TDTS06: Computer Networks

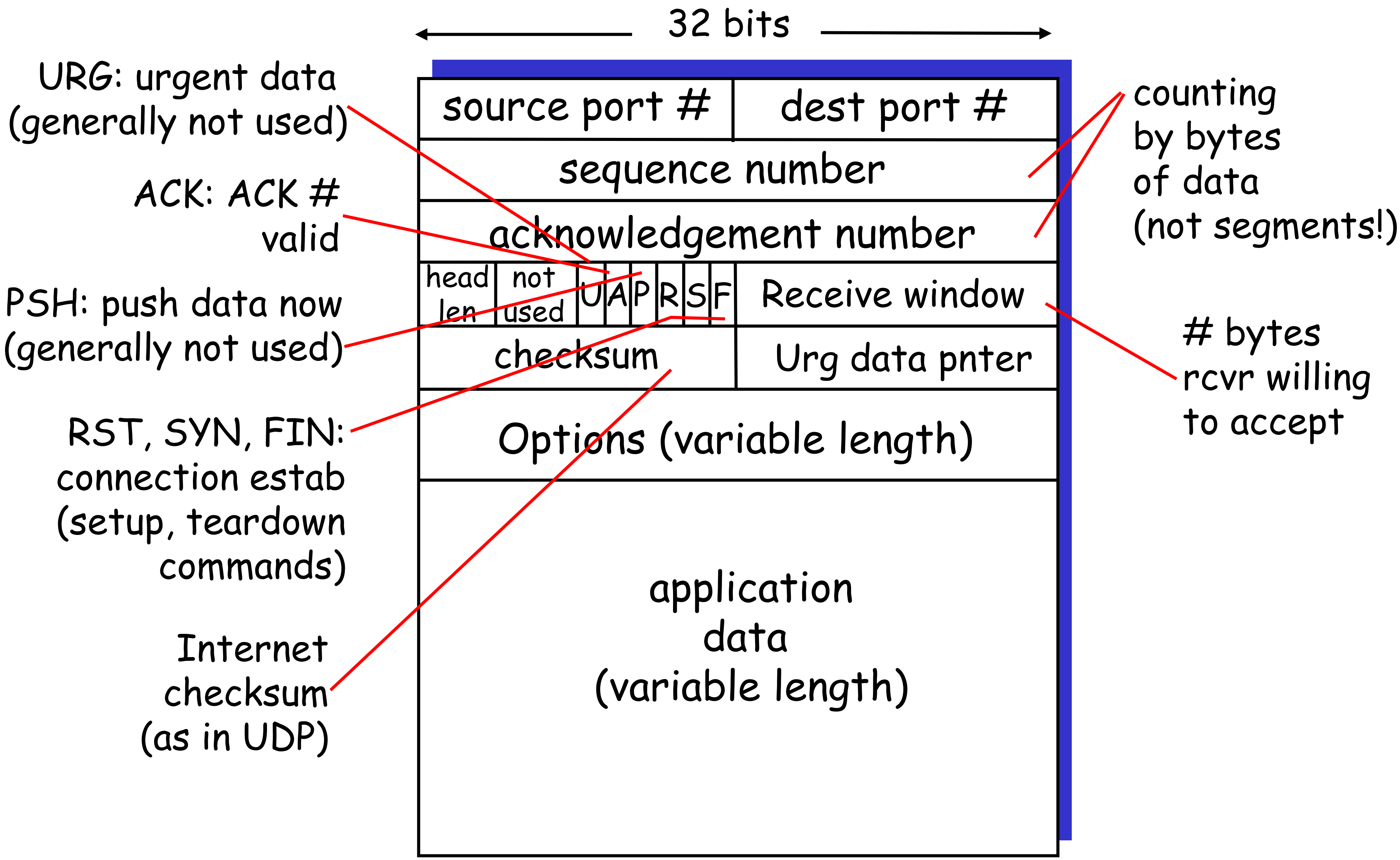
Instructor: Niklas Carlsson
Email: niklas.carlsson@liu.se

Notes derived from "*Computer Networking: A Top Down Approach*", by Jim Kurose and Keith Ross, Addison-Wesley.

The slides are adapted and modified based on slides from the book's companion Web site, as well as modified slides by Anirban Mahanti and Carey Williamson.

Transmission Control Protocol

TCP segment structure



Sequence and Acknowledgement Number

- TCP views data as unstructured, but ordered stream of bytes.
- Sequence numbers are over bytes, not segments
- Initial sequence number is chosen randomly
- TCP is full duplex – numbering of data is independent in each direction
- Acknowledgement number – sequence number of the next byte expected from the sender
- ACKs are cumulative

TCP seq. #'s and ACKs

Seq. #'s:

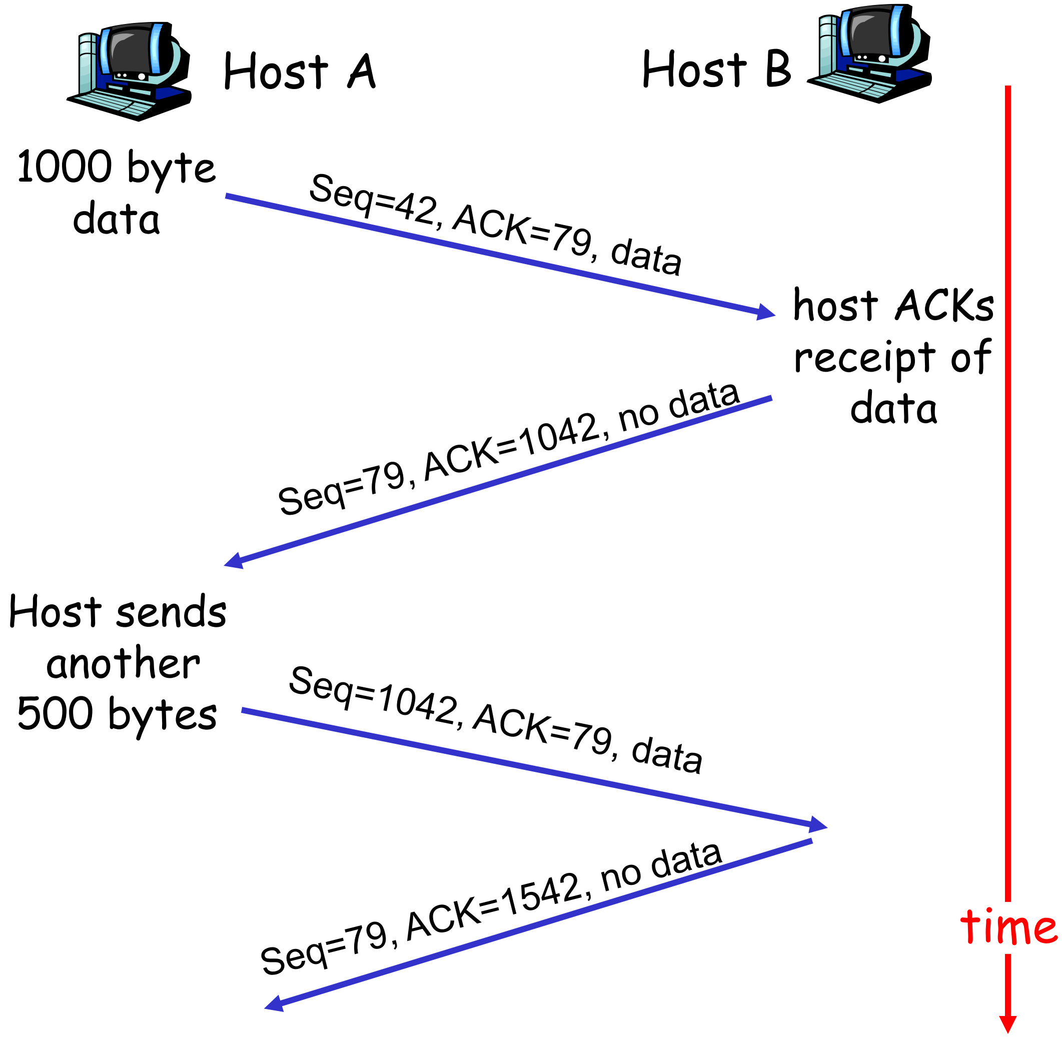
- byte stream "number" of first byte in segment's data

ACKs:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. **`RcvWindow`**)
- *client:* connection initiator

```
Socket clientSocket = new
 Socket("hostname","port

 number");
```

- *server:* contacted by client

```
Socket connectionSocket =
 welcomeSocket.accept();
```

TCP Connection Establishment

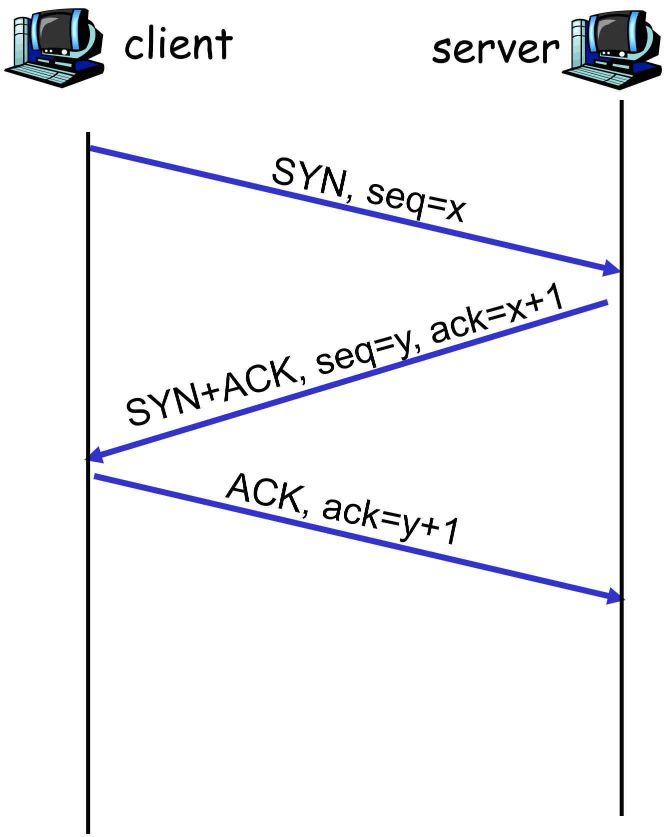
Three way handshake:

Step 1: client host sends TCP SYN segment to server
- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

TCP Connection Establishment



Three way handshake:

Step 1: client host sends TCP SYN segment to server

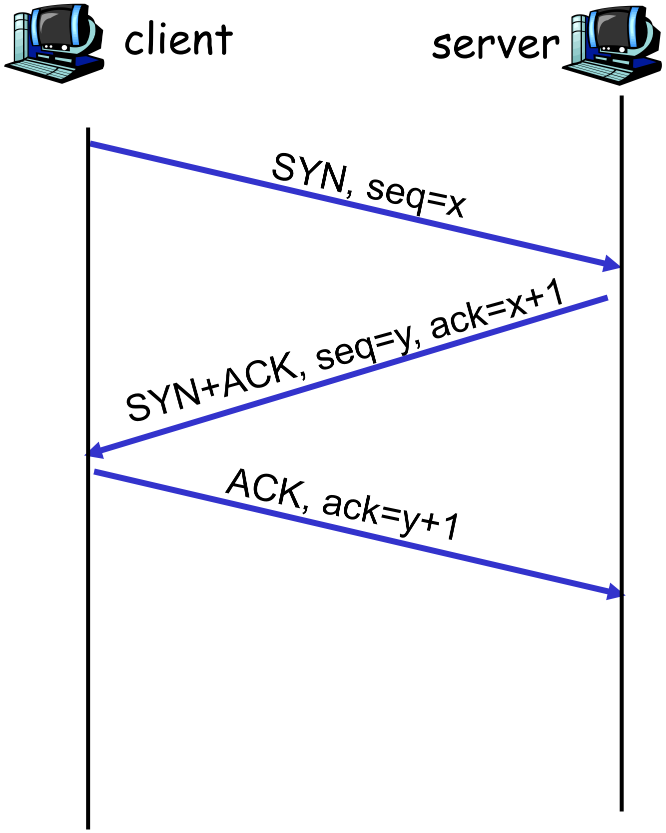
- specifies initial seq #
- no data

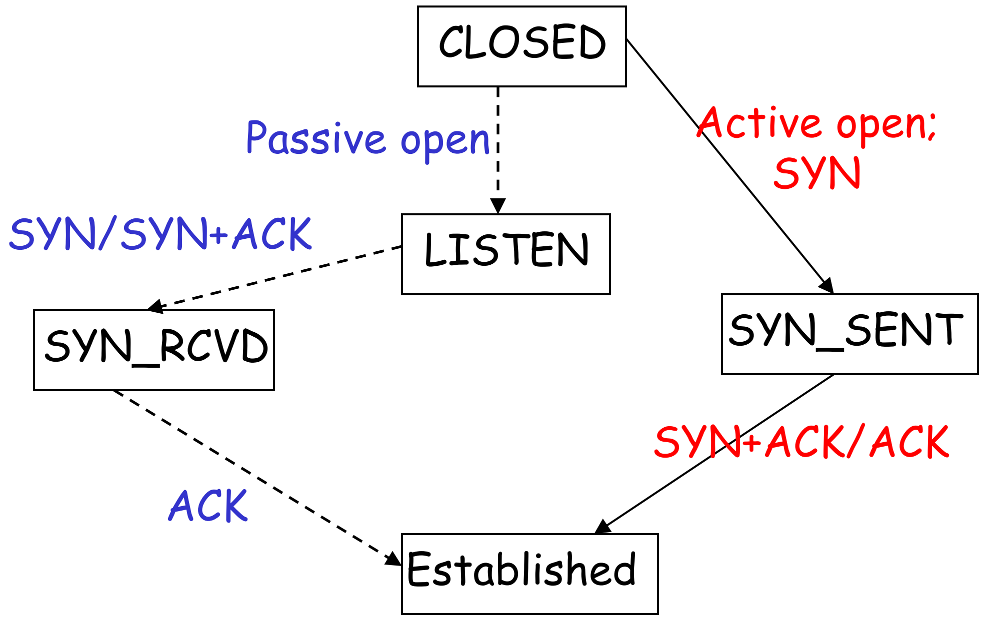
Step 2: server host receives SYN, replies with SYNACK segment

- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

TCP Connection Establishment

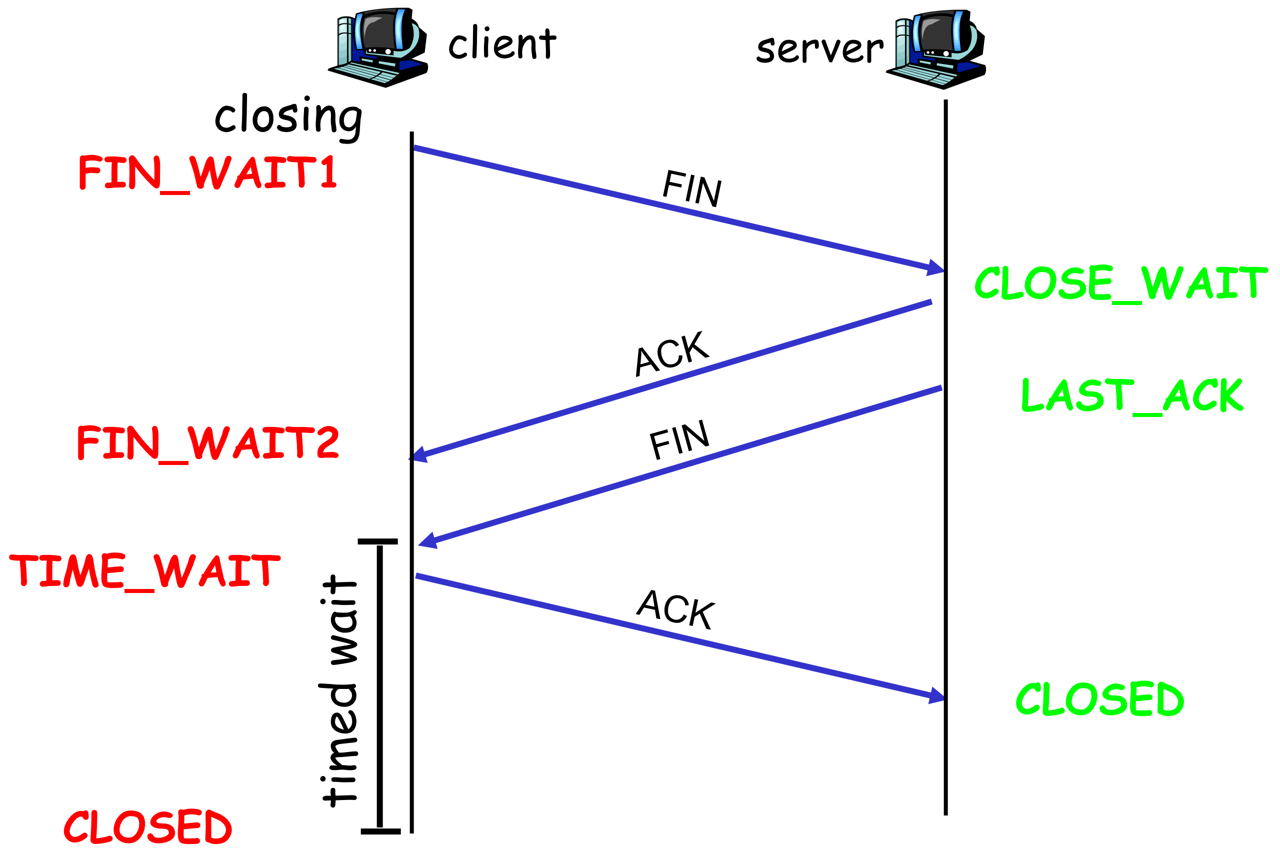




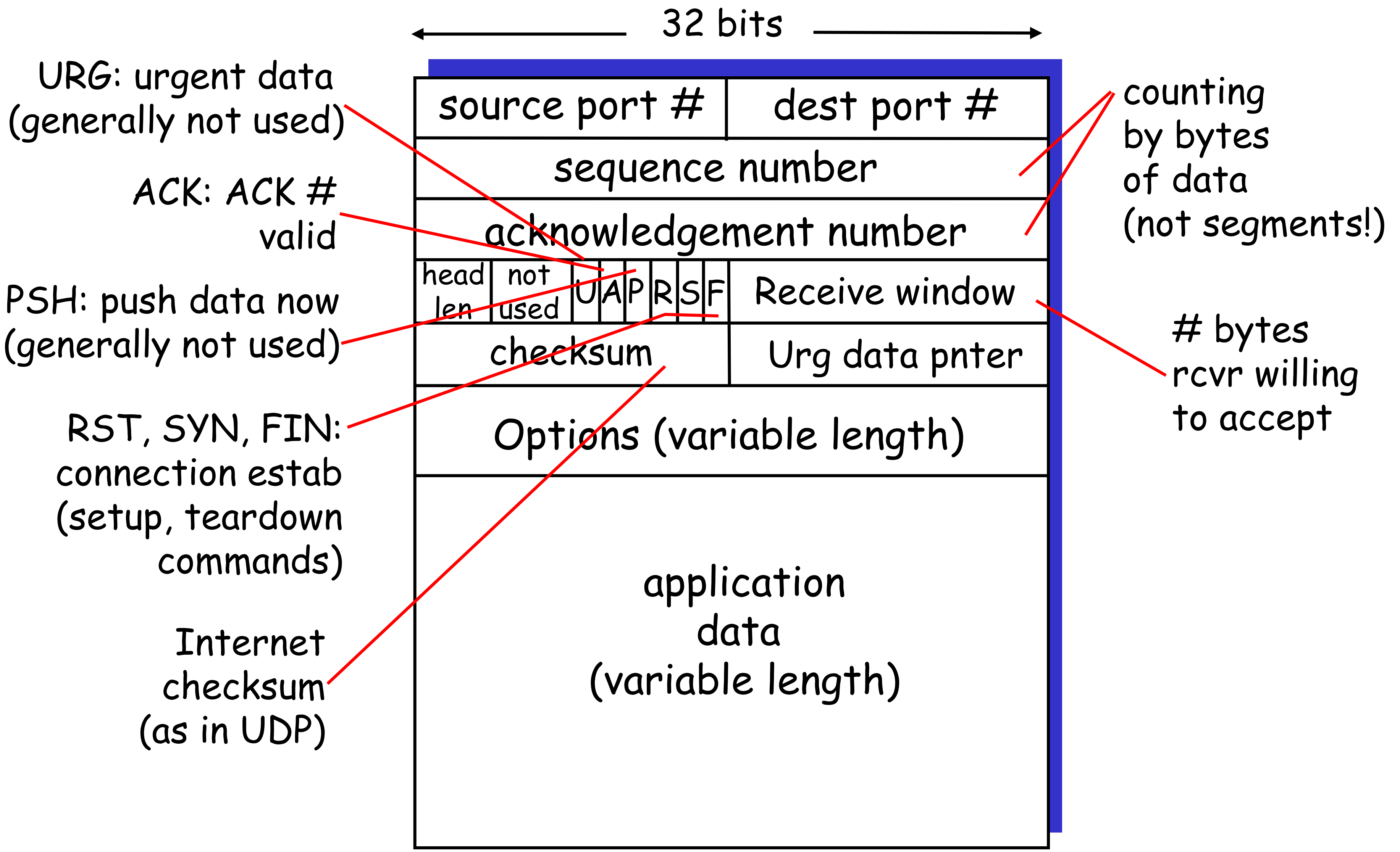
Solid line for client

Dashed line for server

TCP Connection Termination



TCP segment structure



TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

TCP sender events:

1) data rcvd from app:

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

2) timeout:

- retransmit segment that caused timeout
- restart timer

3) ack rcvd:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

 loop (forever) {
   switch(event)

   event: data received from application above
         create TCP segment with sequence number NextSeqNum
         if (timer currently not running)
              start timer
         pass segment to IP
         NextSeqNum = NextSeqNum + length(data)

    event: timer timeout
         retransmit not-yet-acknowledged segment with
                smallest sequence number
         start timer

    event: ACK received, with ACK field value of y
         if (y > SendBase) {
              SendBase = y
             if (there are currently not-yet-acknowledged segments)
                     start timer
             }

  }  /* end of loop forever */
```

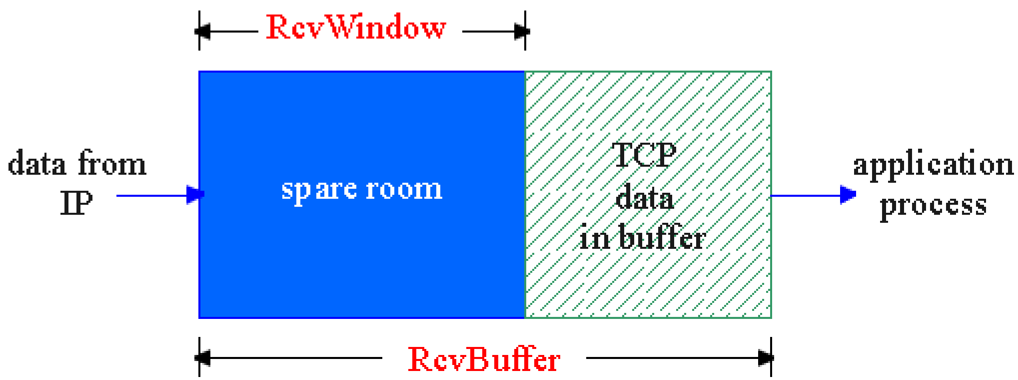
TCP Flow Control

flow control

sender won't overflow receiver's buffer by transmitting too much, too fast

TCP Flow Control

- receive side of TCP connection has a receive buffer:



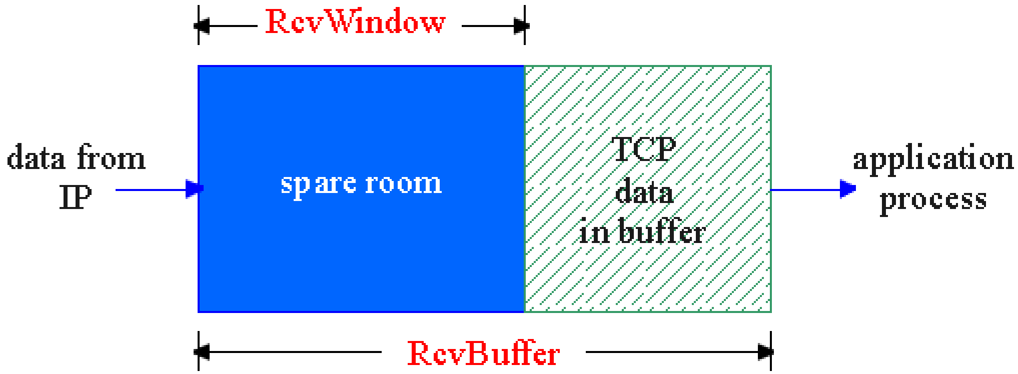
- app process may be slow at reading from buffer

flow control

sender won't overflow receiver's buffer by transmitting too much, too fast

- speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow control: how it works



(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer

= **RcvWindow**

= **RcvBuffer-[LastByteRcvd - LastByteRead]**

- Rcvr advertises spare room by including value of **RcvWindow** in segments
- Sender limits unACKed data to **RcvWindow**
  - guarantees receive buffer doesn't overflow

Silly Window Syndrome

- Recall: TCP uses sliding window
- "Silly Window" occurs when small-sized segments are transmitted, resulting in inefficient use of the network pipe
- For e.g., suppose that TCP sender generates data slowly, 1-byte at a time

Silly Window Syndrome

- Recall: TCP uses sliding window
- "Silly Window" occurs when small-sized segments are transmitted, resulting in inefficient use of the network pipe
- For e.g., suppose that TCP sender generates data slowly, 1-byte at a time
- Solution: wait until sender has enough data to transmit – "Nagle's Algorithm"

Nagle's Algorithm

1. TCP sender sends the first piece of data obtained from the application (even if data is only a few bytes).

2. Wait until enough bytes have accumulated in the TCP send buffer or until an ACK is received.

3. Repeat step 2 for the remainder of the transmission.

Silly Window Continued …

- Suppose that the receiver consumes data slowly
    - Receive Window opens slowly, and thus sender is forced to send small-sized segments
- Solutions

Silly Window Continued …

- Suppose that the receiver consumes data slowly
  - Receive Window opens slowly, and thus sender is forced to send small-sized segments
- Solutions
  - Delayed ACK
  - Advertise Receive Window = 0, until reasonable amount of space available in receiver's buffer

Historical Perspective

- October 1986, Internet had its first congestion collapse
- Link LBL to UC Berkeley
  - 400 yards, 3 hops, 32 Kbps
  - throughput dropped to 40 bps
  - factor of ~1000 drop!
- Van Jacobson proposes TCP Congestion Control:
  - Achieve high utilization
  - Avoid congestion
  - Share bandwidth

Principles of Congestion Control

- **Congestion:** informally: "too many sources sending too much data too fast for *network* to handle"
- Different from flow control!
- Manifestations:
  - Packet loss (buffer overflow at routers)
  - Increased end-to-end delays (queuing in router buffers)
- Results in unfairness and poor utilization of network resources
  - Resources used by dropped packets (before they were lost)
  - Retransmissions
  - Poor resource allocation at high load

Congestion Control: Approaches

- Goal: Throttle senders as needed to ensure load on the network is "reasonable"
- 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 (e.g., ECN)
  - explicit rate sender should send at

TCP Congestion Control: Overview

- end-end control (no network assistance)
- Limit the number of packets in the network to window W
- Roughly,

$$\text{rate} = \frac{W}{RTT} \text{ Bytes/sec}$$

- W is dynamic, function of perceived network congestion

TCP Congestion Controls

- Tahoe (Jacobson 1988)
  - Slow Start
  - Congestion Avoidance
  - Fast Retransmit
- Reno (Jacobson 1990)
  - Fast Recovery
- SACK
- Vegas (Brakmo & Peterson 1994)
  - Delay and loss as indicators of congestion
- Cubic and many other ...

Slow Start

- "Slow Start" is used to reach the equilibrium state
- Initially: W = 1 (slow start)
- On each successful ACK:

  W ← W + 1

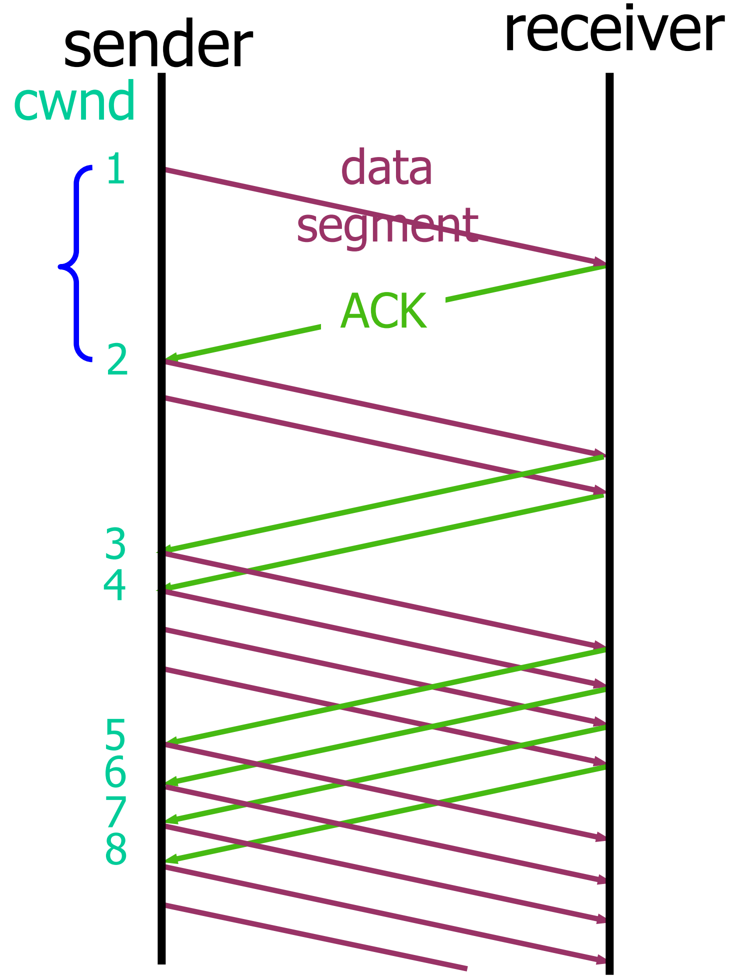
- Exponential growth of W

  each RTT: W ← 2 x W

- Enter CA when

  W >= ssthresh

- ssthresh: window size after which TCP cautiously probes for bandwidth



Congestion Avoidance

- Starts when

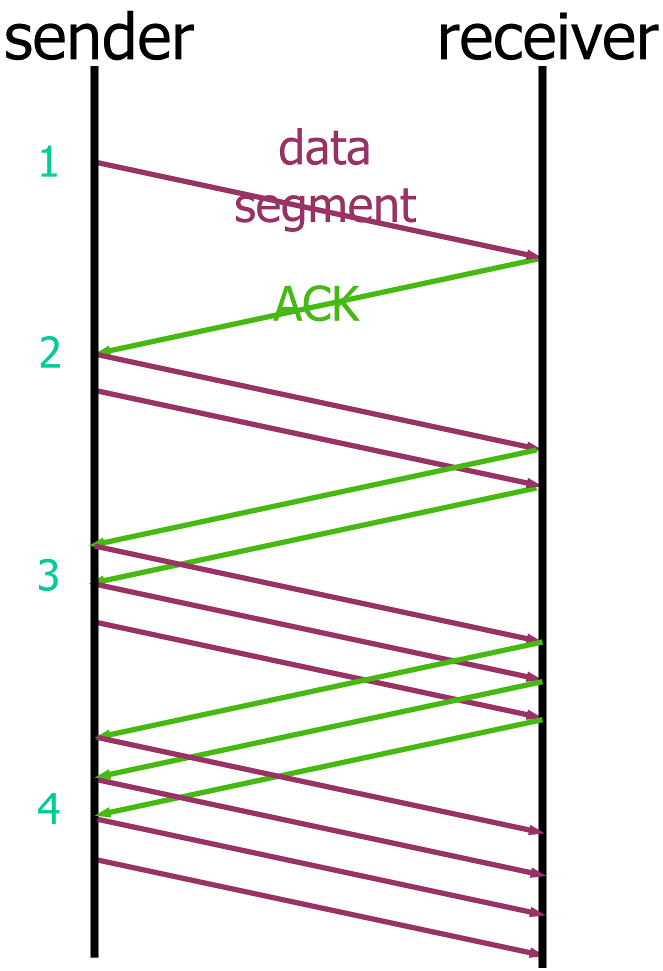
  $W \geq ssthresh$

- On each successful ACK

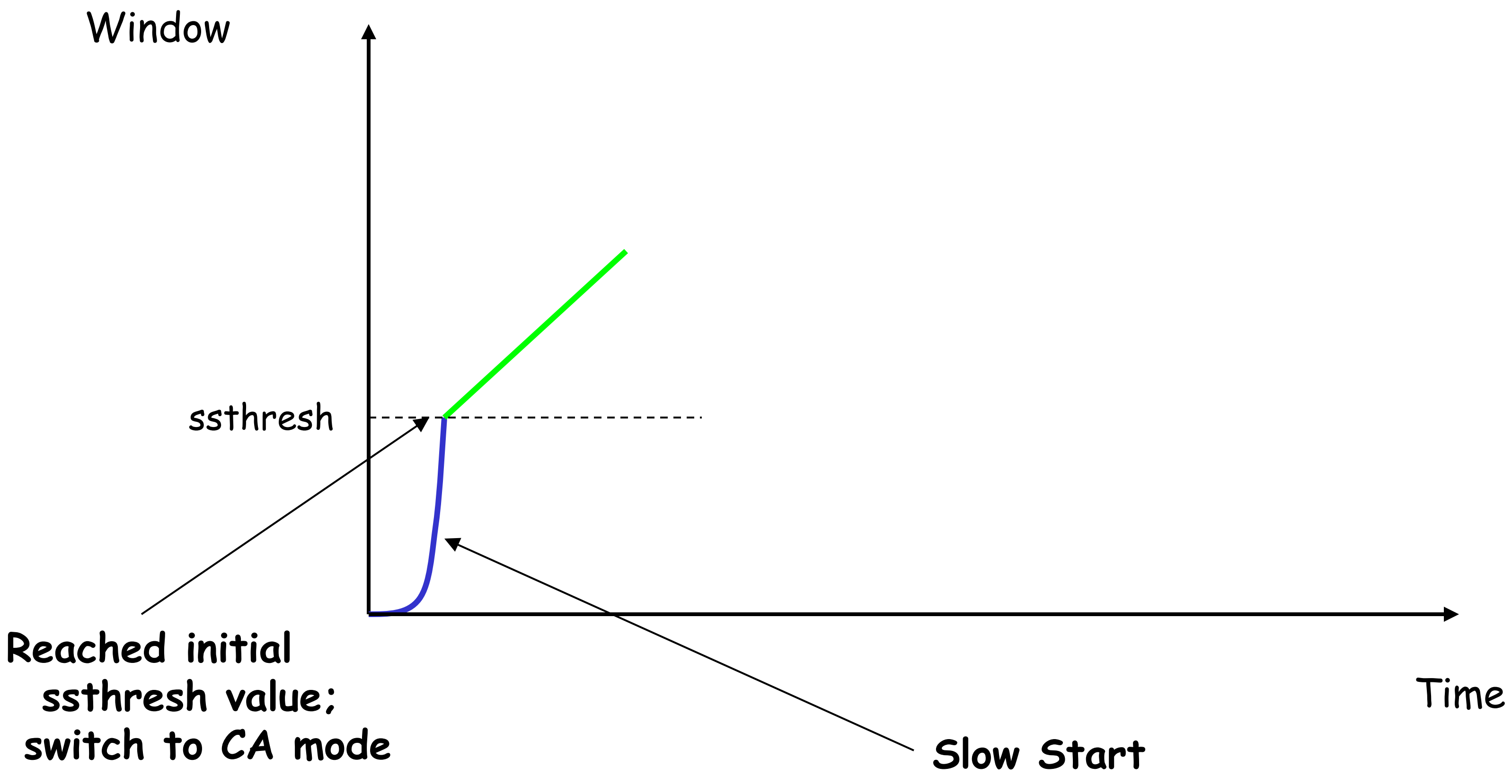
  $W \leftarrow W + 1/W$

- Linear growth of W each RTT

  $W \leftarrow W + 1$



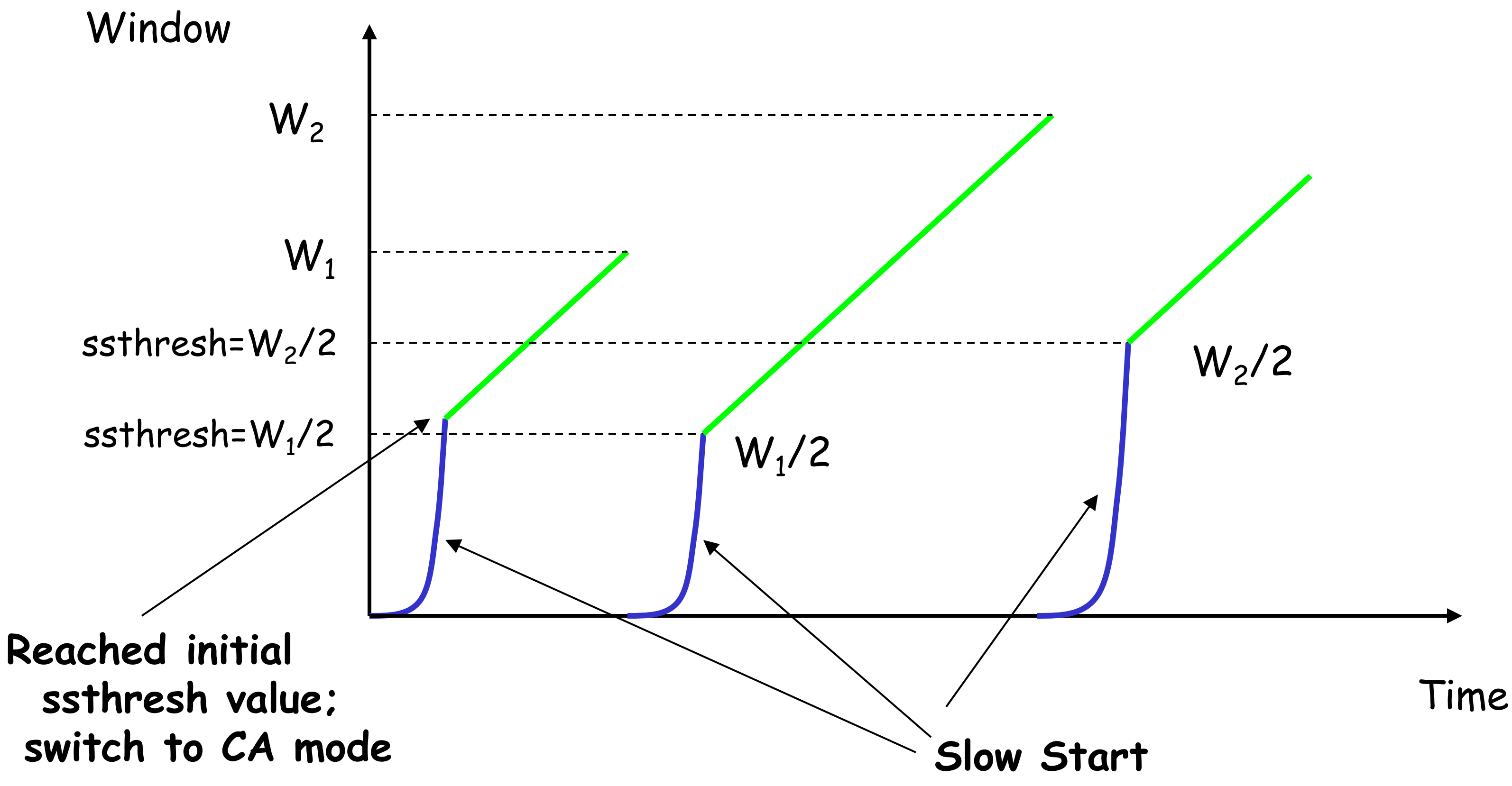
TCP (initial version without loss)



CA: Additive Increase, Multiplicative Decrease

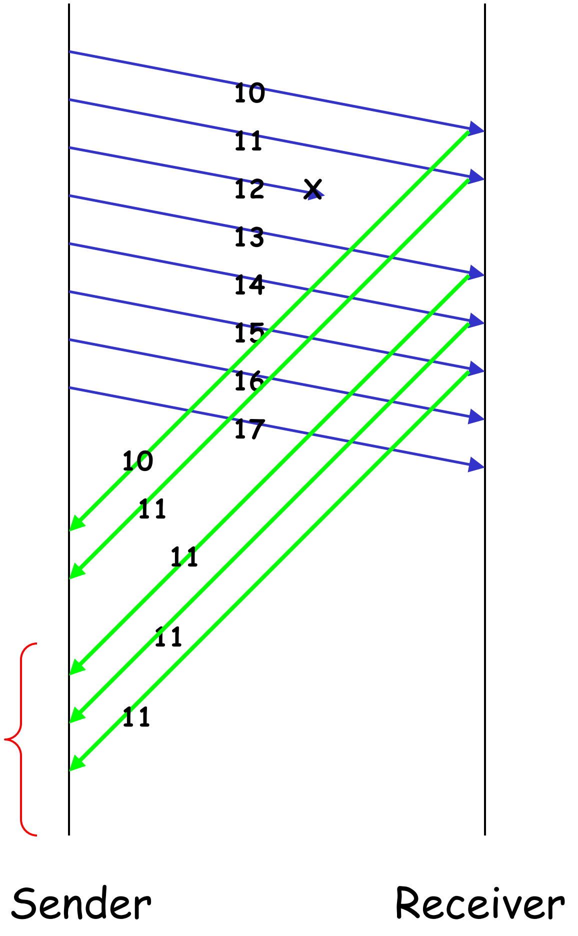
- We have "additive increase" in the absence of loss events
- After loss event, decrease congestion window by half – "multiplicative decrease"
  - ssthresh = W/2
  - Enter Slow Start

TCP Tahoe (more on losses next …)



Detecting Packet Loss

- Assumption: loss indicates congestion
- Option 1: time-out
  - Waiting for a time-out can be long!
- Option 2: duplicate ACKs
  - How many? At least 3.



Fast Retransmit

- Wait for a timeout is quite long
- Immediately retransmits after 3 dupACKs without waiting for timeout
- Adjusts ssthresh

    ssthresh ← W/2

- Enter Slow Start

    W = 1

How to Set TCP Timeout Value?

- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

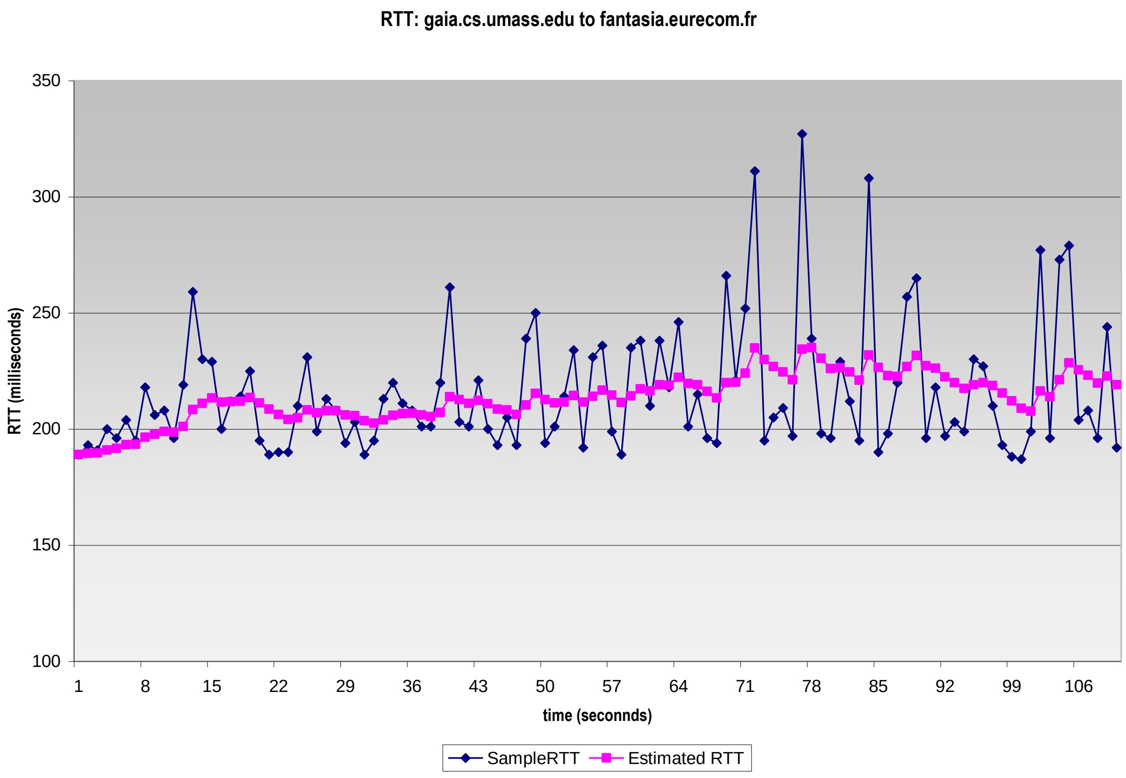
How to Estimate RTT?

- **SampleRTT**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **SampleRTT** will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **SampleRTT**

TCP Round-Trip Time and Timeout

