CS162
Operating Systems and Systems Programming
Lecture 17
Reliability, TCP, Flow Control

March 21, 2012

Anthony D. Joseph and Ion Stoica

http://inst.eecs.berkeley.edu/~cs162

Placing Network Functionality

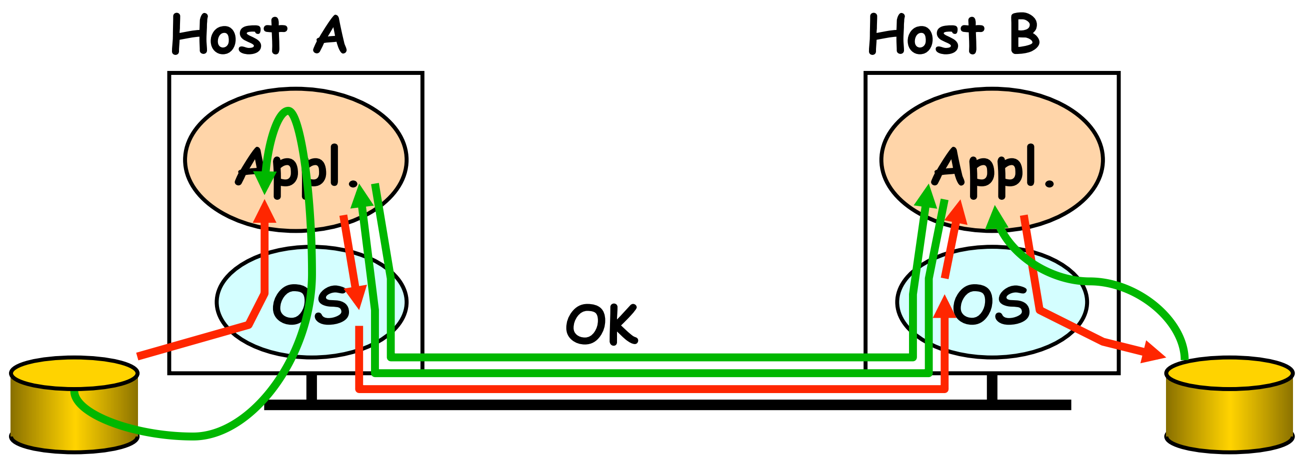
- Hugely influential paper: "End-to-End Arguments in System Design" by Saltzer, Reed, and Clark ('84)

- "Sacred Text" of the Internet
  – Endless disputes about what it means
  – Everyone cites it as supporting their position

Basic Observation

- Some types of network functionality can only be correctly implemented end-to-end
  - Reliability, security, etc
- Because of this, end hosts:
  - Can satisfy the requirement without network's help
  - Will/**must** do so, since can't ***rely*** on network's help
- Therefore **don't** go out of your way to implement them in the network
- Note: By "network" here we mean **network layer**

Example: Reliable File Transfer



- Solution 1: make each step reliable, and then concatenate them
- Solution 2: end-to-end **check** and try again if necessary

Discussion

- Solution 1 is incomplete
  - What happens if memory is corrupted?
  - Receiver has to do the check anyway!
- Solution 2 is complete
  - Full functionality can be entirely implemented at application layer with no need for reliability from lower layers
- *Is there any need to implement reliability at lower layers?*
  - Well, it could be more efficient

End-to-End Principle

Implementing this functionality in the network:

- Doesn't reduce host implementation complexity
- Does increase network complexity
- Probably imposes delay and overhead on all applications, even if they don't need functionality

- However, implementing in network can enhance performance in some cases
  - E.g., very lossy link

Conservative Interpretation of E2E

- Don't implement a function at the lower levels of the system unless it can be completely implemented at this level

- Unless you can relieve the burden from hosts, don't bother

Moderate Interpretation

- Think twice before implementing functionality in the network
- If hosts can implement functionality correctly, implement it in a lower layer only as a performance enhancement
- But do so only if it does not impose burden on applications that do not require that functionality
- This is the interpretation we are using

Summary

- Layered architecture powerful abstraction for organizing complex networks
- Internet: 5 layers
  - Physical: send bits
  - Datalink: Connect two hosts on same physical media
  - Network: Connect two hosts in a wide area network
  - Transport: Connect two processes on (remote) hosts
  - Applications: Enable applications running on remote hosts to interact
- Narrow waist: only one network layer in the Internet
  - Enables the higher layer (Transport and Applications) and lower layers (Datalink and Physical) to evolve indpendently

Summary

- E2E argument encourages us to keep IP simple
- If higher layer can implement functionality correctly, implement it in a lower layer only if
  - it improves the performance significantly for application that need that functionality, and
  - it does not impose burden on applications that do not require that functionality

Goals for Today

- Reliable Transfer & flow control
- TCP
  - Open connection (3-way handshake)
  - Tear-down connection
  - Flow control

Reliable Transfer

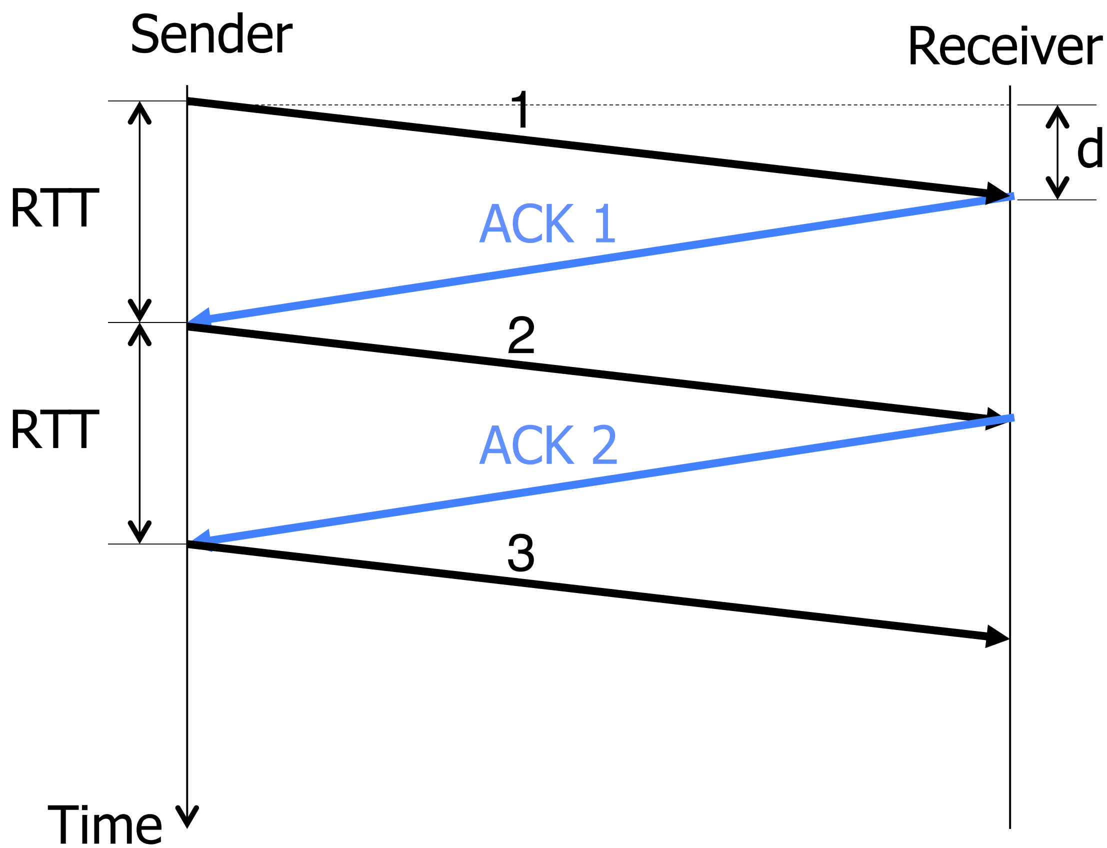
- Retransmit missing packets
  – Numbering of packets and ACKs
- Do this efficiently
  – Keep transmitting whenever possible
  – Detect missing packets and retransmit quickly
- Two schemes
  – Stop & Wait
  – Sliding Window (Go-back-n and Selective Repeat)

Detecting Packet Loss?

