

TELCOM 2310: Applications of Networks

Lecture 6: Transport Layer Introduction

Objectives

- Understand the basic multiplexing / demultiplexing service provided by the transport layer
- Understand UDP's connectionless transport service
- Understand principles of reliable data transfer

Learning objectives

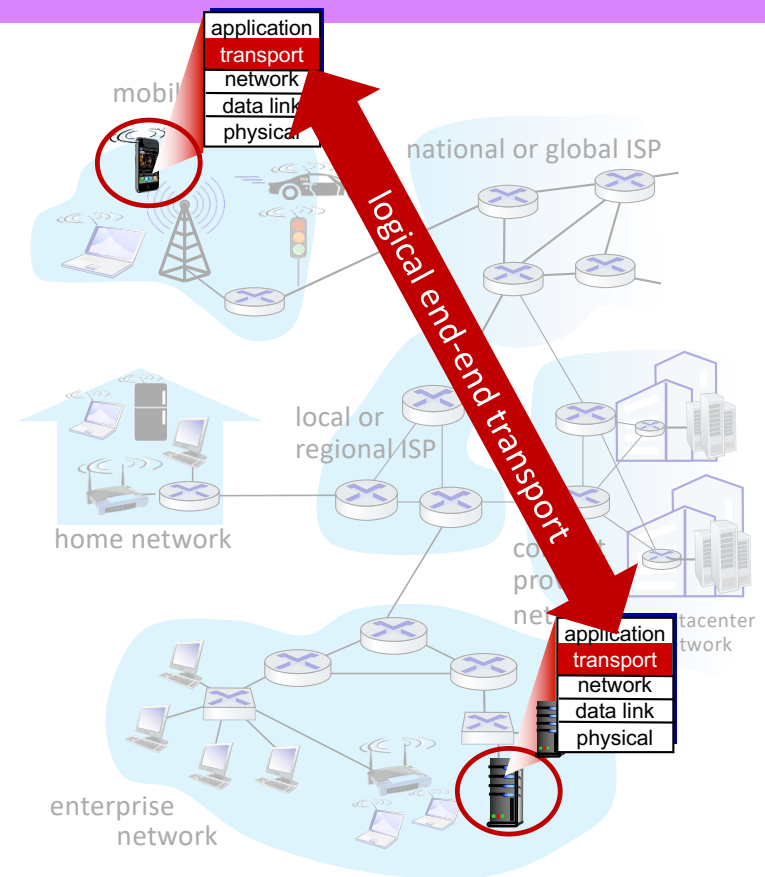
- After this meeting, you should be able to
 - Explain why a transport layer is needed
 - Explain why multiplexing is needed in TCP
 - Describe how the port mechanism for multiplexing functions
 - Describe the services offered by UDP
 - Critically evaluate the checksum in UDP

Rationale for a transport layer

- Provide generic *end-to-end* services to application layer
- Factors to consider
 - Speed
 - Throughput
 - Latency
 - Reliability
- We have different transport layer protocols because heterogeneous requirements across these factors!
- Economically speaking, factors such as web page engagement are strongly dependent on performance

Transport Layer Intro

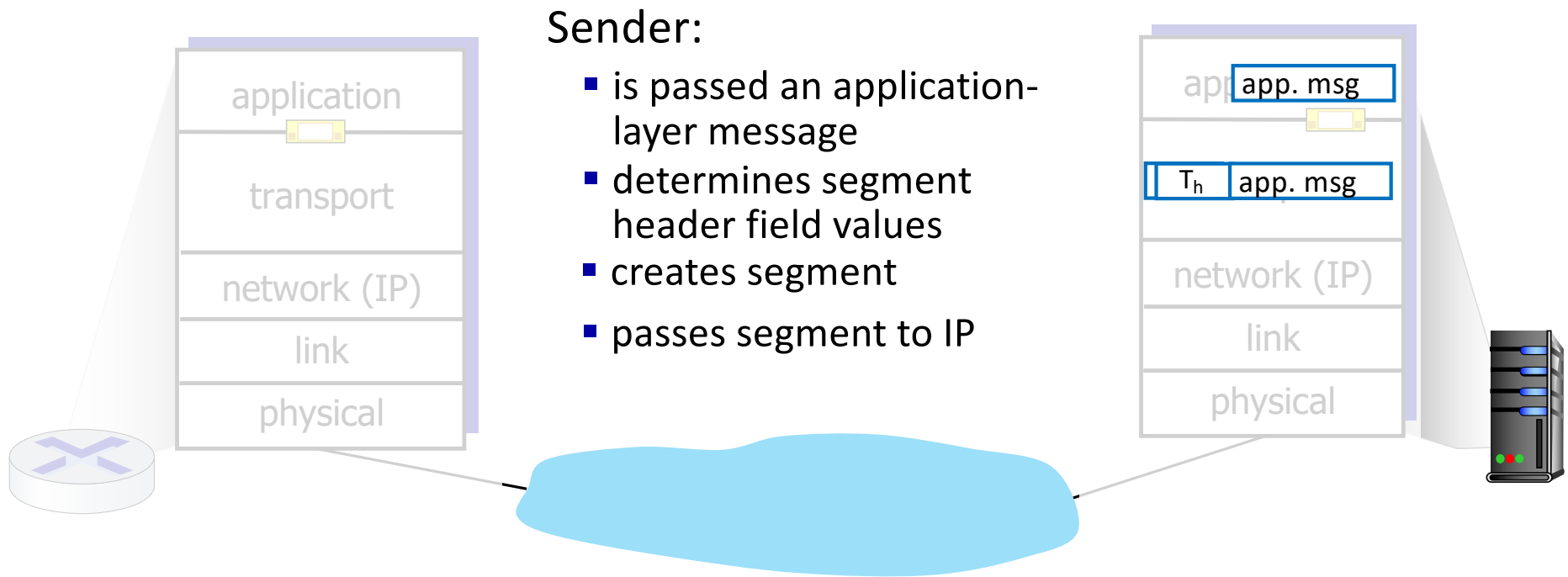
- Provides **logical communication** between **application processes**
 - Abstracts out network details; applications can act as if directly connected
- Built on top of **network layer** service
 - Provides *best-effort* (unreliable) communication between **hosts**
- Different transport protocols can offer different services
 - On the Internet: UDP vs TCP



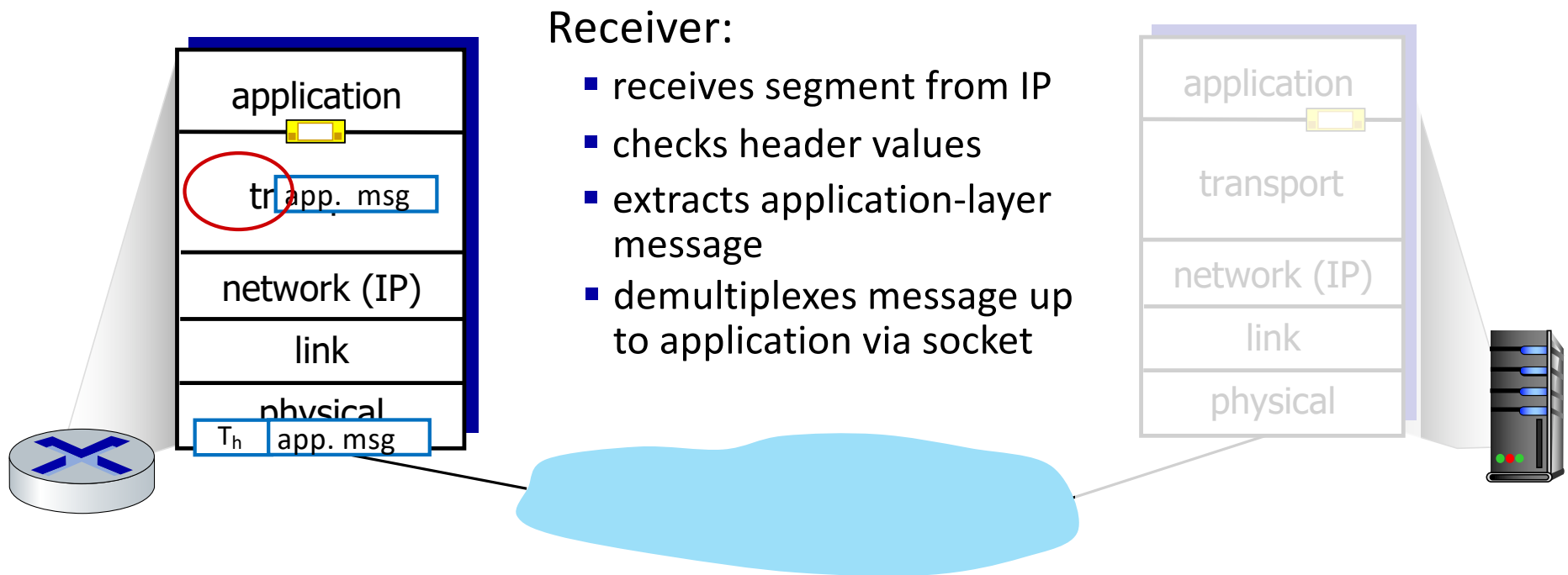
Transport Layer Services

- Core service: **Multiplexing/Demultiplexing**
 - Provide process-to-process communication on top of network layer's host-to-host communication
- **Error detection** (provided by UDP and TCP)
- TCP additionally offers:
 - **Reliable data transfer**
 - **Flow control**
 - **Congestion control**

