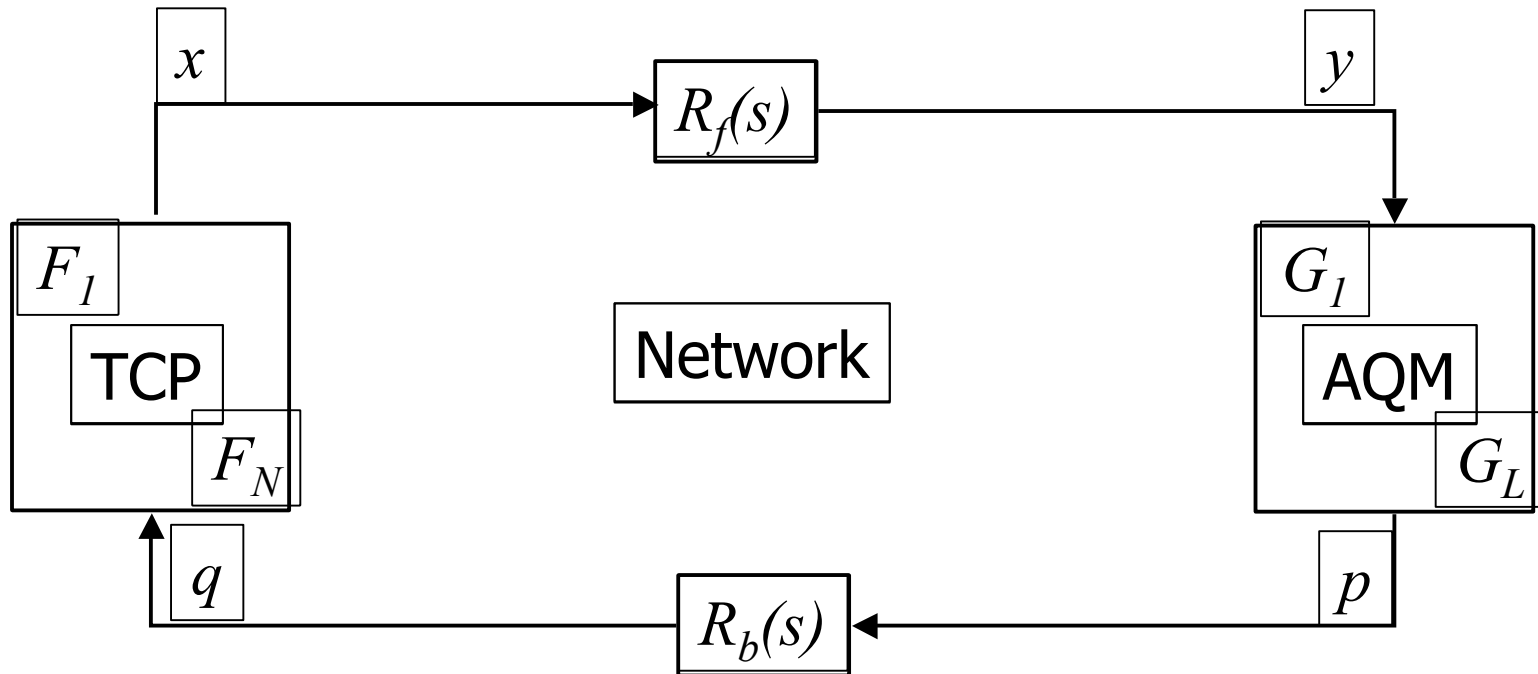
- Congestion control as an optimization problem
- Primal-dual framework to study TCP congestion control
- Modeling existing TCP implementations
- Theoretical analysis on FAST TCP
- FAST TCP Implementation

# TCP/AQM



- Congestion control has two components
  - TCP: adjusts rate according to congestion
  - AQM: feeds back congestion based on utilization
- Distributed feed-back system
  - equilibrium and stability properties determine system performance

# Network Model



- Components: TCP and AQM algorithms, and routing matrices
- Each TCP source sees an aggregate price,  $q$
- Each link sees an aggregate incoming rate

# FAST and Other DCAs

---

- FAST is one implementation within the more general primal-dual framework
- Queueing delay is used as an explicit feedback from the network
- FAST does not use queueing delay to predict or avoid packet losses
- FAST may use other forms of price in the future when they become available

# Optimization Model

---

- Network bandwidth allocation as utility maximization
- Optimization problem

$$\begin{aligned} \max_{x_s \geq 0} \quad & \sum_s U_s(x_s) \\ \text{subject to} \quad & y_l \leq c_l, \quad \forall l \in L \end{aligned}$$

- Primal-dual components

$$\begin{aligned} x(t+1) &= F(q(t), x(t)) && \text{Source} \\ p(t+1) &= G(y(t), p(t)) && \text{Link} \end{aligned}$$

# Use of Queuing Delay in FAST

---

- Each FAST TCP flow has a target number of packets to maintain in network buffers in equilibrium
- Queueing delay allows FAST to estimate the number of packets currently buffered and estimate its distance from the target

# FAST TCP

---

- Flow level
    - Understood and Synthesized first
  - Packet level
    - Designed and implemented later
- Design flow level equilibrium & stability
  - Implement flow level goals at packet level

# Window Control Algorithm

---

$$W \leftarrow W \cdot \frac{baseRTT}{RTT} + \alpha$$

- RTT: exponential moving average with weight of  $\min \{1/8, 3/cwnd\}$
- baseRTT: latency, or minimum RTT
- $\alpha$  determines fairness and convergence rate

# Packet Level

---

- **Reno**  
AIMD(1, 0.5)  
ACK:  $W \leftarrow W + 1/W$   
Loss:  $W \leftarrow W - 0.5 W$
- **HSTCP**  
AIMD( $a(w)$ ,  $b(w)$ )  
ACK:  $W \leftarrow W + a(w)/W$   
Loss:  $W \leftarrow W - b(w) W$
- **STCP**  
MIMD( $a$ ,  $b$ )  
ACK:  $W \leftarrow W + 0.01$   
Loss:  $W \leftarrow W - 0.125 W$
- **FAST**  
RTT:  $W \square W \cdot \frac{\text{baseRTT}}{\text{RTT}} + \square$

# Architecture

---

Each component

- designed independently
- upgraded asynchronously



**TCP Protocol Processing**

# Outline

---

- The problem
- Our solution

# Experiments

---

- In-house dummynet testbed
- PlanetLab Internet experiments
- Internet2 backbone experiments
- ns-2 simulations