- Timeouts
  – Sender timeouts on not receiving ACK
- Missing ACKs
  – Sender ACKs each packet
  – Receiver detects a missing packet when seeing a gap in the sequence of ACKs
  – Need to be careful! Packets and acks might be reordered
- NACK: Negative ACK
  – Receiver sends a NACK specifying a packet its missing

Stop & Wait w/o Errors

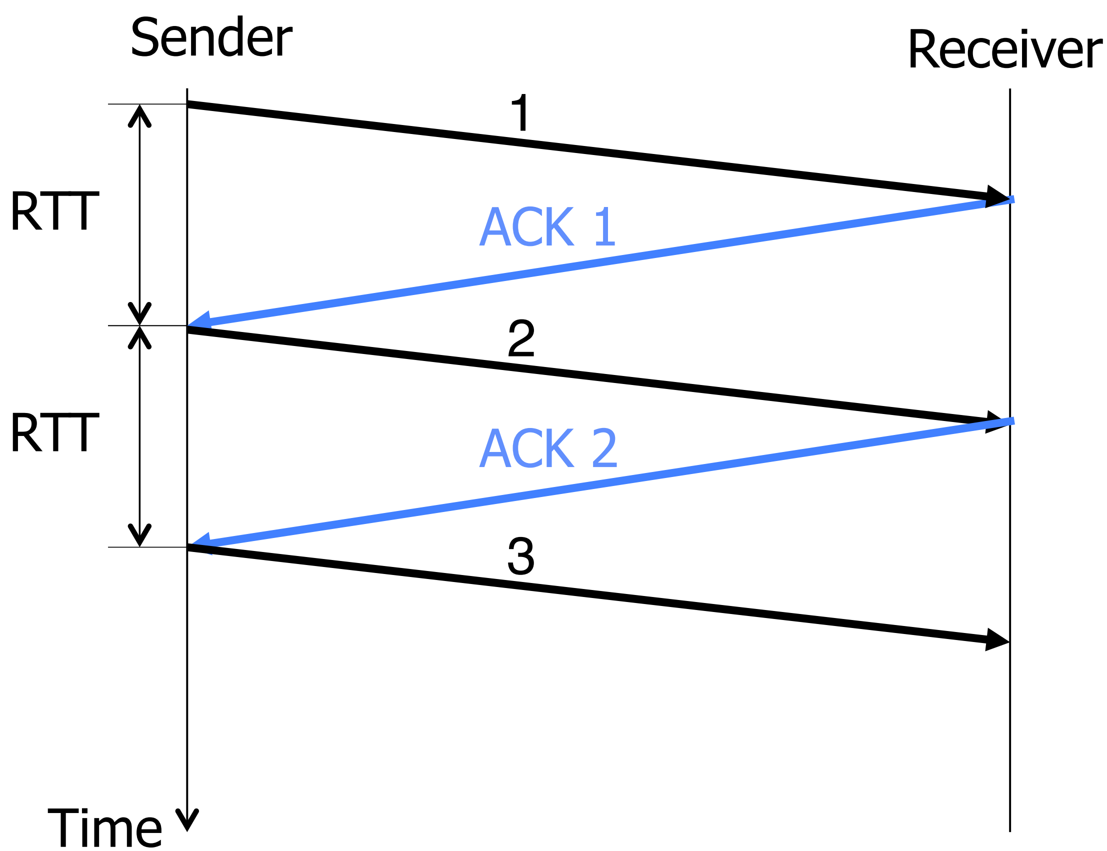
- Send; wait for ack; repeat
- RTT: Round Trip Time (RTT): time it takes a packet to travel from sender to receiver and back
  - One-way latency (d): one way delay from sender and receiver



RTT = 2*d (if latency is symmetric)

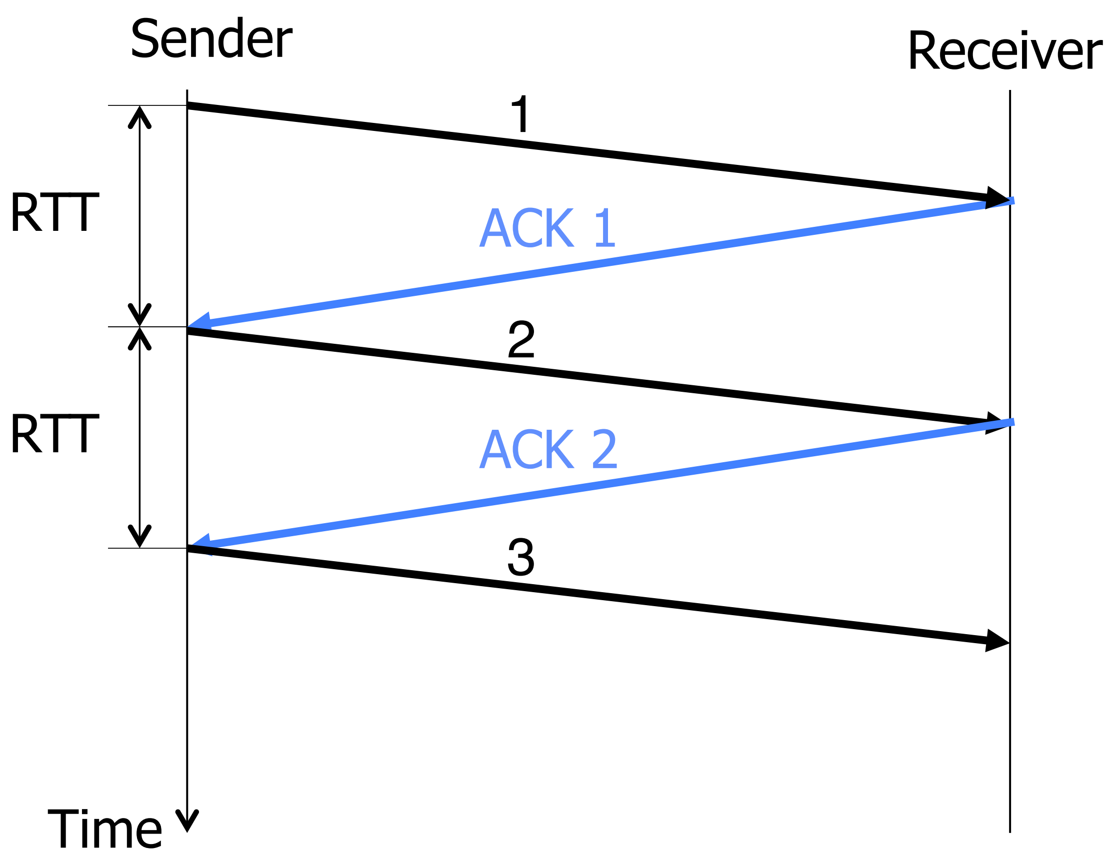
Stop & Wait w/o Errors

- How many packets can you send?
- 1 packet / RTT
- Throughput: number of bits delivered to receiver per sec



