# Wireless Networks for Mobile Applications

Prof. Claudio Palazzi
cpalazzi@math.unipd.it

Università degli Studi di Padova

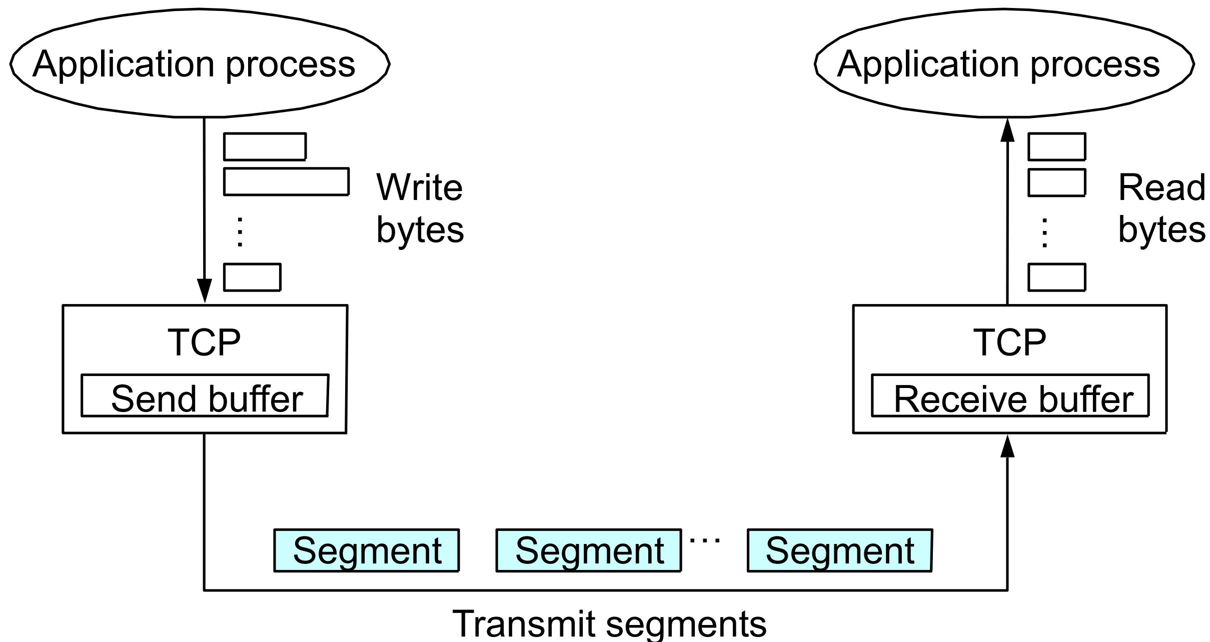
# TCP Overview

- TCP is the most widely used Internet protocol
  - Web, Peer-to-peer, FTP, telnet, …
- A two way, reliable, byte stream oriented end-to-end protocol
  - Includes flow and congestion control
- Closely tied to the Internet Protocol (IP)
- A focus of intense study for many years
  - RENO, NEW RENO, SACK are classic, legacy TCP versions
  - Nowadays TCP Cubic (Linux) and TCP Compound (Windows)
  - Hundreds of proposals for new scenarios (e.g., wireless links)

- Not appropriate for current wireless scenarios

# TCP Features

- Connection-oriented
- Byte-stream
  - app writes bytes
  - TCP sends *segments*
  - app reads bytes
- Reliable data transfer
- Full duplex
- Flow control: keep sender from overrunning receiver
- Congestion control: keep sender from overrunning network



# Reliability in TCP

- Checksum used to detect bit level errors
- Sequence numbers used to detect sequencing errors
  - Duplicates are ignored
  - packets can be reordered (or dropped)
  - Lost packets are retransmitted
- Timeouts is one of the two main ways to detect lost packets
  - Requires RTO (retransmission timeout) calculation
  - Requires sender to maintain data until it is ACKed
  - [ The other way is receiving three dupacks (discussed later in these slides) ]

# Timeout-based Recovery

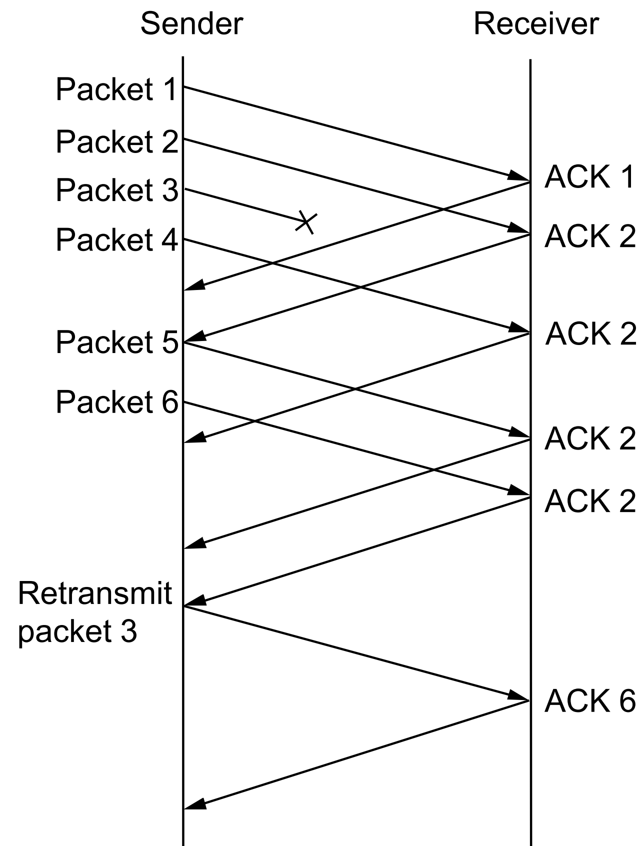
- Wait at least one RTT (Round Trip Time) before retransmitting
- Importance of accurate RTT estimators:
  - Low RTO → unneeded retransmissions
  - High RTO → poor throughput
- RTO estimator must adapt to change in RTT
  - But not too fast, or too slow!
  - About 4 times the RTT

# Fast Retransmit

- Duplicate acks (dupacks) are repeated acks for the same segment
- 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 dupacks as indication of loss
  - Don’t wait for timeout to retransmit packet

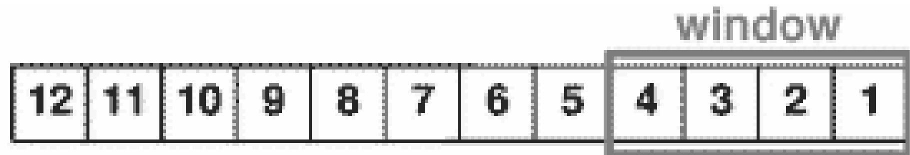
# Fast Retransmit and Fast Recovery: 3 Dupacks

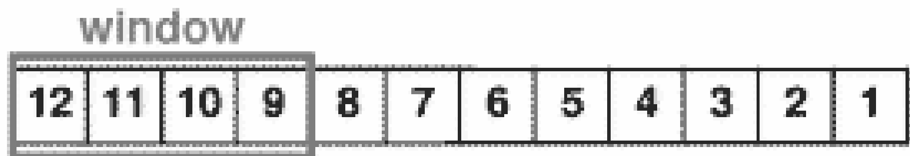
- Problem: coarse-grain TCP timeouts lead to idle periods
- Fast retransmit: use 3 duplicate ACKs (dupacks) to trigger retransmission
- Fast recovery: start at SSTHRESH (send out ssthresh new packets) and do additive increase after fast retransmit
  - Instead of starting from 1 packet only as with timeouts