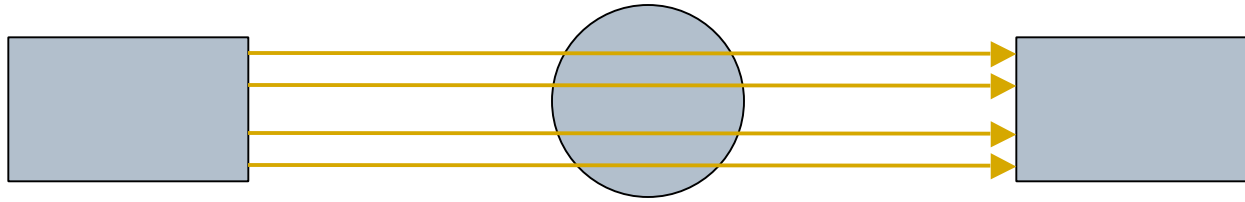
# Lesions

---

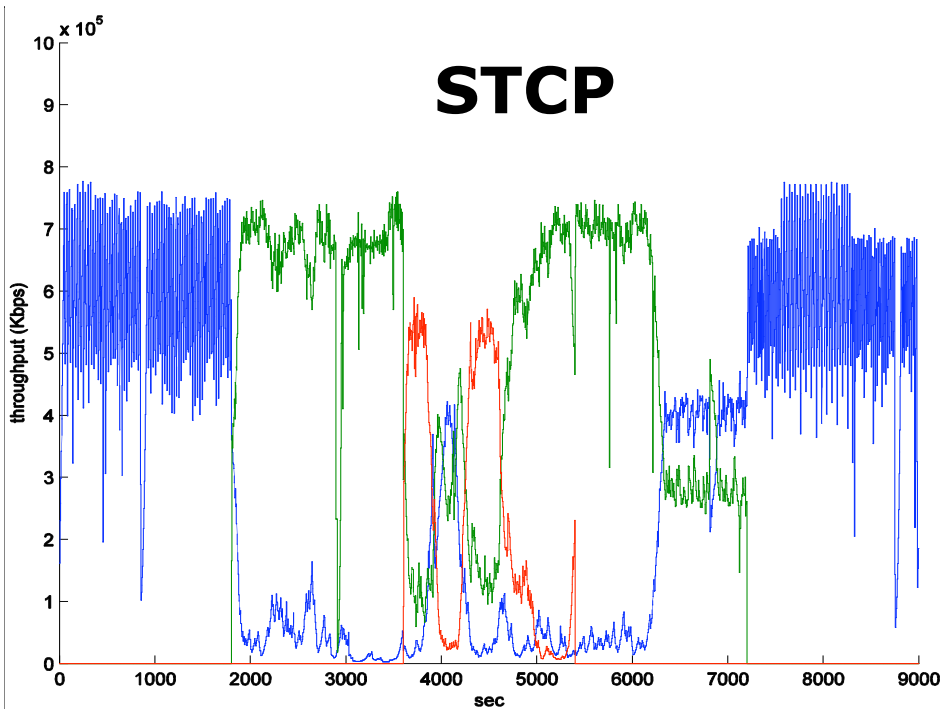
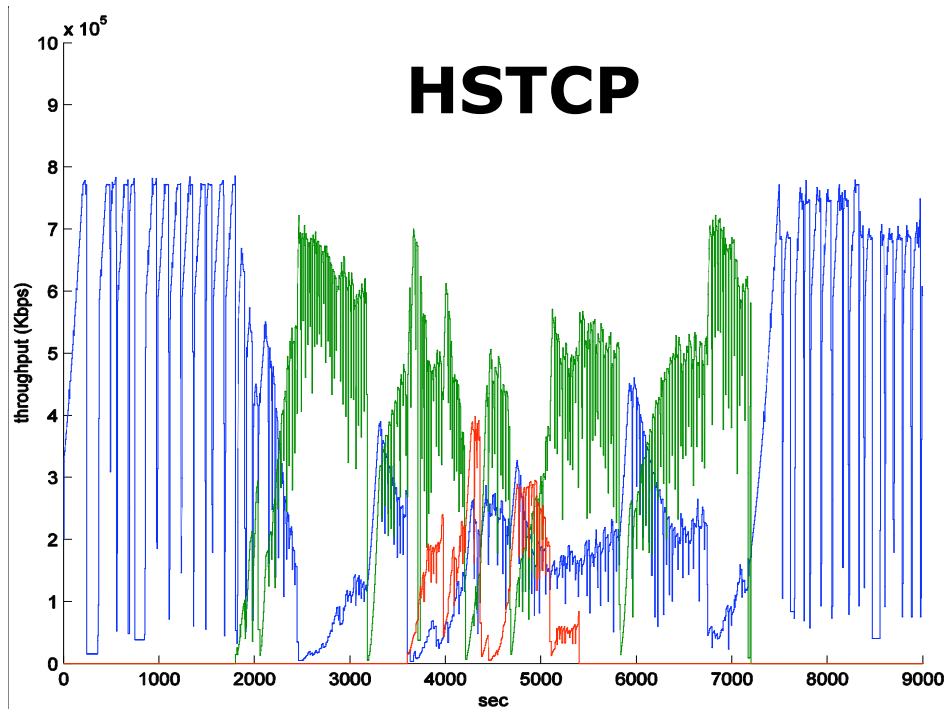
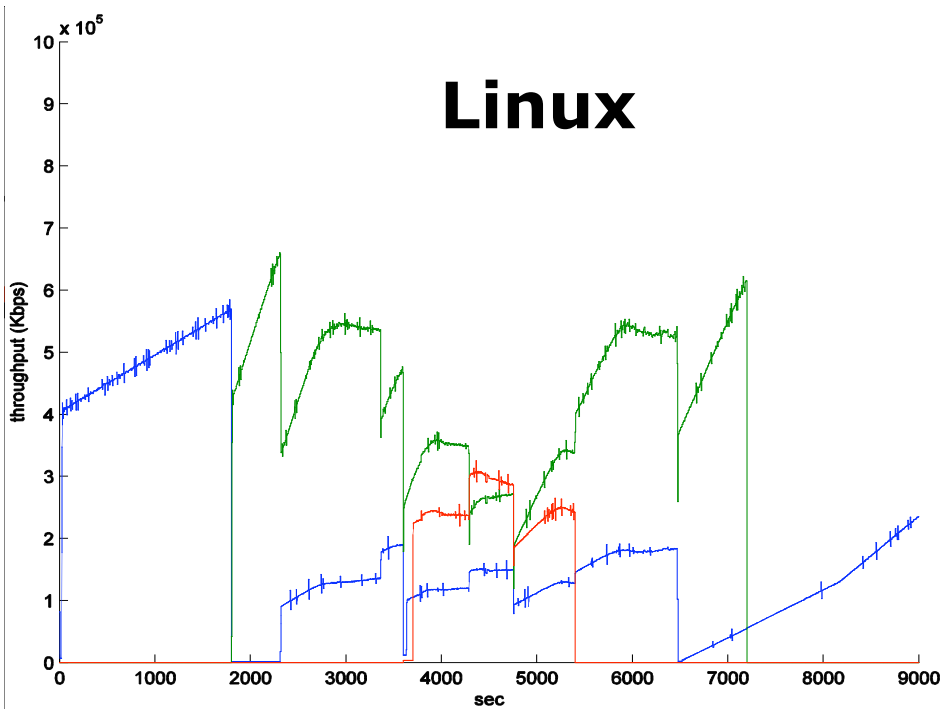
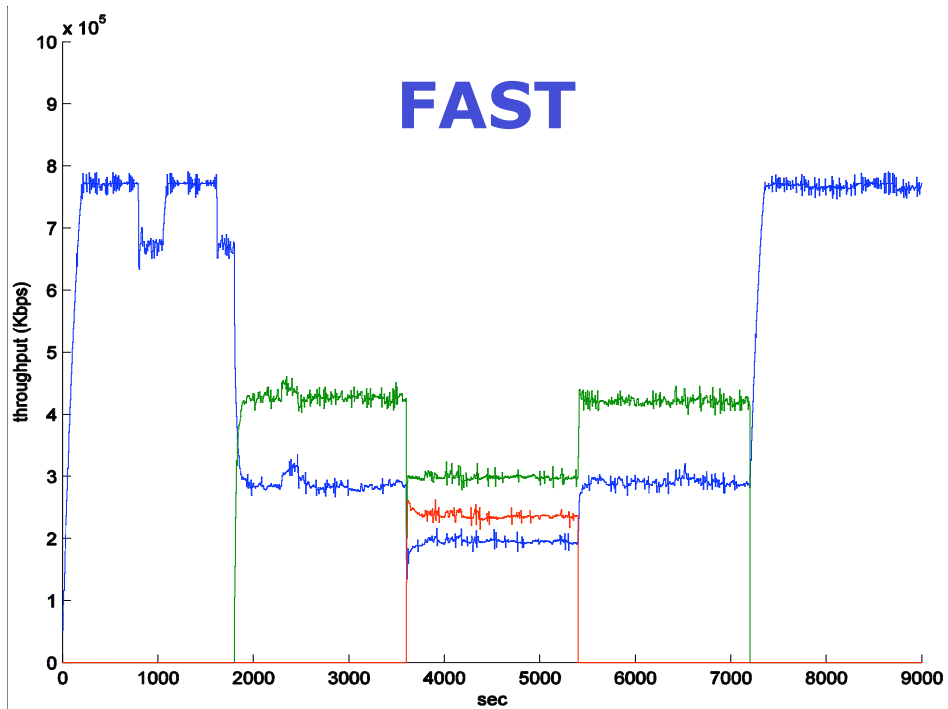
- FAST performs well under normal network conditions
- Well-known scenarios where FAST doesn't perform well
- Network behavior is important
- Dynamic scenarios are important
- Host implementation (Linux) also important

# Experimental Setup

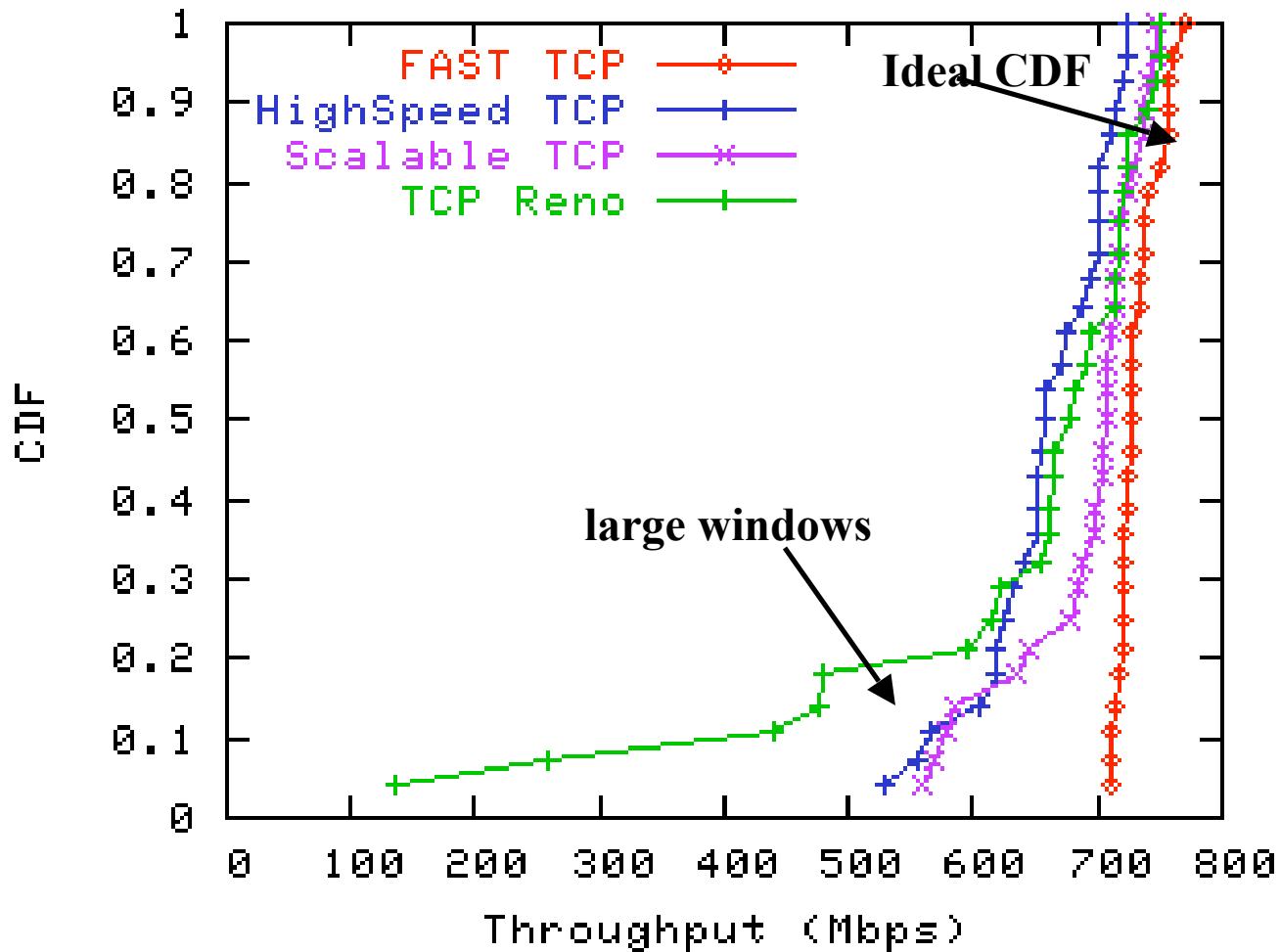
---



- Single bottleneck link, multiple path latencies
- Iperf for memory-to-memory transfers
- Intra-protocol testings
- Dynamic network scenarios
- Instrumentation on the sender and the router