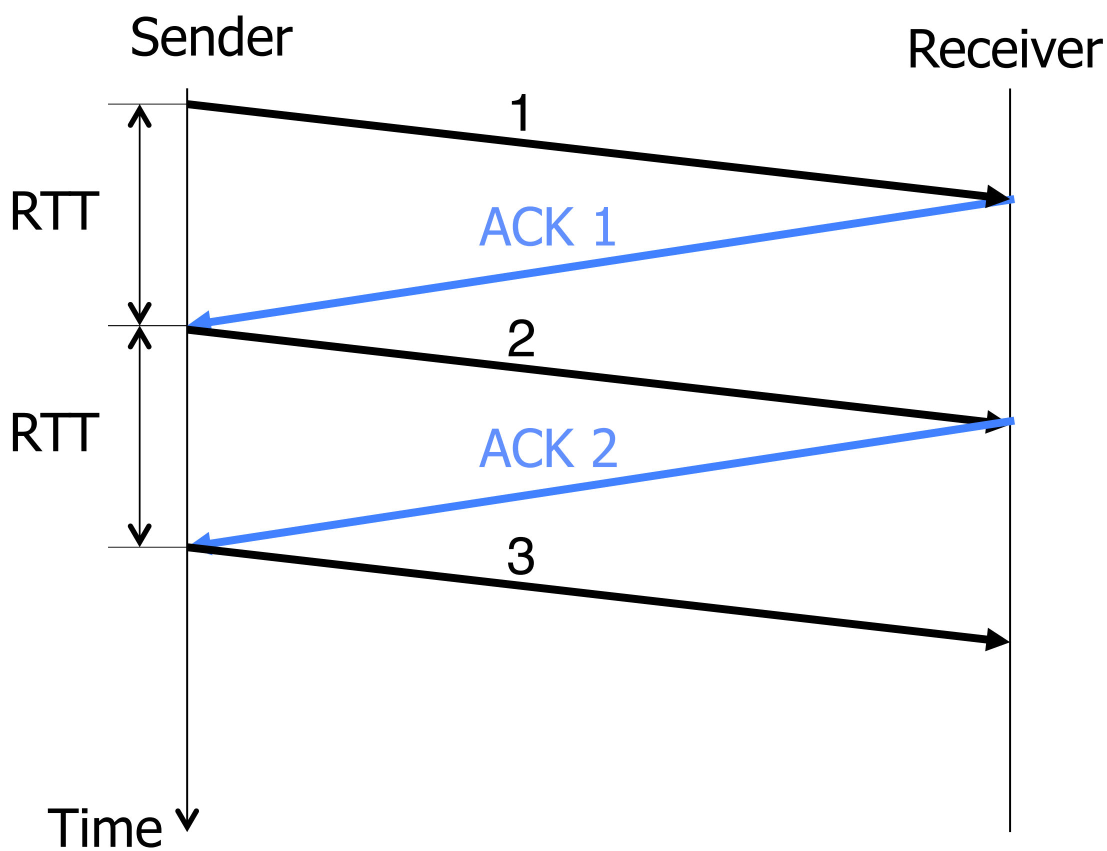
Stop & Wait w/o Errors

- Say, RTT = 100ms
- 1 packet = 1500 bytes
- Throughput = 1500*8bits/0.1s = 120 Kbps



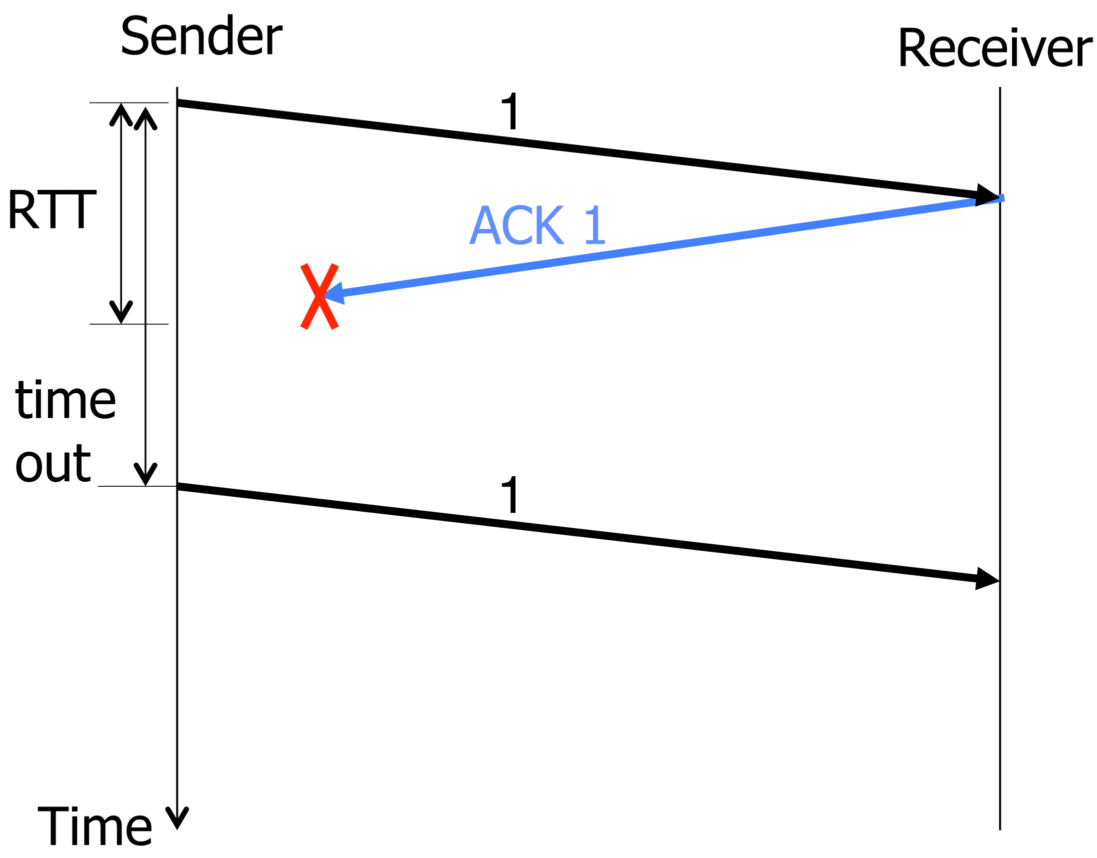
Stop & Wait w/o Errors

- Can be highly inefficient for high capacity links
- Throughput doesn't depend on the network capacity → even if capacity is 1Gbps, we can only send 120 Kbps!