Basic Transport Layer Actions



Basic Transport Layer Actions



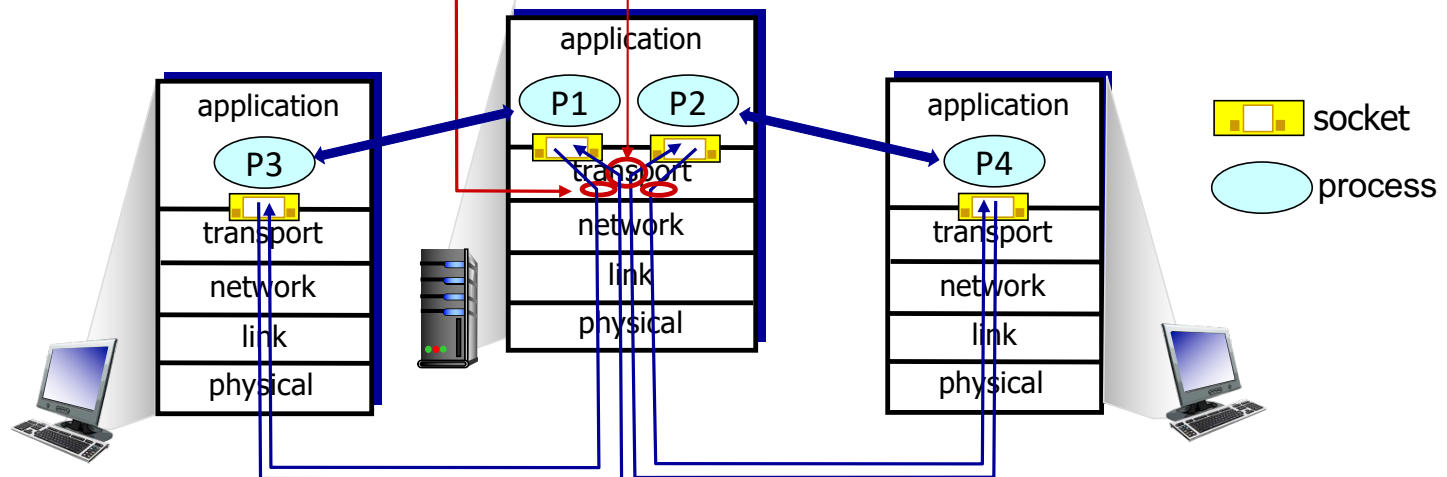
Multiplexing / Demultiplexing

multiplexing at sender:

handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:

use header info to deliver received segments to correct socket

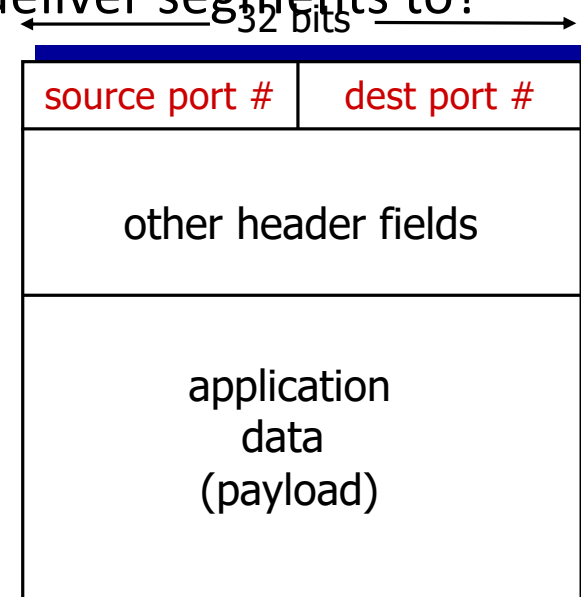


Multiplexing / Demultiplexing

- How do we identify the specific socket to deliver segments to?

Multiplexing / Demultiplexing

- How do we identify the specific socket to deliver segments to?
 - Based on **port numbers**



TCP/UDP segment format

Connectionless demultiplexing

Recall:

- when creating socket, must specify *host-local* port #:

```
DatagramSocket mySocket1  
= new DatagramSocket(12534);
```

- when creating datagram to send into UDP socket, must specify
 - destination IP address
 - destination port #

when receiving host receives *UDP* segment:

- checks destination port # in segment
- directs UDP segment to socket with that port #

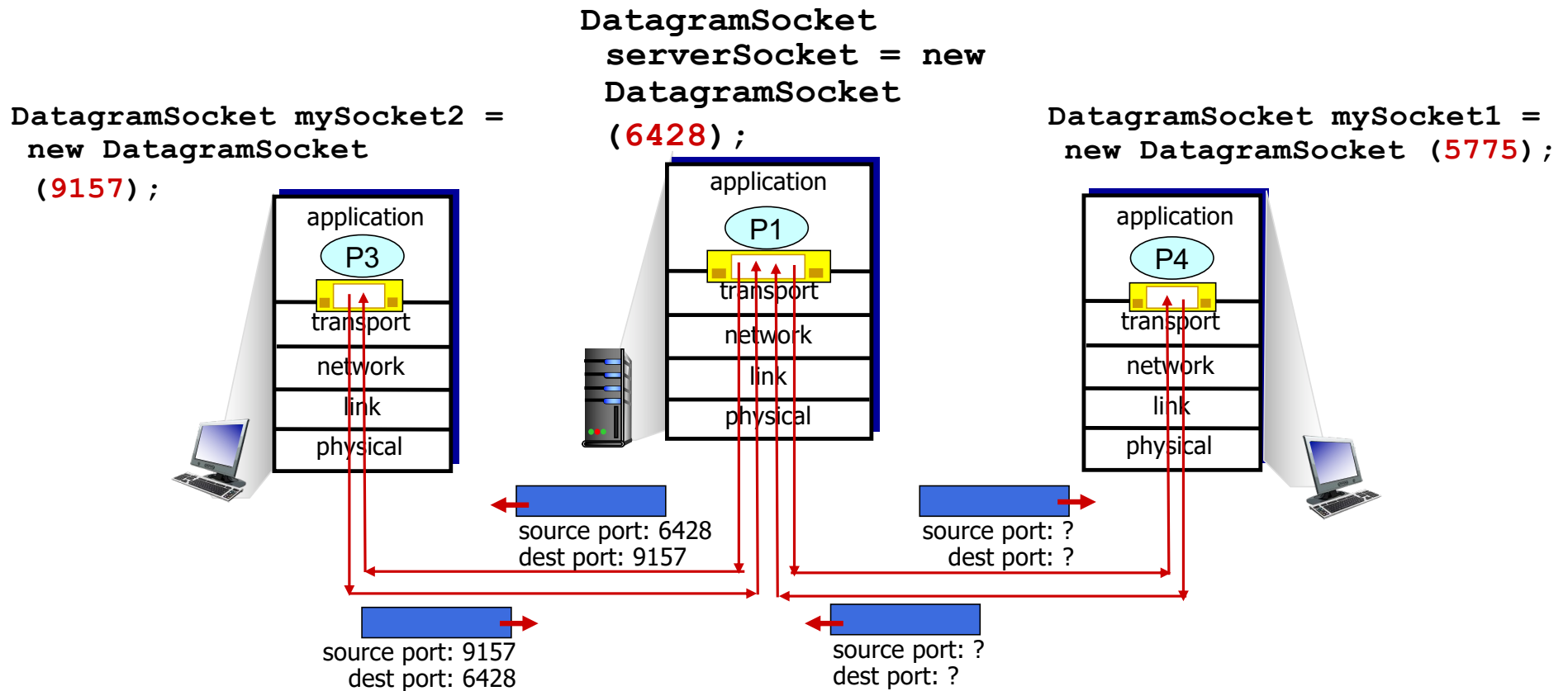


IP/UDP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

Connectionless (UDP) Demultiplexing

- When creating datagram to send into UDP socket, must specify:
 - destination IP address
 - destination port number
- When a host receives a UDP segment it:
 - checks the destination port number in the segment
 - delivers the UDP segment to the socket with that port number
- UDP segments arriving at a host with the **same destination port** are directed to **same socket** (regardless of source IP address/port)

Connectionless demultiplexing: an example



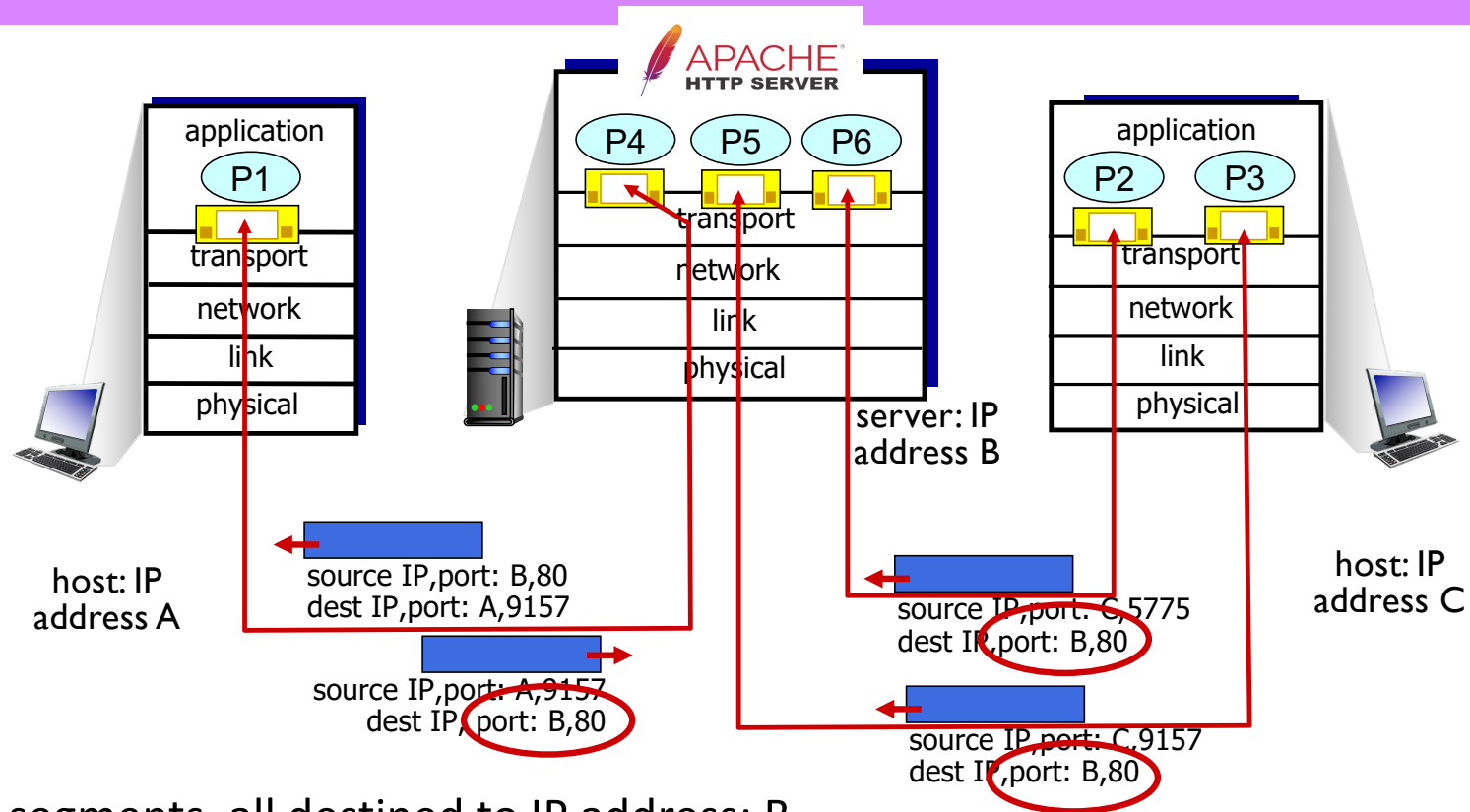
Connection-Oriented (TCP) Demultiplexing

- **Can we use the same approach?**
- Consider a webserver listening for incoming client connections on port 80
 - Because TCP is **connection-oriented**, each client gets its own socket / connection
 - But, all of them will be sending to port 80 on the webserver...**how do we differentiate?**

Connection-Oriented (TCP) Demultiplexing

- Each TCP socket is identified by a 4-tuple:
 - source IP address
 - source port number
 - destination IP address
 - destination port number
- Demultiplexing uses **all four** of these values to direct segments to the appropriate sockets