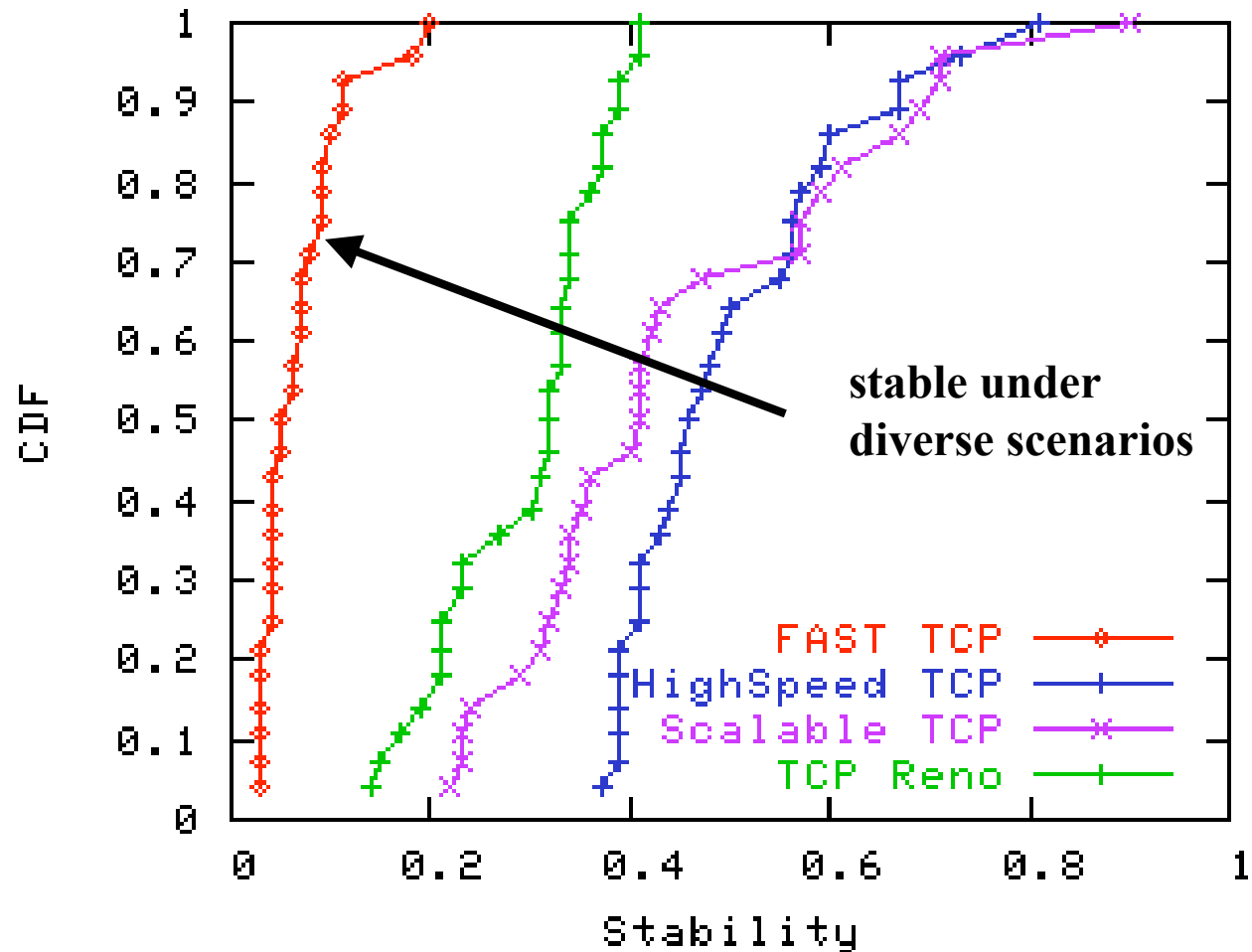


# Aggregate Throughput



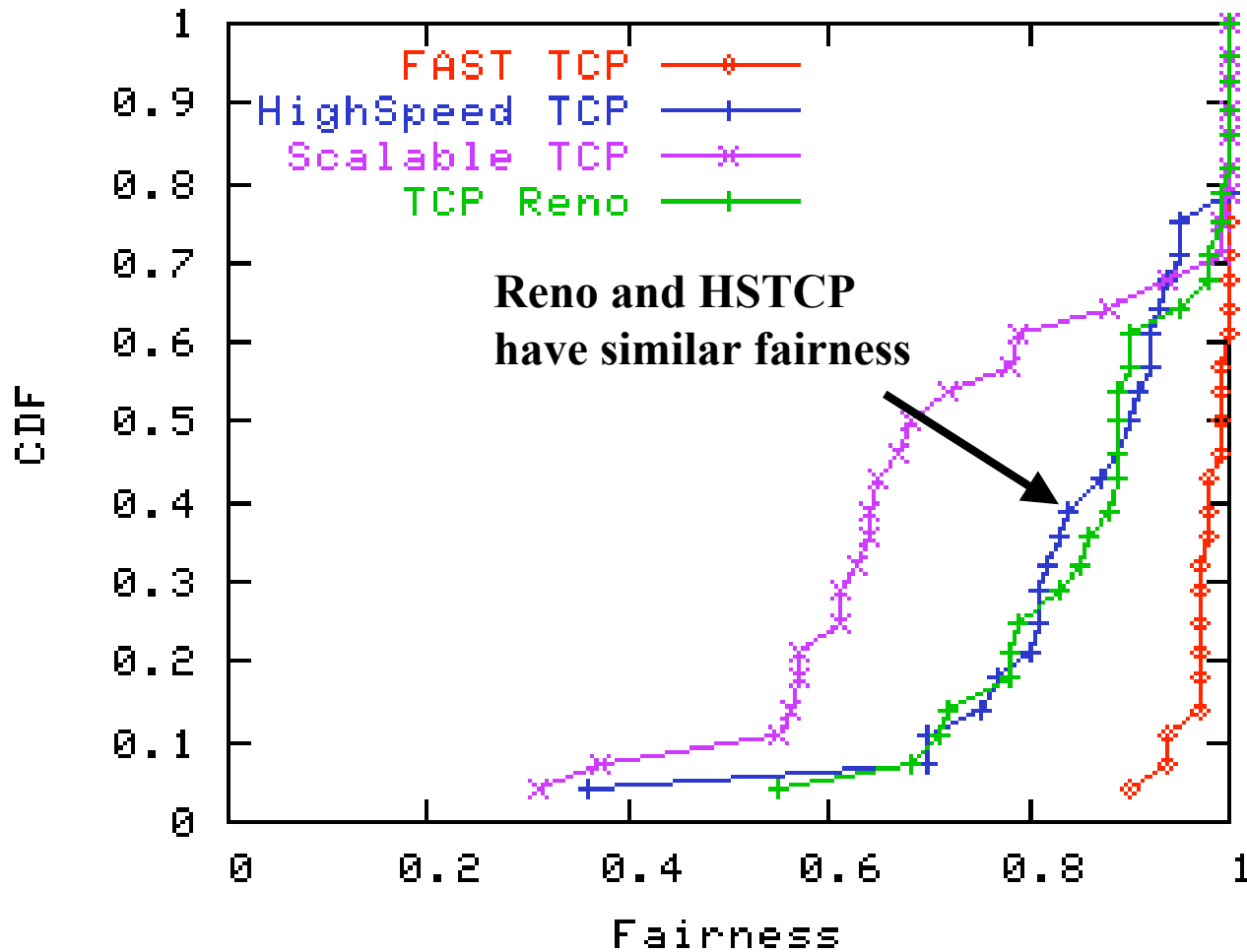
Dummysnet: cap = 800Mbps; delay = 50-200ms; #flows = 1-14; 29 expts