Stop & Wait with Errors

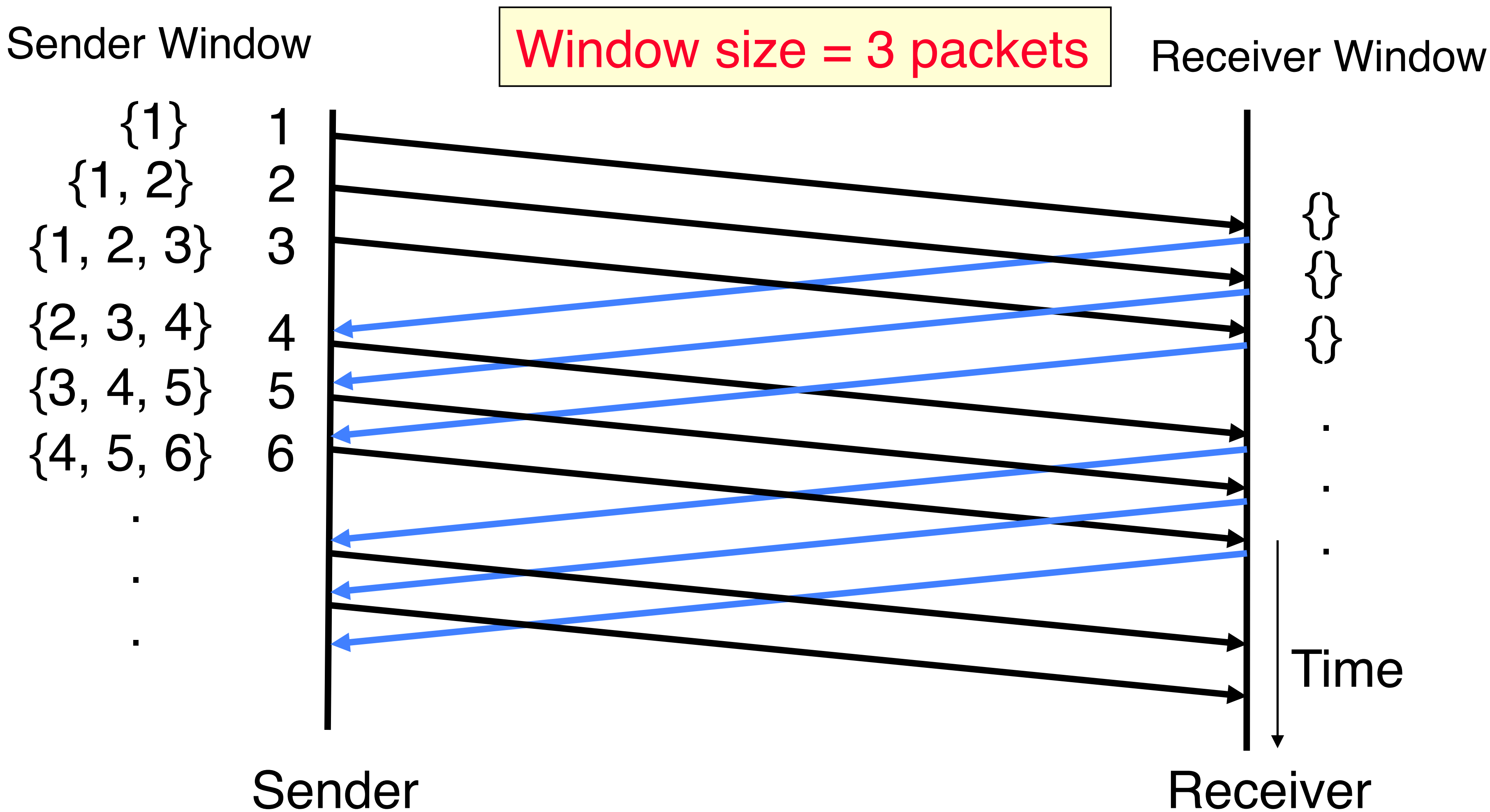
- If a loss wait for a retransmission timeout and retransmit
- Ho do you pick the timeout?



Sliding Window

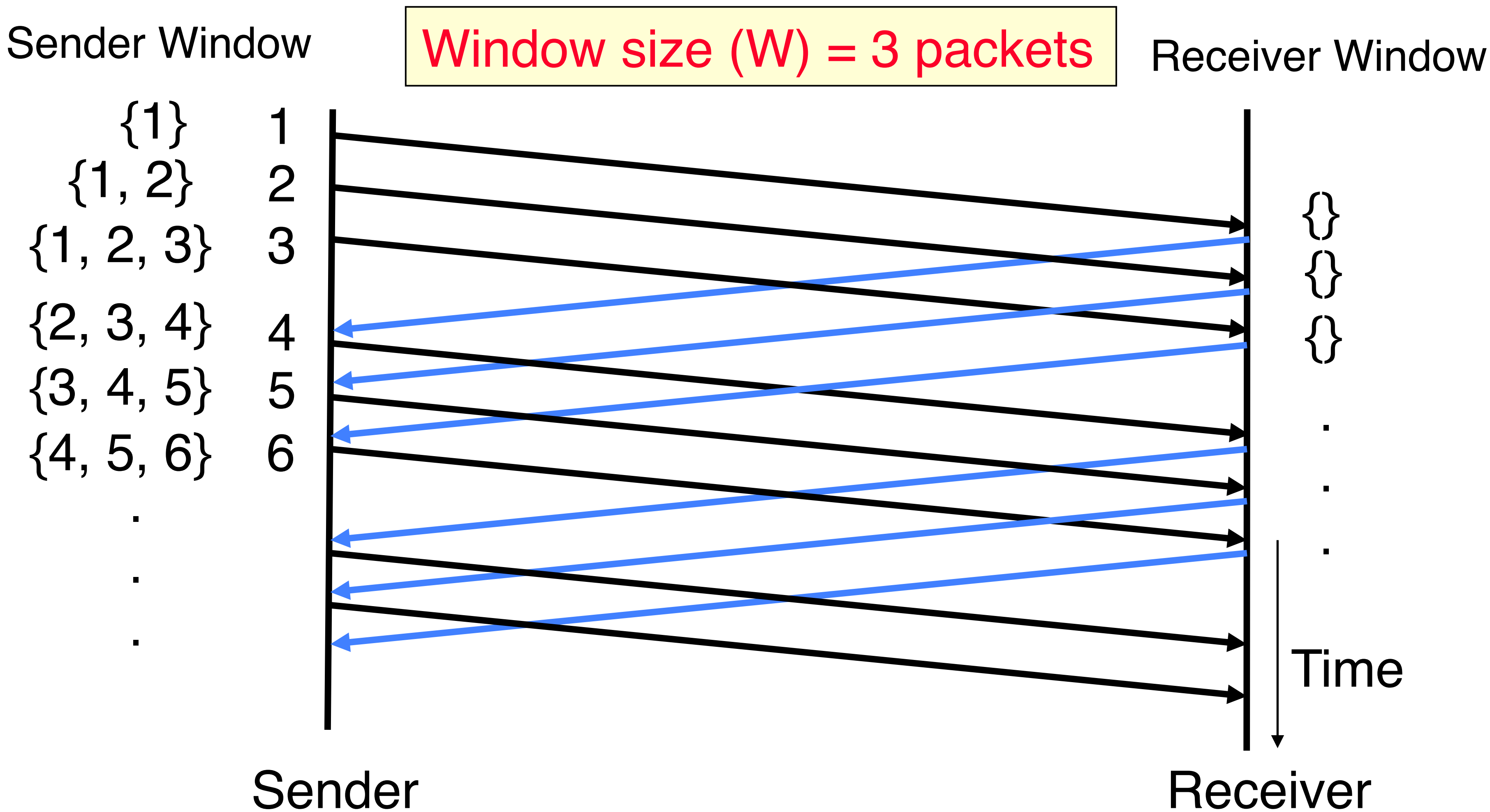
- *window* = set of adjacent sequence numbers
- The size of the set is the *window size*
- Assume window size is n
- Let A be the last ack'd packet of sender without gap; then window of sender = {A+1, A+2, …, A+n}
- Sender can send packets in its window
- Let B be the last received packet without gap by receiver, then window of receiver = {B+1,…, B+n}
- Receiver can accept out of sequence, if in window

Sliding Window w/o Errors



Sliding Window w/o Errors

- Throughput = W*packet_size/RTT



Example: Sliding Window w/o Errors

- Assume
  - – Link capacity, C = 1Gbps
  - – Latency between end-hosts, RTT = 80ms
  - – packet_length = 1000 bytes
- What is the window size W to match link's capacity, C?