# Sequence Numbers

- Stop&go vs. sliding windows
- The *window* indicates how many packets can be transmitted/travelling without having received the corresponding ack yet
- Any time an ack is received the window moves on including new packets that can be transmitted
- Data divided into packets (segments)
  - Generally of 1500 B





# TCP Flow Control

- The **Flow Control** blocks the sender from overwhelming the receiver
  - The receiver can inform the sender about its receiving capacity
  - The Advertised Window field in the ACKs is used to this aim

- Receiving side
  - The **Advertised Window** (in returning ACKs) is set by the receiver as the max number of bytes that it can receive; basically, how much space is left in its buffer

- Sending side
  - The sender's actual window (**Sending Window**) represents the actual bytes sent out and corresponds to the minimum between its computed window (**Congestion Window**) and the receiver's one (**Advertised Window**)

# Delay-Bandwitdth Product

- Delay x Bandwidth Product represents the pipe size
  - Link between sender and receiver seen as a pipe of length = Delay (or RTT) and section = Bandwidth

  - Represents the max number of data that can be traveling on the link at any time
  - Similar to the amount of water filling up a pipe

- With TCP of the the RTT-Bandwidth product is used
  - RTT is twice the Delay
  - It considers the fact that half the traffic is traveling and half reached the receiver and ACKs (which have small/negligible size with respect to data packets) are coming back

# TCP Congestion Control

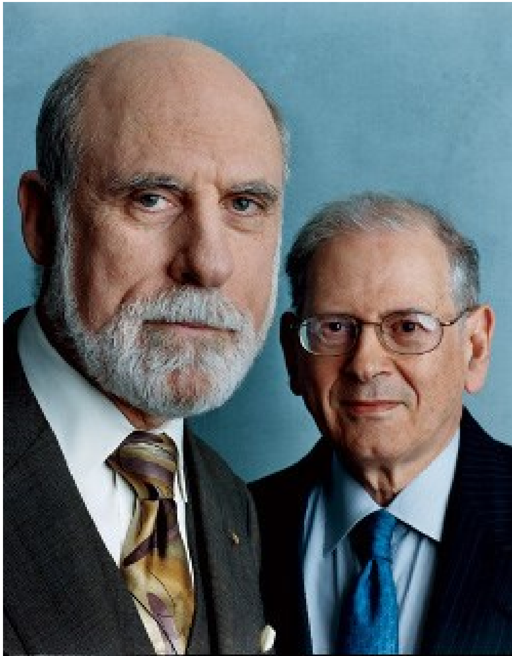
- The **Congestion Control** blocks the sender from overwhelming the network/Internet

- Idea
  - assumes best-effort network (FIFO routers) each source determines network capacity for itself
  - uses implicit feedback
  - ACKs pace transmission (*self-clocking*)

- Challenge
  - determining the available capacity in the first place
  - adjusting to changes in the available capacity

# Scenario

- Internet is a black box
  - Intelligence is at the boundaries (end nodes)
- 1972: TCP protocol (by Vint Cerf and Bob Kahn)
  - No clue about wireless revolution
  - Any loss in wired links is due to congestion (no error losses)



**1987: Gekko's cellphone**

# Additive Increase and Multiplicative Decrease

- Objective: adjust to changes in the available capacity
- New state variable per connection: **`CongestionWindow`**
  - limits how much data source has in transit

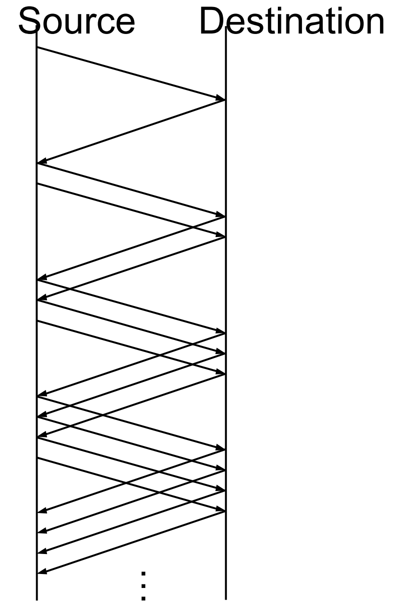
```
SendingWindow = MIN(CongestionWindow, AdvertisedWindow)
```

- Idea:
  - increase **`CongestionWindow`** when congestion goes down
  - decrease **`CongestionWindow`** when congestion goes up

# AIMD (cont)

- Question: how does the source determine whether or not the network is congested?
- Answer: timeout/dupacks
  - timeout signals that a packet (or more than one) was lost
    - probably with some serious congestion or other problem (disconnection?)
  - Three dupacks signals that a packet was lost but others are still passing through
    - Probably a minor congestion or packets that are seldom lost due to transmission error
  - In general it is assumed that lost packet implies congestion
    - Not true in wireless environments!

# AIMD (cont)



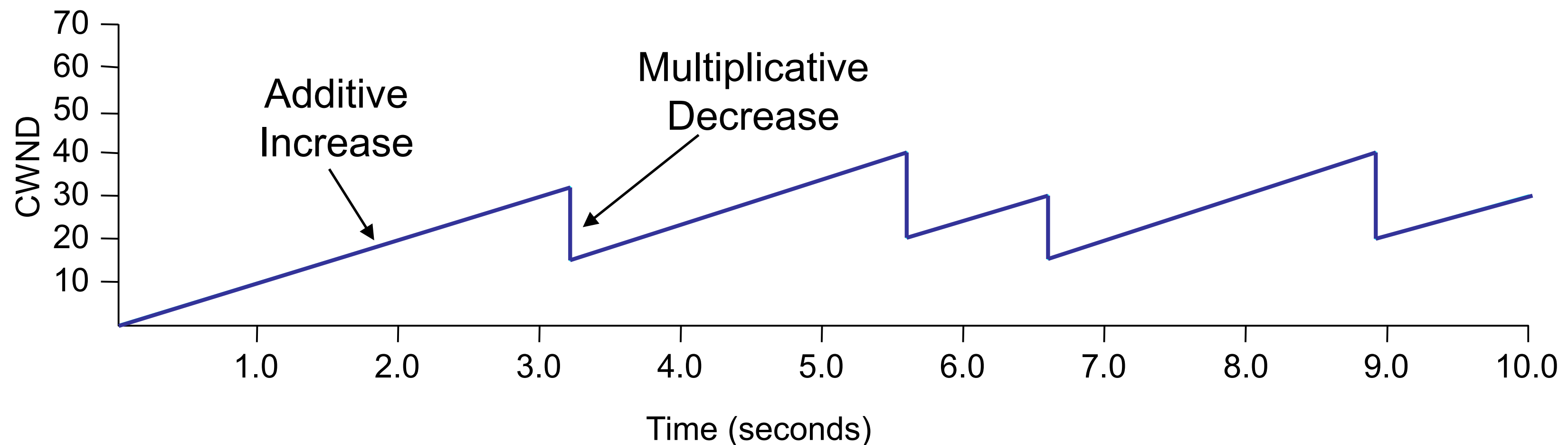

- Algorithm
  - increment **`CongestionWindow`** by one packet per RTT (*linear increase*)
  - divide **`CongestionWindow`** by two whenever a timeout occurs (*multiplicative decrease – fast!!*)