```
EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT
```

- EWMA
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP Round Trip Time and Timeout

[Jacobson/Karels Algorithm]

Setting the timeout

- **EstimtedRTT** plus "safety margin"
  - large variation in **EstimatedRTT** -> larger safety margin
- first estimate how much SampleRTT deviates from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
               β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

Then set timeout interval:

```
TimeoutInterval = μ*EstimatedRTT + Ø*DevRTT
```

Typically, μ =1 and Ø = 4.

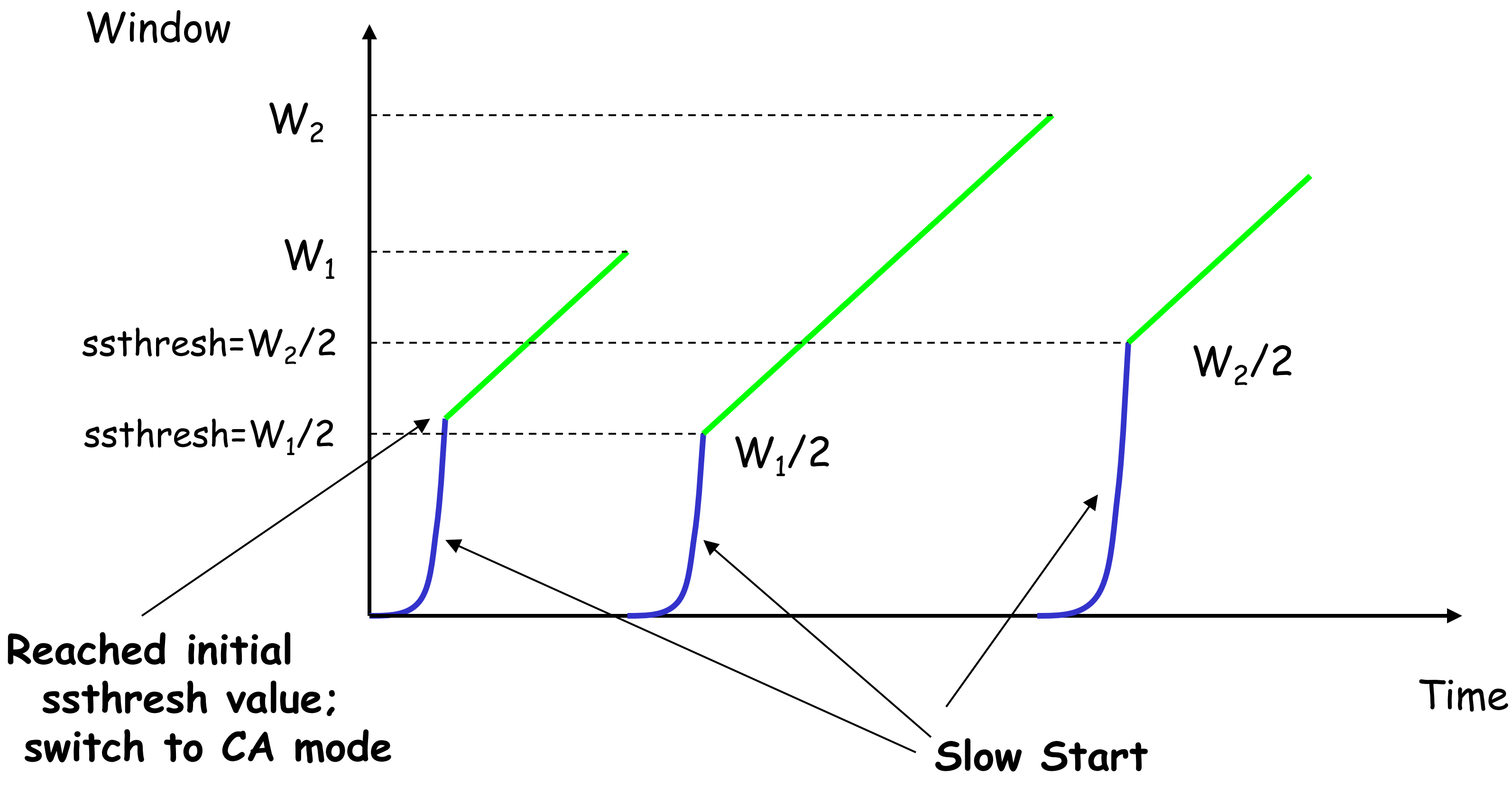
TCP Tahoe: Summary

- Basic ideas
  - Gently probe network for spare capacity
  - Drastically reduce rate on congestion
  - Windowing: self-clocking
  - Other functions: round trip time estimation, error recovery

```
for every ACK {
        if (W < ssthresh) then W++          (SS)
        else            W += 1/W            (CA)