- Solution

  We want Throughput = C

  Throughput = W*packet_size/RTT

  C = W*packet_size/RTT

  W = C*RTT/packet_size = 10^9bps*80*10^{-3}s/(8000b) = 10^4 packets

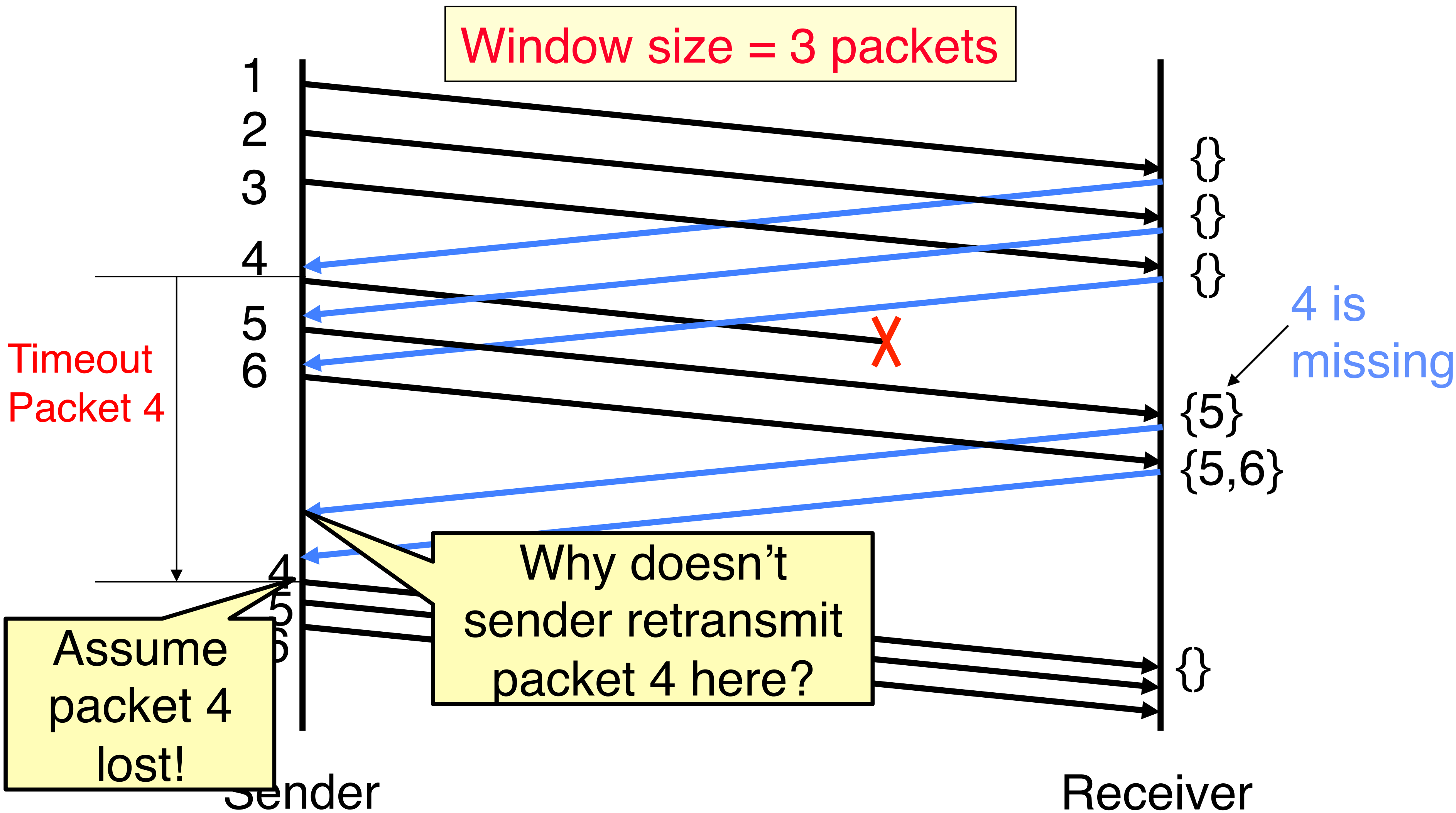
Window size ~ Bandwidth (Capacity), delay (RTT/2) product

Sliding Window with Errors

- Two approaches
  - Go-Back-n (GBN)
  - Selective Repeat (SR)
- In the absence of errors they behave identically

- Go-Back-n (GBN)
  - Transmit up to *n* unacknowledged packets
  - If timeout for ACK(*k*), retransmit *k*, *k+1*, …

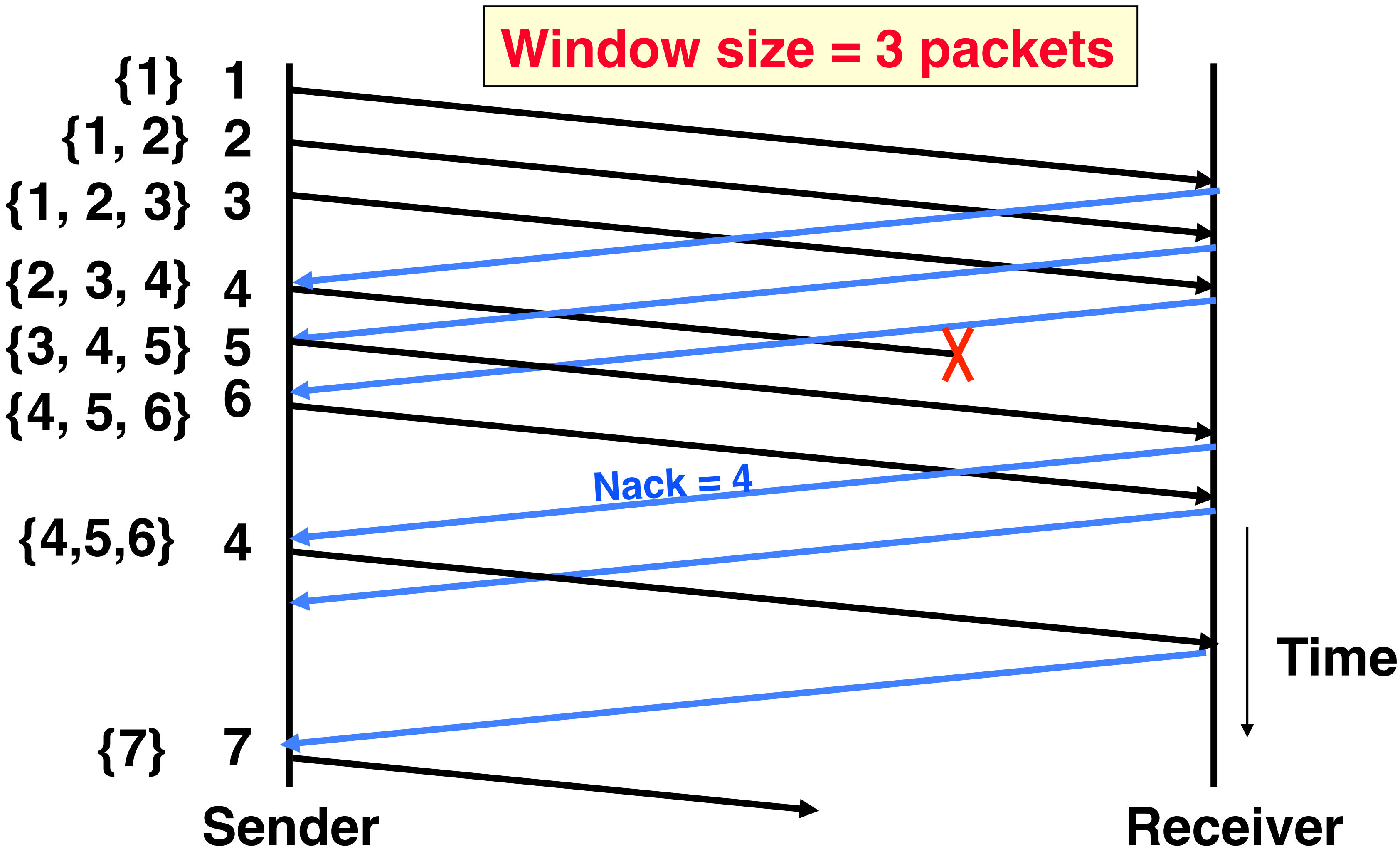
GBN Example with Errors



Selective Repeat (SR)