- In practice: increment a little for each ACK

```
Increment = 1/CongestionWindow
CongestionWindow += Increment
```

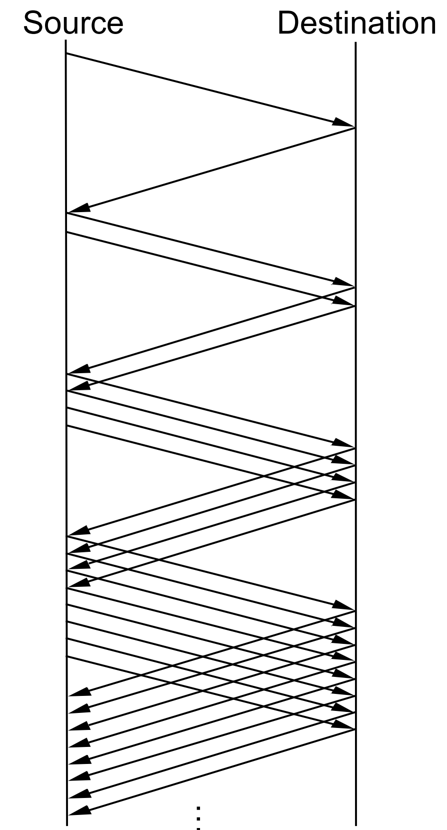
# AIMD (cont)

- AIMD results in a behavior that generates a sawtooth shape trace of transmission data rate



# Slow Start

- Objective: quickly determine the available capacity in the first part of a connection
- Idea:
  - begin with **`CongestionWindow`** = 1 pckt
  - double **`CongestionWindow`** each RTT (increment by 1 packet for each ACK)
  - This is exponential increase to probe for available bandwidth
  - <u>Up to half of cwnd may get lost</u> (when congestion level is reached)
- Used…
  - when first starting connection
  - when connection goes dead waiting for timeout
- SSTHRESH (slow start threshold) indicates when to begin additive increase phase



# SSTHRESH and CWND

- SSTHRESH typically very large on connection setup
- Set to one half of `CongestionWindow` (CWND) on packet loss
  - So, SSTHRESH goes through multiplicative decrease for each packet loss
  - If loss is indicated by timeout, set `CongestionWindow = 1`
  - If loss is indicated by 3 dupacks, set `CongestionWindow` equal to half of the congestion window value prior to the loss event
- After loss, when new data is ACKed, increase CWND
  - Manner depends on whether we are in slow start (exponential) or congestion avoidance (linear)

# Legacy TCP versions

- TCP Tahoe
- TCP Reno
- TCP New Reno
- TCP SACK
- TCP Vegas
- …

# TCP Tahoe Overview

- Standard TCP functions
  - connections, reliability, etc.
- Slow Start
- Congestion control/management
  - Additive Increase/ Multiplicative Decrease (AIMD)
  - Only timeouts to detect losses

# TCP Reno & New Reno

- Fast Retransmit/Fast Recovery
  - Three dupacks to quickly recover from light congestion (1 pkt loss)

- TCP Reno can recover from 1 pkt loss without having a time out

- TCP New Reno
  - Introduces partial ACKs to recover more pkts without resorting to timeouts

# TCP SACK

- TCP New Reno can retransmit only 1 pckt every RTT
  - Needs a partial ack to come back
- TCP SACK (Selective Acknowledgment)
  - Returning acks declares which packets (even non contiguous) were received
  - All non received packets can be retransmitted
    - Recover from multiple losses in just one RTT