Connection-oriented demultiplexing: example



Three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

✓ User Datagram Protocol, Src Port: 57662, Dst Port: 53

Source Port: 57662
Destination Port: 53
Length: 44
Checksum: 0xacad [unverified]
[Checksum Status: Unverified]
[Stream index: 18]
> [Timestamps]
UDP payload (36 bytes)

✓ Domain Name System (query)

Transaction ID: 0xf7dc
> Flags: 0x0120 Standard query
Questions: 1
Answer RRs: 0
Authority RRs: 0
Additional RRs: 1

✓ Queries

> cnn.com: type A, class IN

✓ Additional records

> <Root>: type OPT

[\[Response In: 358\]](#)

Trar

✓ User Datagram Protocol, Src Port: 53, Dst Port: 57662

Source Port: 53
Destination Port: 57662
Length: 108
Checksum: 0x8ede [unverified]
[Checksum Status: Unverified]
[Stream index: 18]
> [Timestamps]
UDP payload (100 bytes)

✓ Domain Name System (response)

Transaction ID: 0xf7dc
> Flags: 0x8180 Standard query response, No error
Questions: 1
Answer RRs: 4
Authority RRs: 0
Additional RRs: 1

✓ Queries

> cnn.com: type A, class IN

✓ Answers

> cnn.com: type A, class IN, addr 151.101.131.5
> cnn.com: type A, class IN, addr 151.101.3.5
> cnn.com: type A, class IN, addr 151.101.67.5
> cnn.com: type A, class IN, addr 151.101.195.5

✓ Additional records

> <Root>: type OPT

[\[Request In: 357\]](#)

Connectionless Transport: UDP

- Lightweight transport protocol
- Provides **multiplexing/demultiplexing** and **error detection**
- **No reliability guarantees**

Connectionless Transport: UDP

- Benefits compared with TCP:
 - No connection setup overhead
 - Recall Lab 1 results...DNS uses UDP for this reason, HTTP moving toward it as well (QUIC protocol)
 - No need to maintain connection state / buffers
 - Lower load on operating system
 - Smaller packet overhead
 - 8-byte header vs 20-byte header
 - Losses don't block/slow delivery of new data
 - Good fit for loss-tolerant real-time applications (e.g. video/audio)

UDP: User Datagram Protocol

- UDP use
 - When speed is more important than reliability
 - When underlying network is reliable
- Examples
 - streaming multimedia apps (loss tolerant, rate sensitive)
 - DNS
 - SNMP
 - HTTP/3
- if reliable transfer needed over UDP (e.g., HTTP/3):
 - add needed reliability at application layer
 - add congestion control at application layer

UDP Specification

RFC 768

INTERNET STANDARD

J. Postel
ISI
28 August 1980

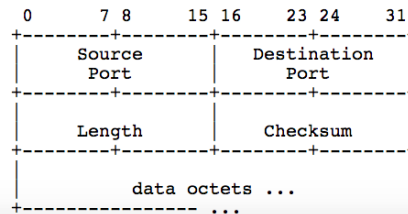
User Datagram Protocol

Introduction

This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

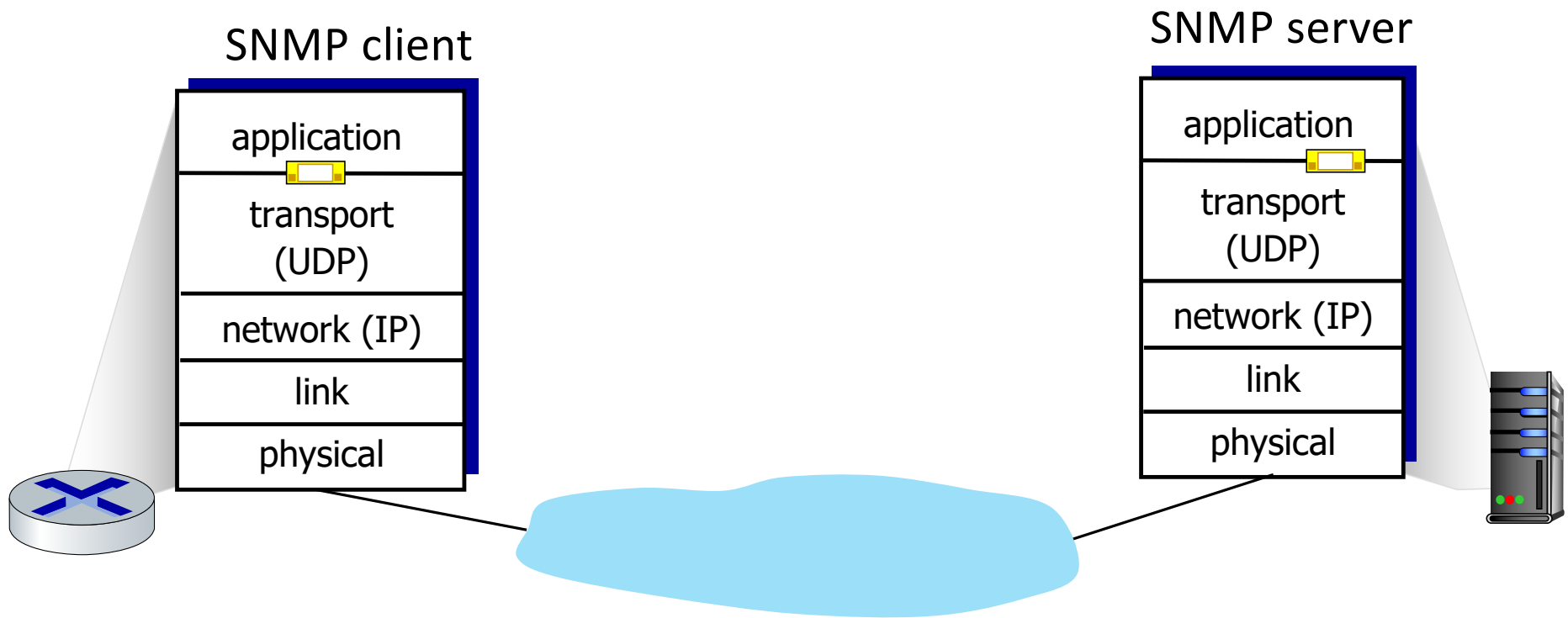
This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