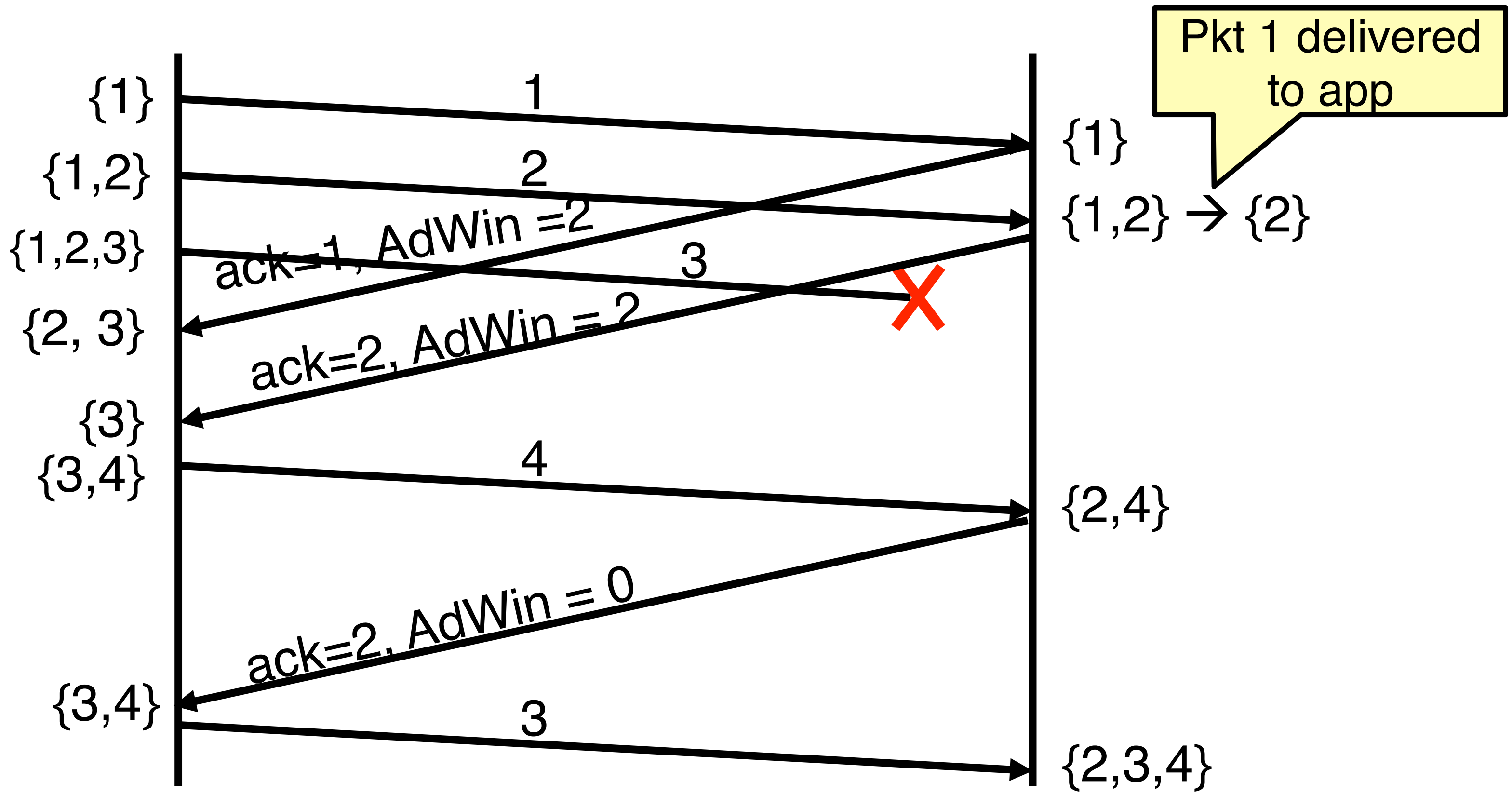
- Sender: transmit up to *n* unacknowledged packets;
- Assume packet *k* is lost
- Receiver: indicate packet *k* is missing
- Sender: retransmit packet *k*