# TCP Tahoe

Wasted…

Sequence No

Time Out and retransmission

Time

# Fast Retransmit

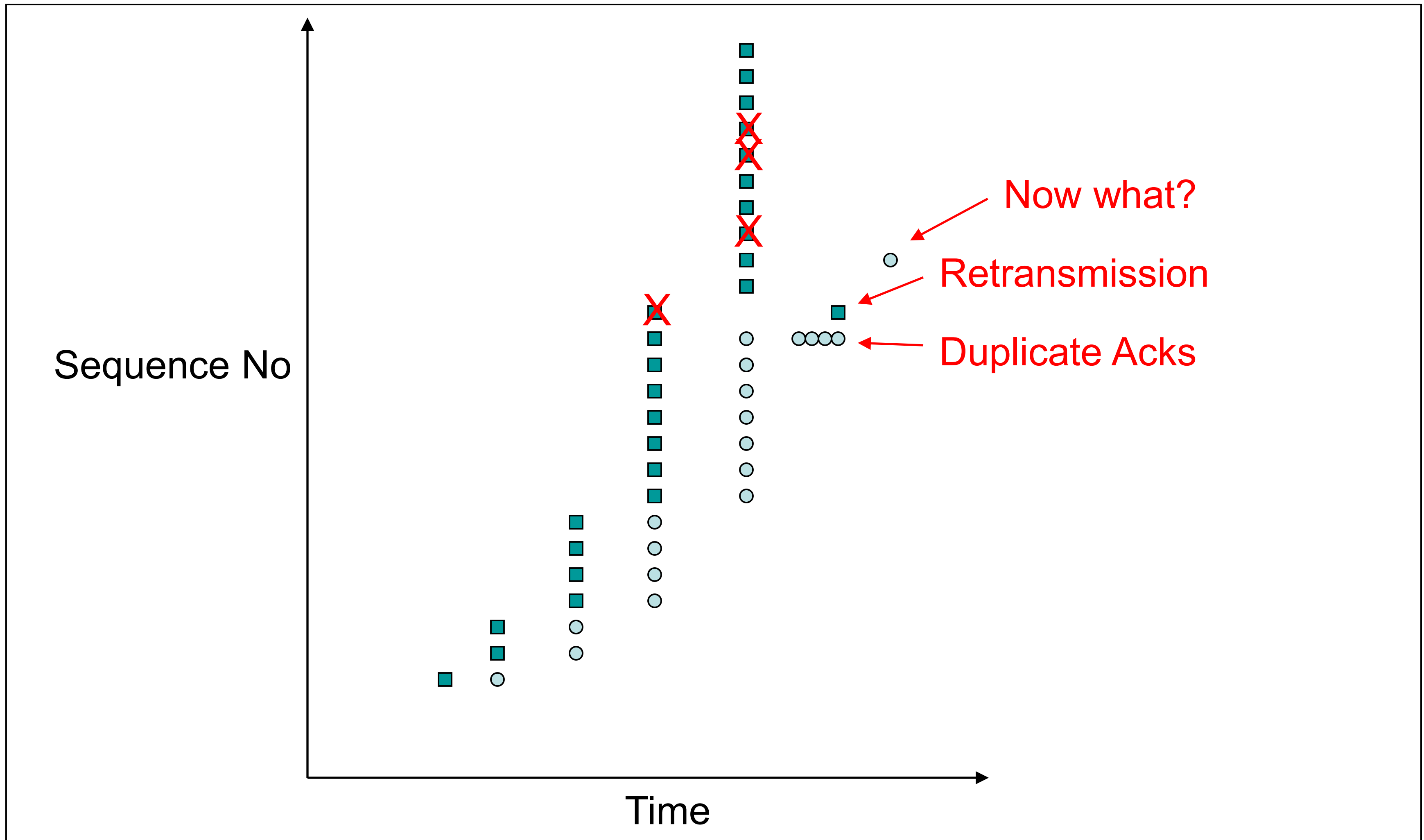
Retransmission

Duplicate Acks

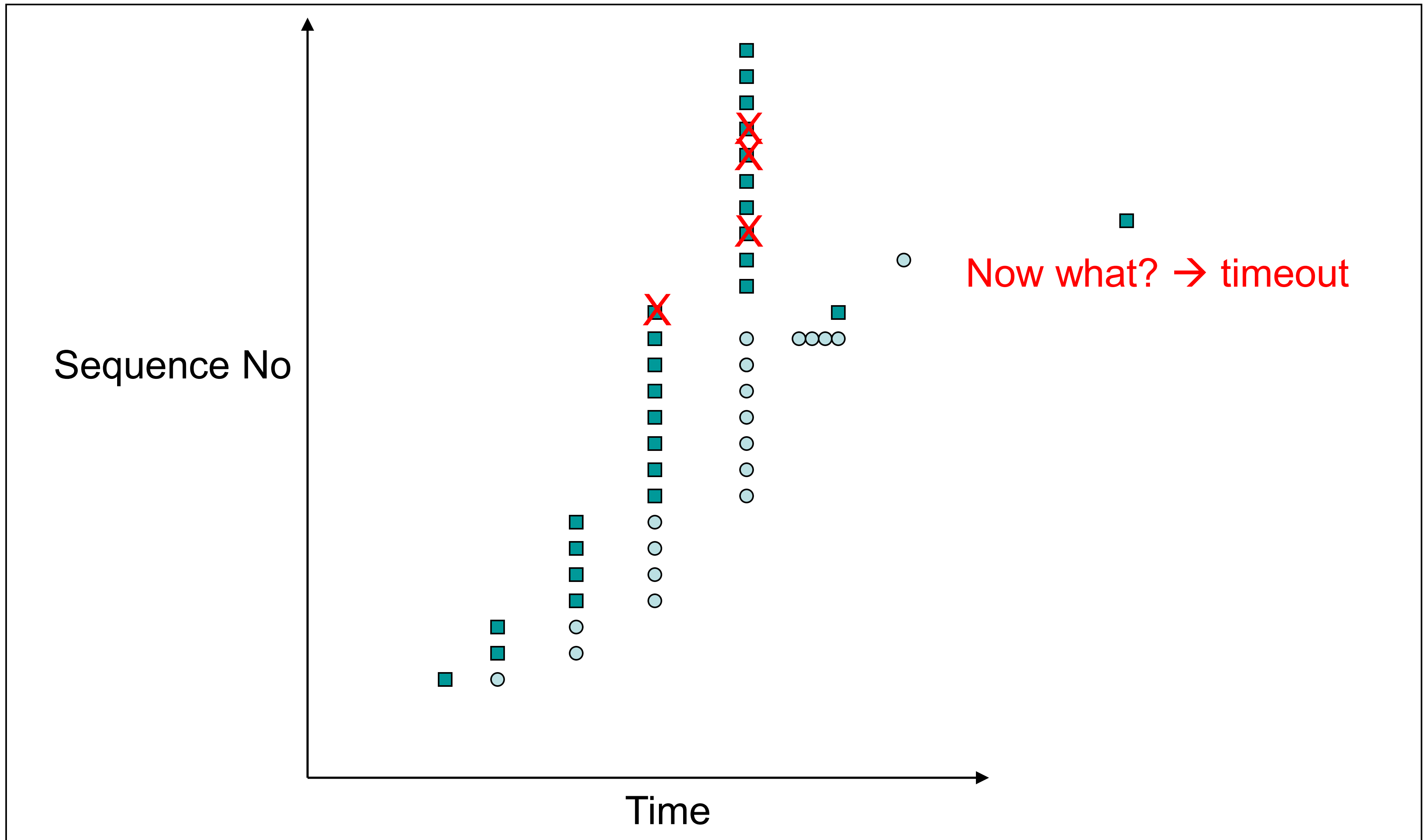
Sequence No

Time

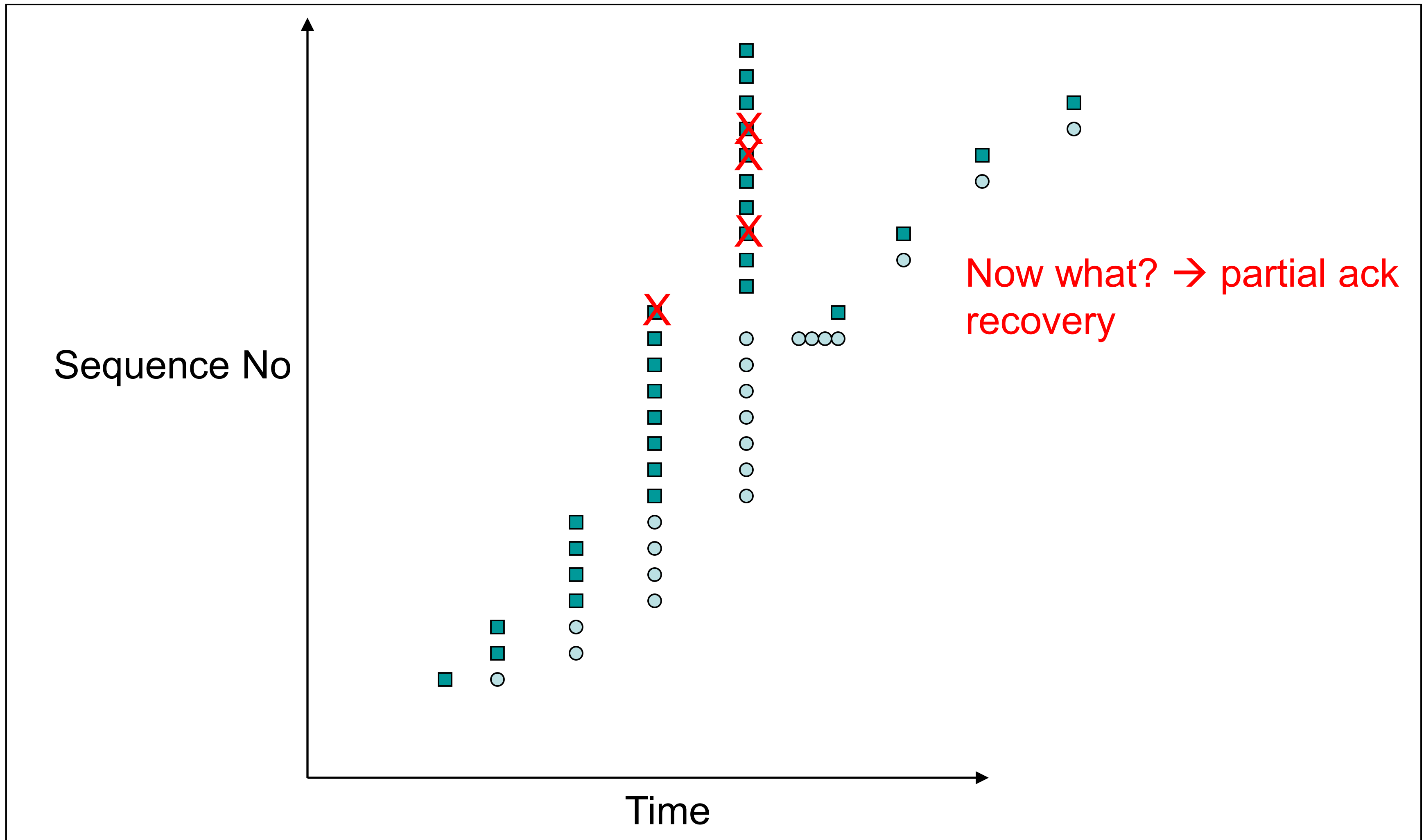
# Multiple Losses



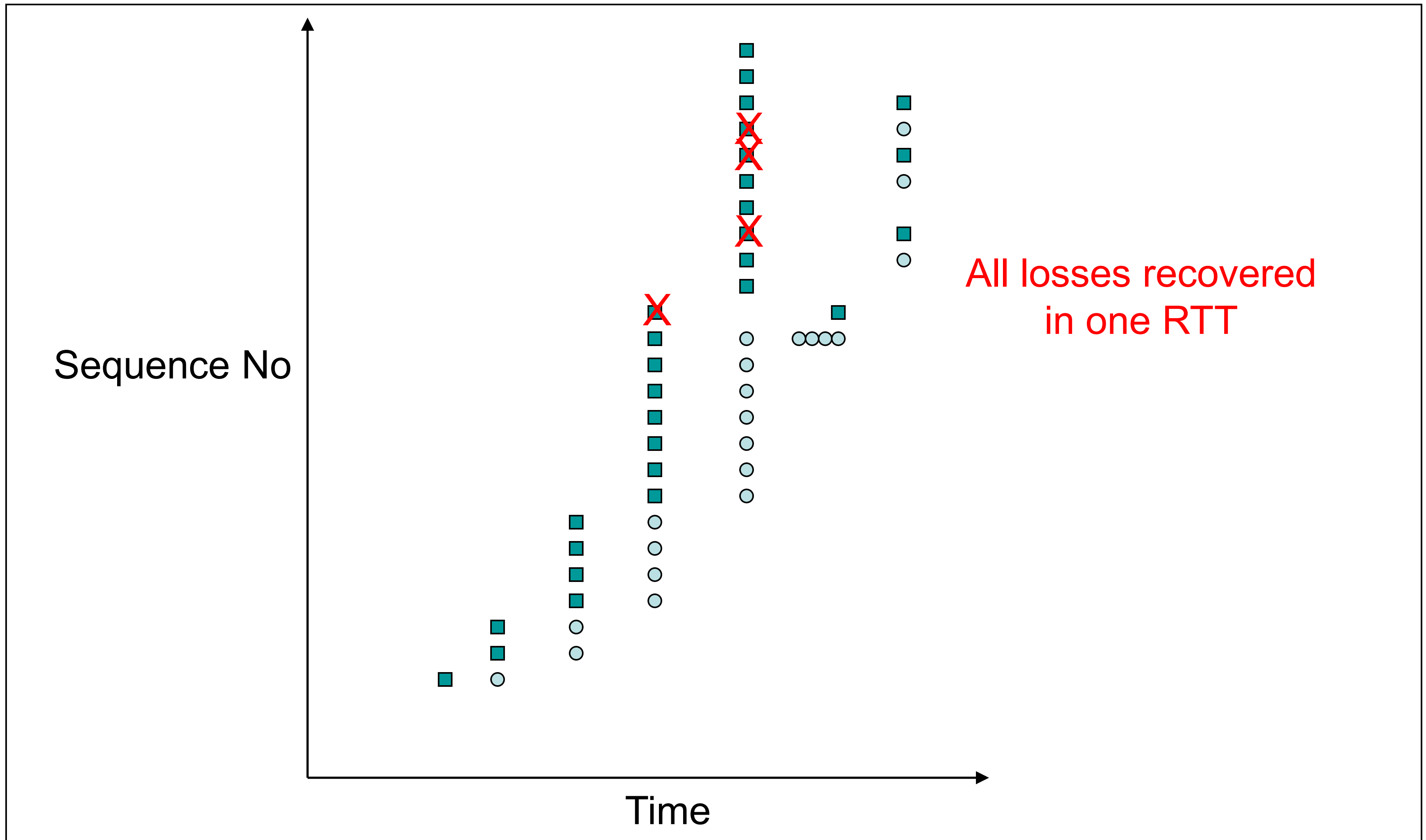
# TCP Reno



# TCP New Reno



# TCP SACK



# Congestion Control Functionality

**Until cwnd ≤ slow_start_threshold**

- Slow Start phase (exponential growth)
  - Each returning ACK, a new pckt is transmitted
    - cwnd -> cwnd + 1
  - Every RTT
    - cwnd -> 2 cwnd

**When cwnd > slow_start_threshold**

- Congestion avoidance phase (linear growth)
  - Each returning ACK, a new pckt is transmitted
    - cwnd -> cwnd + (1/cwnd)
  - Every RTT
    - cwnd -> cwnd + 1