    }
    for every loss {
            ssthresh = W/2
            W  = 1
    }
```

TCP Tahoe


Window
W2
W1
ssthresh=W2/2
ssthresh=W1/2
W1/2
W2/2
Time
Reached initial ssthresh value; switch to CA mode
Slow Start

Questions?

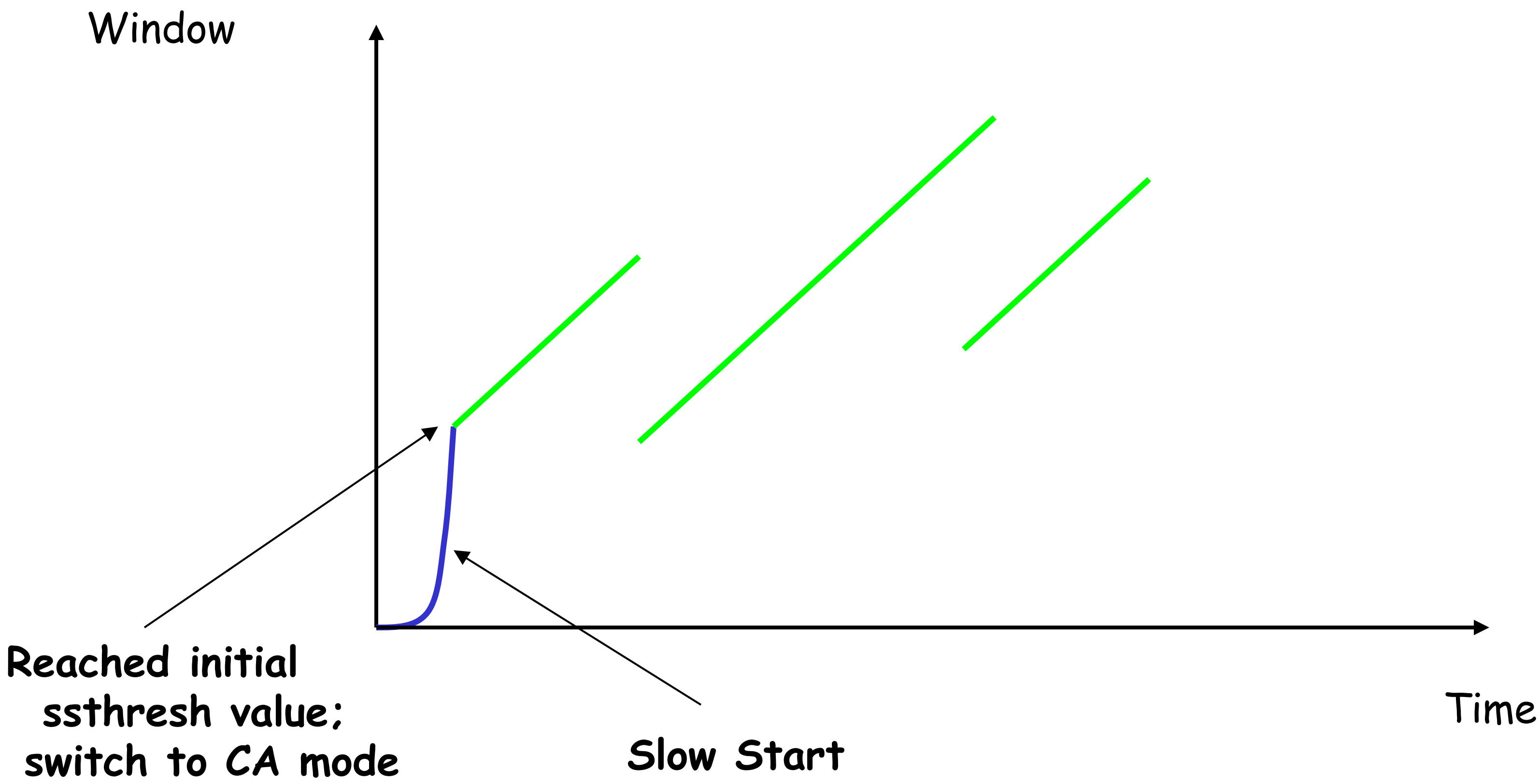
- Q. 1. To what value is ssthresh initialized to at the start of the algorithm?

- Q. 2. Why is "Fast Retransmit" triggered on receiving 3 duplicate ACKs (i.e., why isn't it triggered on receiving a single duplicate ACK)?

- Q. 3. Can we do better than TCP Tahoe?

TCP Reno

Note how there is “Fast Recovery” after cutting Window in half



TCP Reno: Fast Recovery

- Objective: prevent `pipe' from emptying after fast retransmit
  - each dup ACK represents a packet having left the pipe (successfully received)
  - Let's enter the "FR/FR" mode on 3 dup ACKs

ssthresh ← W/2
retransmit lost packet
W ← ssthresh + ndup (window inflation)
Wait till W is large enough; transmit new packet(s)
On non-dup ACK (1 RTT later)
    W ← ssthresh (window deflation)
    enter CA mode

TCP Reno: Summary

- Fast Recovery along with Fast Retransmit used to avoid slow start
- On 3 duplicate ACKs
  - Fast retransmit and fast recovery
- On timeout
  - Fast retransmit and slow start

TCP Throughput

- What's the average throughout ot TCP as a function of window size and RTT?
  - Ignore slow start
- Let W be the window size when loss occurs.