Format

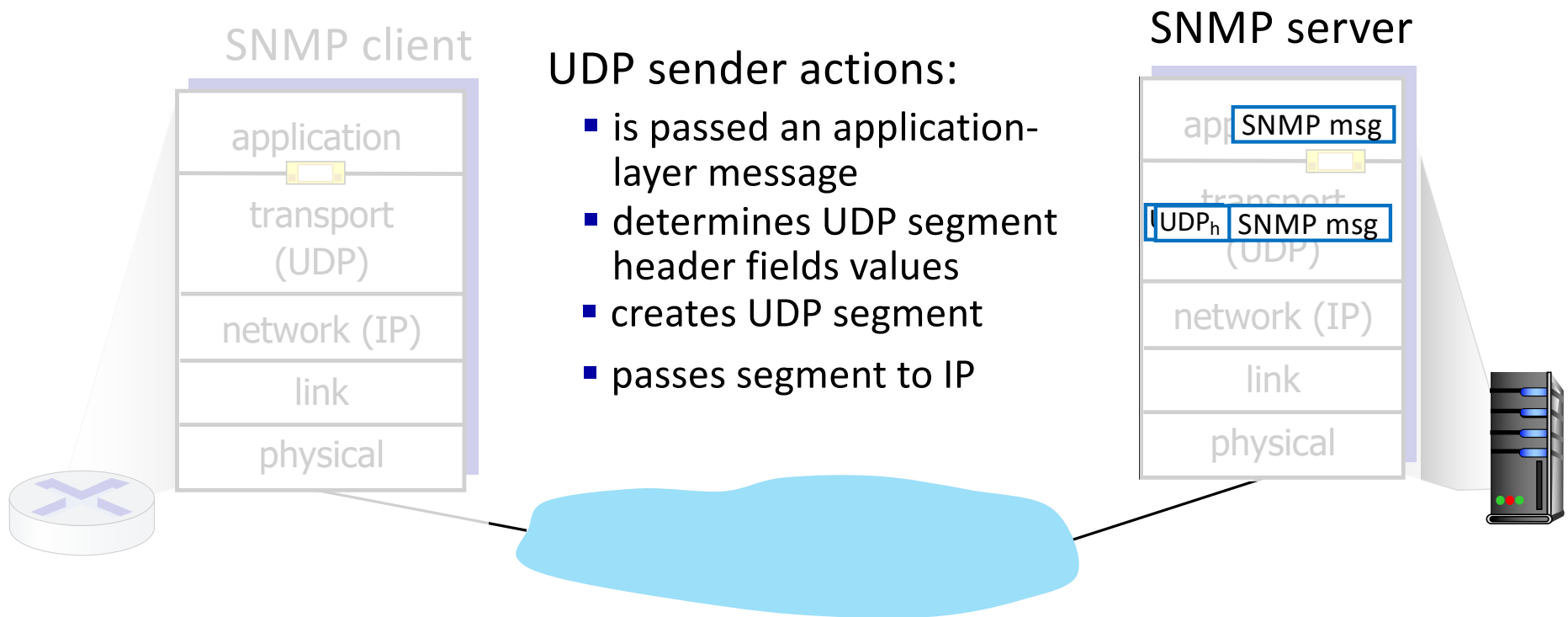


<https://www.ietf.org/rfc/rfc768.txt>

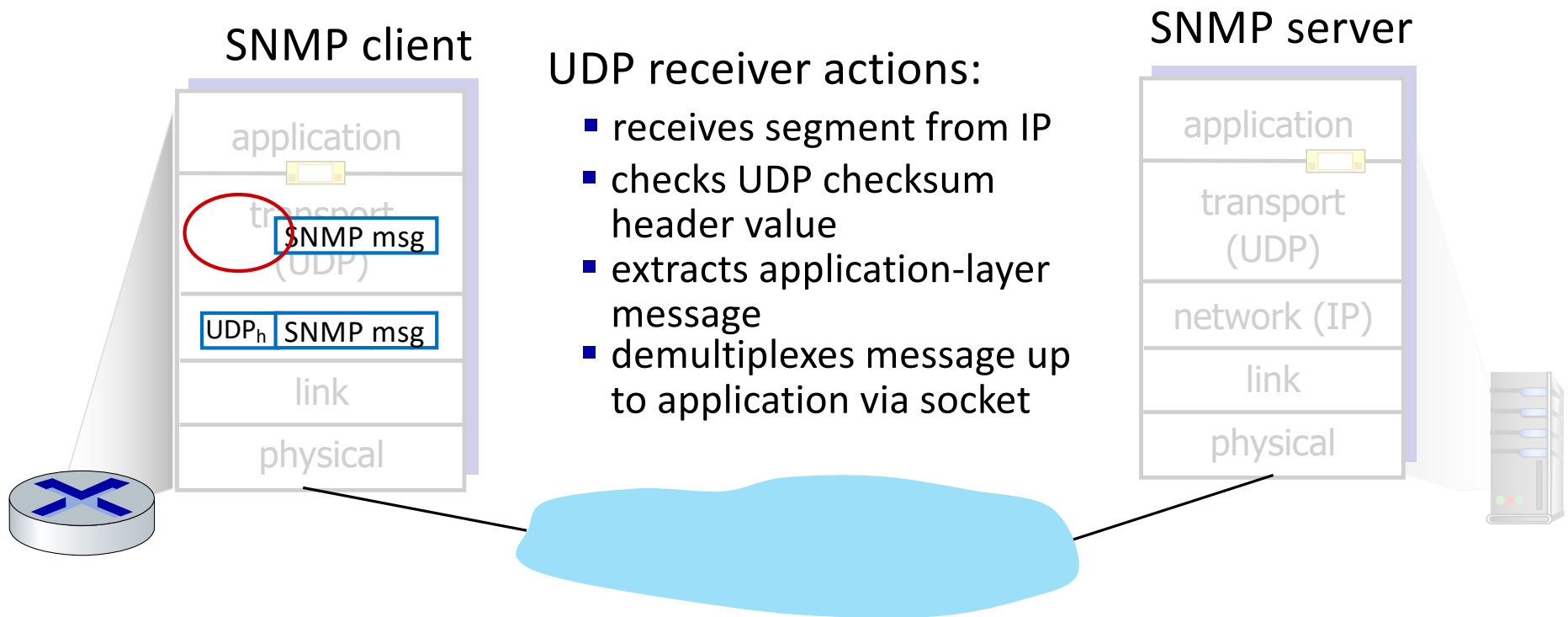
UDP: Transport Layer Actions



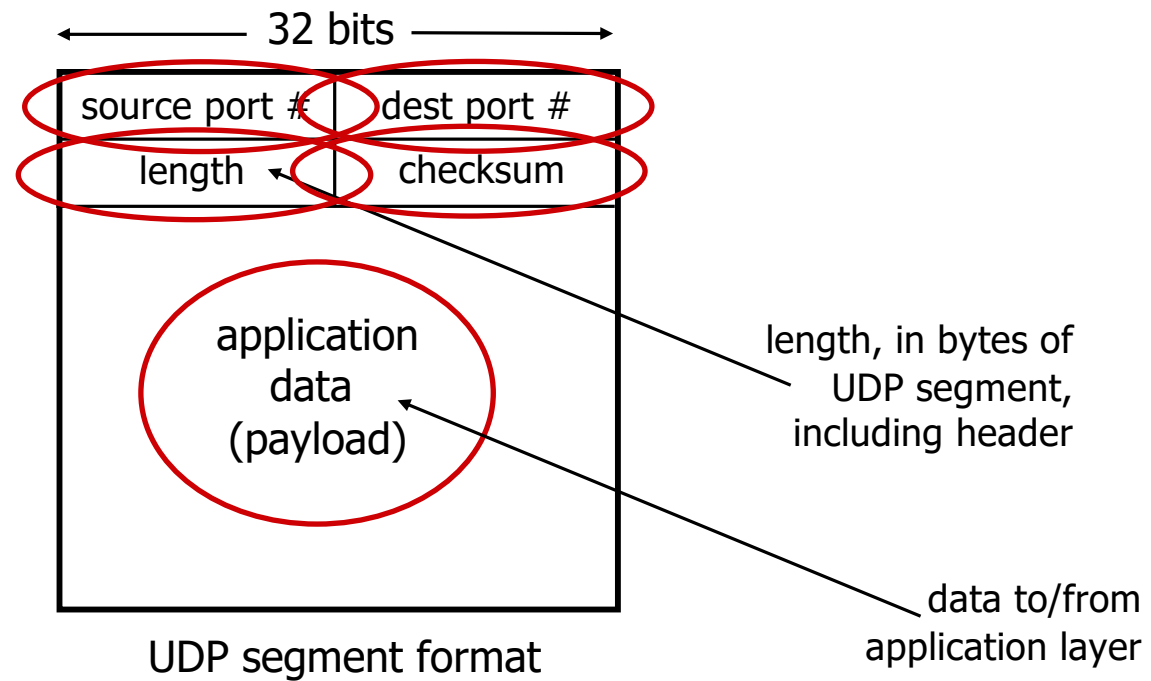
UDP: Transport Layer Actions



UDP: Transport Layer Actions

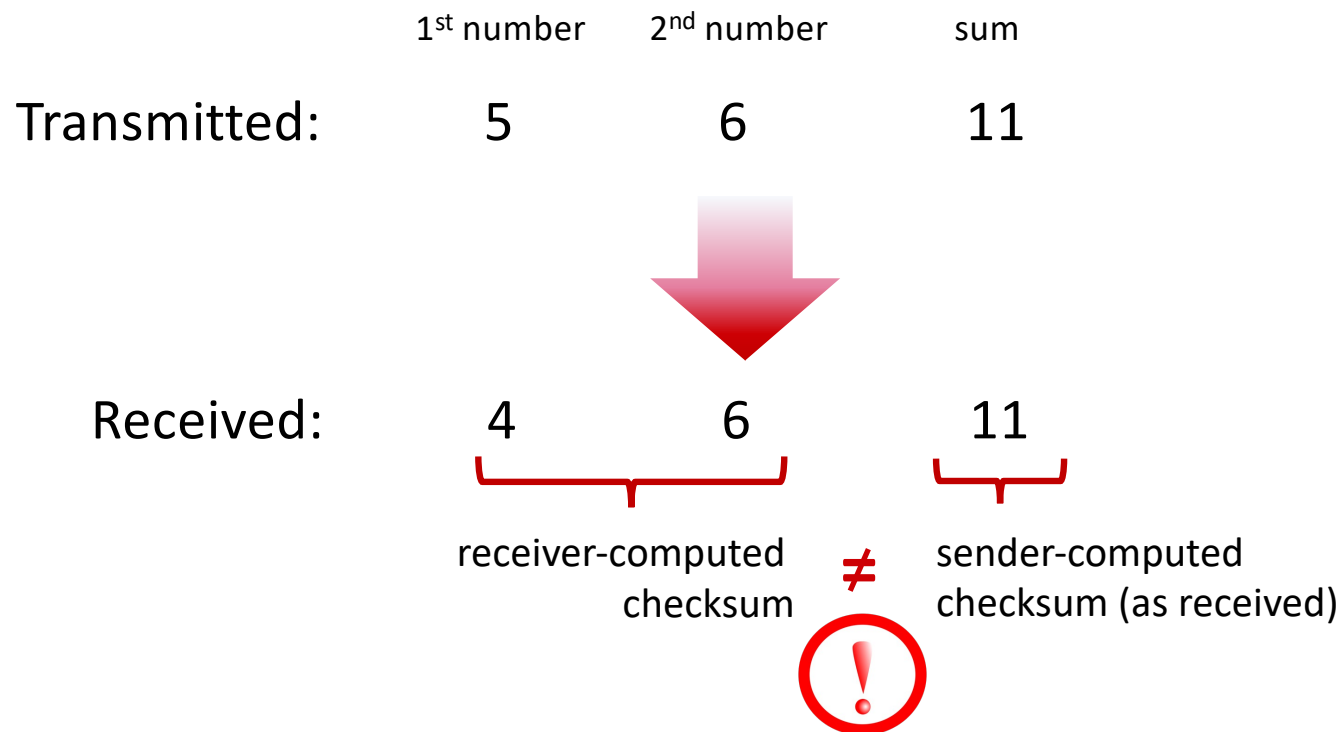


UDP Segment Structure



UDP checksum

Goal: detect errors (*i.e.*, flipped bits) in transmitted segment



UDP Checksum

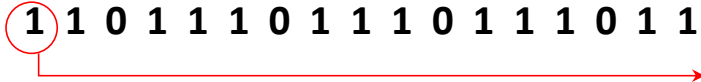
- **Goal:** **detect errors** in received segment
 - Error = flipped bit (i.e. 1 where there should be 0 or vice versa)
 - does not correct errors...just discard corrupted packets (looks the same as loss to application)
- **Mechanism** (high level):
 - Sender calculates some value based on all the bits in the segment (plus some fields of IP header) and stores this in the **checksum** field
 - Receiver can perform the same computation
 - **If result calculated at receiver doesn't match segment checksum**, segment was **corrupted**

UDP Checksum

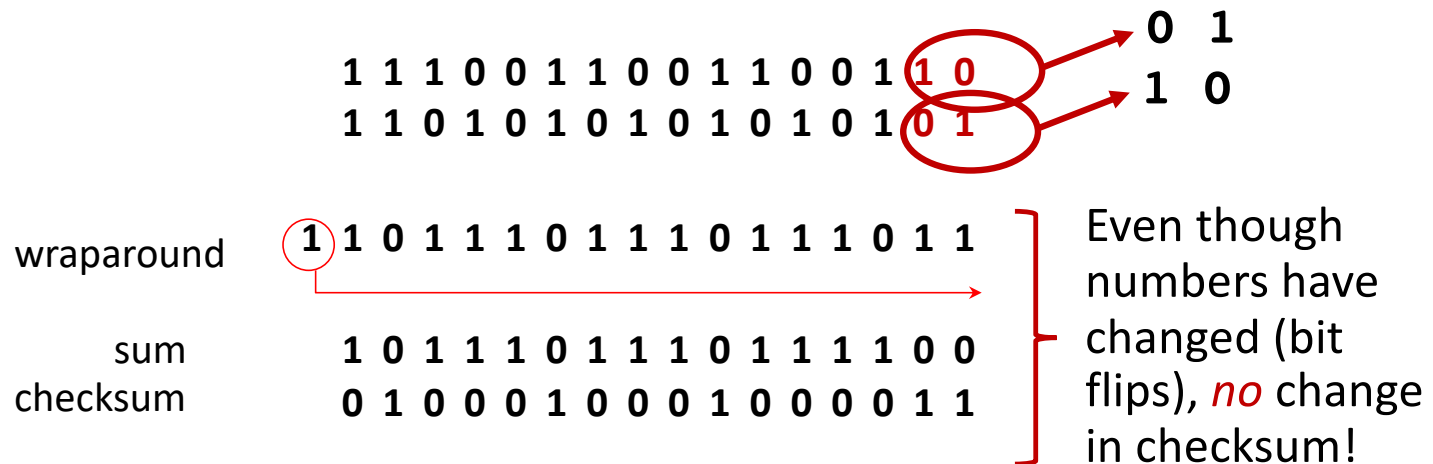
- **Mechanism** (more detailed):
 - Sender computes **checksum** by:
 - Breaking segment into 16-bit words
 - Adding them (overflow is wrapped around)
 - Taking 1s complement (convert all 0s to 1s and all 1s to zeros)
 - Receiver can verify by re-computing through same procedure
 - Slightly faster/simpler: add up all 16-bit words (including checksum). Result should be 1111111111111111

Example: Checksum computation

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1



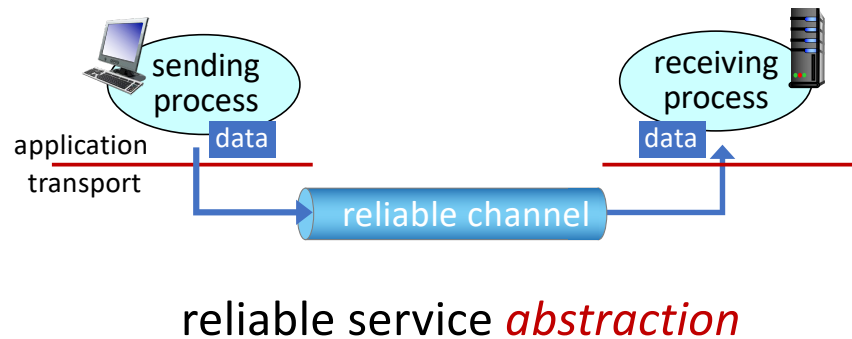
Example: Checksum computation



Will an error **always** be detected?