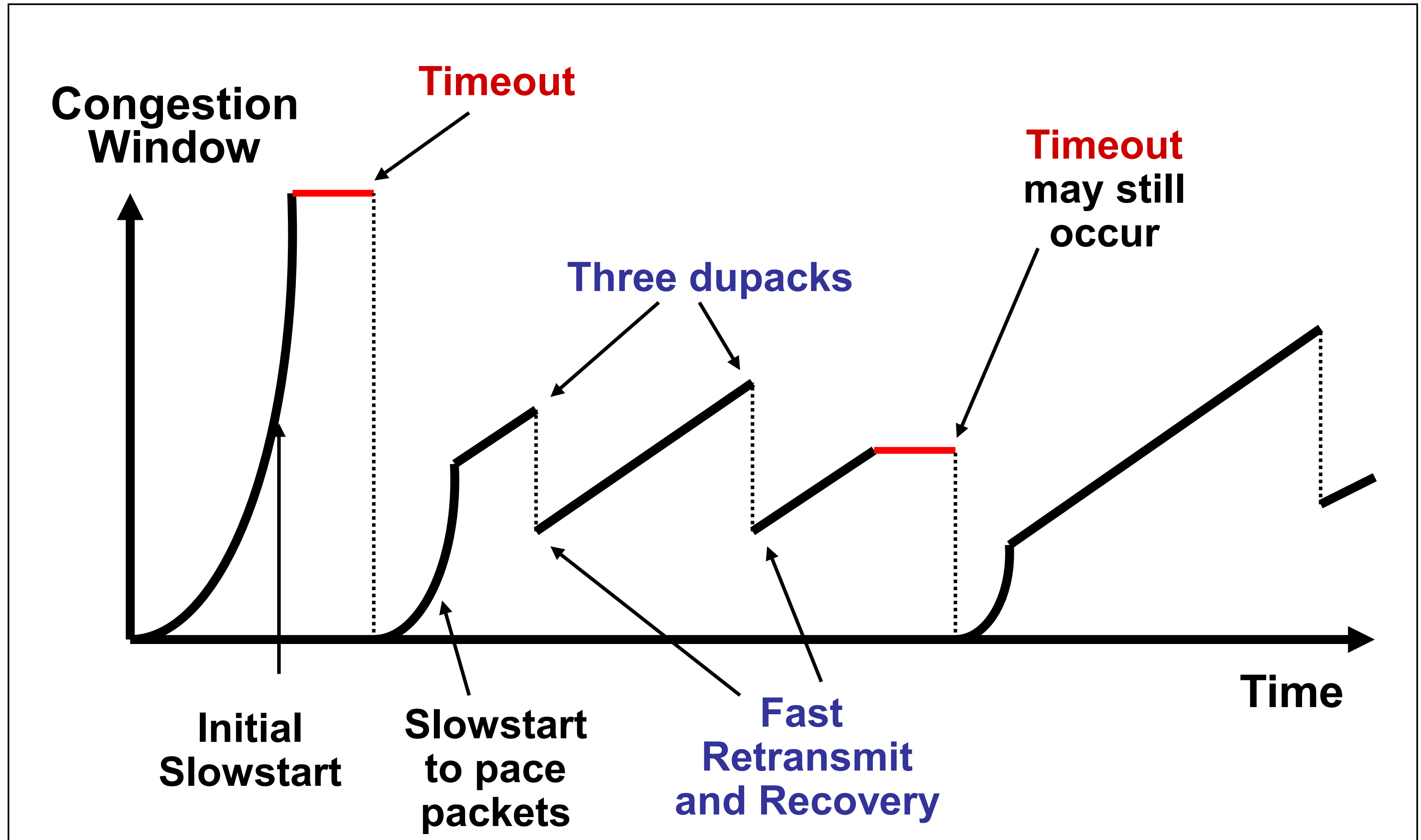
# Loss Recovery: Summary

- Two ways to detect losses
  - Time outs
  - Three dupacks

- With timeout expiration
  - ssthresh = cwnd / 2
  - cwnd = 1 (so, restart in slow start phase)

- With three dupacks
  - ssthresh = cwnd / 2
  - cwnd = cwnd / 2 (so, restart in cong. avoidance phase)

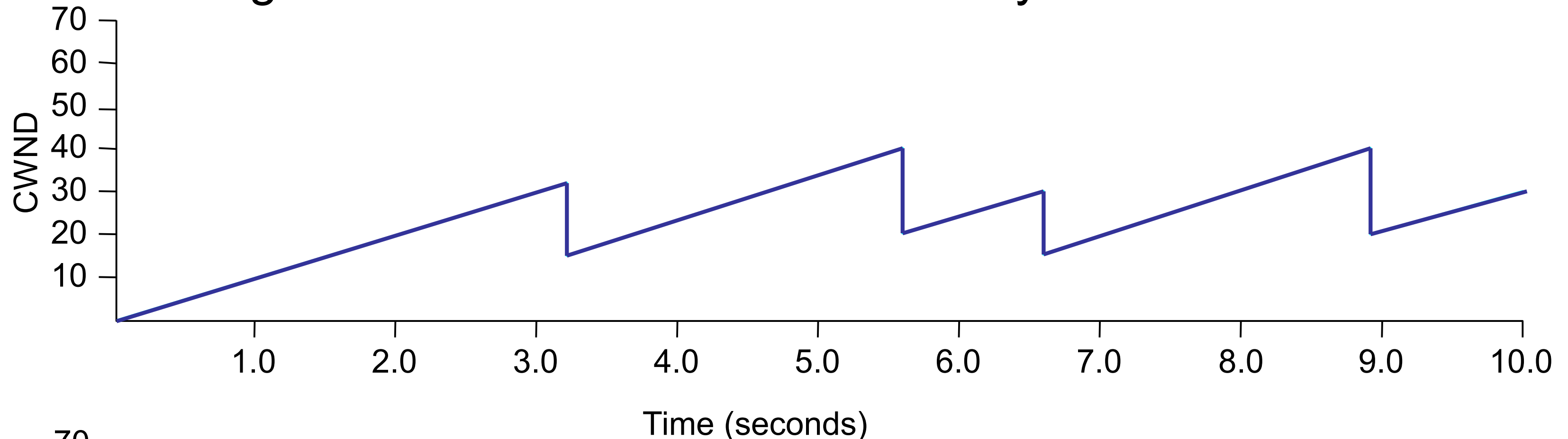
# TCP Saw Tooth

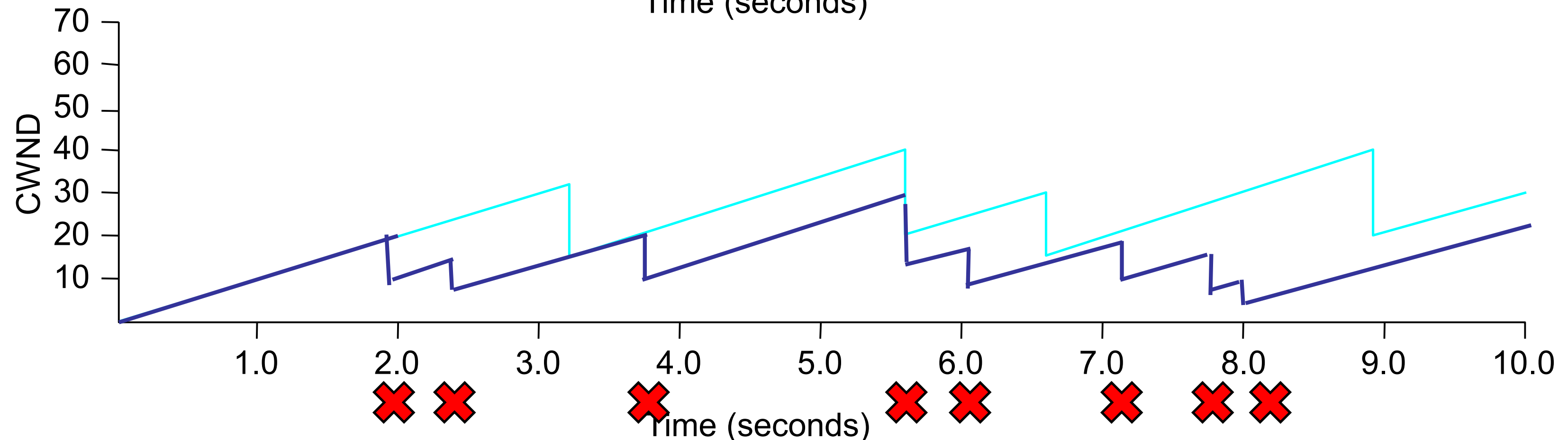



**Wireless environments have error losses as well**

# Wired vs Wireless

- Let's assume for simplicity only congestion avoidance
  - Each packet loss determined through 3 dupacks and recovered through fast retransmit and fast recovery




wireless errors causing throughput degradation

# TCP … summary

- Flow control vs congestion control
- Sliding window
- Slow start vs congestion avoidance
- Losses: time out vs three dupacks
- Windows: congestion vs sending vs advertised
- …

# TCP Vegas Congestion Avoidance

- Reaction per congestion episode not per loss
- Congestion avoidance vs. control
- Use change in observed end-to-end delay to detect onset of congestion
  - Compare expected to actual throughput
  - Expected = window size / round trip time
  - Actual = acks / round trip time
- Modifications:
  - Modified Congestion Avoidance
  - Aggressive Retransmission
  - Aggressive Window Adaptation
  - Modified Slow-Start