# Stability



Dummysnet: cap = 800Mbps; delay = 50-200ms; #flows = 1-14; 29 expts

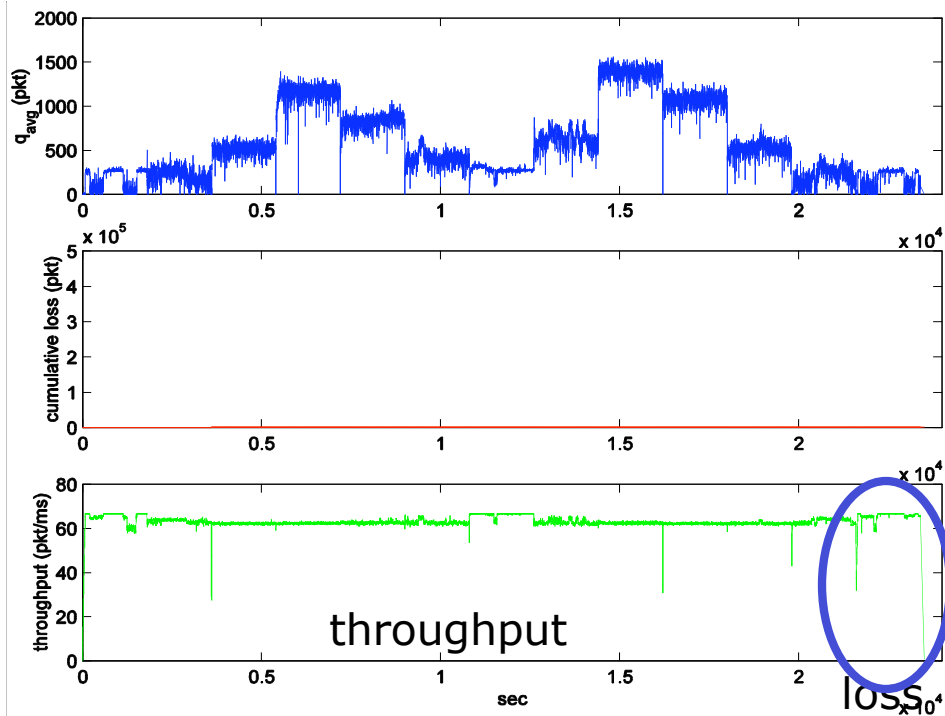
# Fairness



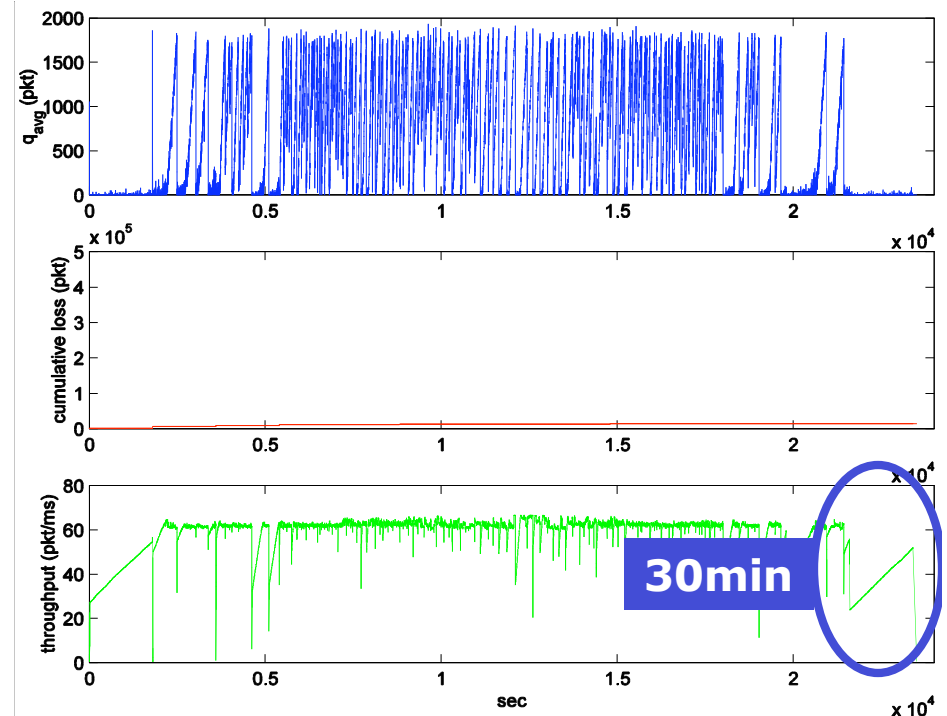
Dummysnet: cap = 800Mbps; delay = 50-200ms; #flows = 1-14; 29 expts



## FAST



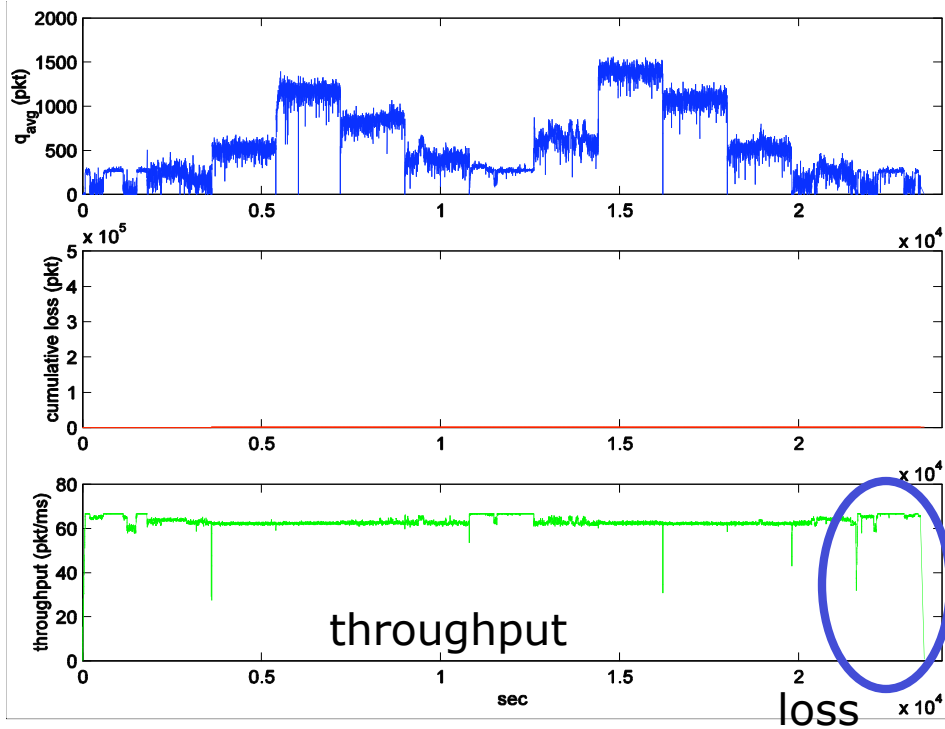
## Linux



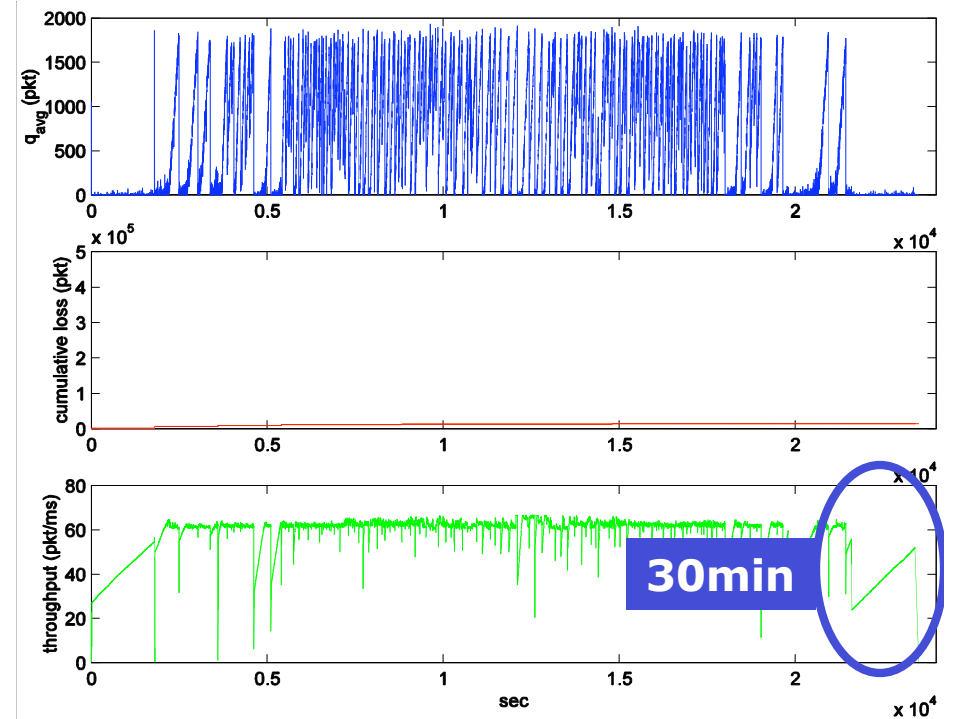
capacity = 800Mbps, delay=120ms, 14 flows,  
Linux 2.4.x (HSTCP: UCL)

# Impact of Packet Loss

**FAST**



**Linux**



capacity = 800Mbps, delay=120ms, 14 flows,  
Linux 2.4.x (HSTCP: UCL)

# 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

# Forward Error Correction

---

- Desired properties of FEC:
  - Given  $k$  original packets, constructs  $r$  redundancy packets such that given any  $(1+e)k$  packets probability of reconstruction rapidly approaches 1 as  $e$  increases.
  - The value  $e$  is essentially “overhead”.
  - Typically between 0.01 and 0.05
  - Very fast to compute on receiving side.