- When window is W, throughput is W/RTT
- Just after loss, window drops to W/2, throughput to W/2RTT.
- Average throughout: .75 W/RTT

TCP Futures

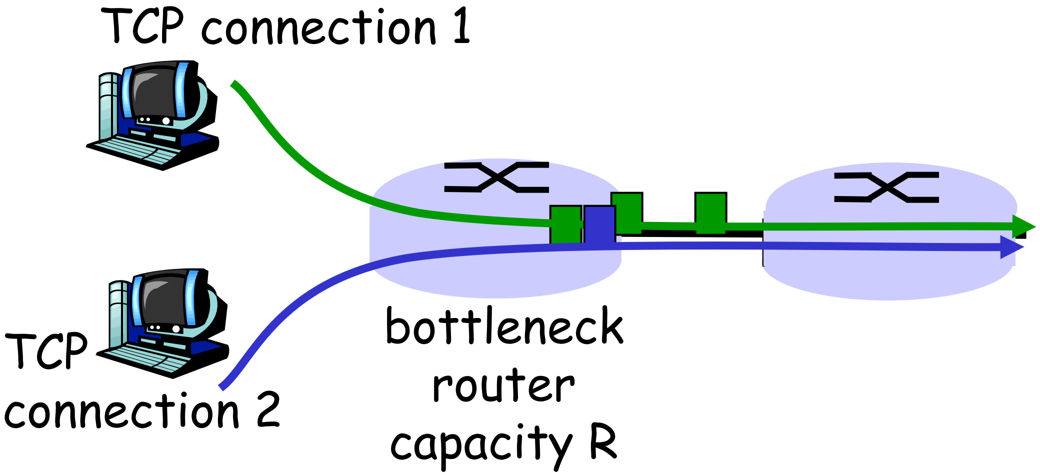
- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- Requires window size W = 83,333 in-flight segments
- Throughput in terms of loss rate:

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- ➜ $L = 2 \cdot 10^{-10}$ *Wow*
- New versions of TCP for high-speed needed!

TCP Fairness

Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Fairness (more)

- TCP fairness: dependency on RTT
  - Connections with long RTT get less throughput
- Parallel TCP connections
- TCP friendliness for UDP streams
  - Similar throughput (and behavior) as TCP; e.g.,

$$throughput \propto \frac{1}{RTT\sqrt{L}}$$

Chapter 3: Summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet
  - UDP
  - TCP

Next:

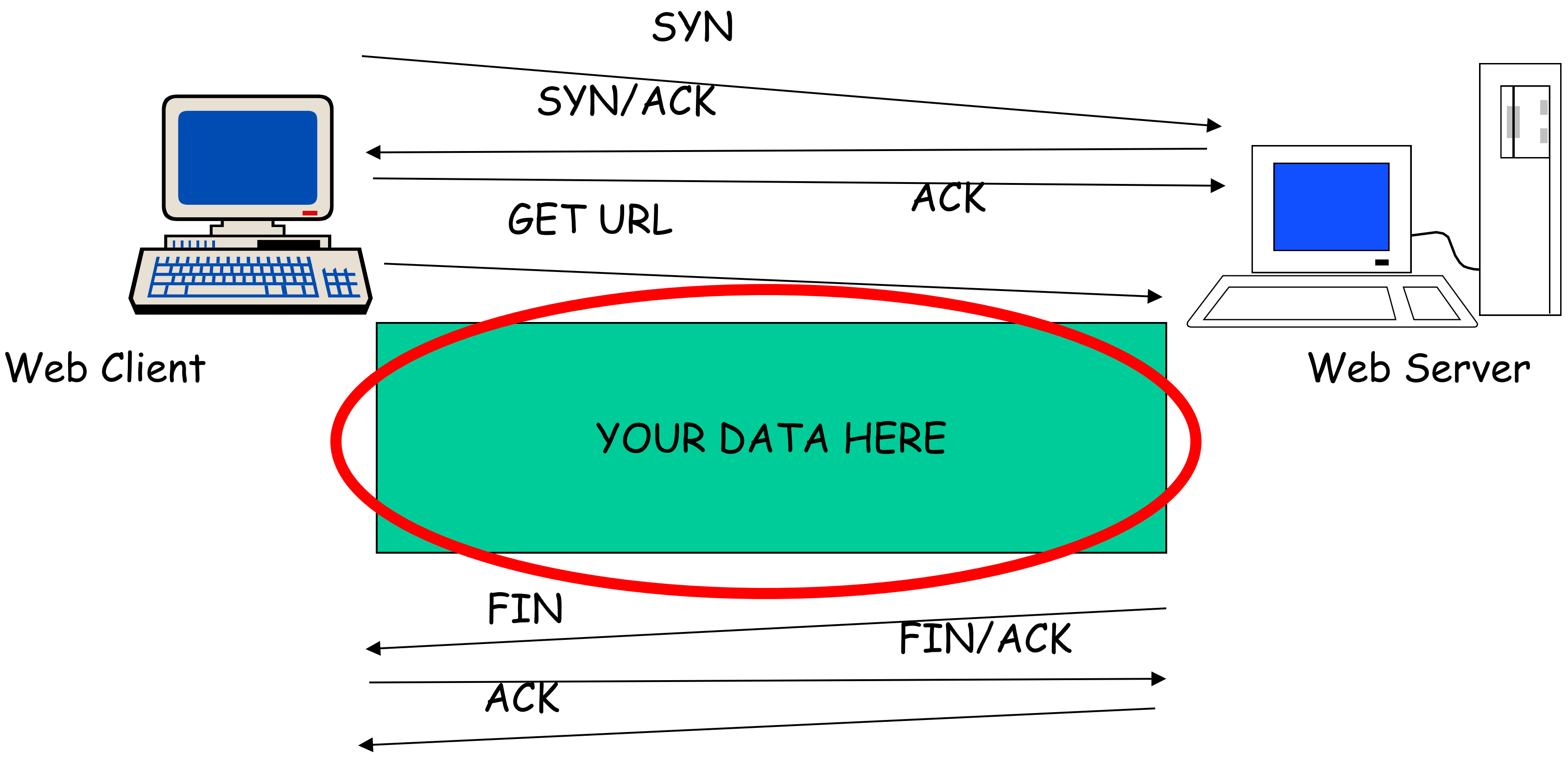
- leaving the network "edge" (application, transport layers)
- into the network "core"

Tutorial: TCP 101

- The Transmission Control Protocol (TCP) is the protocol that sends your data reliably
- Used for email, Web, ftp, telnet, p2p,…
- Makes sure that data is received correctly: right data, right order, exactly once
- Detects and recovers from any problems that occur at the IP network layer