SR Example with Errors



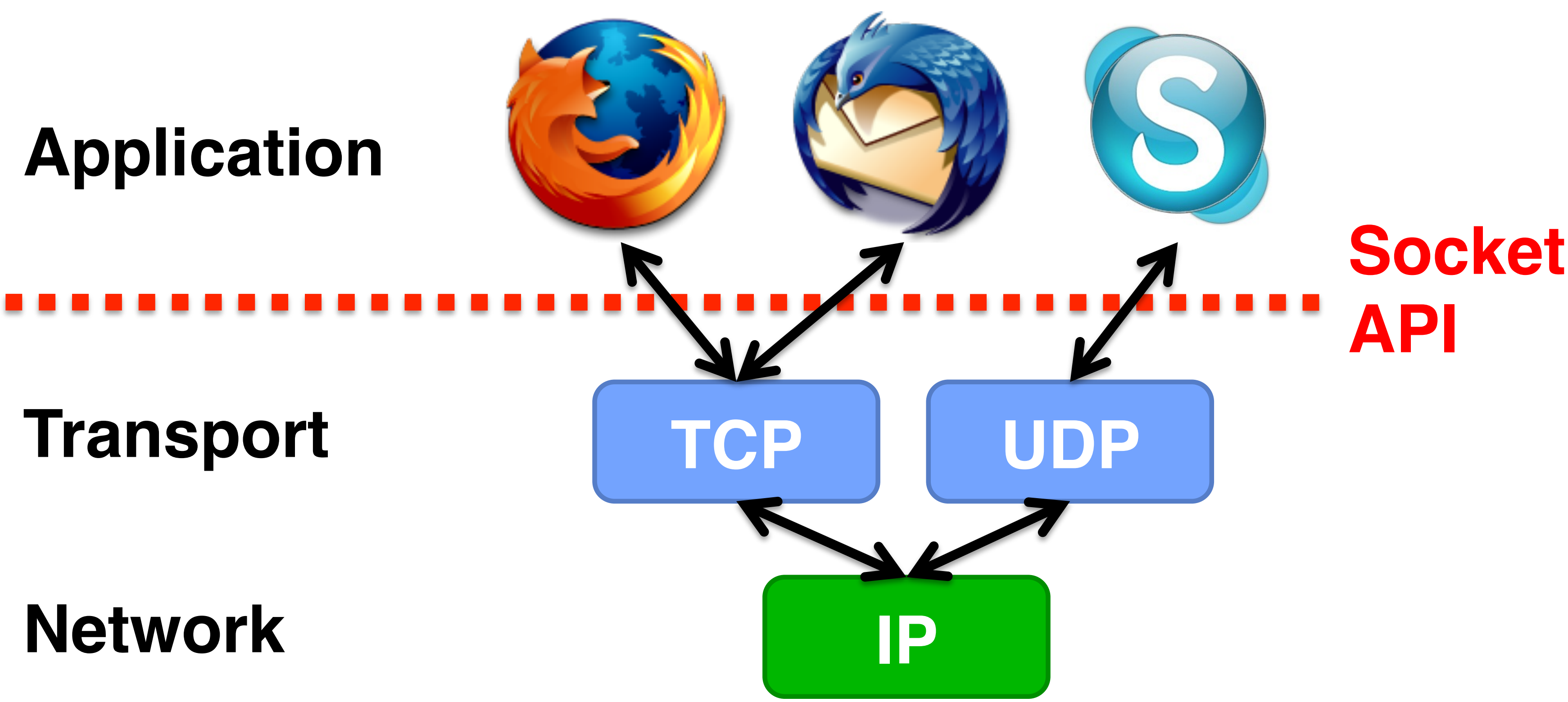
Flow Control

- Sliding window already implements flow control
  - Advertised Window (AdWin): receiver buffer
  - Ack packet specifies the seq. number of last packet received in sequence



Socket API

- Socket API
  - – Network programming interface



BSD Socket API

- Created at UC Berkeley (1980s)
- Most popular network API
- Ported to various OSes, various languages
  – Windows Winsock, BSD, OS X, Linux, Solaris, …
  – Socket modules in Java, Python, Perl, …
- Similar to Unix file I/O API
  – In the form of *file descriptor* (sort of handle).
  – Can share the same `read()`/`write()`/`close()` system calls

TCP: Transport Control Protocol

- Reliable, in-order, and at most once delivery
- Stream oriented: messages can be of arbitrary length
- Provides multiplexing/demultiplexing to IP
- Provides congestion control and avoidance
- Application examples: file transfer, chat

TCP Service

1) Open connection: 3-way handshaking

2) Reliable byte stream transfer from (IPa, TCP_Port1) to (IPb, TCP_Port2)
   - Indication if connection fails: Reset

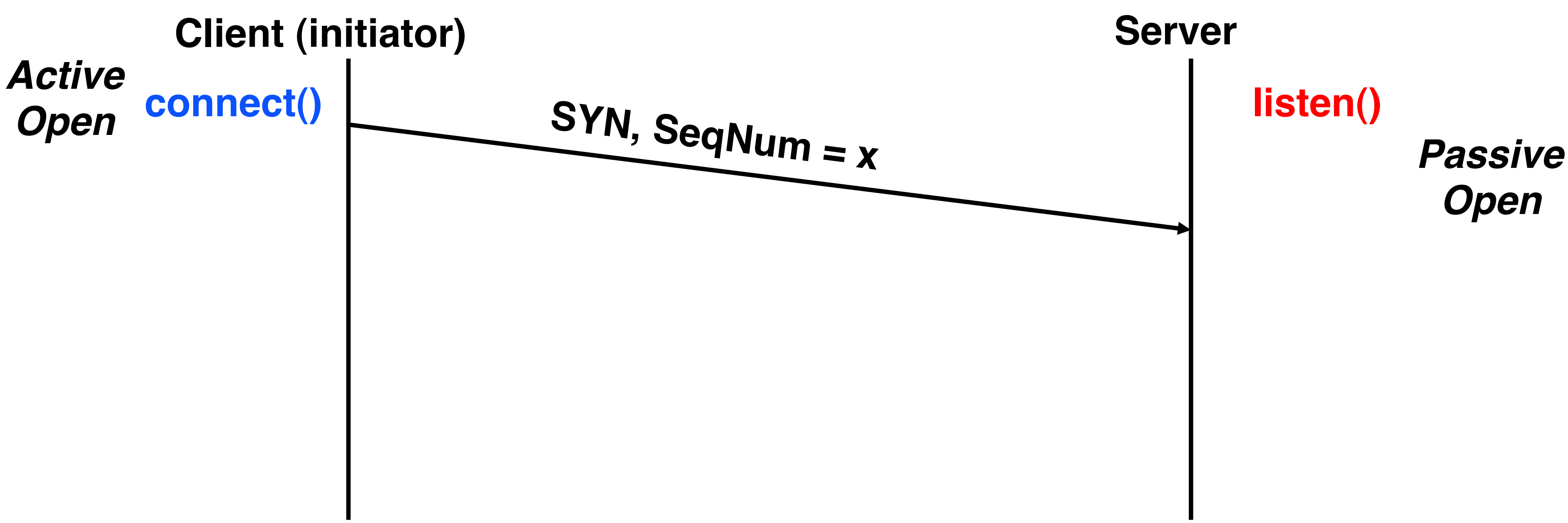
3) Close (tear-down) connection

Open Connection: 3-Way Handshaking

- Goal: agree on a set of parameters, i.e., the start sequence number for each side
  - Starting sequence number: sequence of first byte in stream
  - Starting sequence numbers are random

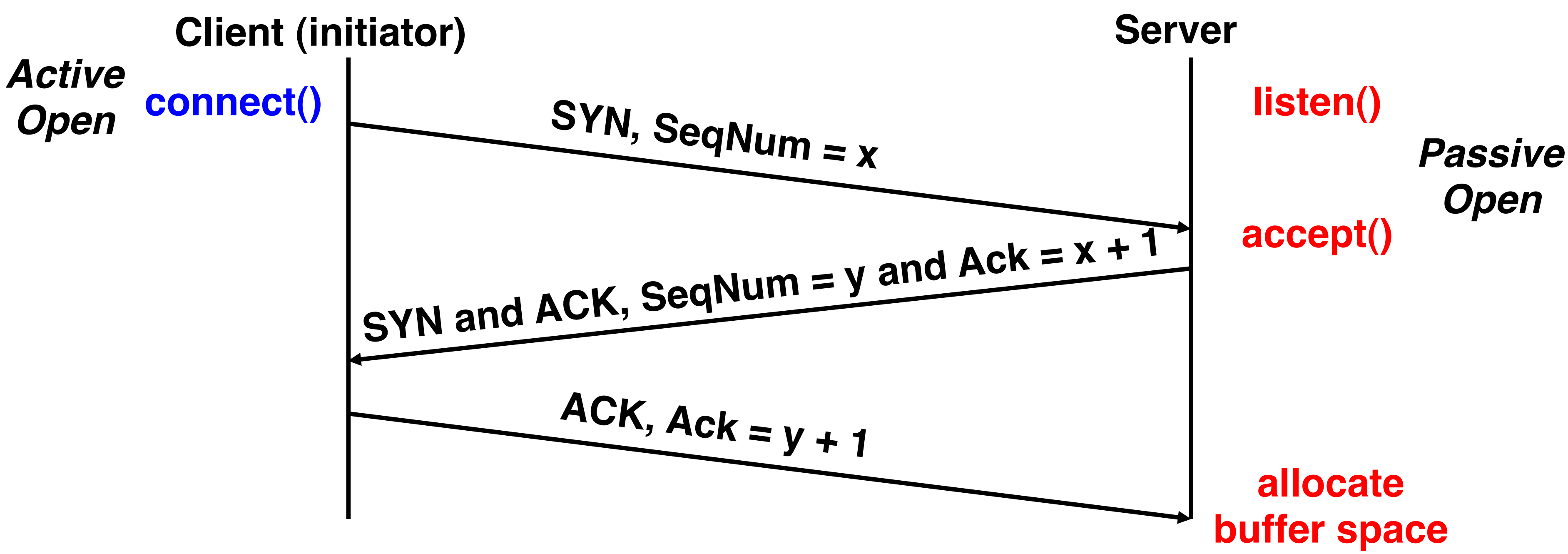
Open Connection: 3-Way Handshaking

- Server waits for new connection calling listen()
- Sender call connect() passing socket which contains server's IP address and port number
  - OS sends a special packet (SYN) containing a proposal for first sequence number, x