# Primary Objectives

---

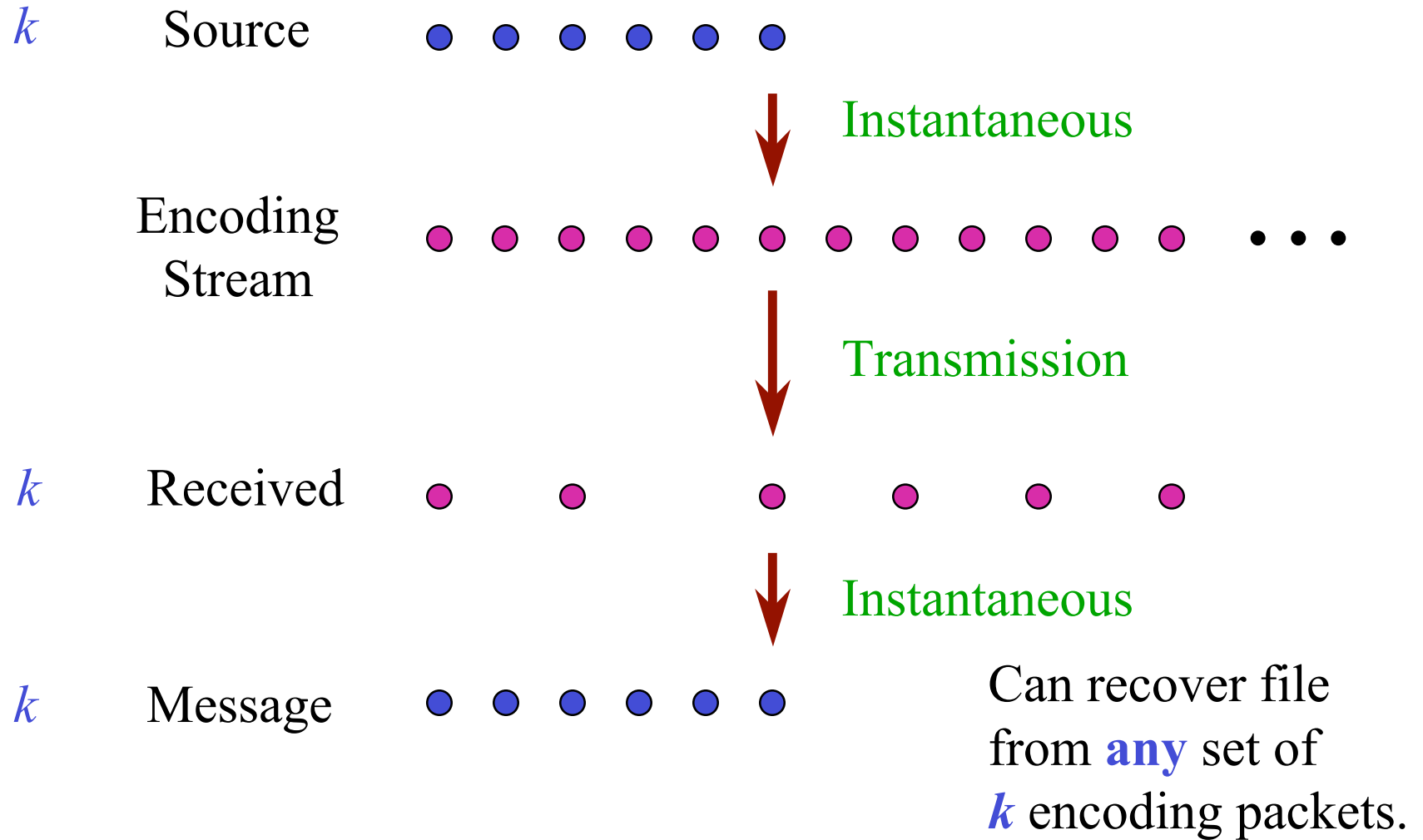
- Scale to vast numbers of clients
  - No ARQs or NACKs
  - Minimize use of network bandwidth
- Minimize overhead at receivers:
  - Computation time
  - Useless packets
- Compatibility
  - Networks: Internet, satellite, wireless
  - Scheduling policies, i.e. congestion control

# Impediments

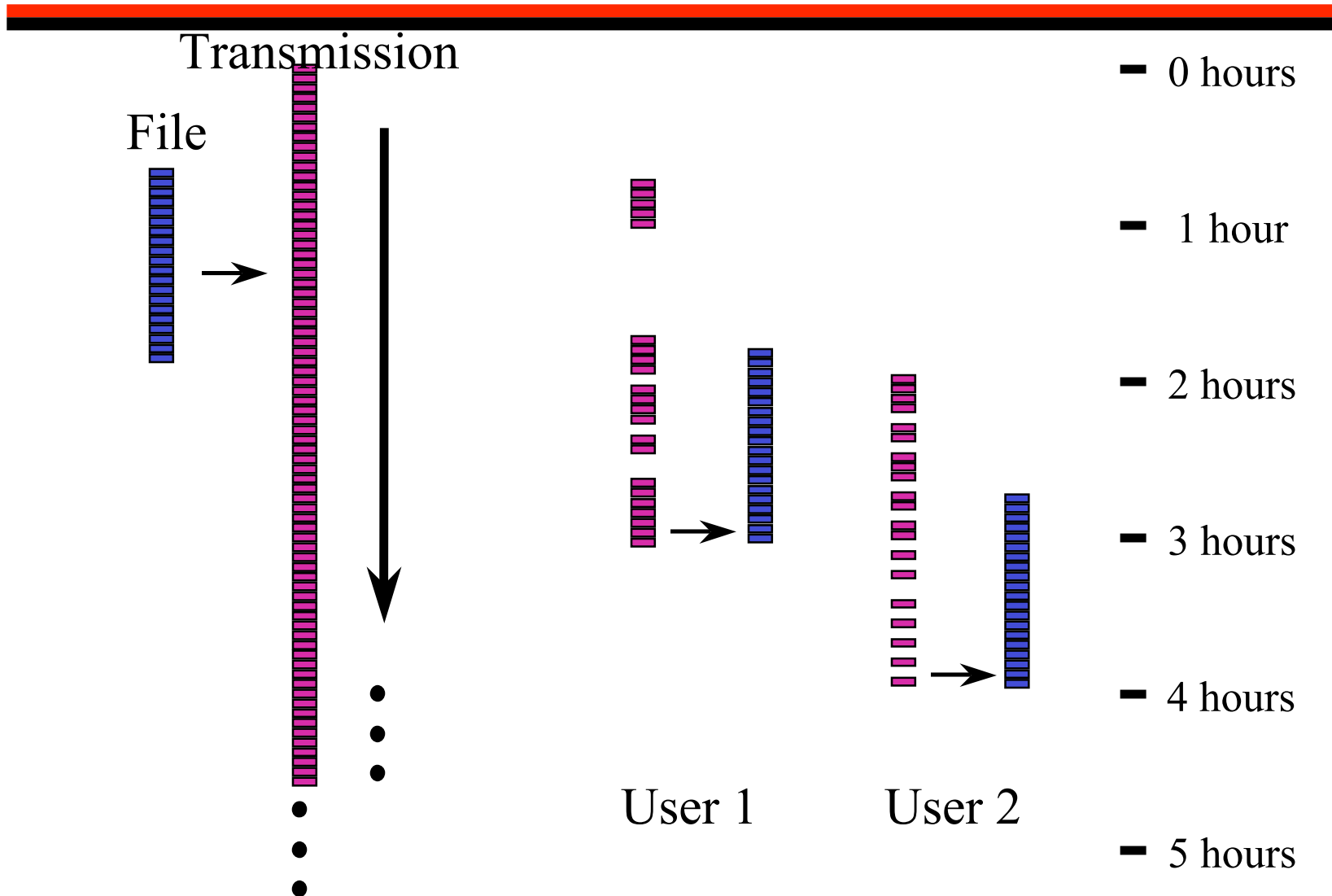
---

- Packet loss
  - wired networks: congestion
  - satellite networks, mobile receivers
- Receiver heterogeneity
  - packet loss rates
  - end-to-end throughput
- Receiver access patterns
  - asynchronous arrivals and departures
  - overlapping access intervals

# Digital Fountain



# Digital Fountain Solution



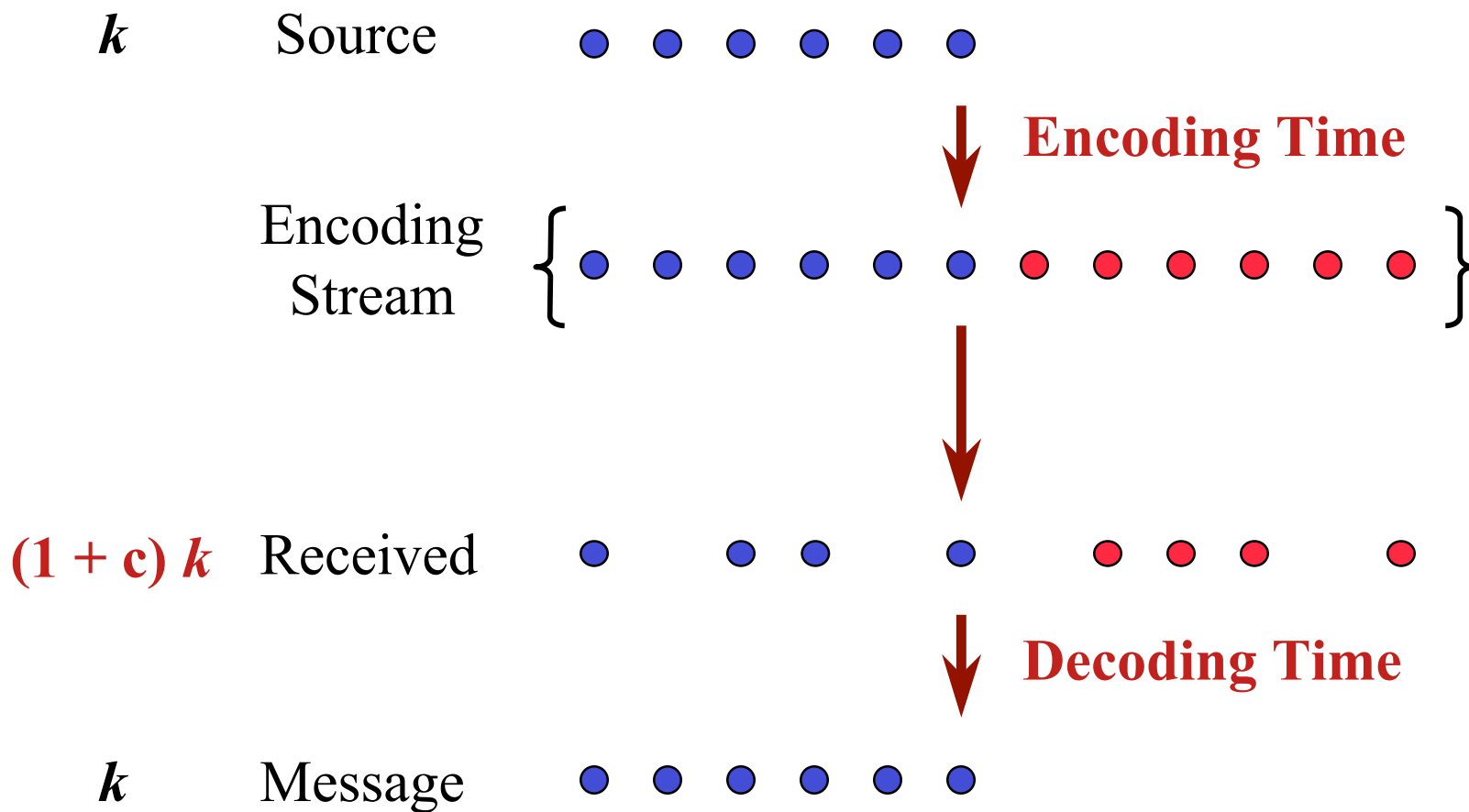
# Is FEC Inherently Bad?