# TCP Vegas

actual can only be equal or less than expected

- expected is transmission rate with no other traffic/queue

<u>Monitor transmission rate (throughput, goodput):</u>

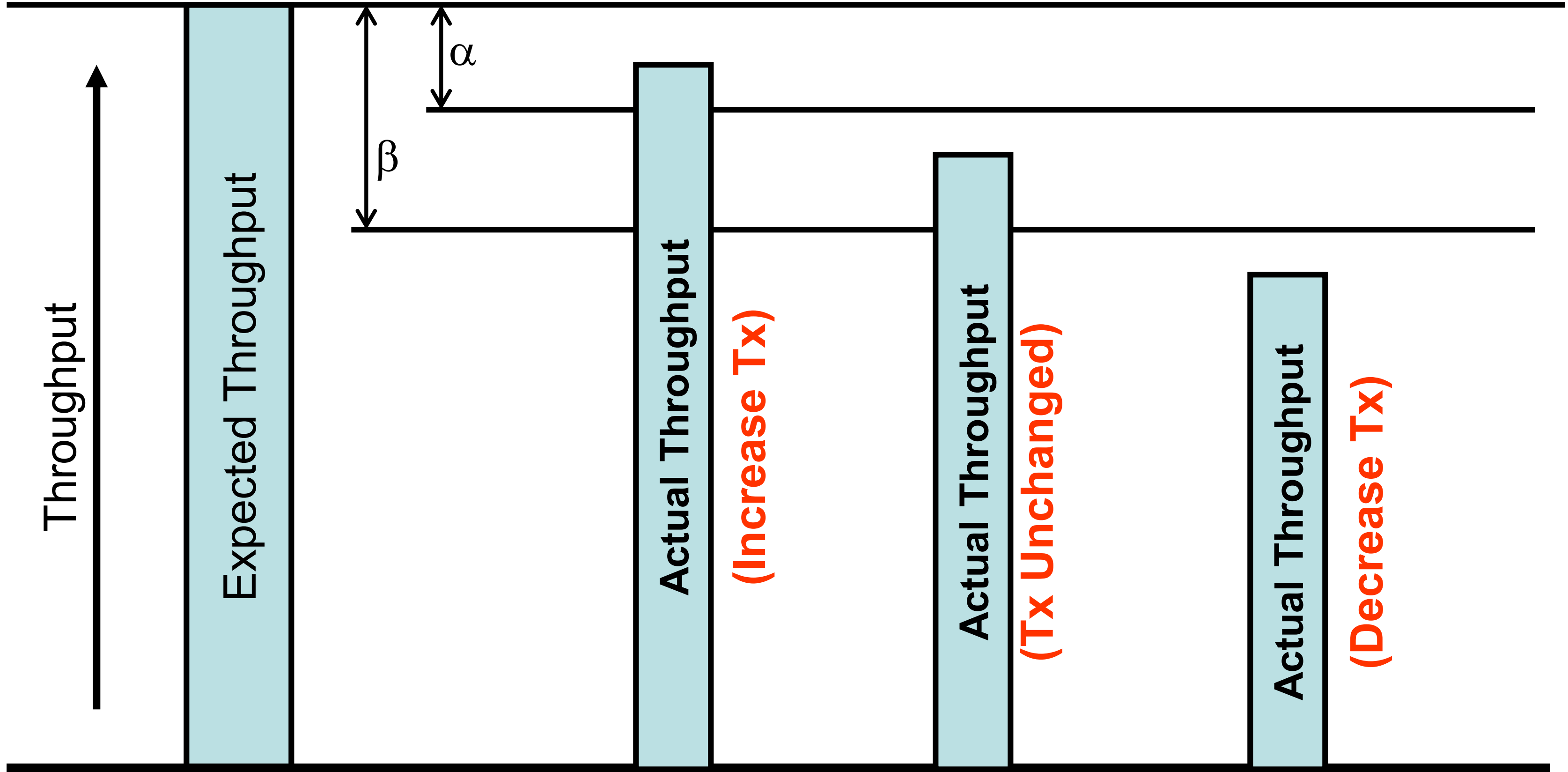
- Thresholds of $\alpha$ and $\beta$ correspond to how many packets Vegas is willing to have in queues
  - $\alpha < \beta$ so… expected - $\beta$ < expected - $\alpha$ < expected
- If expected - $\alpha$ < actual < expected
  - Queues decreasing $\rightarrow$ increase rate
- If expected - $\beta$ < actual < expected - $\alpha$
  - Don’t do anything
- If actual < expected - $\beta$
  - Queues increasing $\rightarrow$ decrease rate before packet drop

# Vegas: Modified Congestion Avoidance

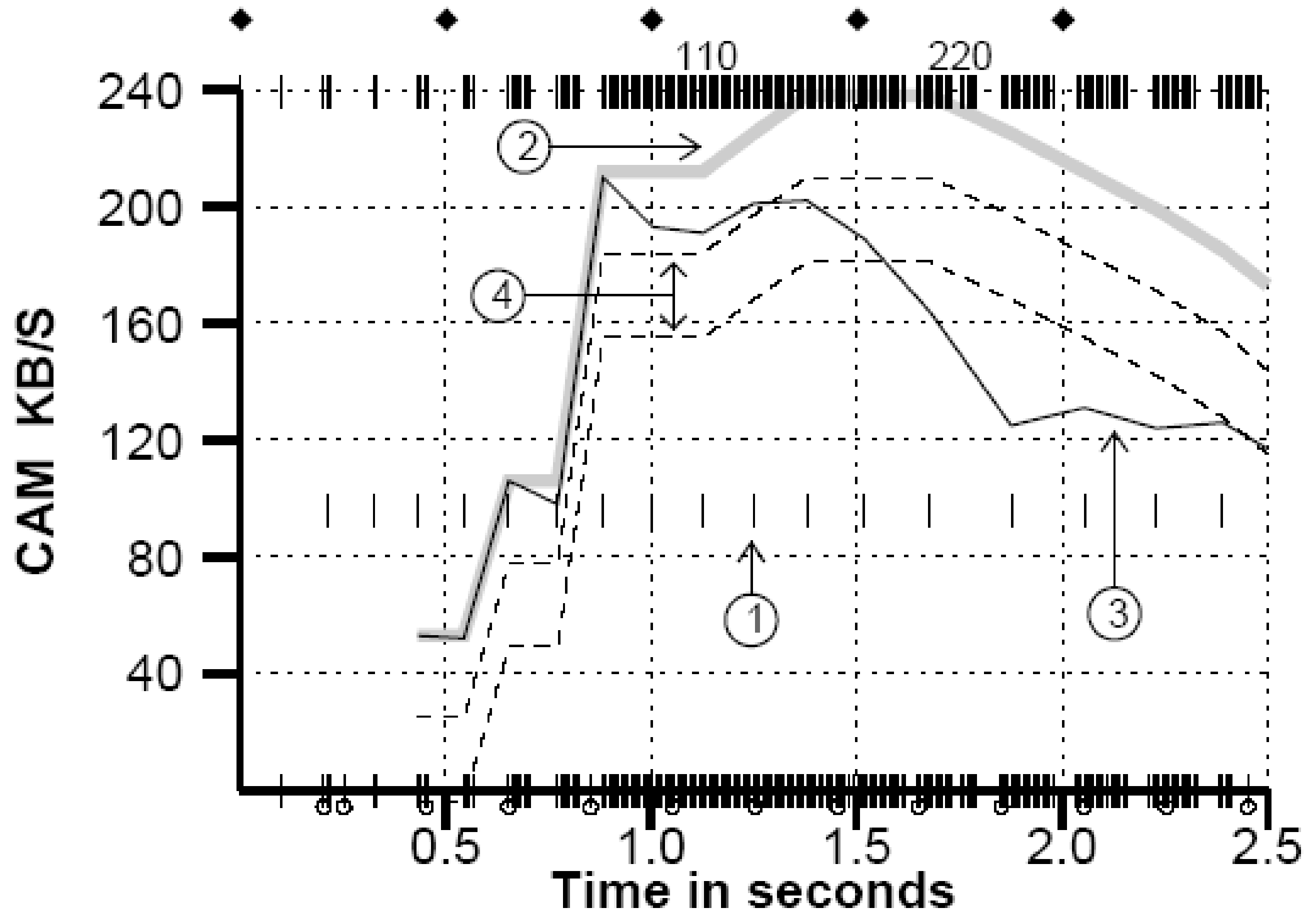
- **Vegas Calculates (Once per RTT):**
  - *Expected Throughput*=WindowSize/BaseRTT
  - *Actual Throughput*=ActualTransmittedAmount/RTT