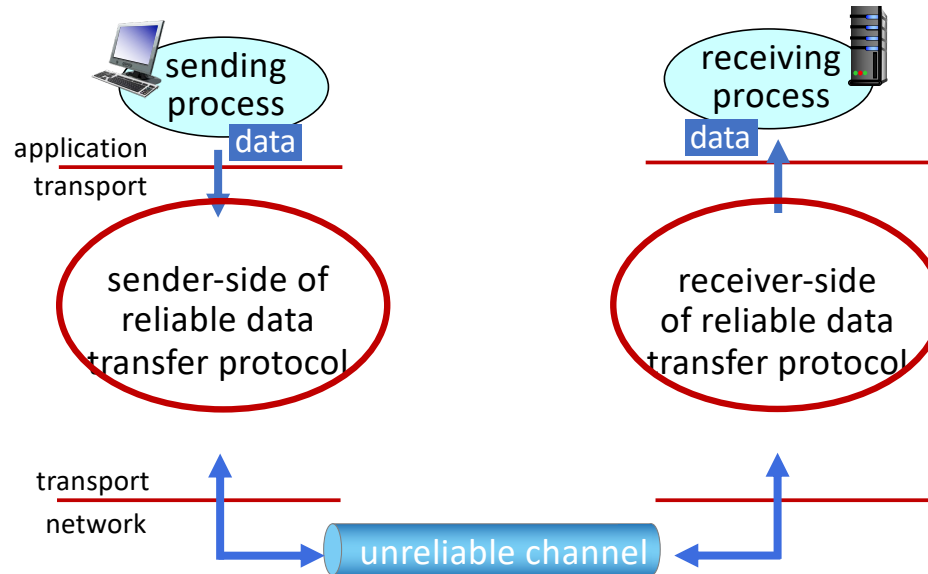
Principles of Reliable Data Transfer

- Goal: implement a **reliable service abstraction** over an **unreliable channel**



Principles of Reliable Data Transfer

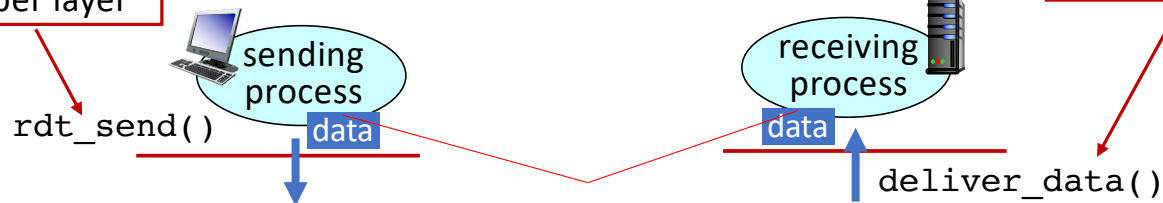
- Goal: implement a **reliable service abstraction** over an **unreliable channel**



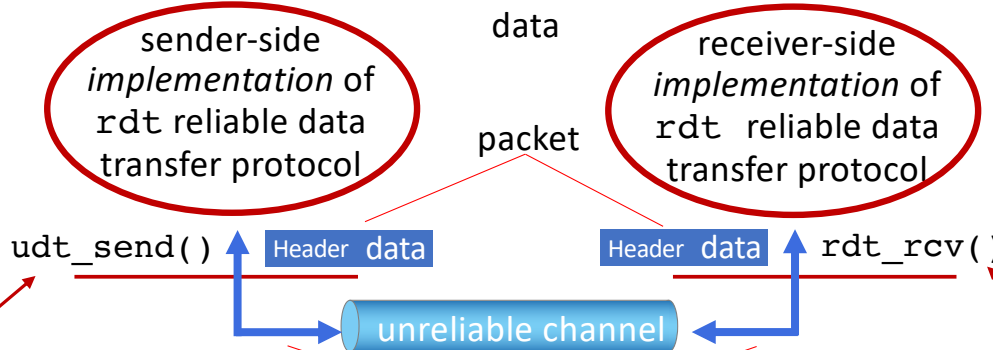
reliable service *implementation*

Reliable Data Transfer: Interfaces

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer



deliver_data(): called by rdt to deliver data to upper layer



udt_send(): called by rdt to transfer packet over unreliable channel to receiver

Bi-directional communication over unreliable channel

rdt_rcv(): called when packet arrives on receiver side of channel

In a perfect world...

- Assume we get to build our reliable transport protocol over a **reliable** network channel
- This is easy!

Sender:
Send data

Receiver:
Wait for data

In a perfect world...

- A little more precisely:

Sender:

- Wait for data to send
- Upon receiving data from app:
 - Package data into transport segment/package
 - Send packet

Receiver:

- Wait for packet from sender
- Upon receiving packet:
 - Unpackage and extract application data
 - Deliver data to application

In a perfect world...

- A little more precisely:

Sender:

- Wait for data to send
- Upon **rdt_send(data)**
 - **packet = make_pkt(data)**
 - **udt_send(packet)**

Receiver:

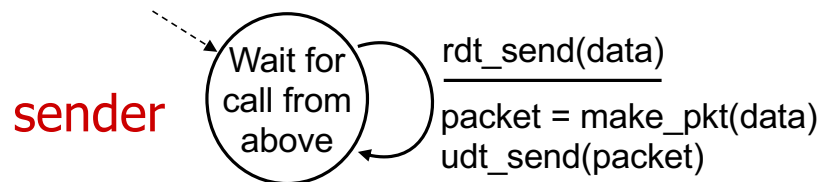
- Wait for packet from sender
- Upon **rdt_rcv(packet)**:
 - **data = extract(packet)**
 - **deliver_data(data)**

In a perfect world...

- A little more precisely:

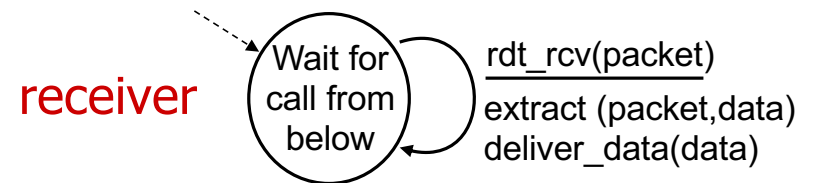
Sender:

- Wait for data to send
- Upon **rdt_send(data)**
 - **packet = make_pkt(data)**
 - **udt_send(packet)**



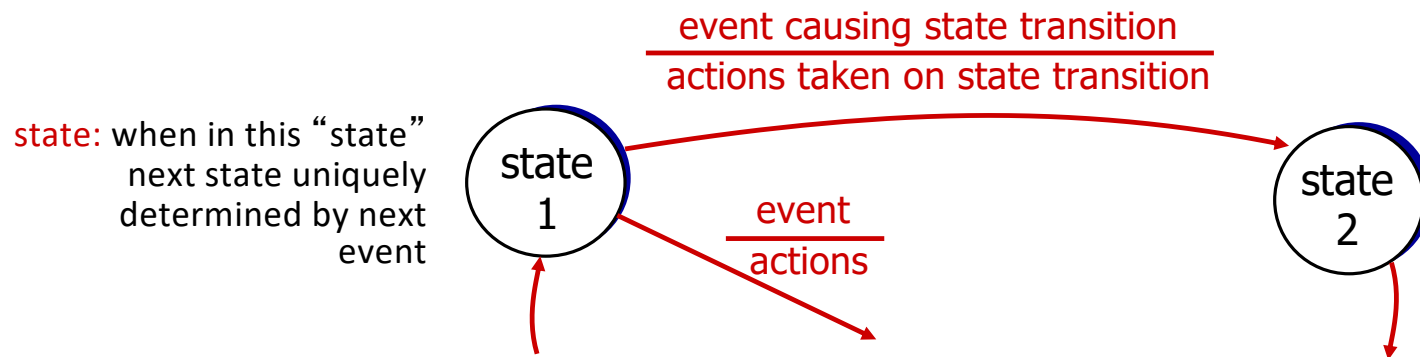
Receiver:

- Wait for packet from sender
- Upon **rdt_rcv(packet)**:
 - **data = extract(packet)**
 - **deliver_data(data)**



Aside...Finite State Machines (FSMs)

- Often used to specify network or distributed systems protocols
- Consist of:
 - **States** entities can be in
 - **Events** that cause transitions between states
 - **Actions** that entities take when transitioning



But in reality...the network is not reliable

- We'll start by assuming packets can be **corrupted (bit errors)** but NOT lost
- Questions:
 - How can the receiver know whether a packet it receives is correct?
 - What should the sender do if the packet was NOT received correctly?

But in reality...the network is not reliable

- Questions:
 - How can the receiver know whether a packet it receives is correct?
 - Try to verify the checksum (as we discussed)
 - What should the sender do if the packet was NOT received correctly?
 - Retransmit (resend) the packet

But in reality...the network is not reliable

- Questions:
 - How can the receiver know whether a packet it receives is correct?
 - Try to verify the checksum (as we discussed)
 - What should the sender do if the packet was NOT received correctly?
 - Retransmit (resend) the packet
 - **BUT**, how does the sender know whether or not the packet was received correctly?