---

- **Faulty Reasoning**
  - FEC adds redundancy
  - Redundancy increases congestion and losses
  - More losses necessitate more transmissions
  - **FEC consumes more overall bandwidth**
- **But...**
  - Each and every packet can be useful to **all** clients
  - Each client consumes minimum bandwidth possible
  - **FEC consumes less overall bandwidth by compressing bandwidth across clients**

# Approximating a Digital Fountain

---



# Work on Erasure Codes

---

- Standard Reed-Solomon Codes
  - Dense systems of linear equations.
  - **Poor** time overhead (**quadratic** in  $k$ )
  - **Optimal** decoding inefficiency of 1
- Tornado Codes [LMSSS '97]
  - Sparse systems of equations.
  - **Fast** encoding and decoding (**linear** in  $k$ )
  - **Suboptimal** decoding inefficiency

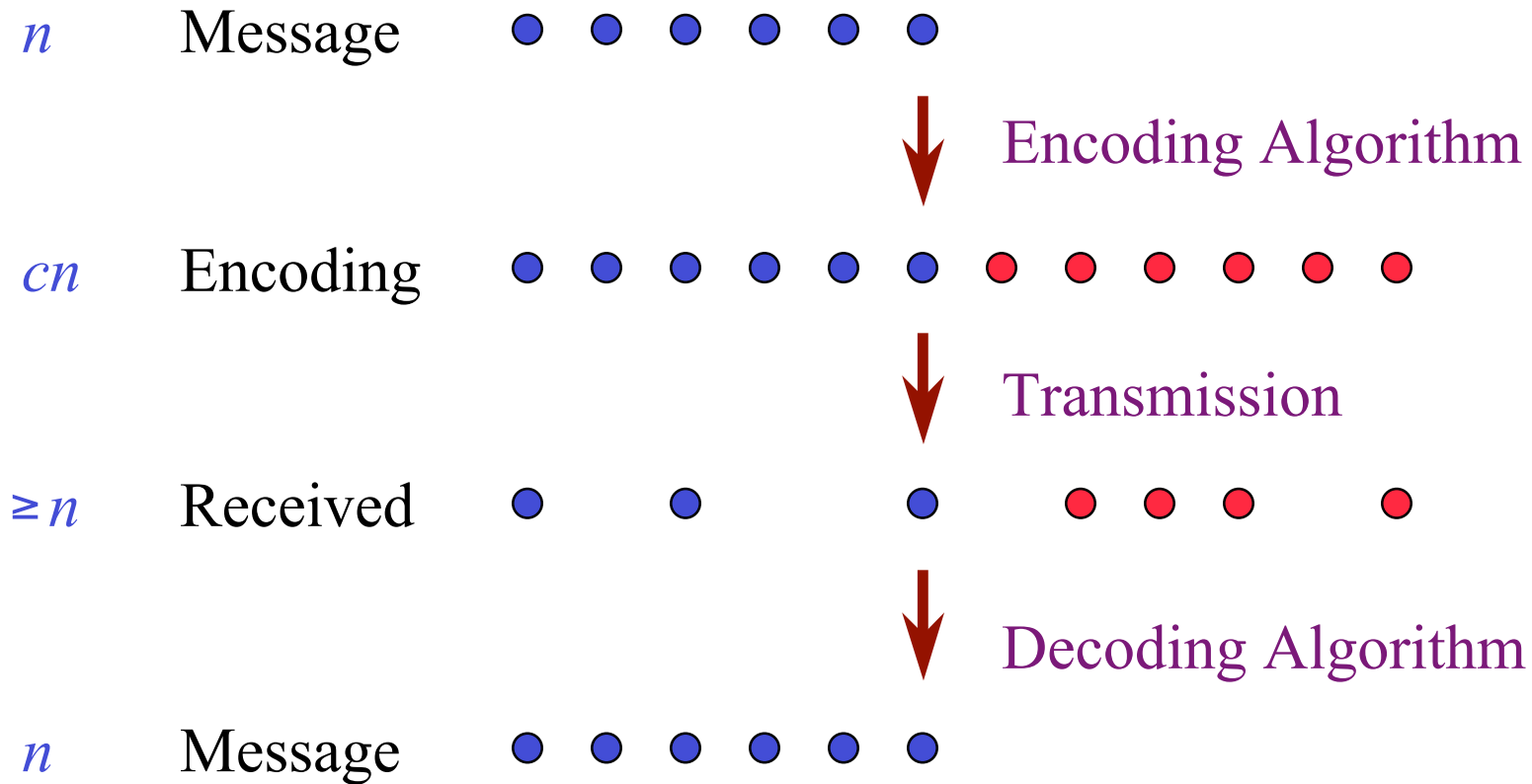
# So, Why Aren't We Using This...

---

- Encoding and decoding are slow for large files -- especially decoding.
- So we need fast codes to use a coding scheme.
- We may have to give something up for fast codes...

# Erasure Codes

---



# Performance Measures

---

- Time Overhead
  - The time to encode and decode expressed as a multiple of the encoding length.
- Reception efficiency
  - Ratio of packets in message to packets needed to decode. Optimal is 1.

# Reception Efficiency

---

- Optimal
  - Can decode from any  $n$  words of encoding.
  - Reception efficiency is 1.
- Relaxation
  - Decode from any  $(1+\epsilon)n$  words of encoding
  - Reception efficiency is  $1/(1+\epsilon)$ .

# Parameters of the Code

---



$n$

Message



$cn$

Encoding



$(1+\epsilon)n$

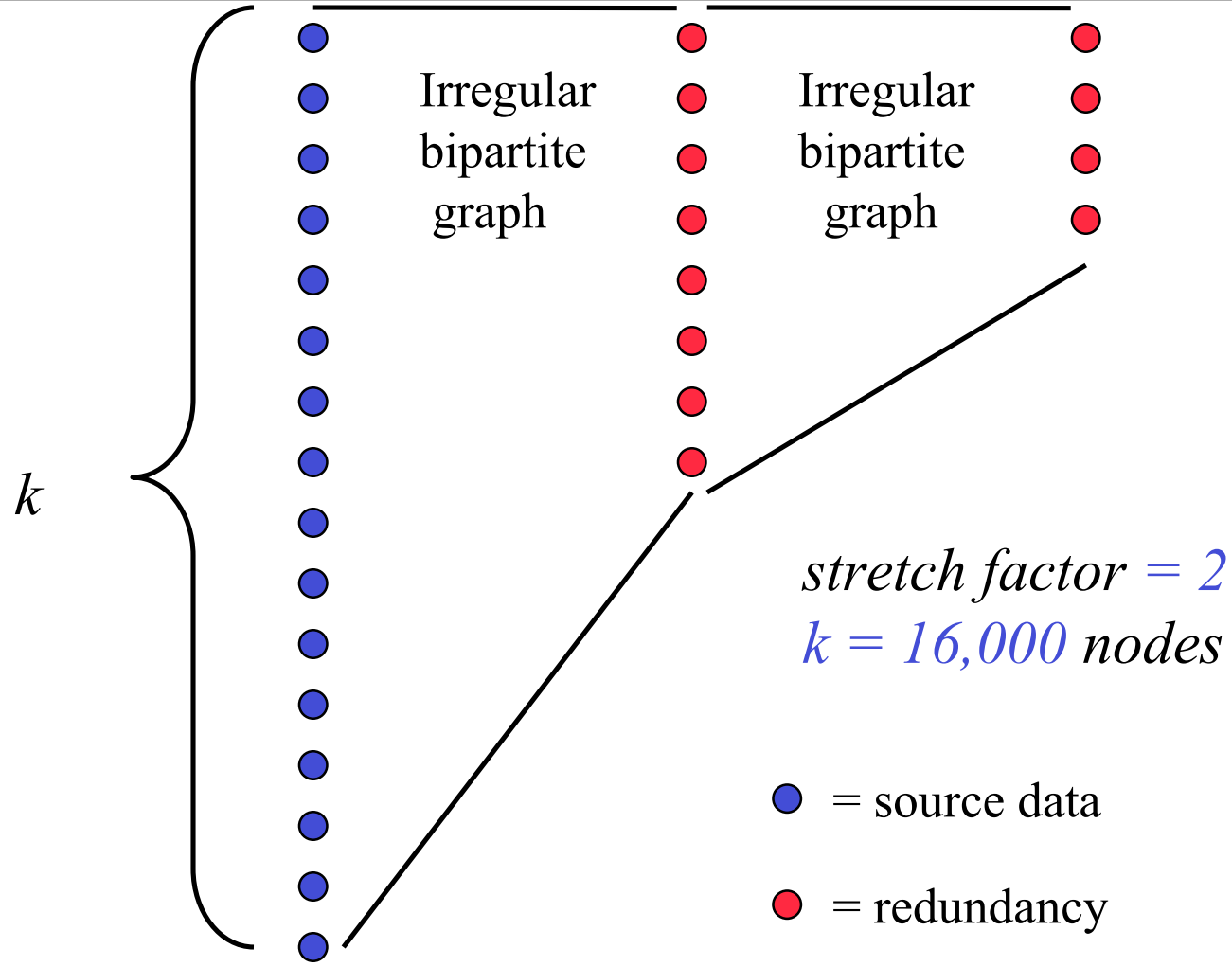
Reception efficiency is  $1/(1+\epsilon)$