- Mechanisms for reliable data transfer: sequence numbers, acknowledgements, timers, retransmissions, flow control…

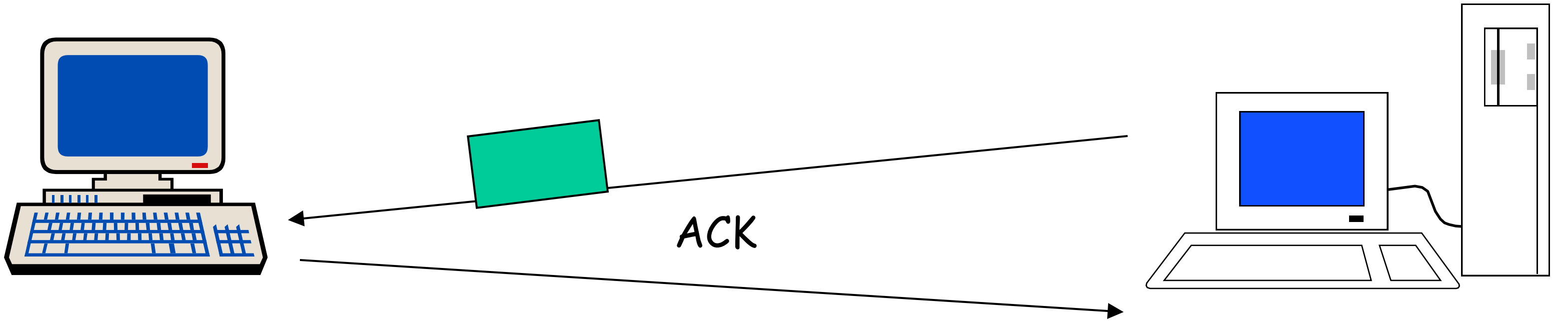
TCP 101 (Cont'd)

- TCP is a connection-oriented protocol



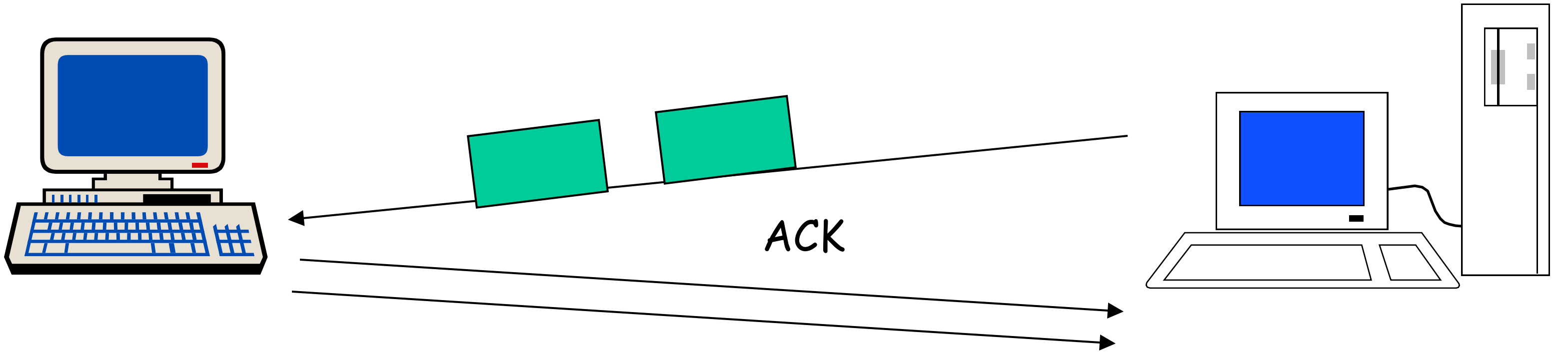
TCP 101 (Cont'd)

- TCP slow-start and congestion avoidance



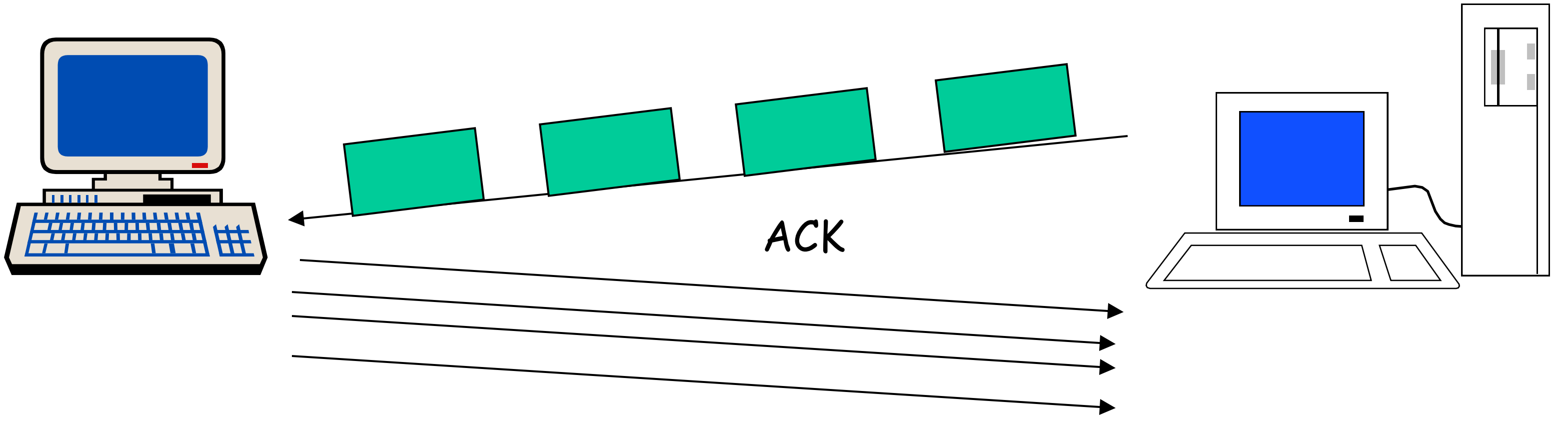
TCP 101 (Cont'd)

- TCP slow-start and congestion avoidance



TCP 101 (Cont'd)

- TCP slow-start and congestion avoidance



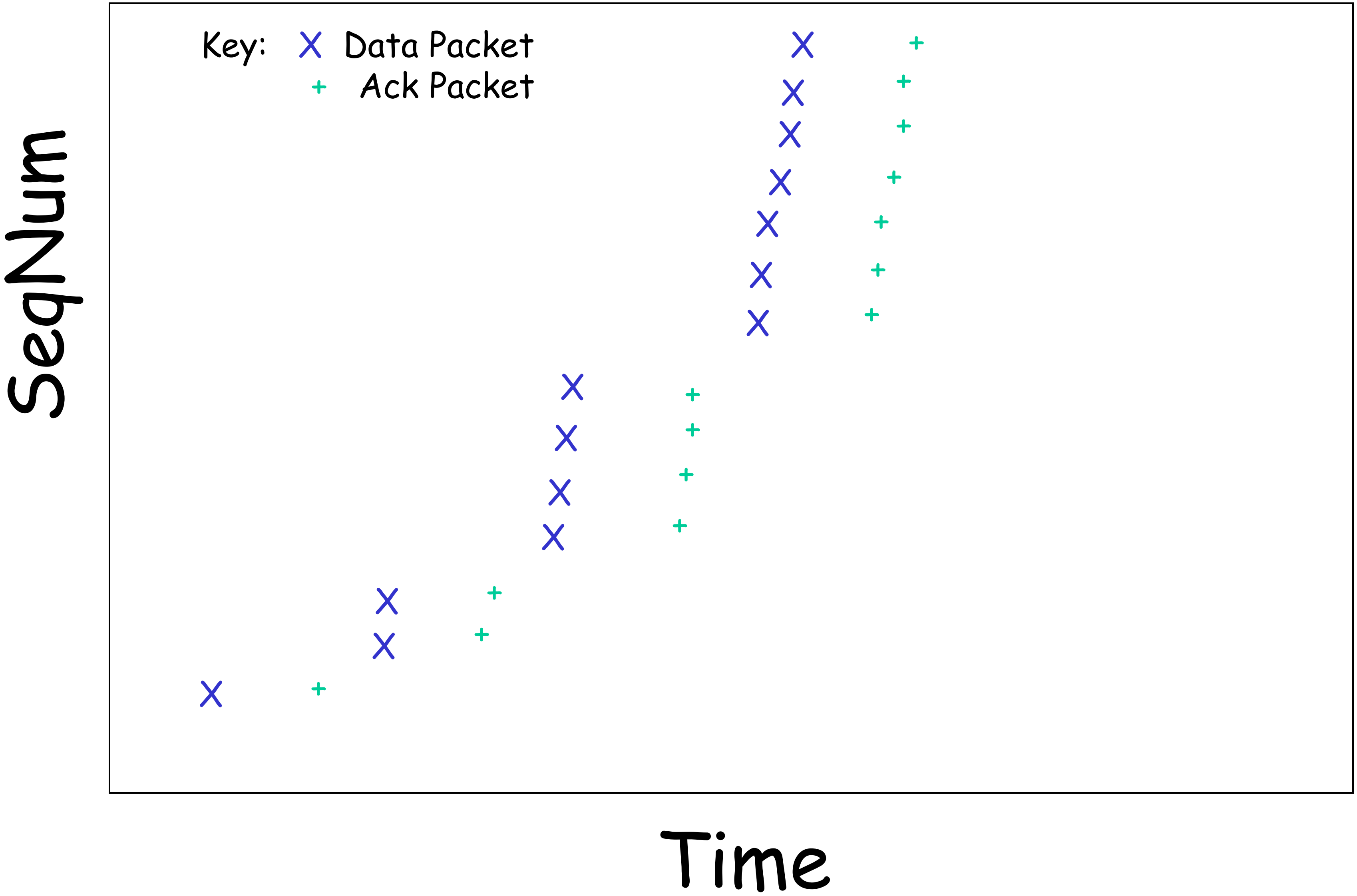
TCP 101 (Cont'd)

- This (exponential growth) "slow start" process continues until either:
  - packet loss: after a brief recovery phase, you enter a (linear growth) "congestion avoidance" phase based on slow-start threshold found
  - limit reached: slow-start threshold, or maximum advertised receive window size
  - all done: terminate connection and go home

TCP 201: Examples …

Tutorial: TCP 301

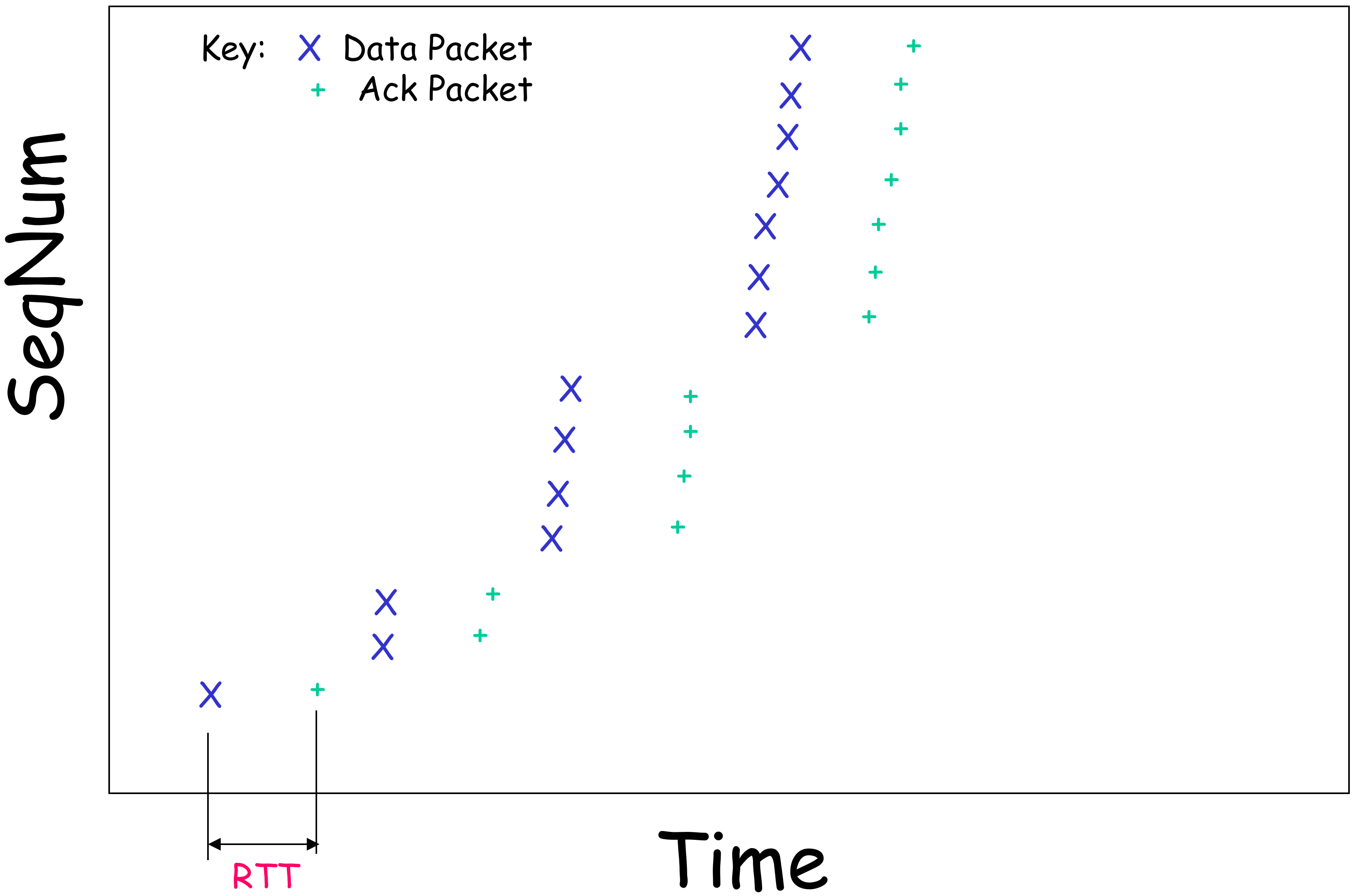
- There is a beautiful way to plot and visualize the dynamics of TCP behaviour
- Called a "TCP Sequence Number Plot"
- Plot packet events (data and acks) as points in 2-D space, with time on the horizontal axis, and sequence number on the vertical axis
- Example: Consider a 14-packet transfer

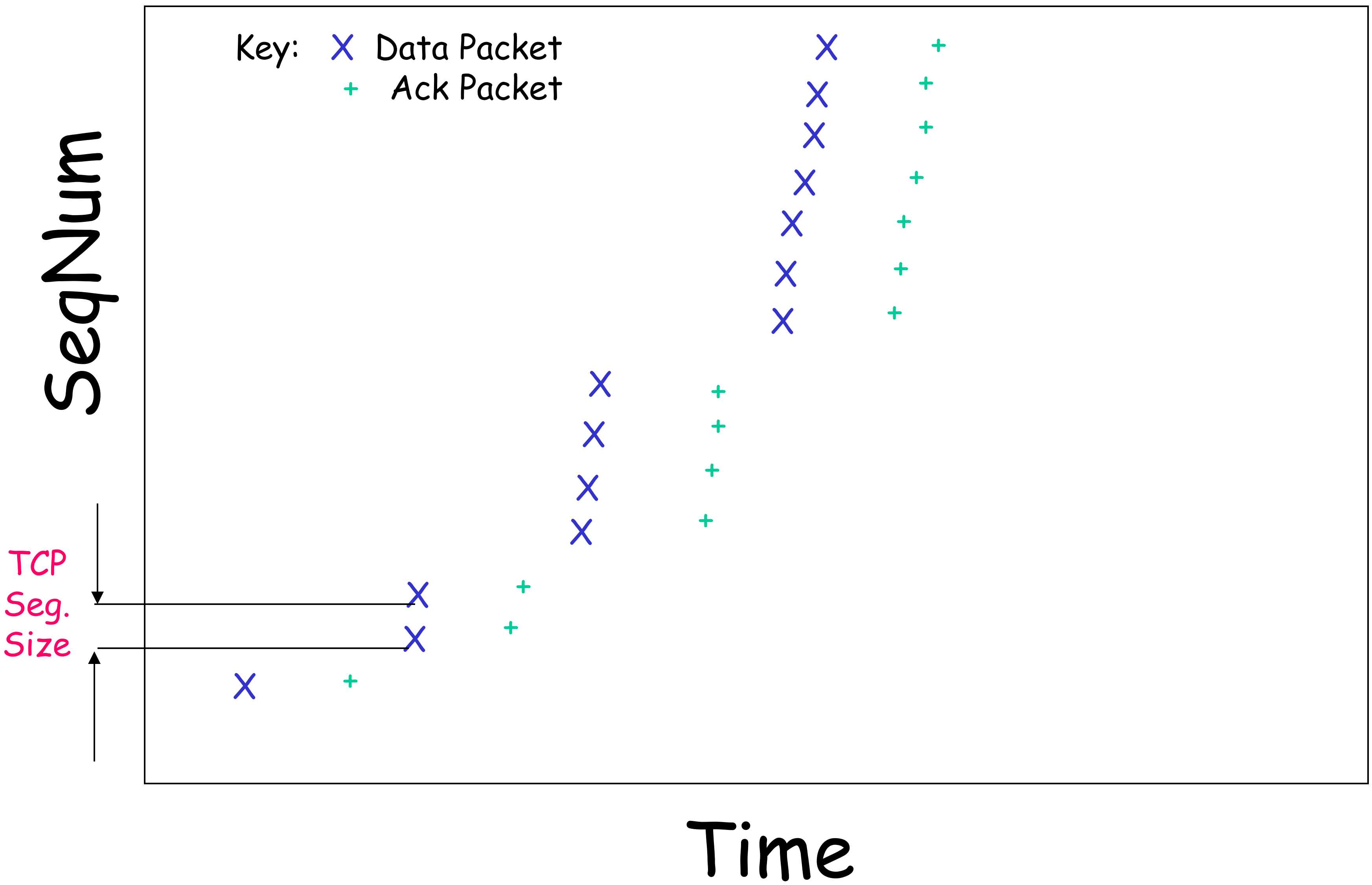

Key:
Data Packet
Ack Packet
SeqNum
Time

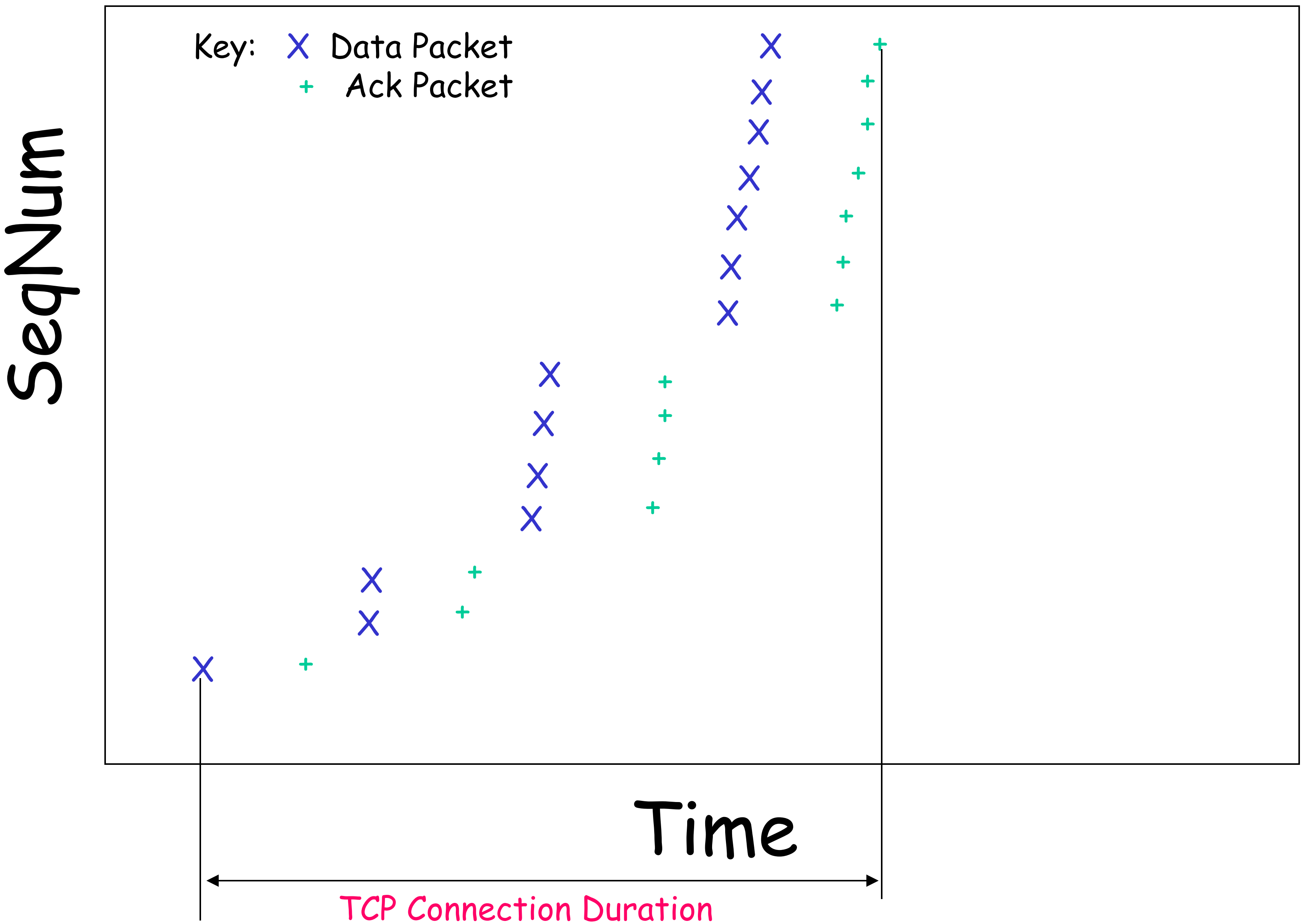
So What?

- What can it tell you?