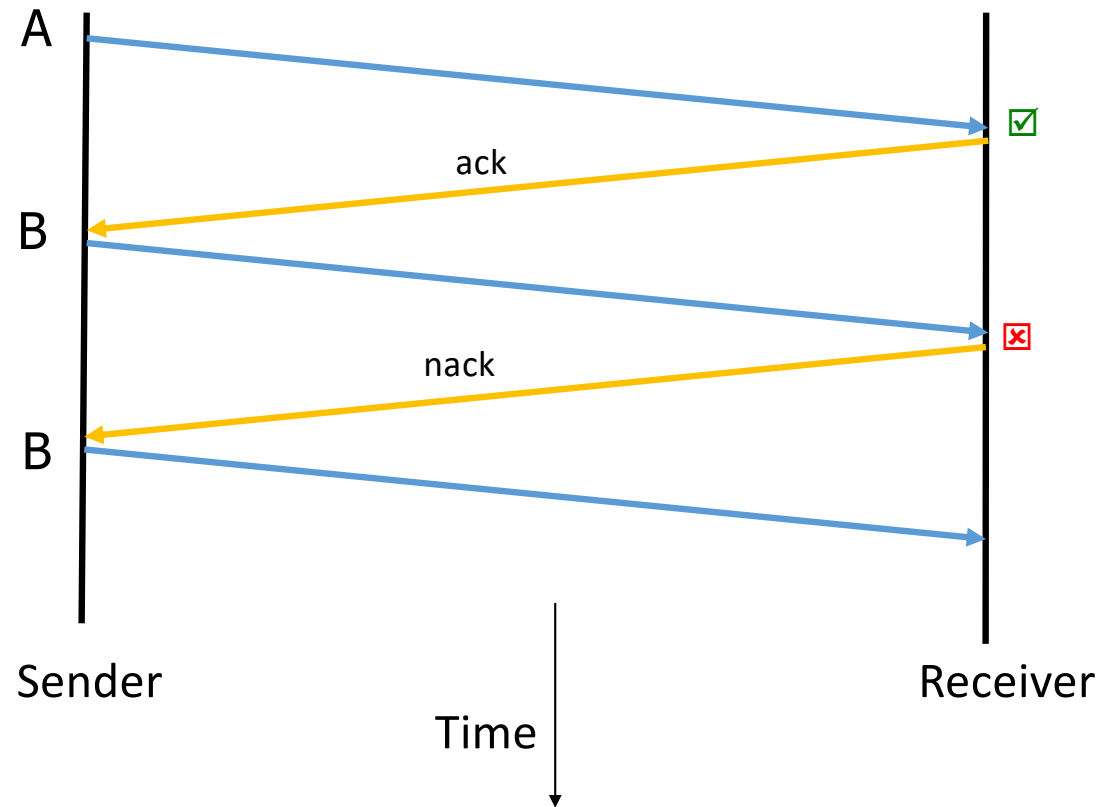
But in reality...the network is not reliable

- Questions:
 - How can the receiver know whether a packet it receives is correct?
 - Try to verify the **checksum** (as we discussed)
 - What should the sender do if the packet was NOT received correctly?
 - **Retransmit** (resend) the packet
 - **BUT, how does the sender know whether or not the packet was received correctly?**
 - **Feedback!**
 - **Acknowledgments (ACKs):** “I received the packet correctly”
 - **Negative Acknowledgements (NAKs):** “I did not receive this packet correctly”

Toward a solution: Stop-and-Wait (v1)

- **Stop-and-wait**: sender sends one packet, then waits for feedback from receiver
 - If feedback is ACK, good to move on to next packet
 - If feedback is NAK, resend the same packet

Toward a Solution: Stop-and-Wait (v1)



Toward a solution: Stop-and-Wait (v1)

- **Stop-and-wait**: sender sends one packet, then waits for feedback from receiver

Sender:

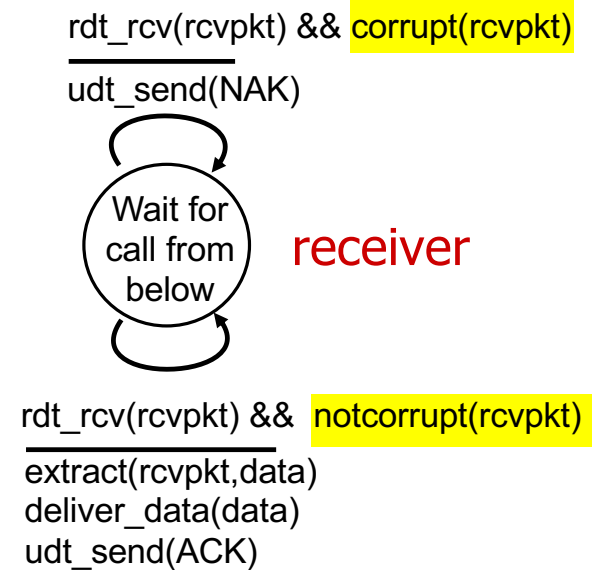
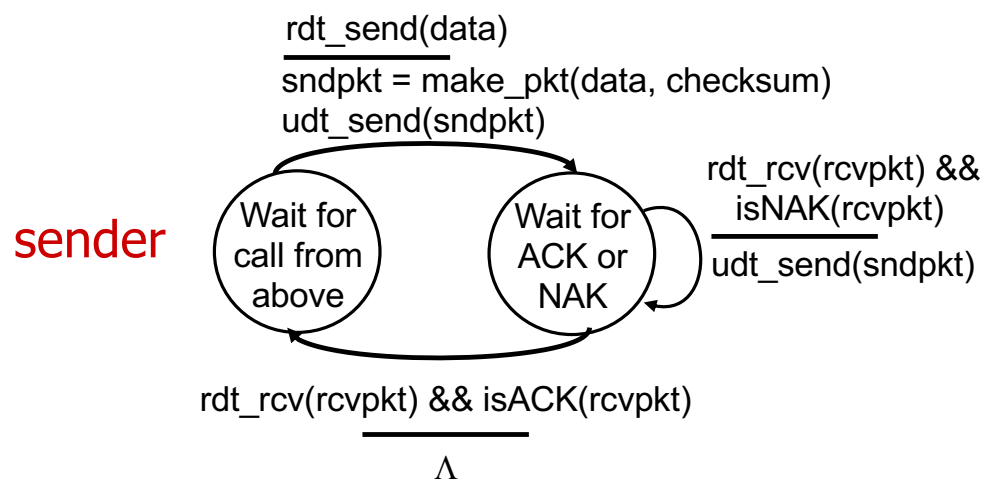
- Wait for data to send **(1)**
- Package data and send packet
- Wait for ACK/NAK **(2)**
 - If ACK:
 - Go wait for more data (1)
 - If NAK:
 - Resend packet and wait for new ACK/NAK (2)

Receiver:

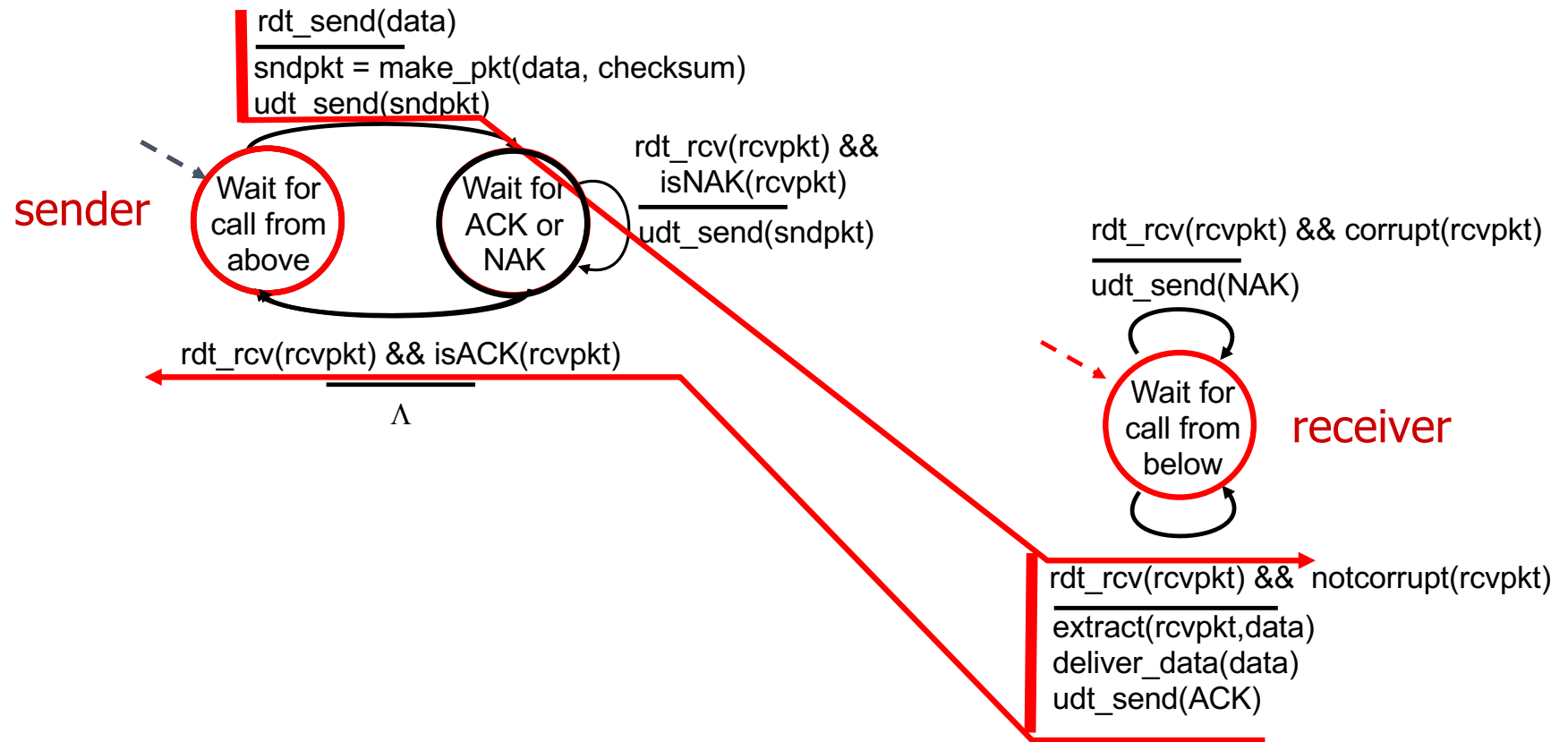
- Wait for packet from sender **(1)**
 - Compute checksum
 - If corrupt:
 - Send NAK
 - Else (not corrupt):
 - Extract and deliver to application
 - Send ACK
- Go wait for new packet from sender (1)

Toward a solution: Stop-and-Wait (v1)