# Previous Work

---

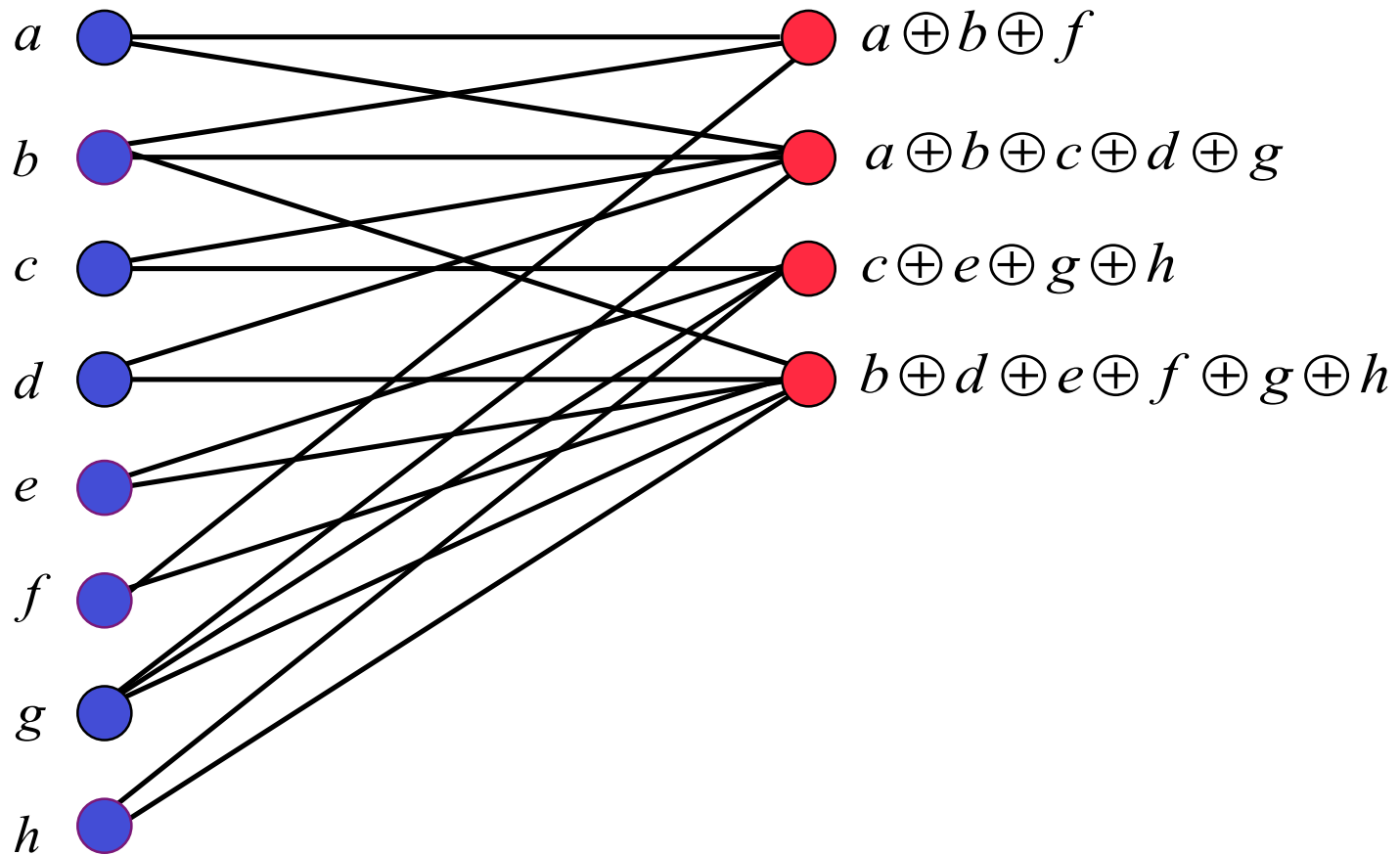
- Reception efficiency is  $1$ .
  - Standard Reed-Solomon
    - Time overhead is number of redundant packets.
    - Uses finite field operations.
  - Fast Fourier-based
    - Time overhead is  $\ln^2 n$  field operations.
- Reception efficiency is  $1/(1+\epsilon)$ .
  - Random mixed-length linear equations
    - Time overhead is  $\ln(1/\epsilon)/\epsilon$ .

# Tornado Z: Encoding Structure



# Encoding/Decoding Process

---



# Timing Comparison

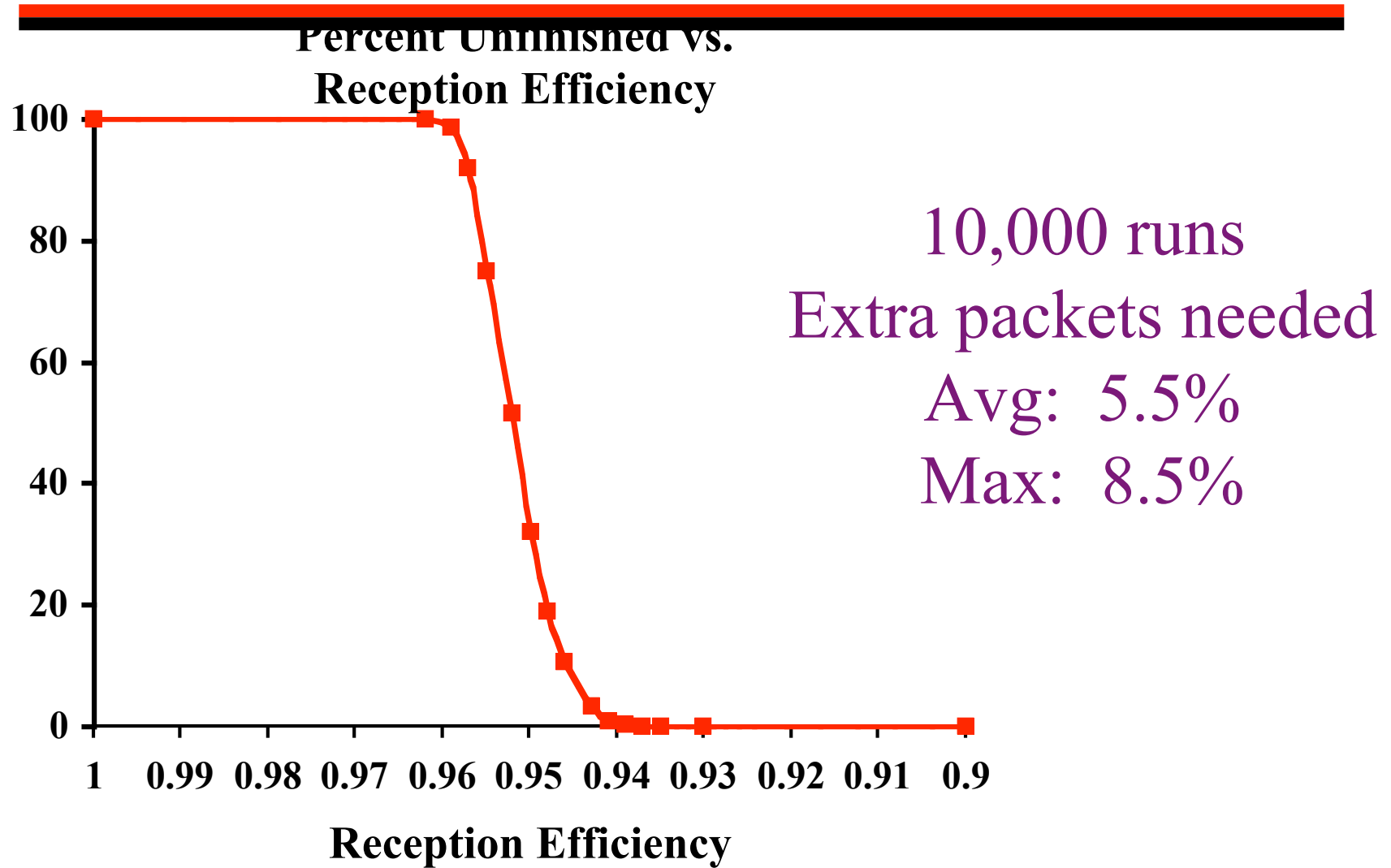
Encoding time, 1K packets		
Size	Reed-Solomon	Tornado Z
250 K	4.6 sec.	0.11 sec.
500 K	19 sec.	0.18 sec.
1 MB	93 sec.	0.29 sec.
2 MB	442 sec.	0.57 sec.
4 MB	30 min.	1.01 sec.
8 MB	2 hrs.	1.99 sec.
16 MB	8 hrs.	3.93 sec.