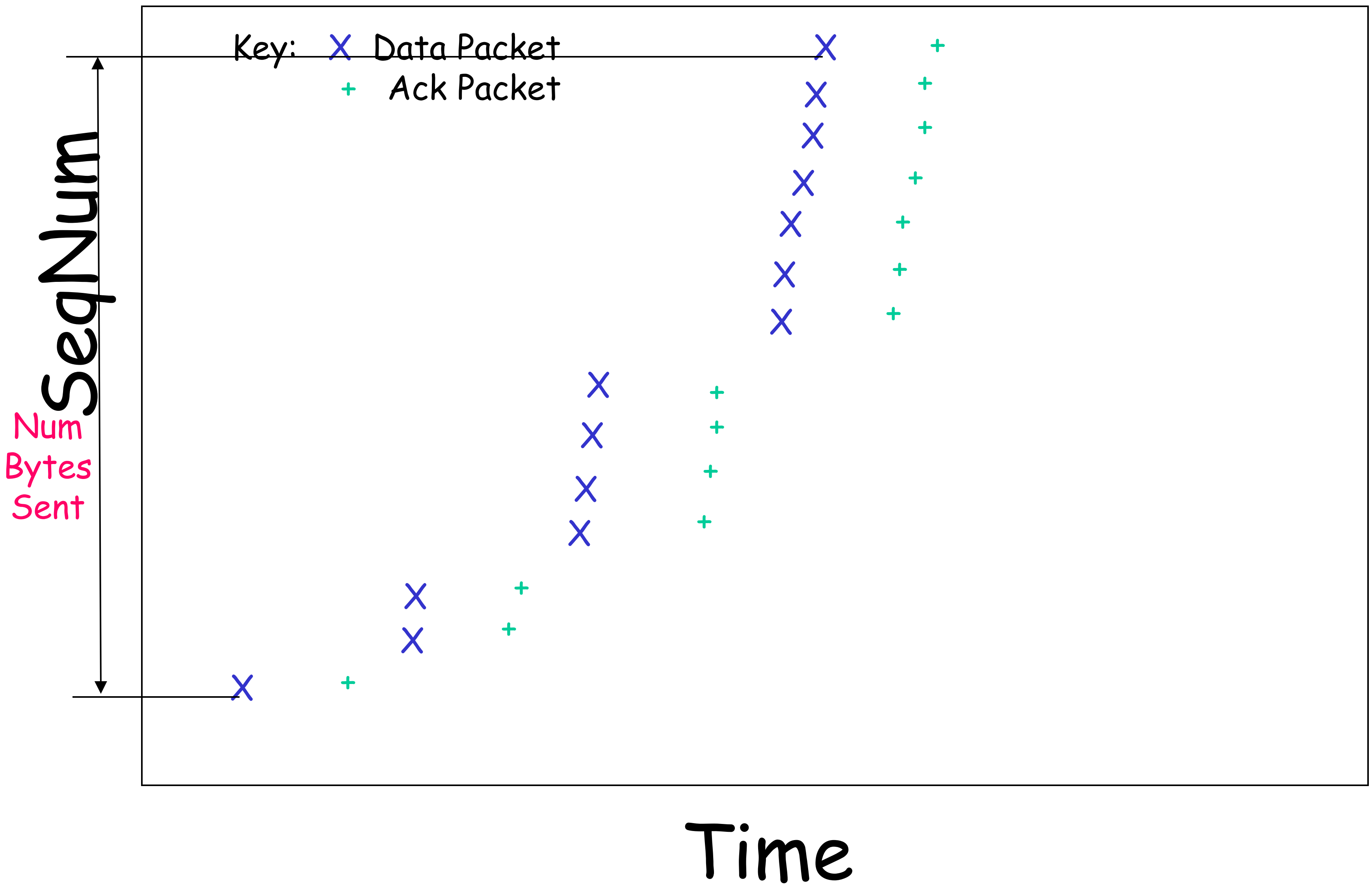
- Everything!!!

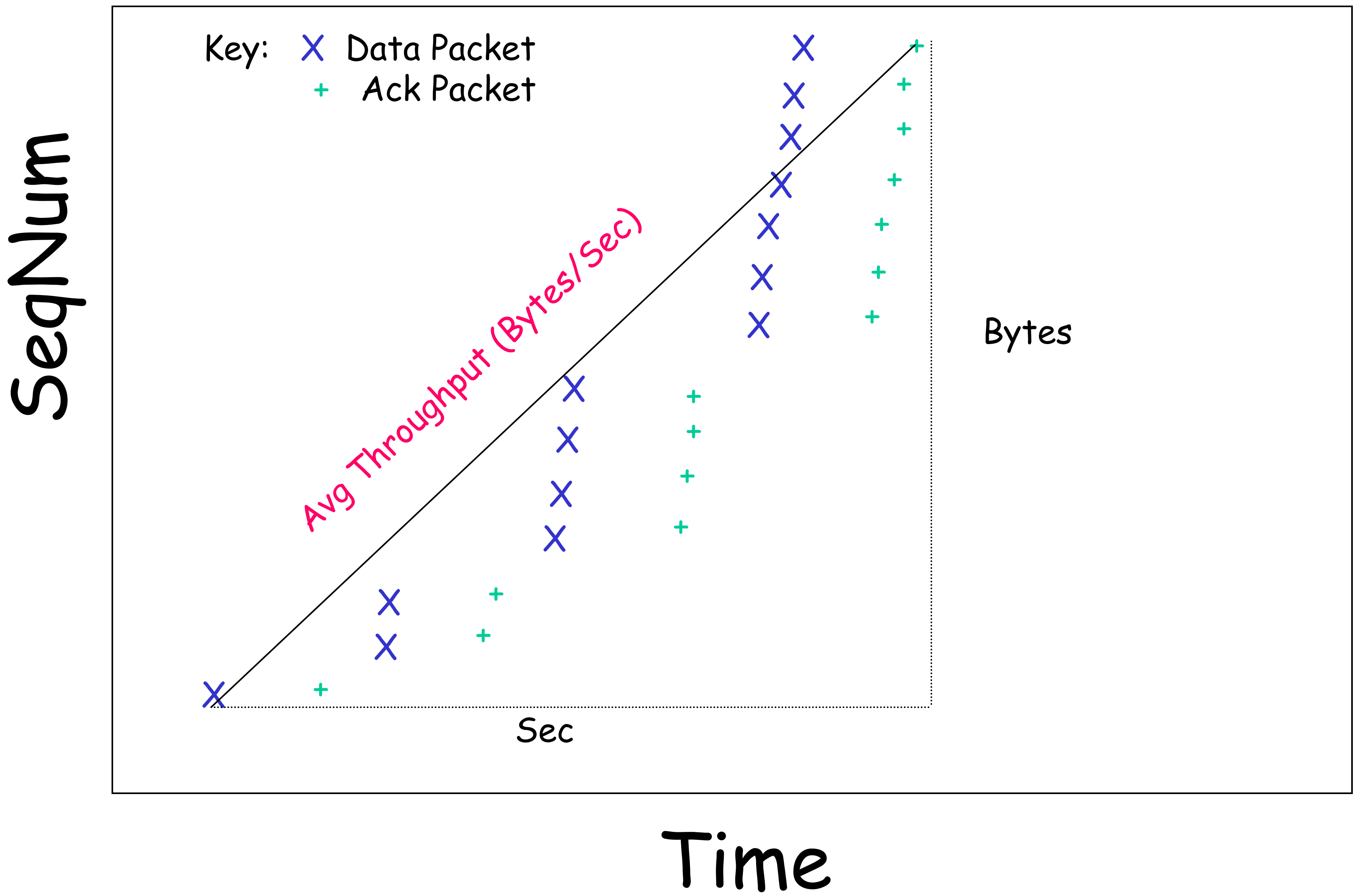

Key:
X Data Packet
+ Ack Packet
SeqNum
Time
RTT


Key:
Data Packet
Ack Packet
SeqNum
TCP
Seg.
Size
Time

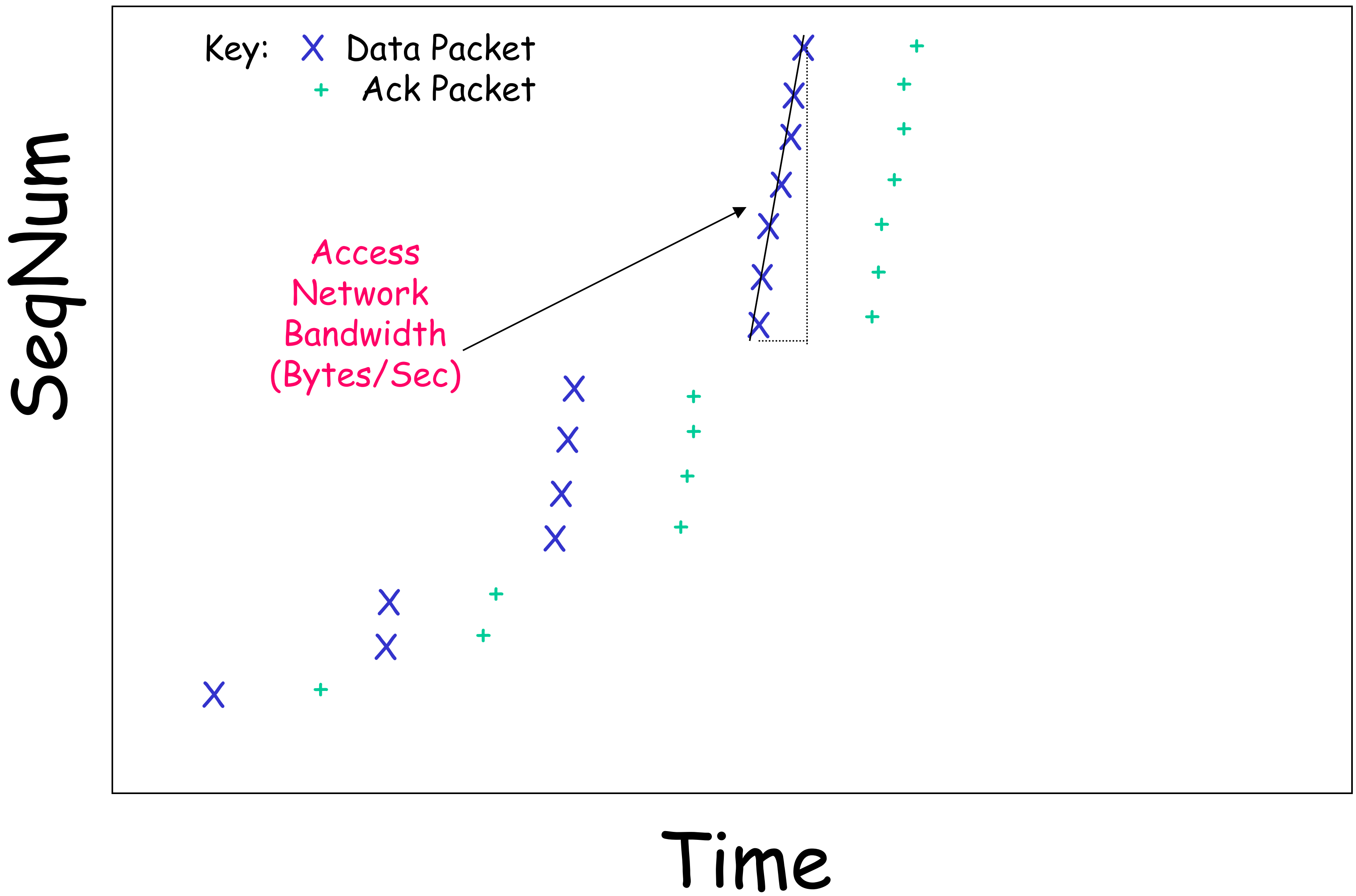


Key:
Data Packet
Ack Packet
SeqNum
Time
TCP Connection Duration

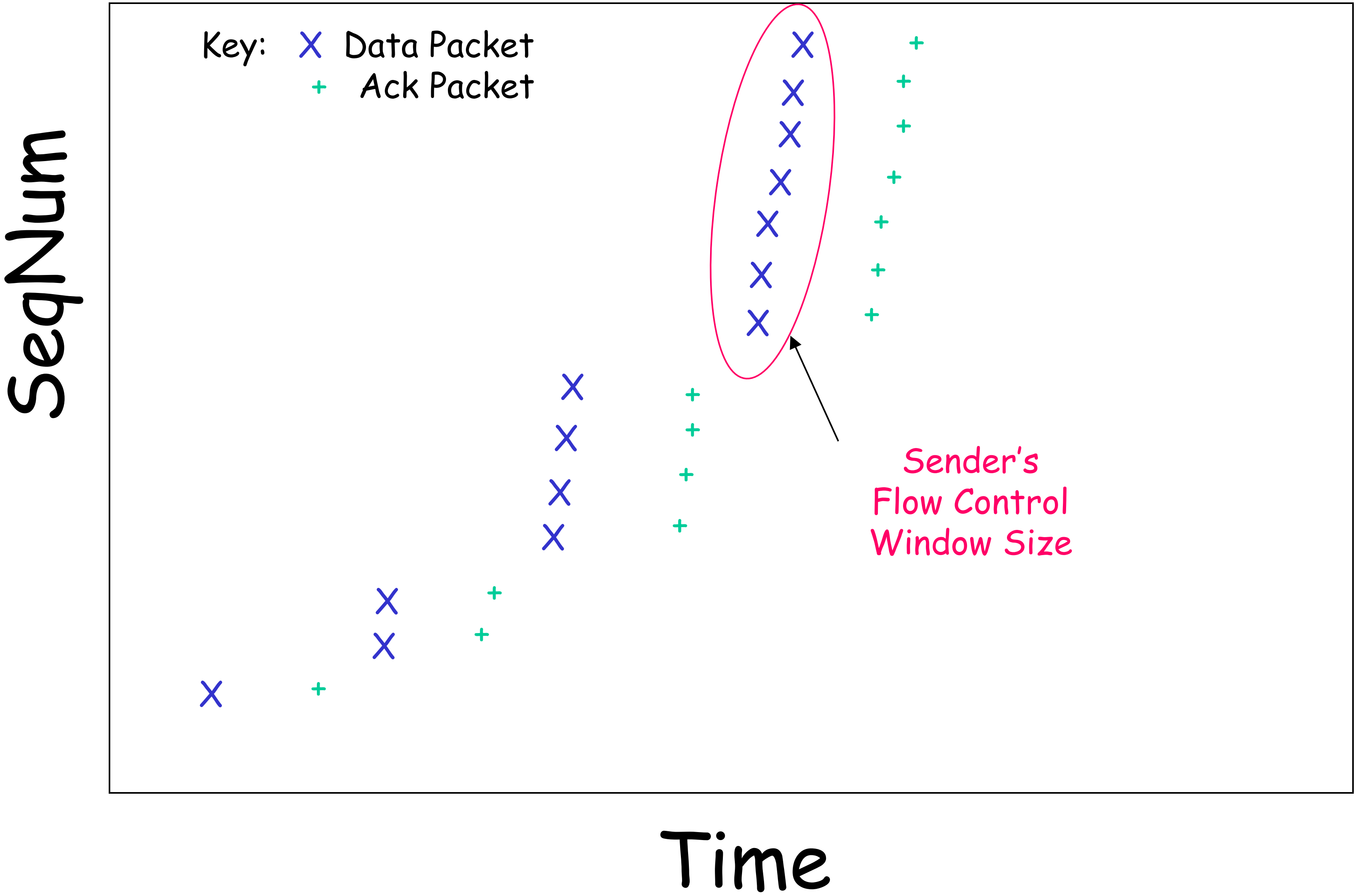

Key: X Data Packet
+ Ack Packet
SeqNum
Num
Bytes
Sent
Time

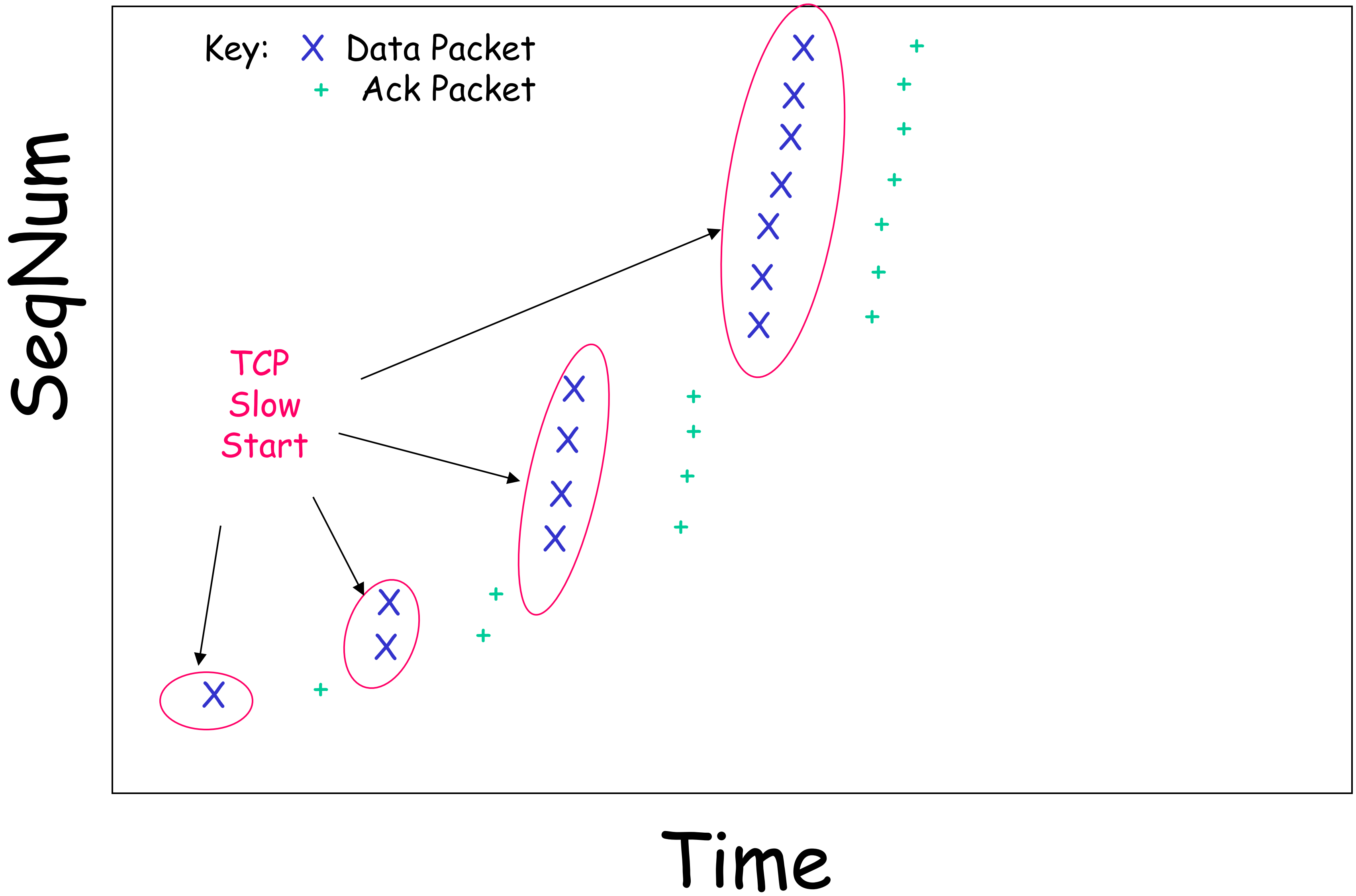


SeqNum
Key: X Data Packet
+ Ack Packet
Avg Throughput (Bytes/Sec)
Bytes
Sec
Time

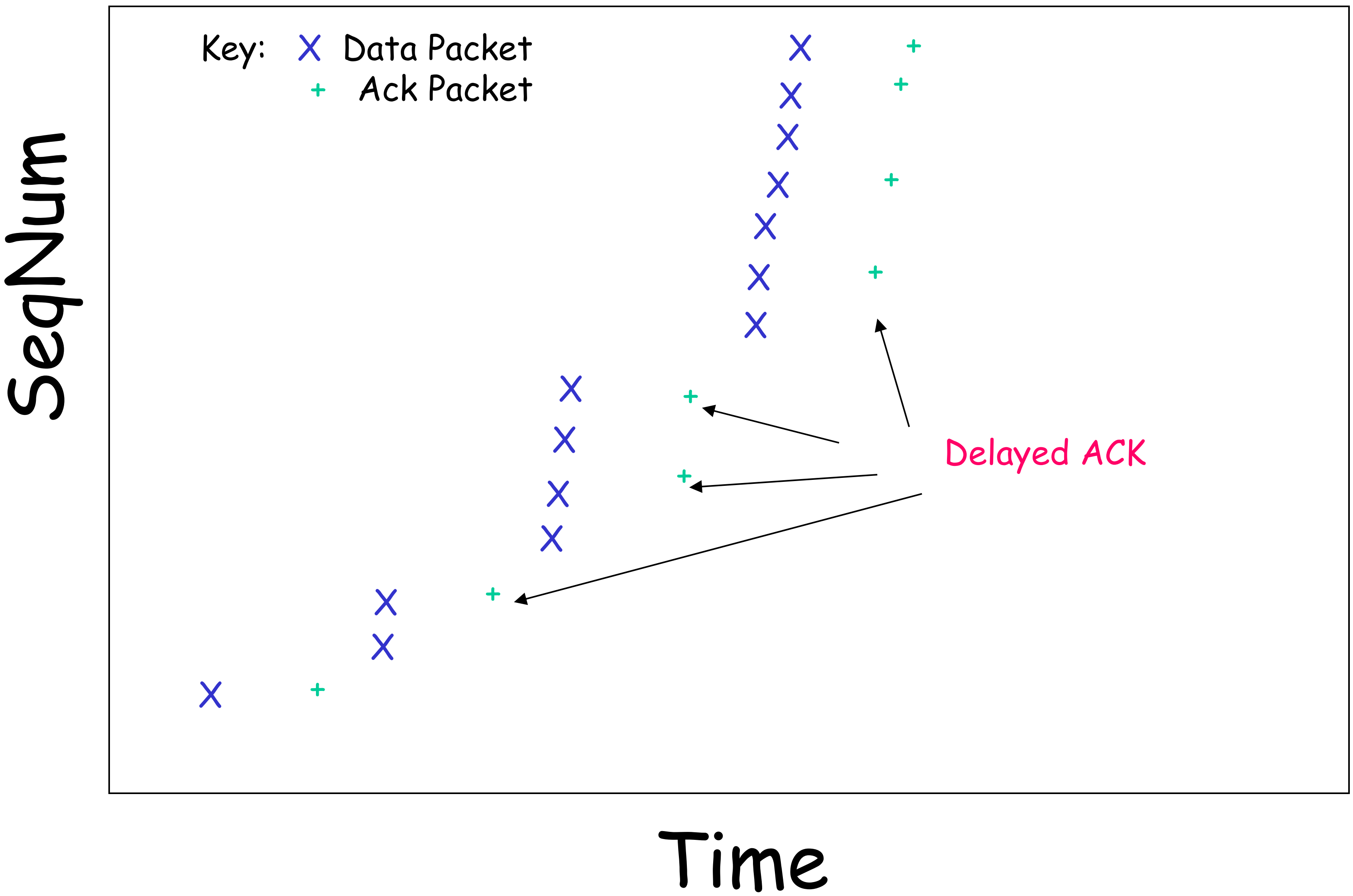


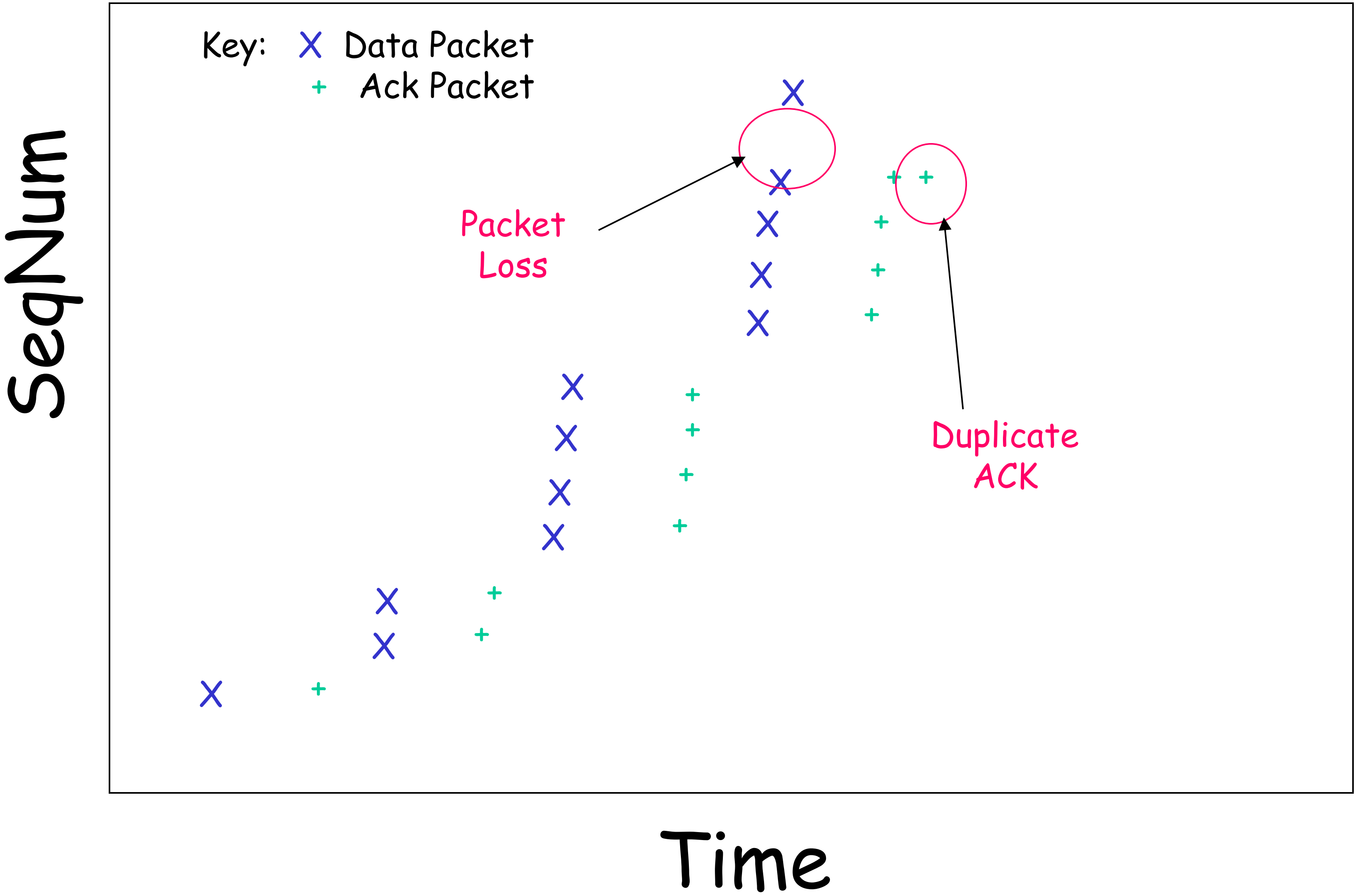
Key:
X Data Packet
+ Ack Packet
Access Network Bandwidth (Bytes/Sec)
SeqNum
Time

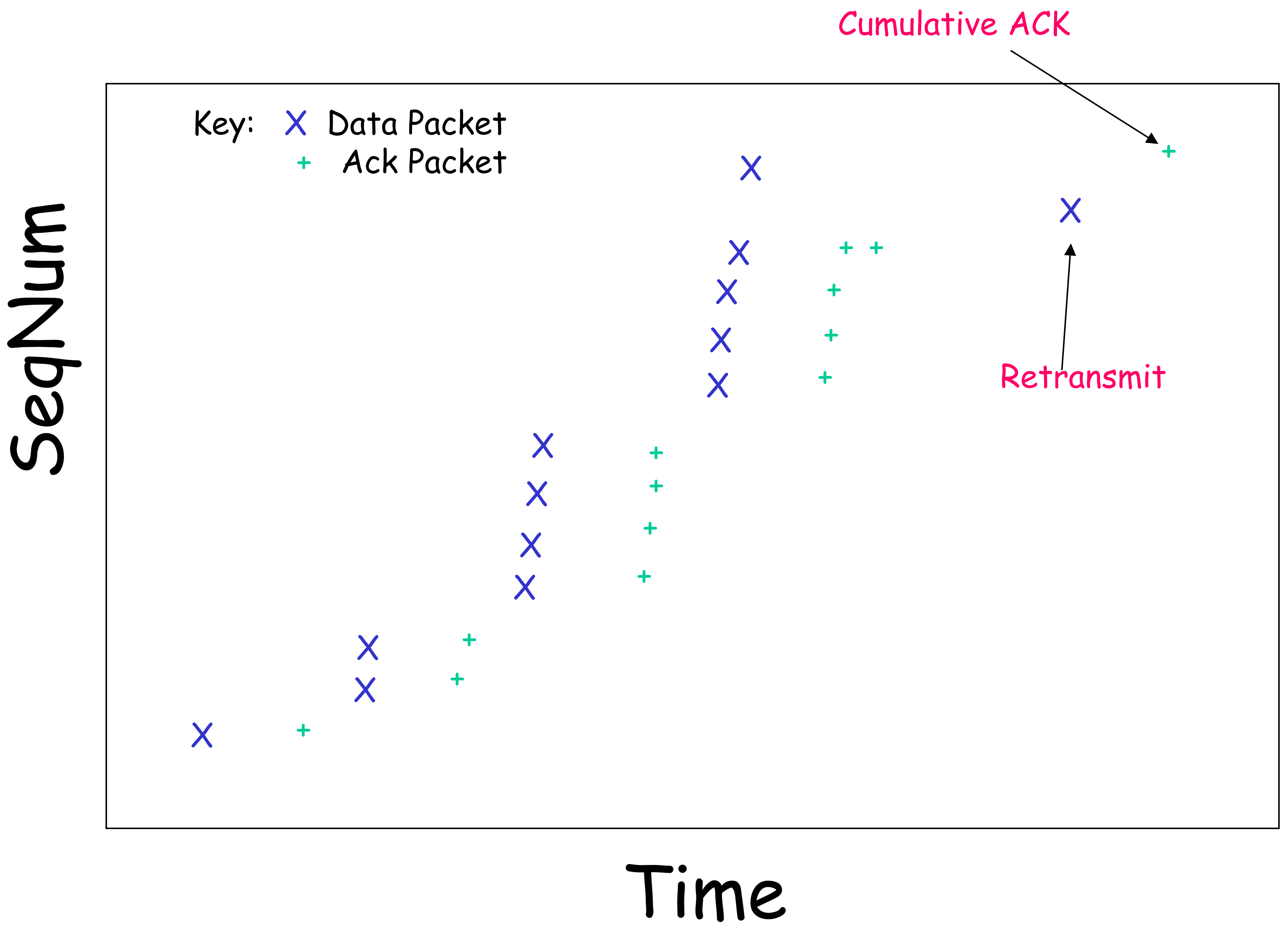

Key:
Data Packet
Ack Packet
SeqNum
Sender's
Flow Control
Window Size
Time



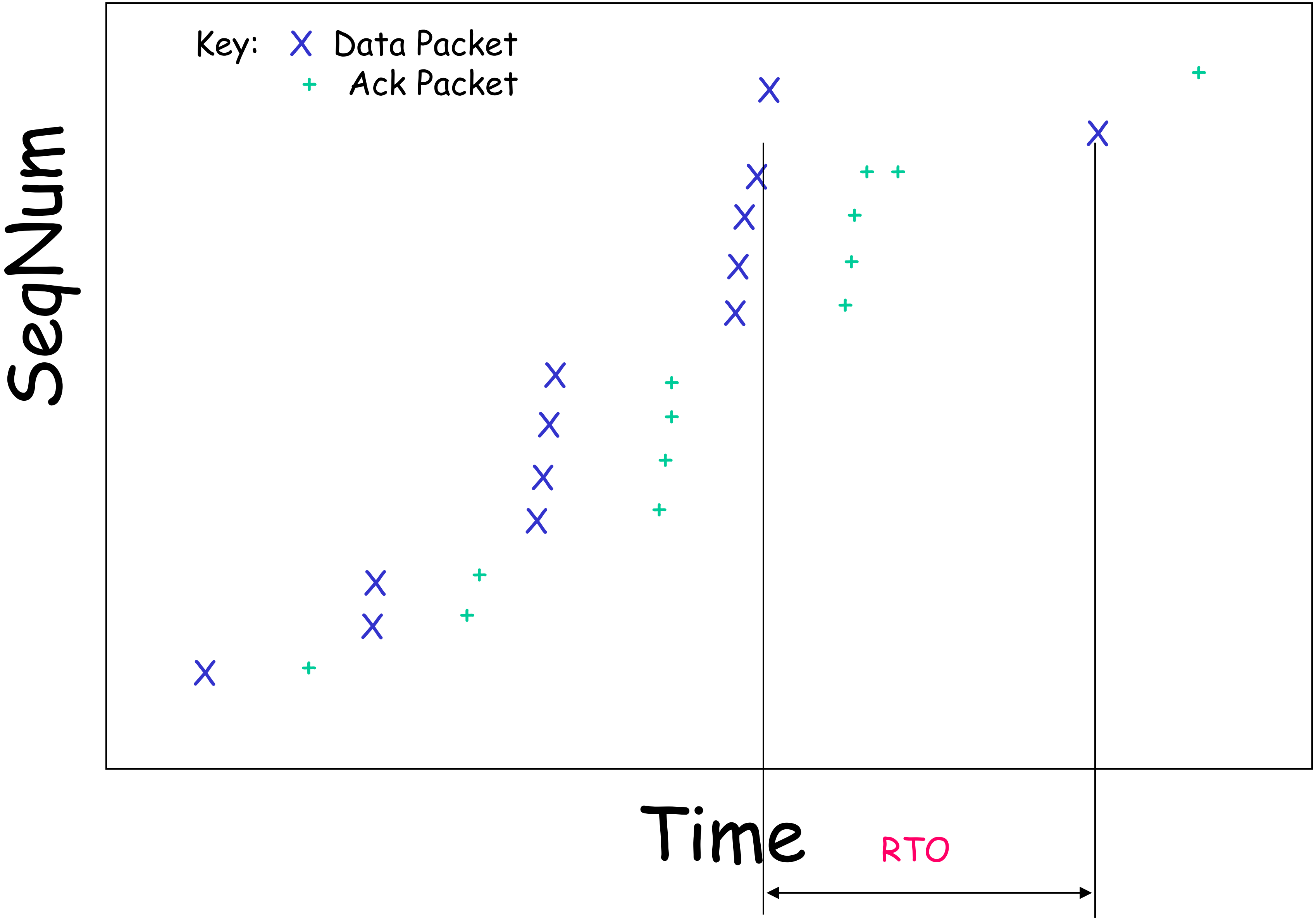
Key: X Data Packet
+ Ack Packet
TCP
Slow
Start
SeqNum
Time


Key:
Data Packet
Ack Packet
Delayed ACK
SeqNum
Time


Key: X Data Packet
+ Ack Packet
Packet Loss
Duplicate ACK
SeqNum
Time

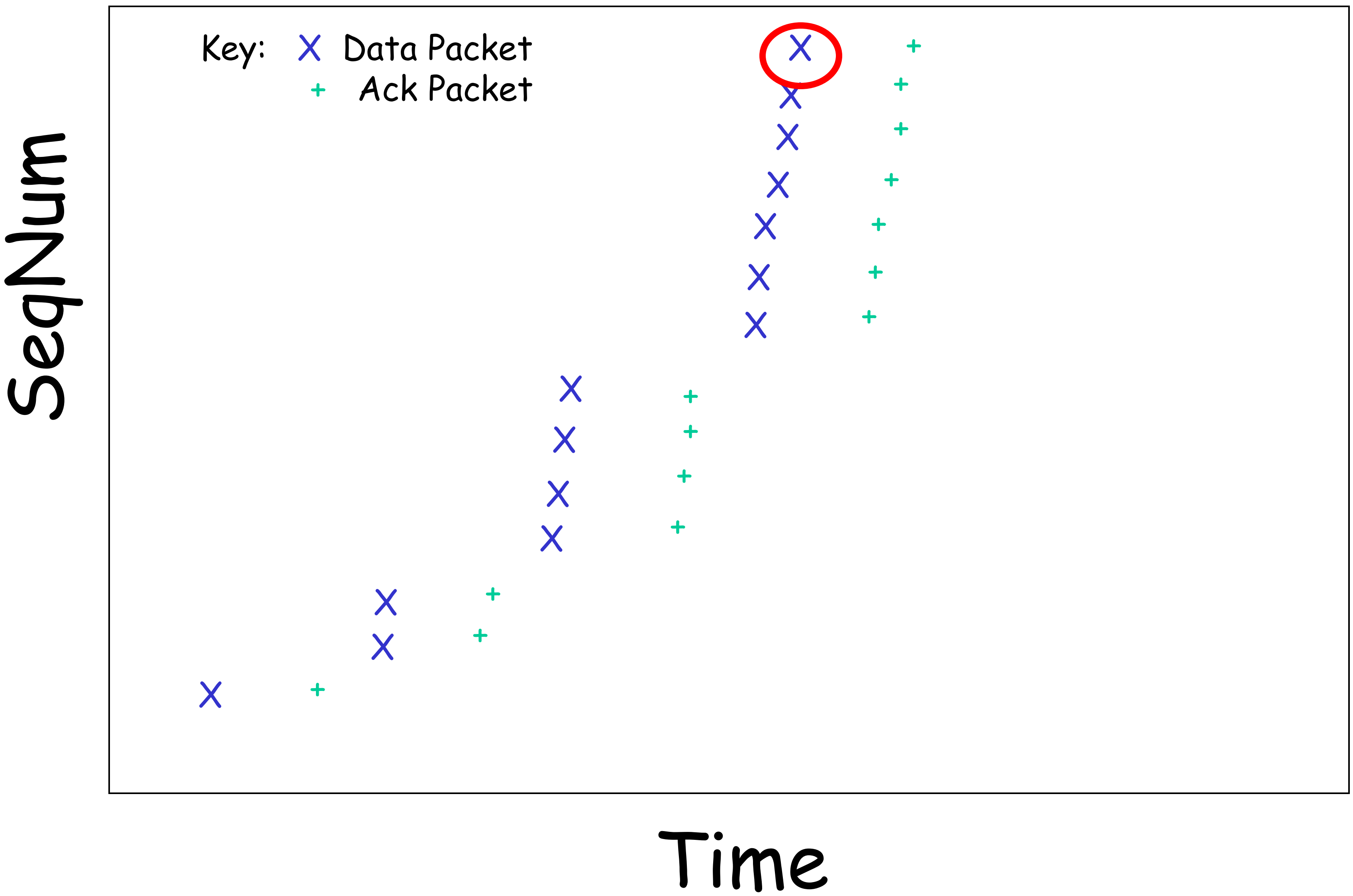


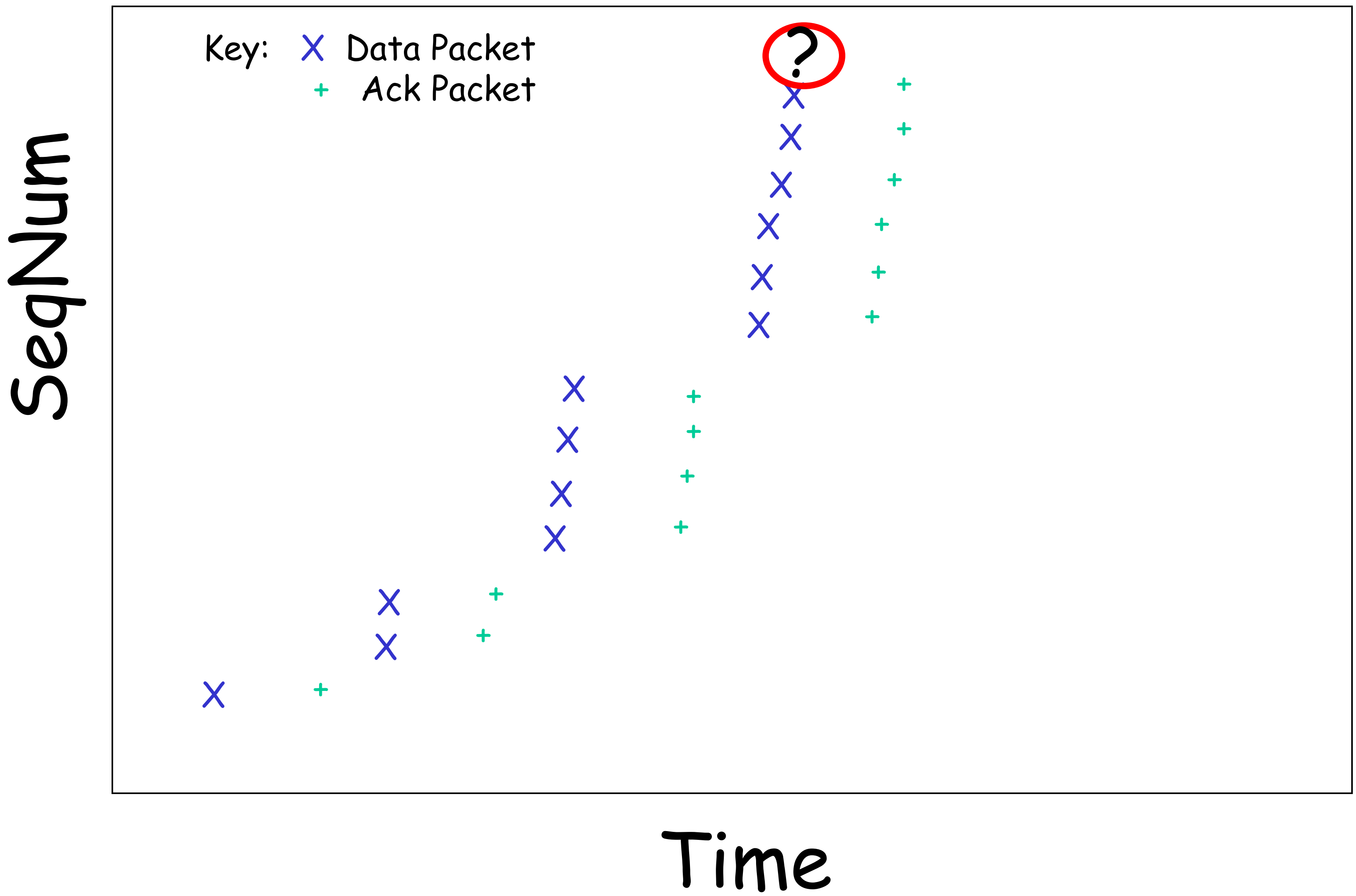
Cumulative ACK
Key: X Data Packet
+ Ack Packet
Retransmit
SeqNum
Time


Key:
Data Packet
Ack Packet
SeqNum
Time
RTO

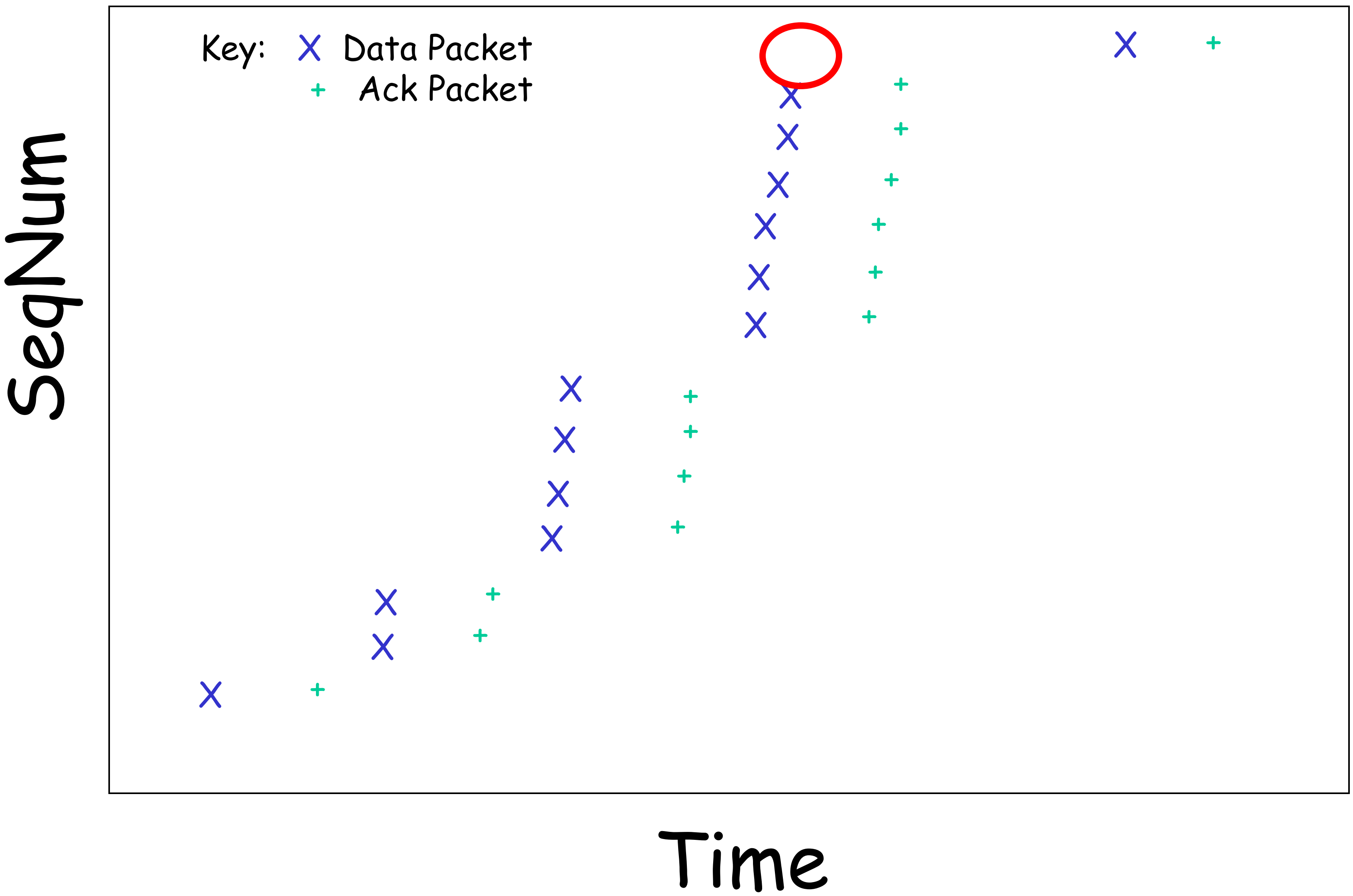
TCP 301 (Cont'd)

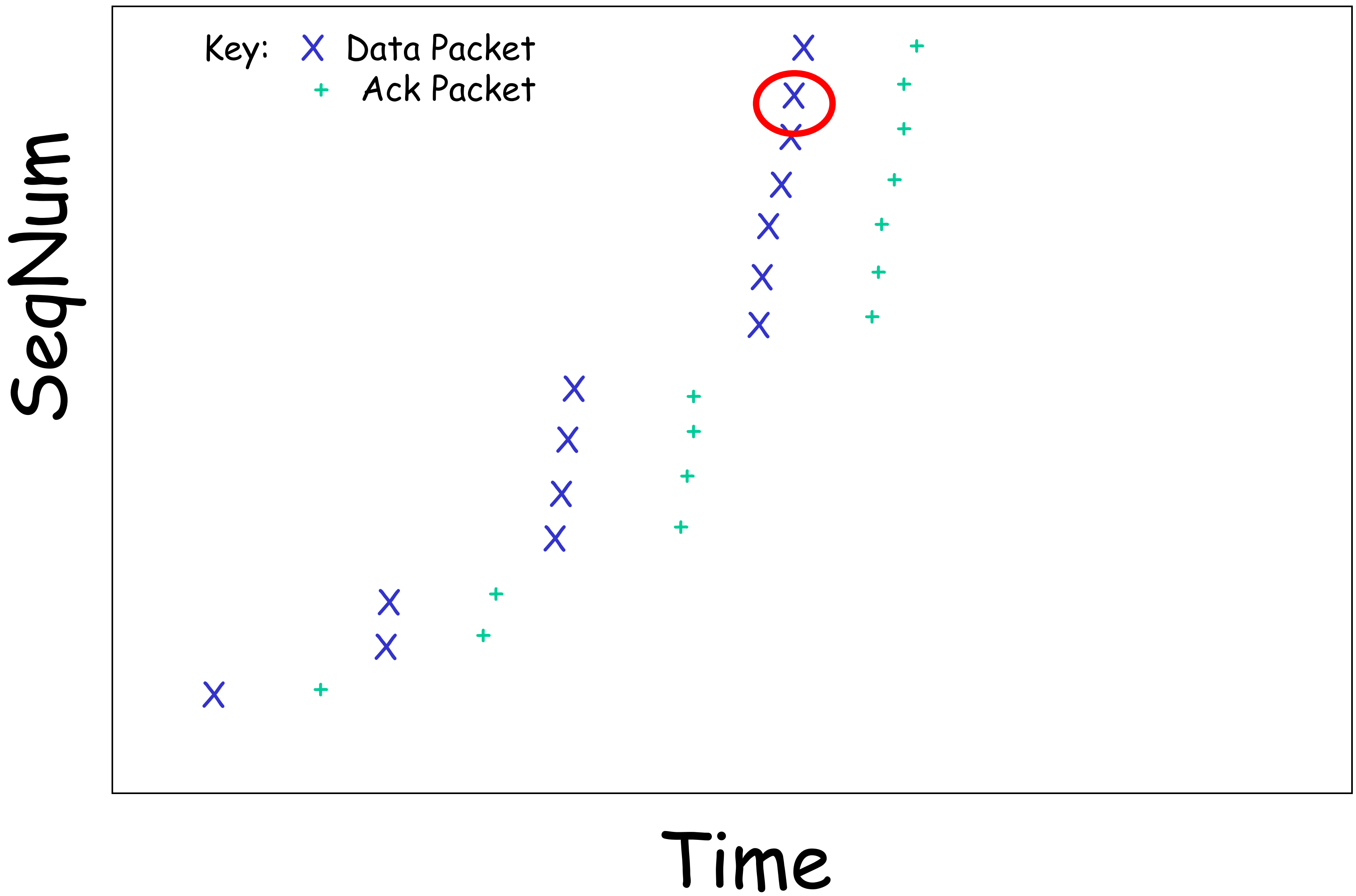
- What happens when a packet loss occurs?