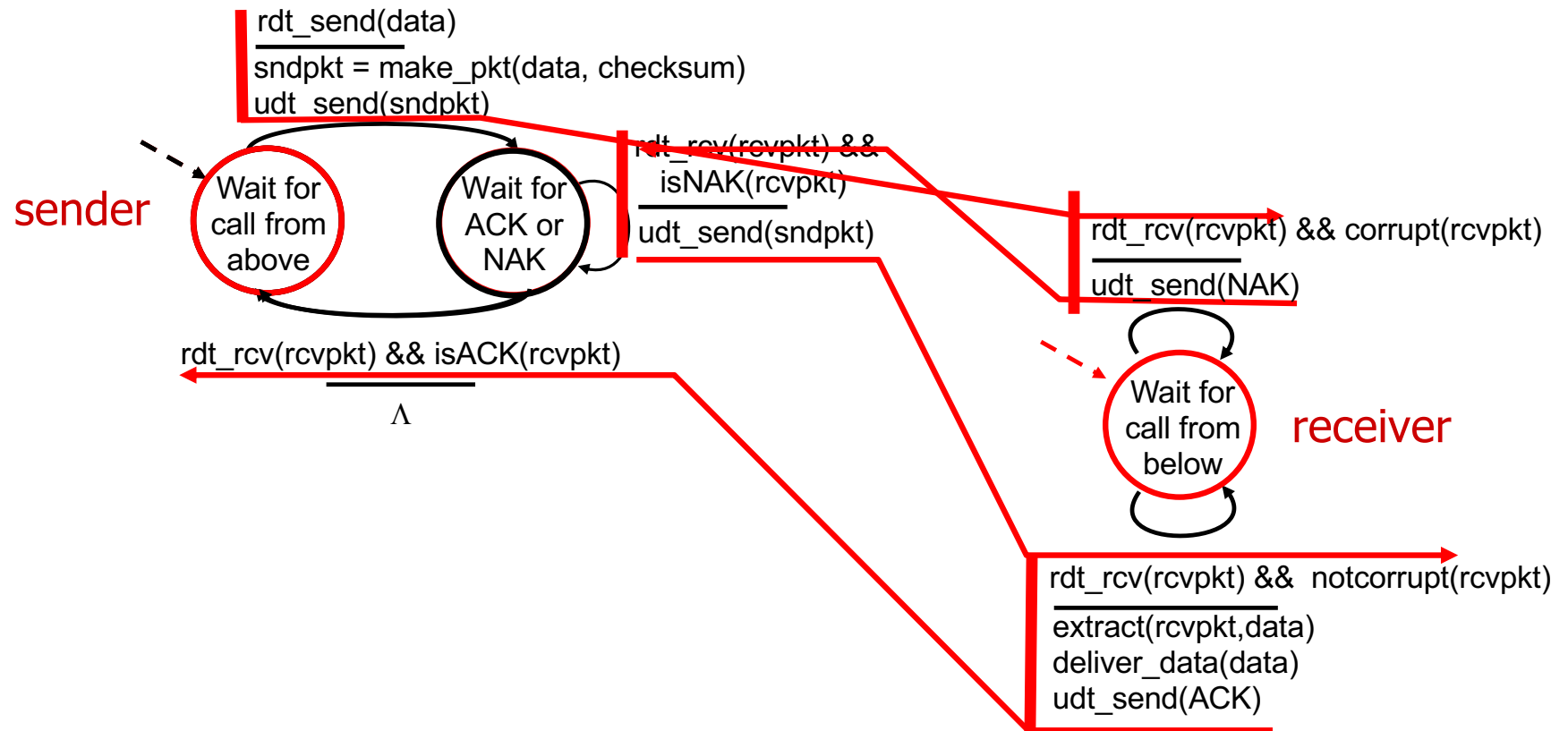
- **Stop-and-wait**: sender sends one packet, then waits for feedback from receiver



Reading the FSM: No errors



Reading the FSM: Packet Corruption Scenario



Good start, but this protocol is broken...

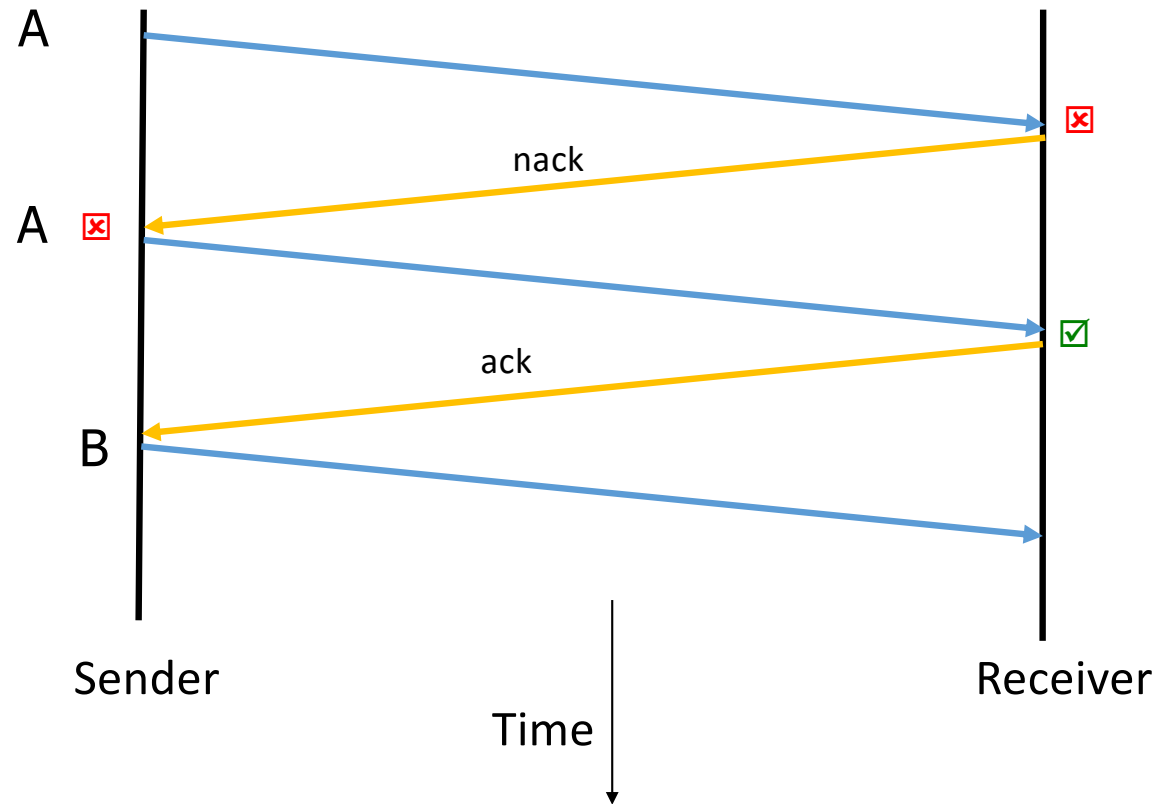
- **What happens if the ACK/NAK gets corrupted?**
- Sender doesn't know whether packet arrived correctly at the receiver
- **How should the sender respond?**

Dealing with corrupted ACK/NAK

- **How should the sender respond?**
 - Definitely NOT safe to assume everything is okay and move on to sending new data
 - We could think of many different solutions that introduce new message types (e.g. request retransmission of the ACK)...but then what if those get corrupted?
 - Simple idea: just retransmit the packet