Open Connection: 3-Way Handshaking

- If it has enough resources, server calls accept() to accept connection, and sends back a SYN ACK packet containing
  - client's sequence number incremented by one, (x + 1)
    - » Why is this needed?
  - A sequence number proposal, y, for the first byte the server will send



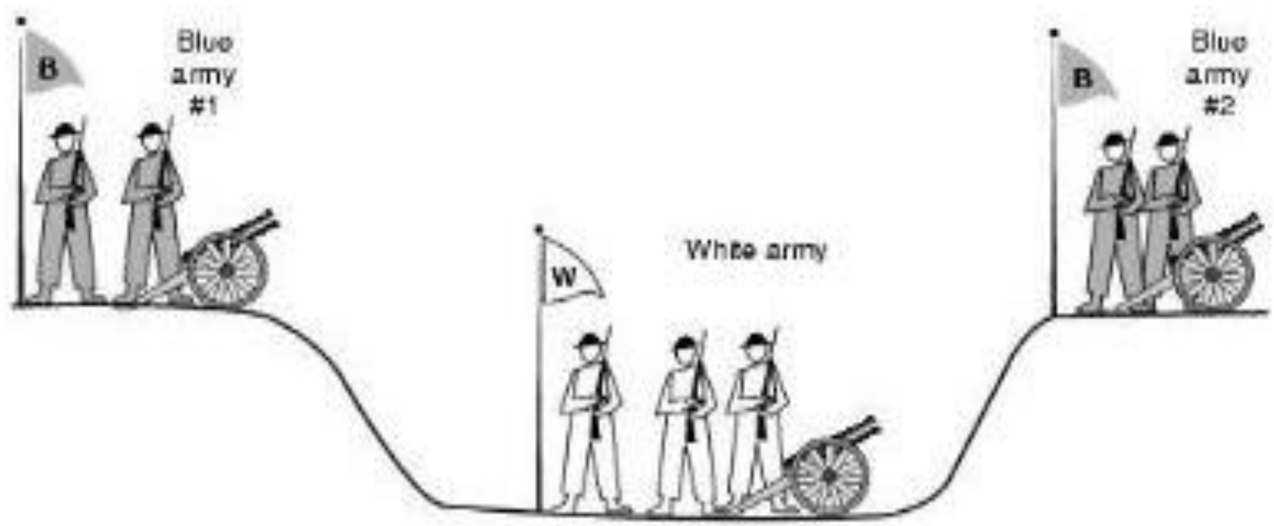
3-Way Handshaking (cont’d)

- Three-way handshake adds 1 RTT delay

- Why?
  - Congestion control: SYN (40 byte) acts as cheap probe
  - Protects against delayed packets from other connection (would confuse receiver)

Close Connection (Two Generals Problem)

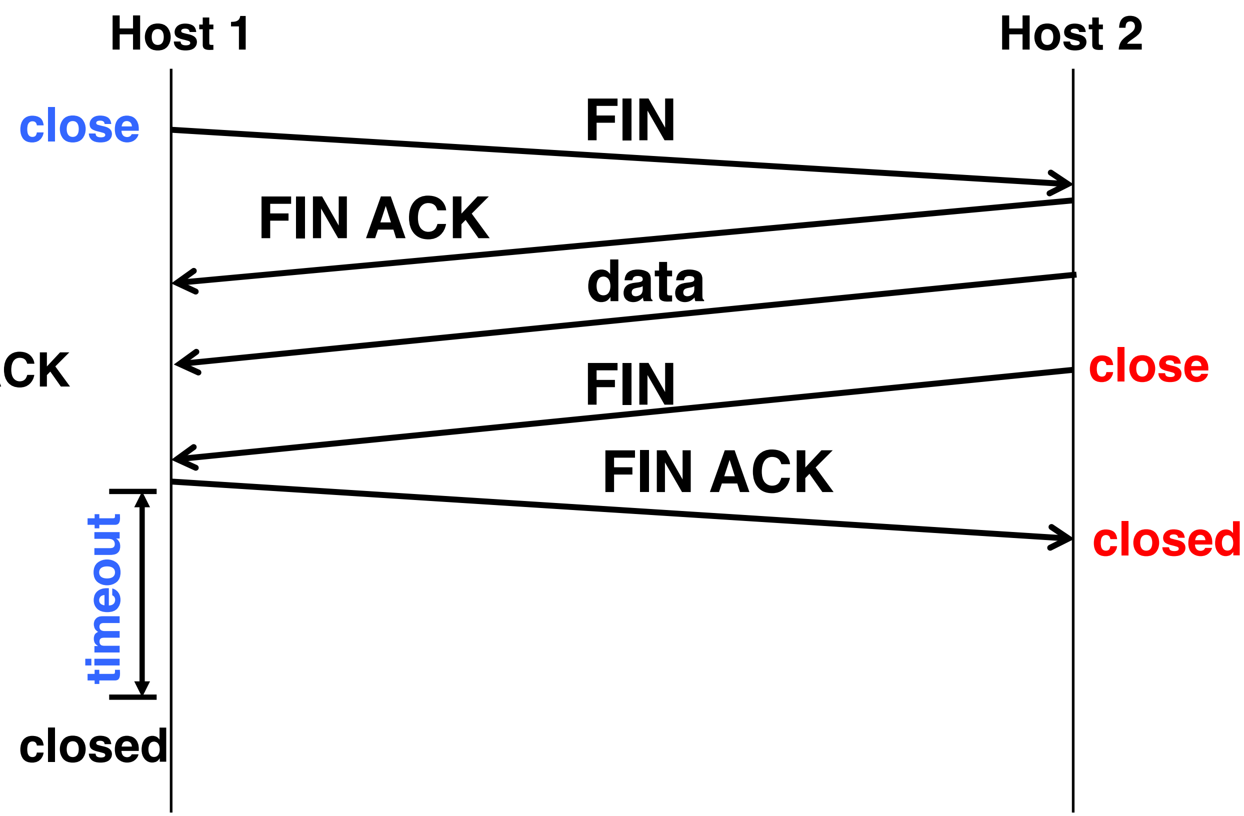
- Goal: both sides agree to close the connection
- Two-army problem:
  - "Two blue armies need to simultaneously attack the white army to win; otherwise they will be defeated. The blue army can communicate only across the area controlled by the white army which can intercept the messengers."



- What is the solution?

Close Connection

- 4-ways tear down connection



- Avoid reincarnation
- Can retransmit FIN ACK if it is lost

Summary

- Reliable transmission
  - S&W not efficient for links with large capacity (bandwidth) delay product
  - Sliding window far more efficient

- TCP: Reliable Byte Stream
  - Open connection (3-way handshaking)
  - Close connection: no perfect solution; no way for two parties to agree in the presence of arbitrary message losses (Byzantine General problem)