- Quiz Time...
  - Consider a 14-packet Web document
  - For simplicity, consider only a single packet loss


Key:
Data Packet
Ack Packet
SeqNum
Time

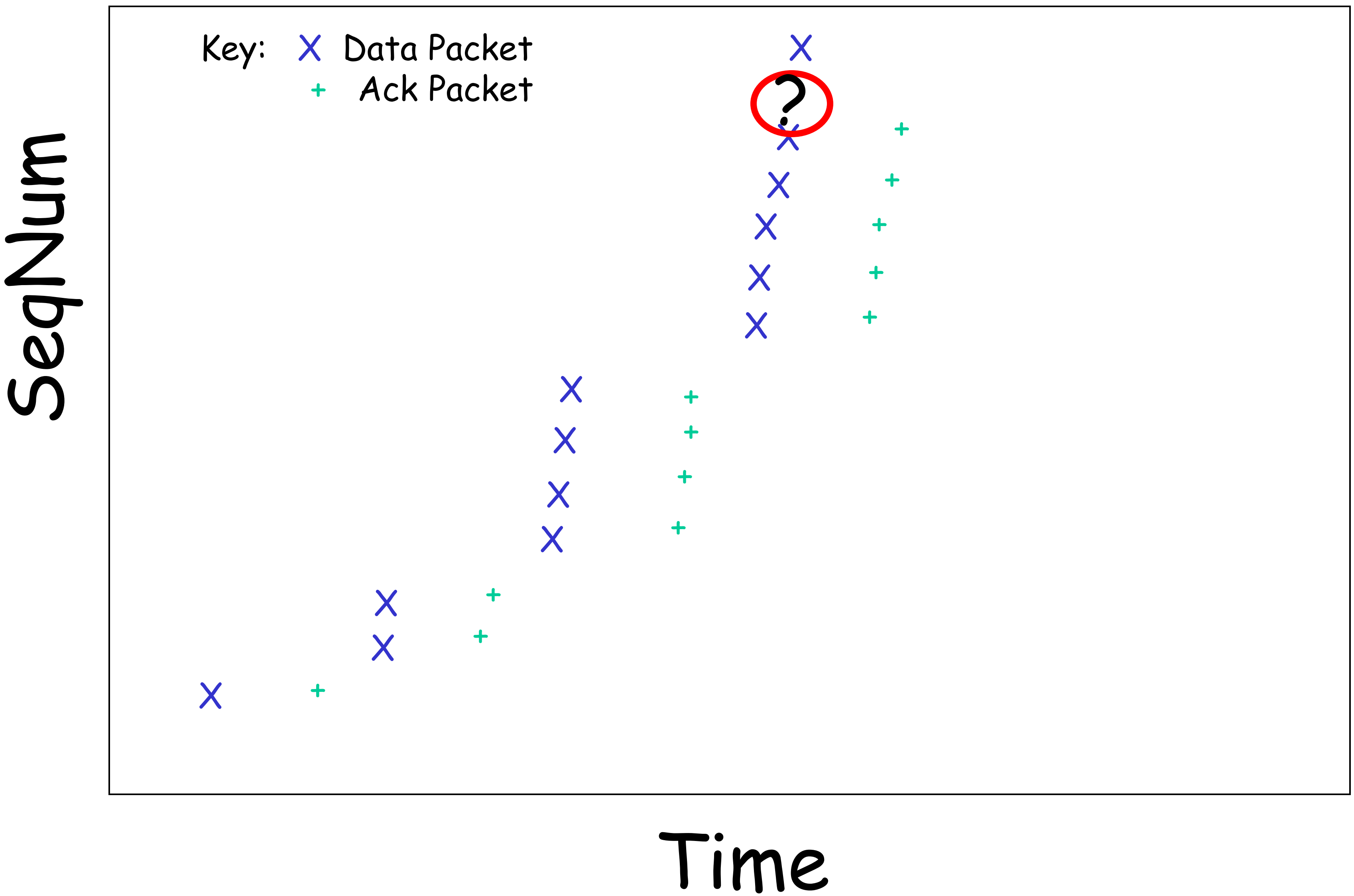


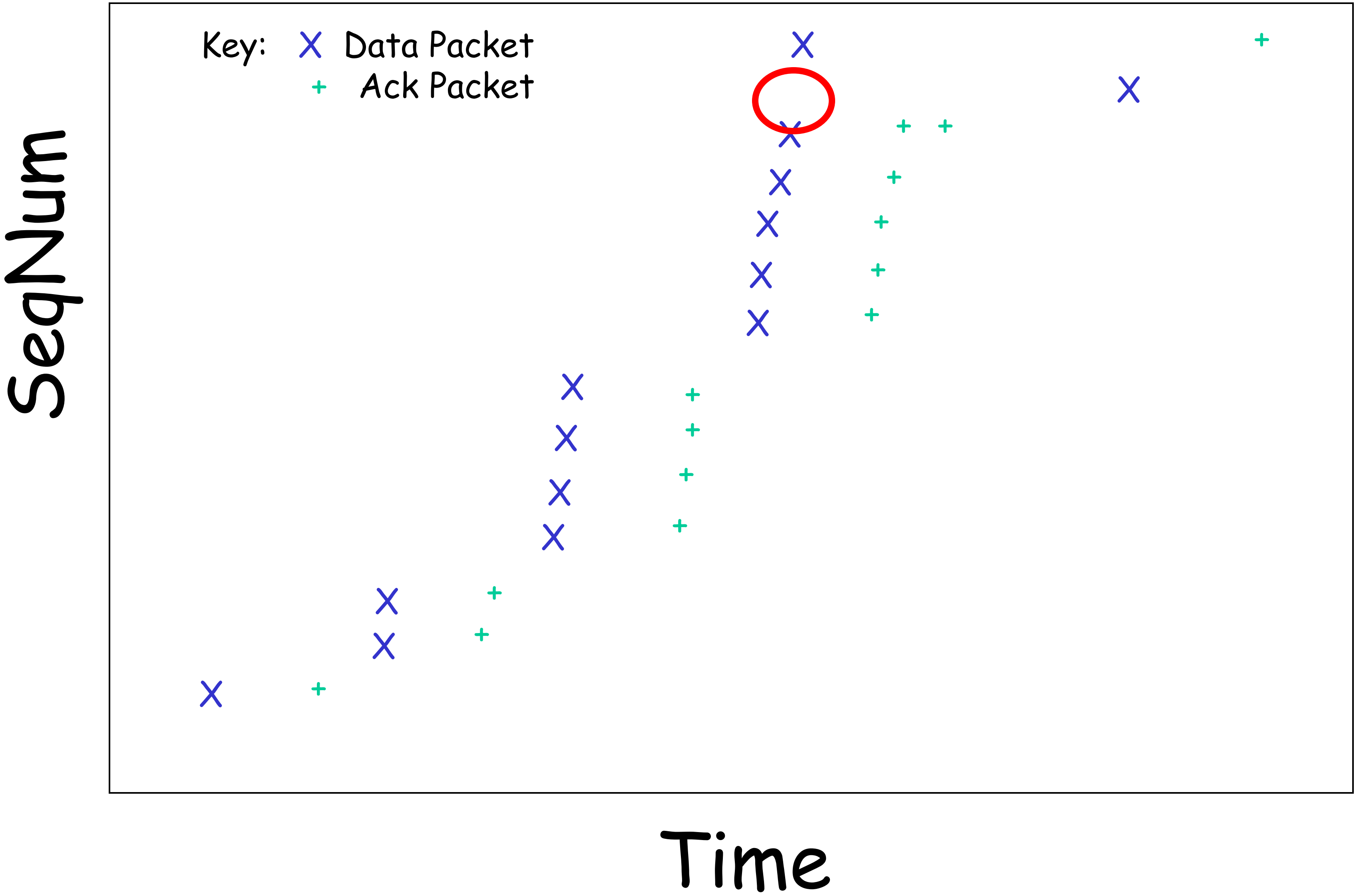
Key:
X Data Packet
+ Ack Packet
?
SeqNum
Time

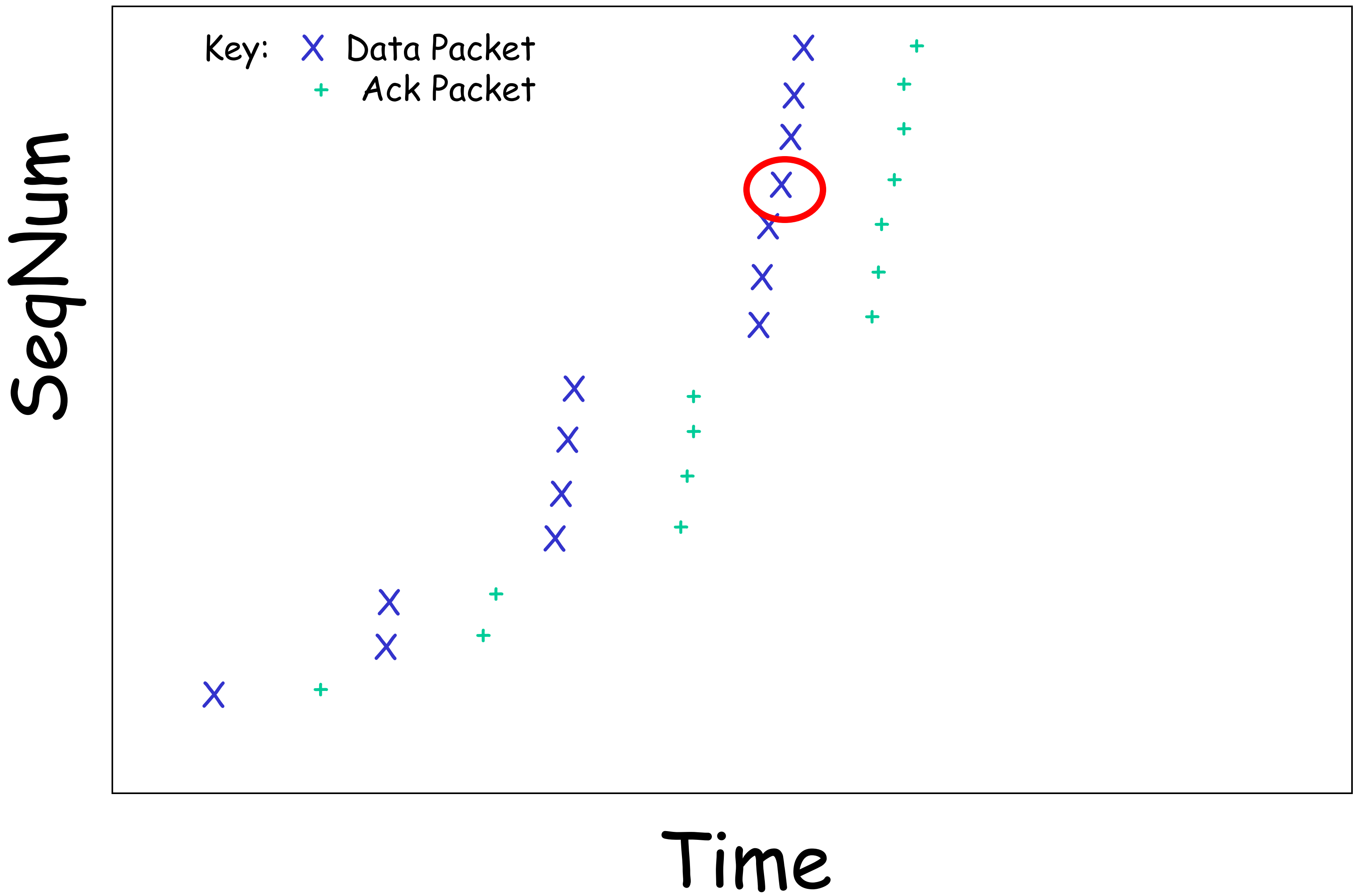

Key:
Data Packet
Ack Packet
SeqNum
Time

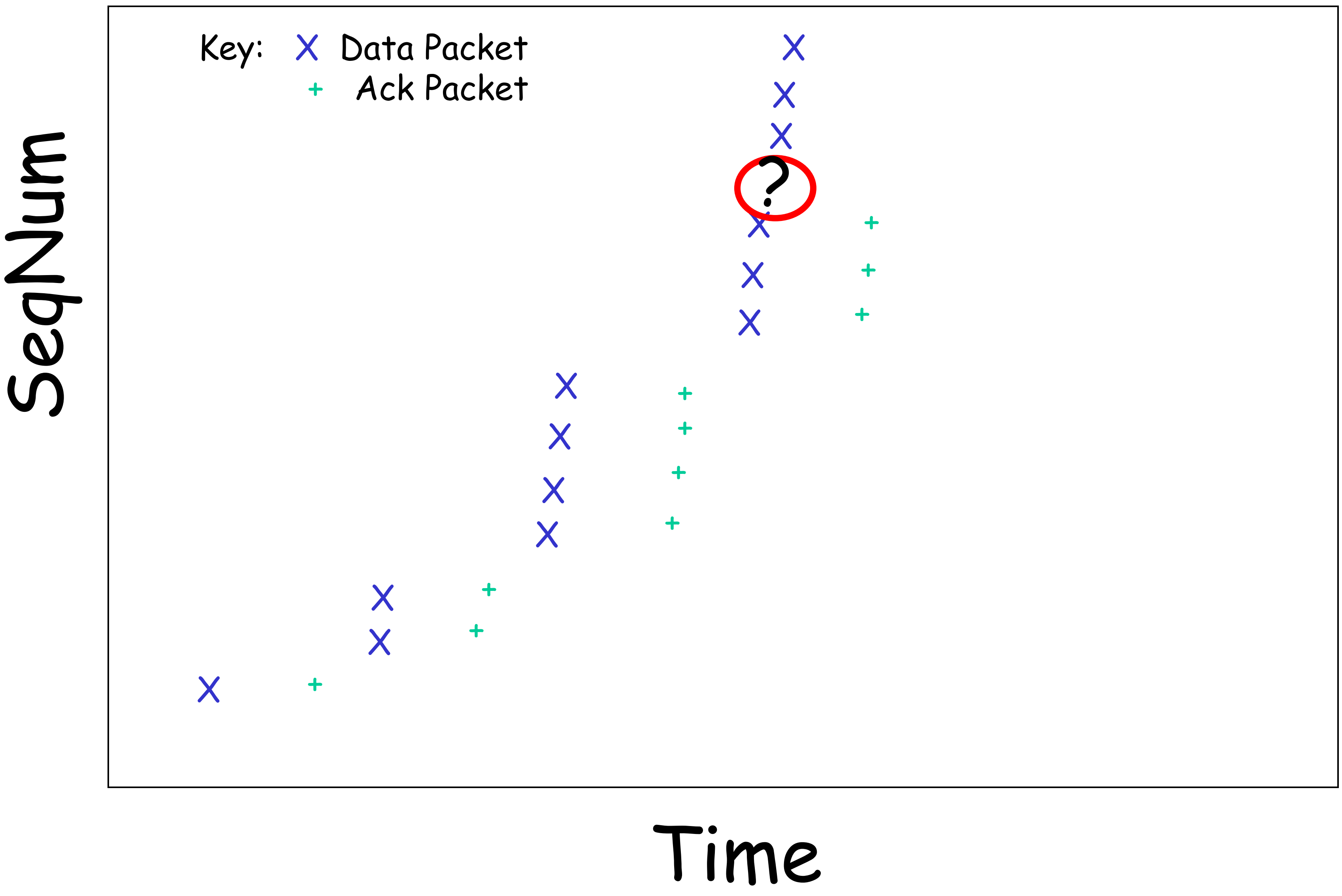


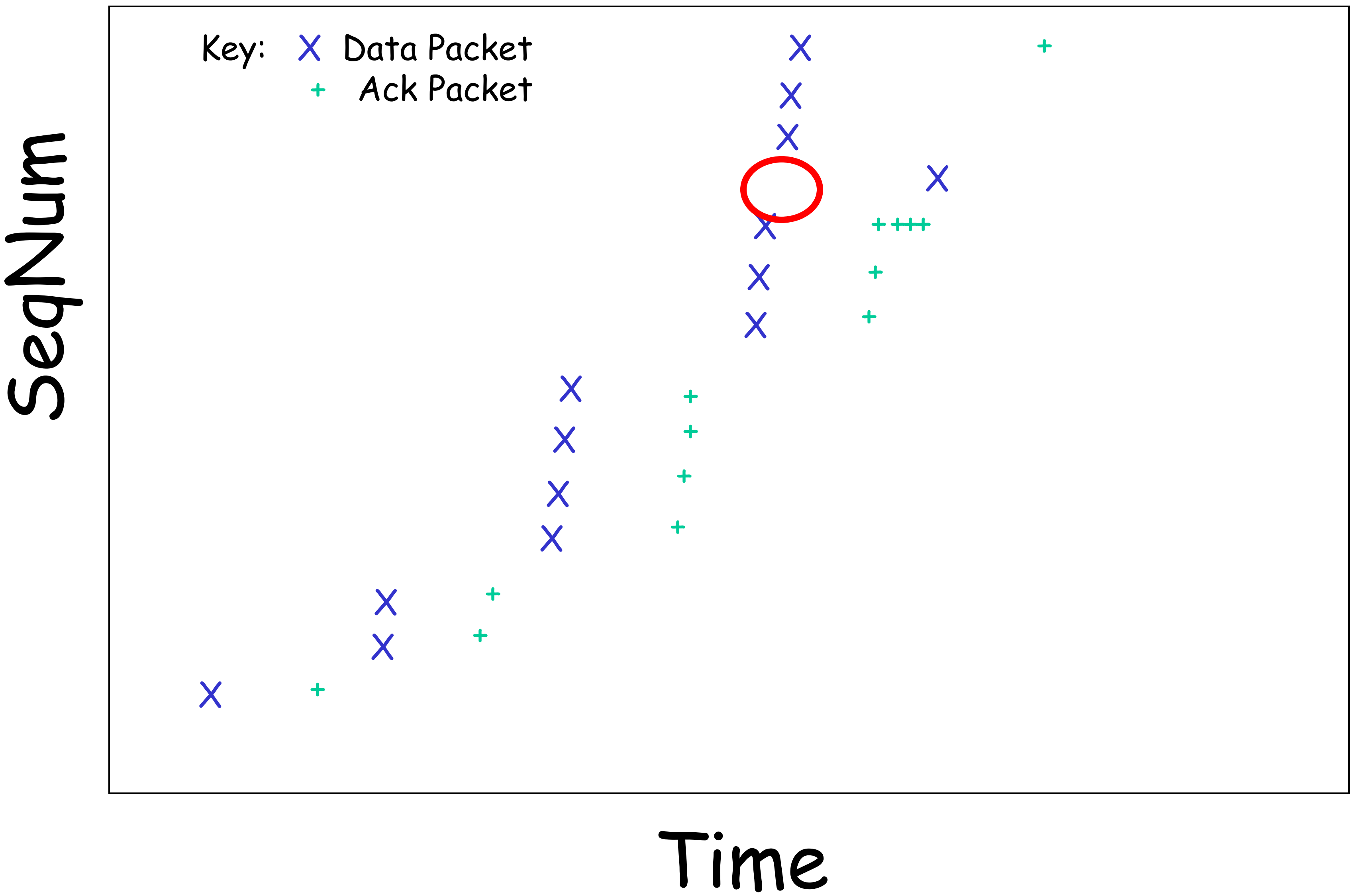
Key:
X Data Packet
+ Ack Packet
SeqNum
Time

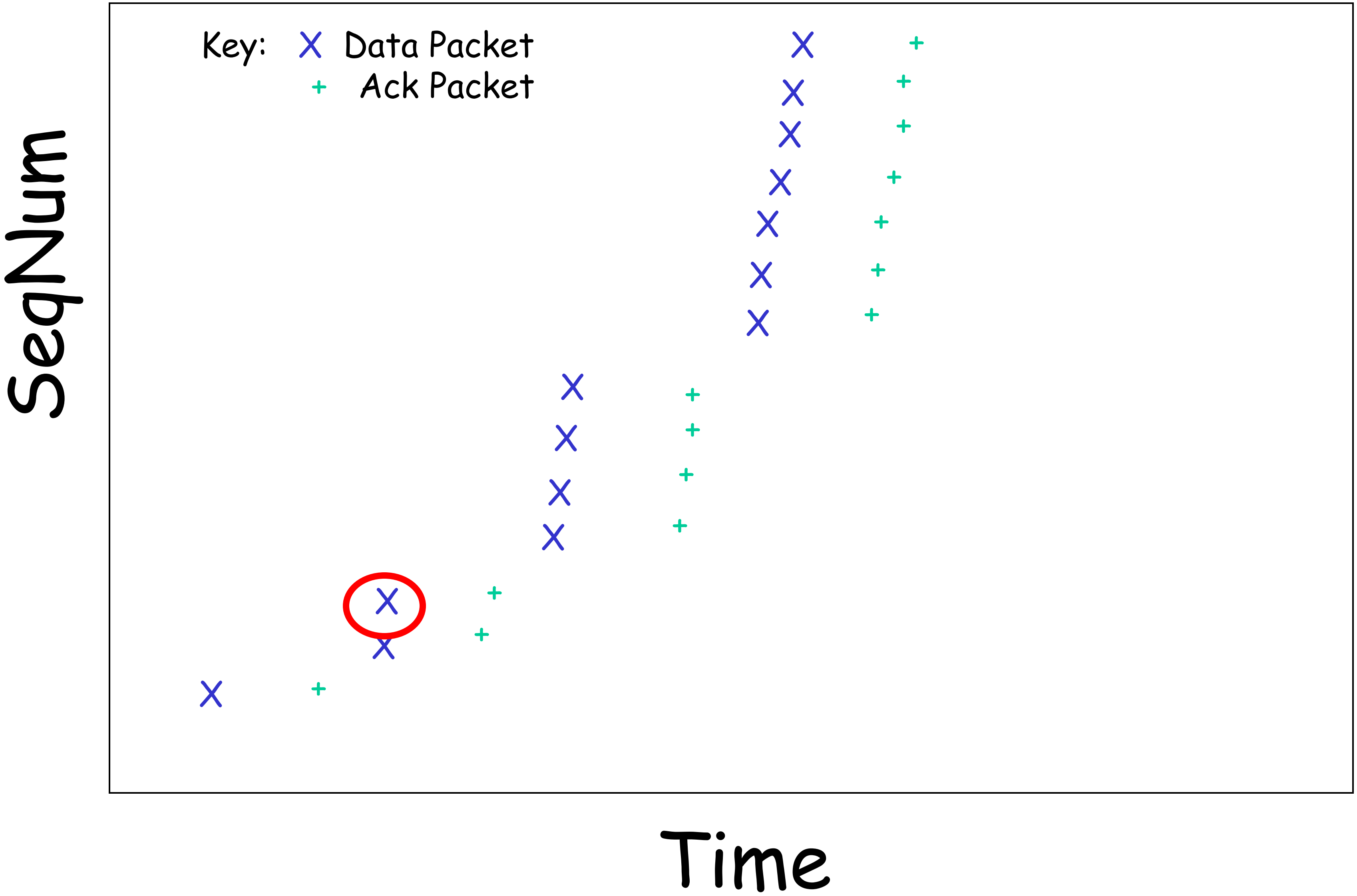

Key: X Data Packet
+ Ack Packet
?
SeqNum
Time

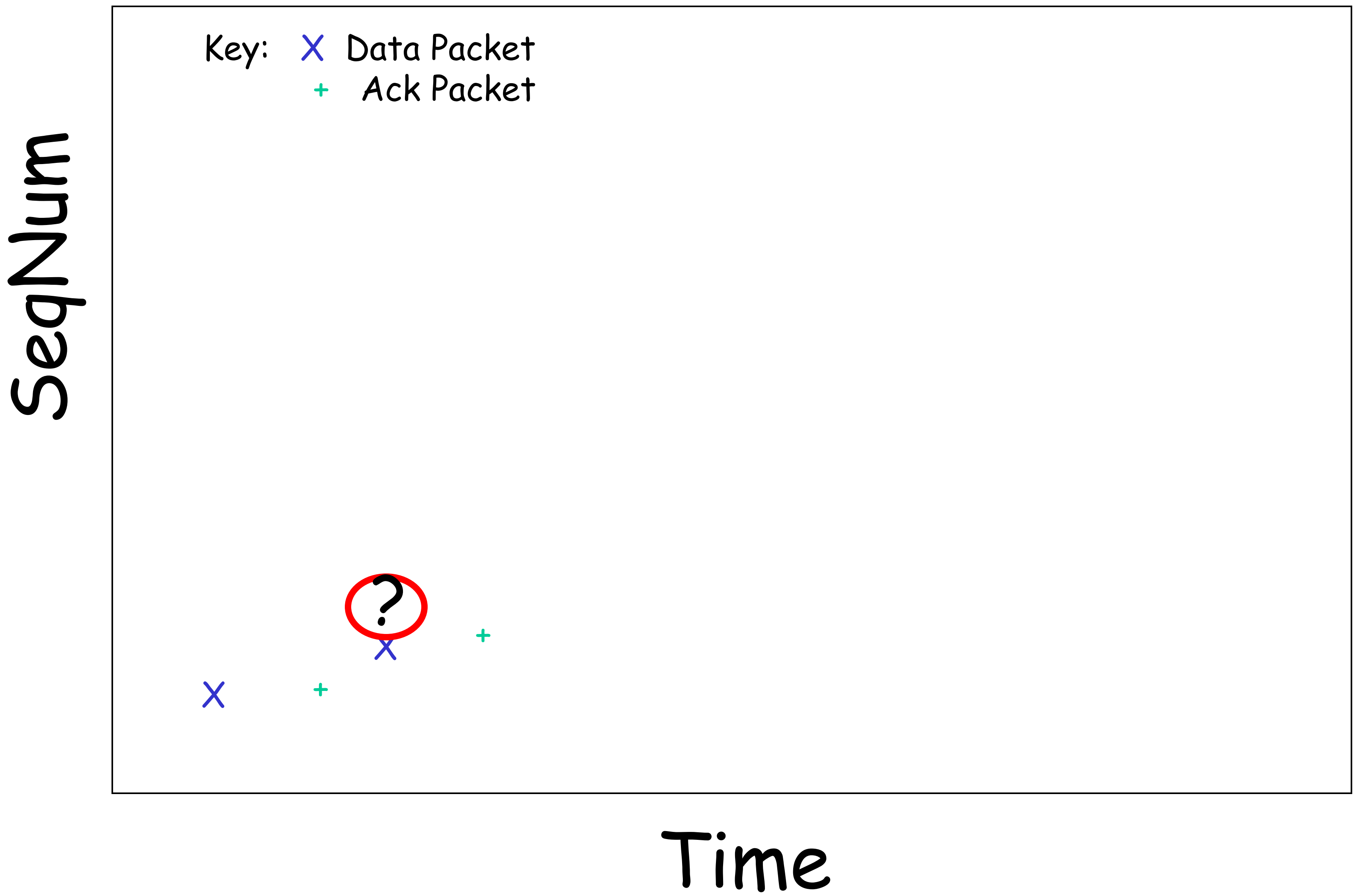

Key: X Data Packet
+ Ack Packet
SeqNum
Time


Key:
X Data Packet
+ Ack Packet
SeqNum
Time

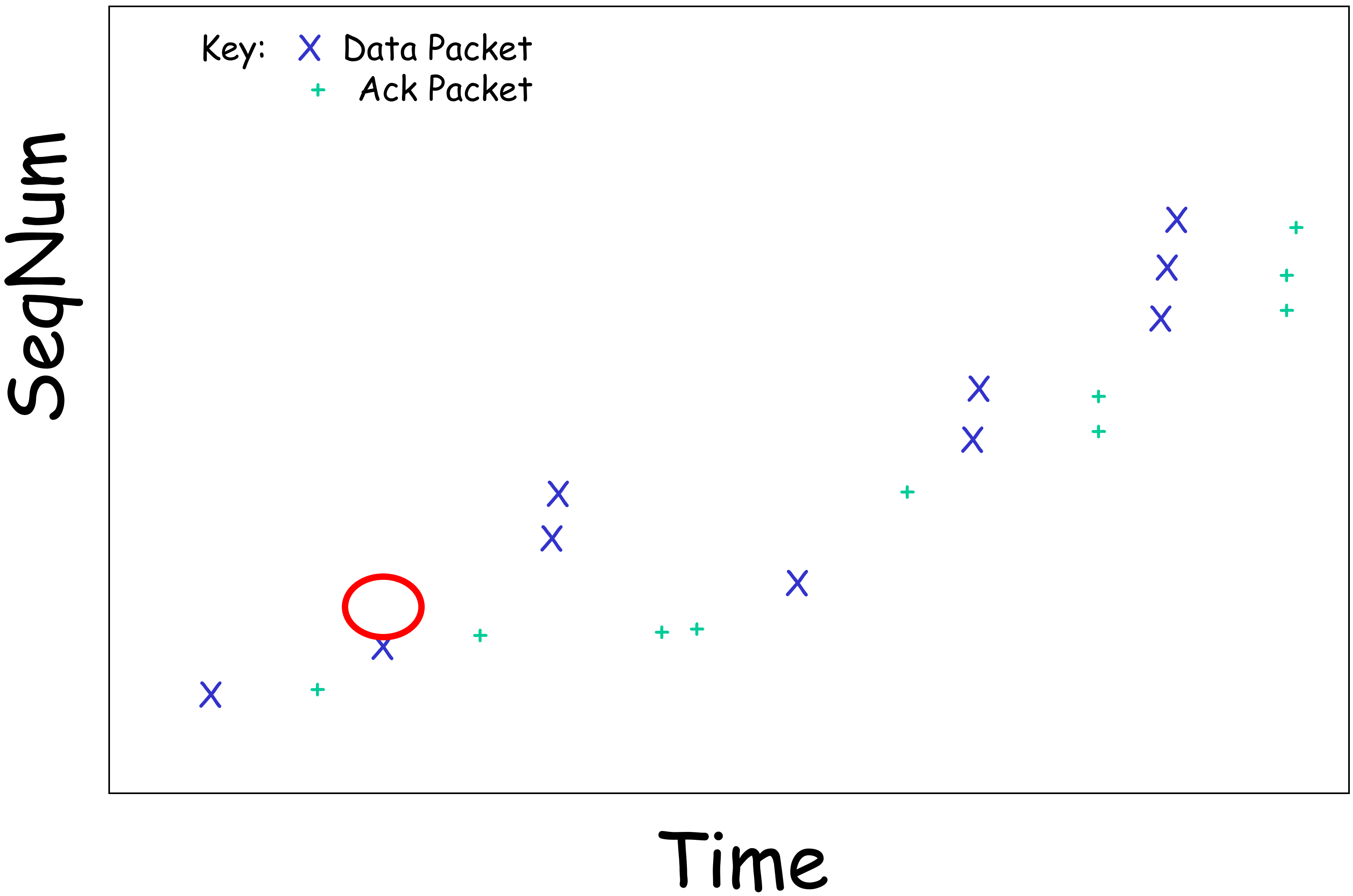

Key: X Data Packet
+ Ack Packet
?
SeqNum
Time


Key:
X Data Packet
+ Ack Packet
SeqNum
Time


Key: X Data Packet
+ Ack Packet
SeqNum
Time



Key:
X Data Packet
+ Ack Packet
?
SeqNum
Time



Key: X Data Packet
+ Ack Packet
SeqNum
Time

TCP 301 (Cont'd)

- Main observation:
  - "Not all packet losses are created equal"
- Losses early in the transfer have a huge adverse impact on the transfer latency
- Losses near the end of the transfer always cost at least a retransmit timeout
- Losses in the middle may or may not hurt, depending on congestion window size at the time of the loss

Congratulations!

- You are now a TCP expert!