Decoding time, 1K packets		
Size	Reed-Solomon	Tornado Z
250 K	2.06 sec.	0.18 sec.
500 K	8.4 sec.	0.24 sec.
1 MB	40.5 sec.	0.31 sec.
2 MB	199 sec.	0.44 sec.
4 MB	13 min.	0.74 sec.
8 MB	1 hr.	1.28 sec.
16 MB	4 hrs.	2.27 sec.

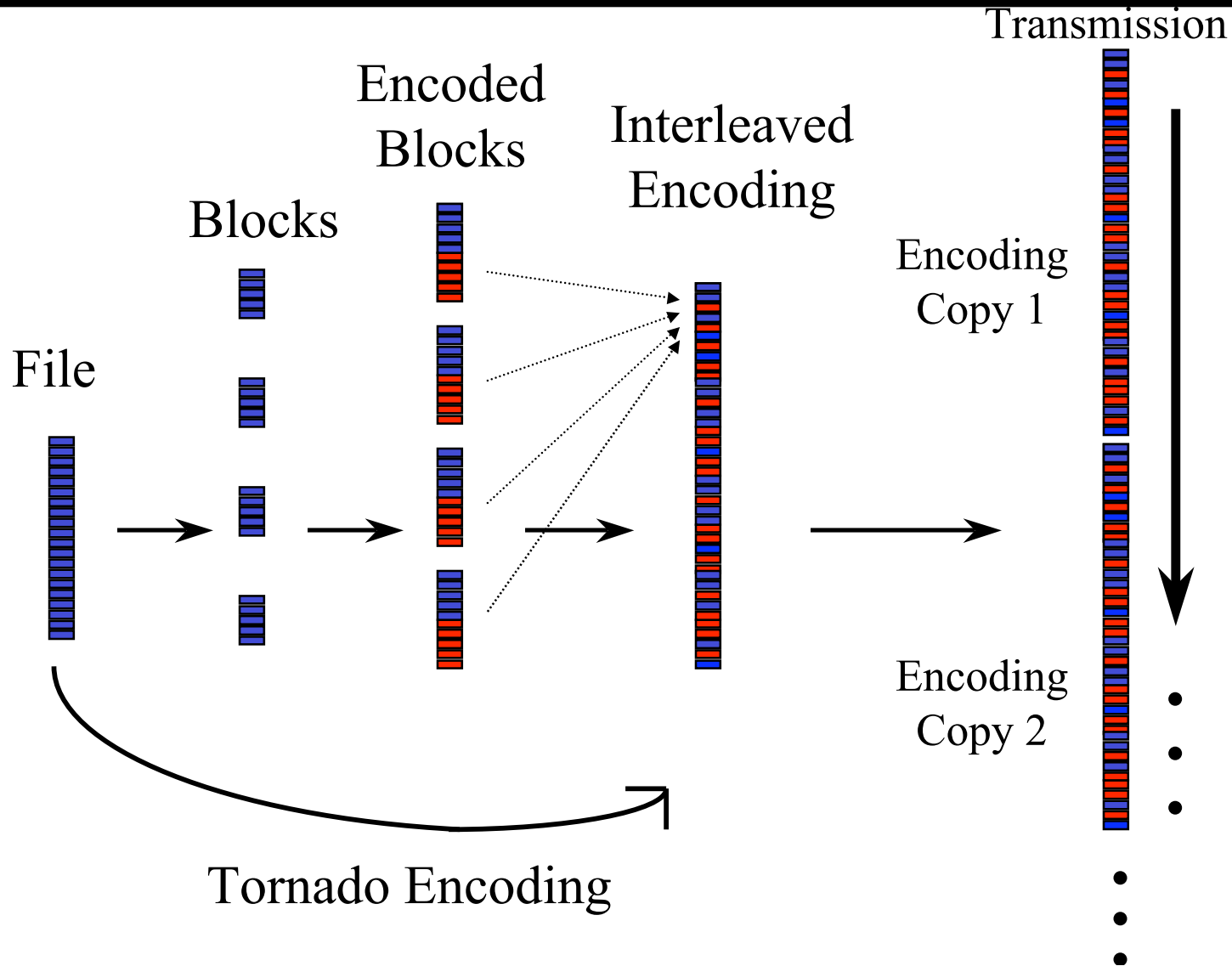
Tornado Z: Average inefficiency = 1.055

Both codes: Stretch factor = 2

# Reception Efficiency

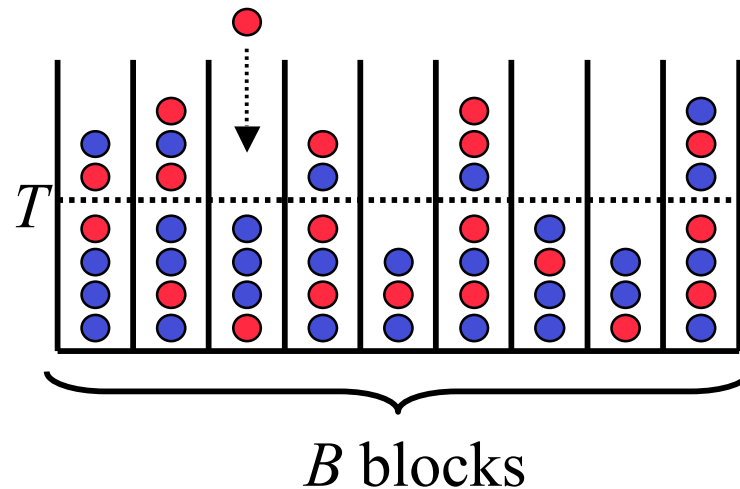


# Cyclic Interleaving



# Cyclic Interleaving: Drawbacks

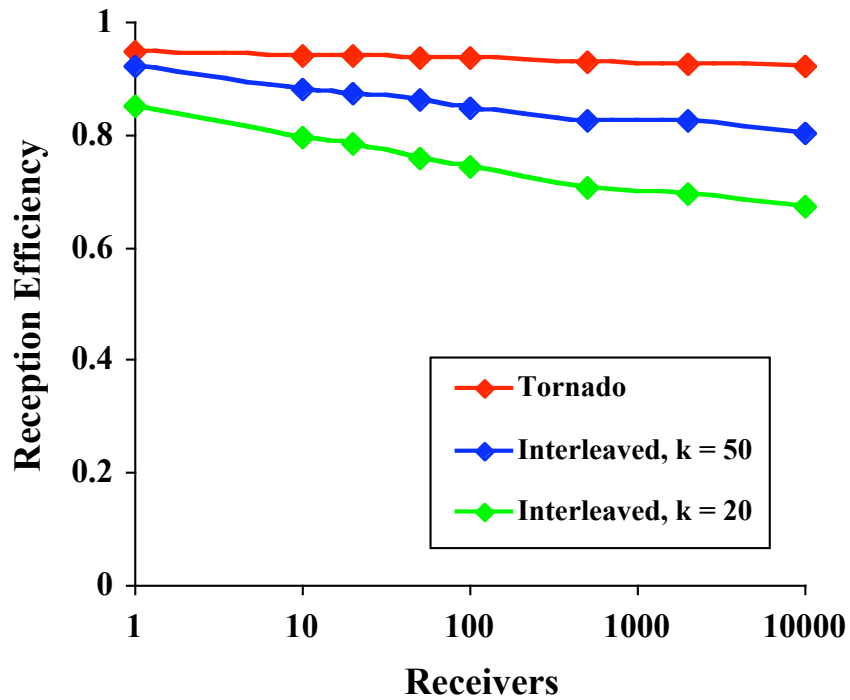
- The Coupon Collector's Problem
  - Waiting for packets from the last blocks:



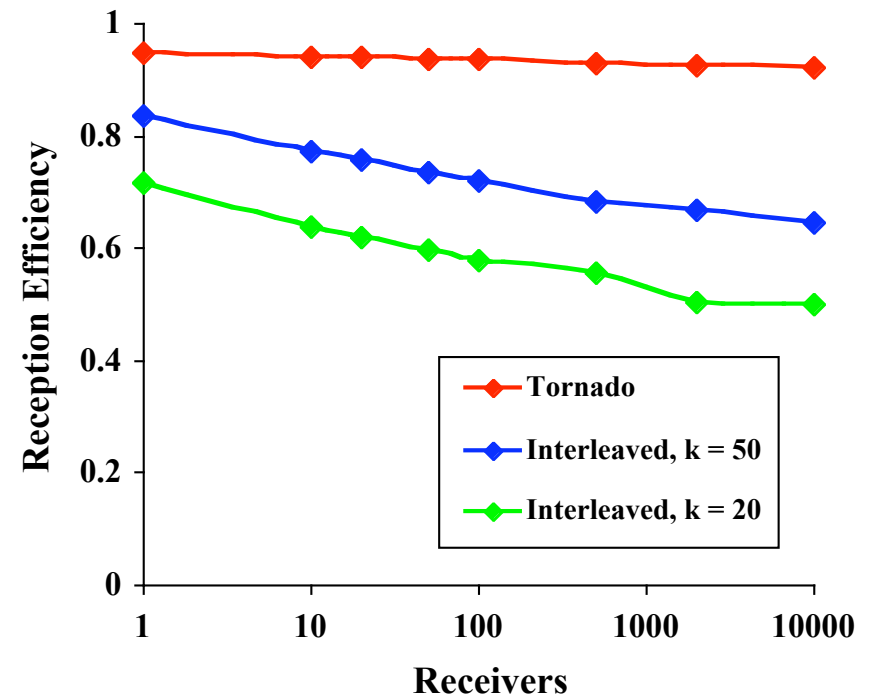
- More blocks: **faster** decoding, **larger** inefficiency

# Scalability over Receivers

Reception Efficiency on a 1MB File,  $p = 0.1$



Reception Efficiency on a 1MB File,  $p = 0.5$



# Scalability over File Size