- **Static Parameters:**
  - $\alpha$ = 1 pkts/RTT
  - $\beta$ = 3 pkts/RTT

# Vegas: Modified Congestion Avoidance



# Vegas: Modified Congestion Avoidance



# Vegas: Modified Congestion Avoidance

- TCP transmission rate = cwnd/RTT
- TCP takes cwnd updating decision once per RTT
- The decision is applied throughput the next RTT for each received ACK as follows:
  - **Increase Tx Rate** (Expected-Actual<$\alpha$): :
    - ❑ cwnd = cwnd + 1/cwnd
  - **Decrease Tx Rate** (Expected-Actual>$\beta$):
    - ❑ cwnd = cwnd - 1/cwnd
  - **Tx Rate Unchanged** ($\alpha$<Expected-Actual<$\beta$): :
    - ❑ cwnd = cwnd

# Vegas: Aggressive Retransmission

- With dupacks
  - When Vegas receives the first dupack or the second dupacks, it checks the fine grained timer expiry
    - More aggressive retransmissions (helps in wireless environments with non-congestion losses)
  - If timer expirers, it retransmits immediately

More appropriate for error prone (wireless) environments

# Vegas: Aggressive cwnd Updating

- With recovery
  - Reduce cwnd by one quarter instead of half when it enters into recovery
- With multiple loss
  - In case of multiple segment loss from a single window, it reduces the cwnd only once
- With Initial setting
  - cwnd is set to 2 instead of 1

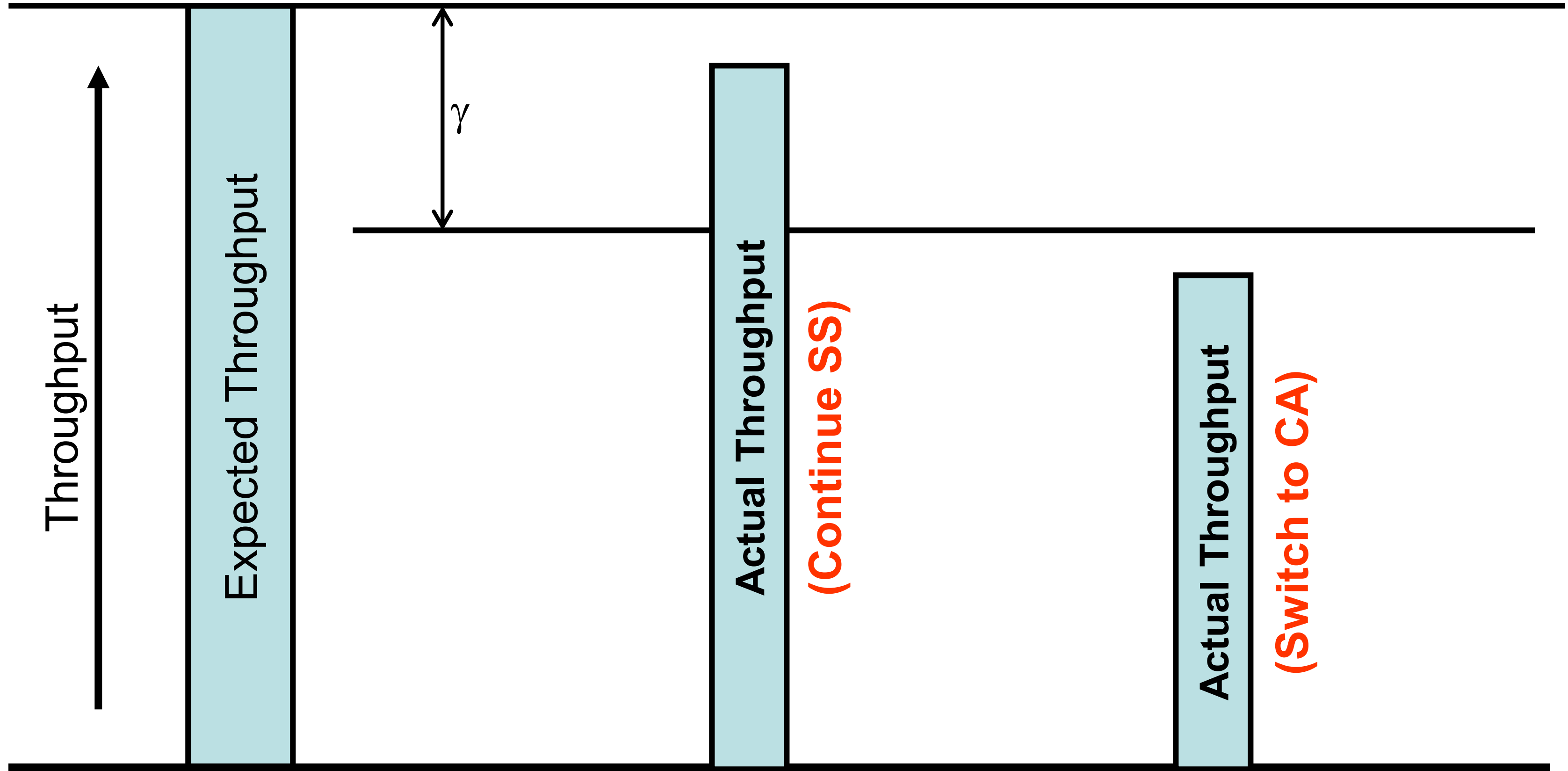
More appropriate for error prone (wireless) environments

# Vegas: Modified Slow-Start

- **Vegas Calculates (in every alternate RTT):**
  - ***Expected Throughput*=WindowSize/BaseRTT**
  - ***Actual Throughput*=ActualSentAmount/RTT**

- **Static Parameters:**
  - $\gamma$ **= 1 pkts/RTT**

# Vegas: Modified Slow-Start



# Vegas: Modified Slow-Start

- TCP keeps the congestion window fixed in every other RTT and it measures the throughput
- On every next RTT, it does the followings:
  - Continue SS (Expected-Actual<$\gamma$):
    - ❑ Exponential Increase
      - ❑ cwnd = cwnd + 1 for each ACK, that is,
      - ❑ cwnd = 2 * cwnd for each RTT
  - Switch to CA (Expected-Actual>$\gamma$):
    - ❑ Set ssthresh=cwnd
    - ❑ Follow the rules of CA

# TCP Vegas

- Flaws
  - Sensitivity to delay variation
  - Cannot coexist with legacy TCP versions
    - When a TCP Vegas flow shares the same bottleneck with a TCP New Reno (for instance), as soon as the pipe is full and packets get buffered TCP Vegas reduces its data rate, thus leaving some more space to TCP New Reno that continues its growth till a congestion episode (packet loss due to buffer overflow)
- Some ideas have been incorporated into more recent implementations
- Overall
  - Some very intriguing ideas
  - Controversies killed it