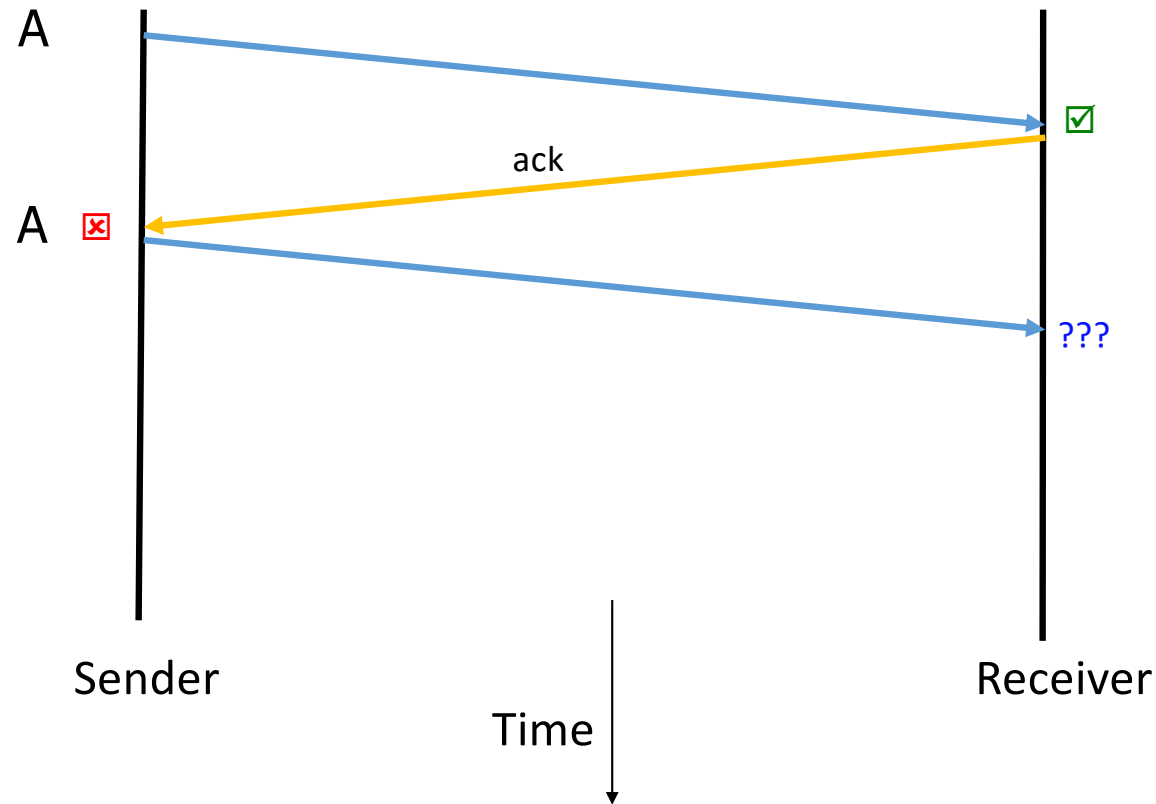
Dealing with corrupted ACK/NAK

Did they receive it
or not?
IDK...so, I'll
resend



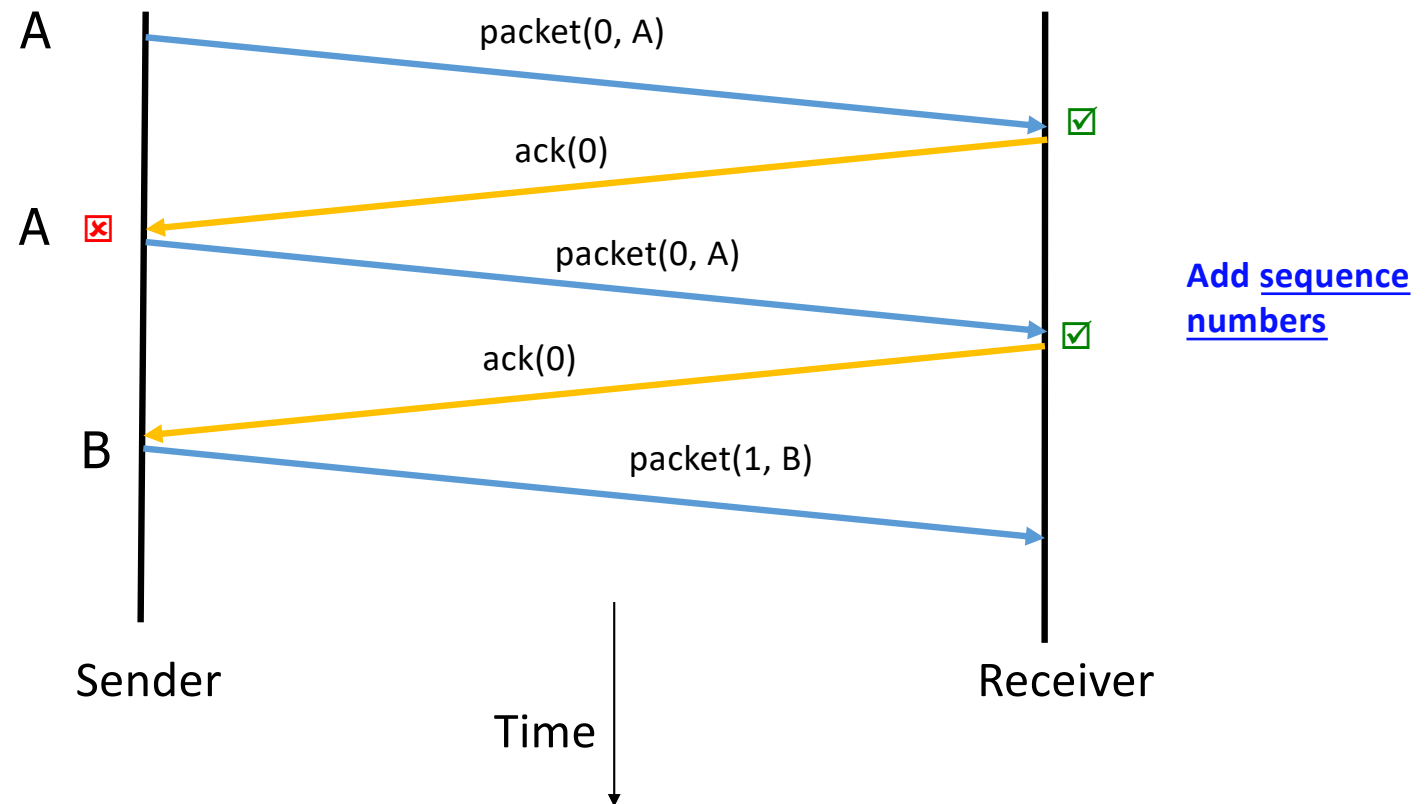
Dealing with corrupted ACK/NAK

Did they receive it
or not?
IDK...so, I'll
resend



How can the
receiver distinguish
new vs duplicate
packets?

Dealing with corrupted ACK/NAK



Sequence Numbers

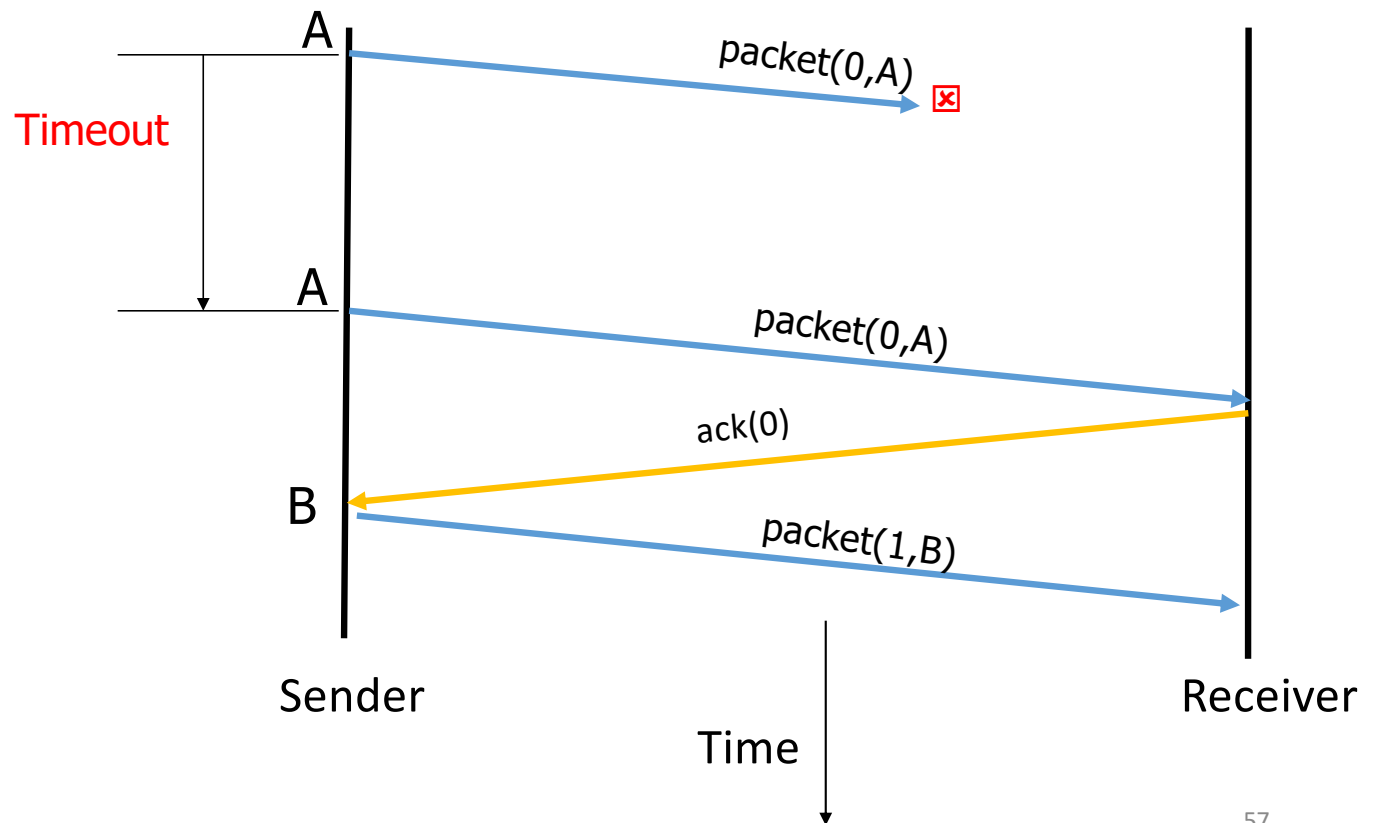
- **Sequence numbers** allow us to identify packets and distinguish new vs old/duplicate/retransmitted packets
 - Sender and receiver must both track next expected sequence number
- Technically, a 1-bit sequence number is enough for our bit-error-only stop-and-wait protocol (i.e. seq is either 0 or 1)
 - **Why? Does it matter?**
- Sequence numbers also allow us to get rid of NAK as packet type
 - Instead, send ACK for last correctly received seq. Duplicate ACK = NAK

Making our channel more realistic: Dealing with Loss

- *New channel assumption:* underlying channel can also *lose* packets (data, ACKs)
 - We still need checksum, ACKs, retransmissions and sequence numbers
- But we need 1 more new mechanism...
 - **What is it?**

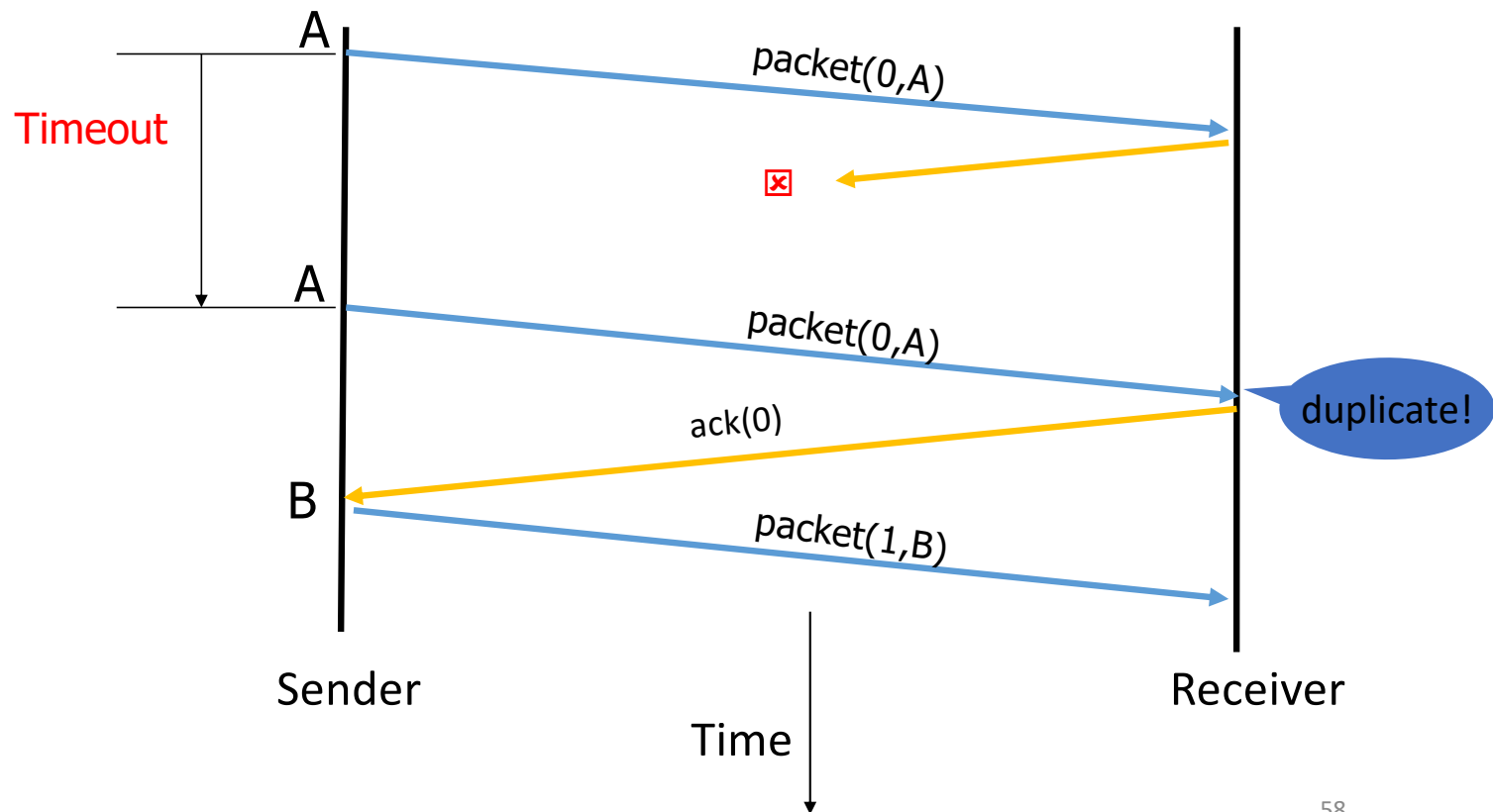
Dealing with Loss: Timeouts

- *Approach:* sender waits “reasonable” amount of time for ACK
 - Retransmits if no ACK received in this time



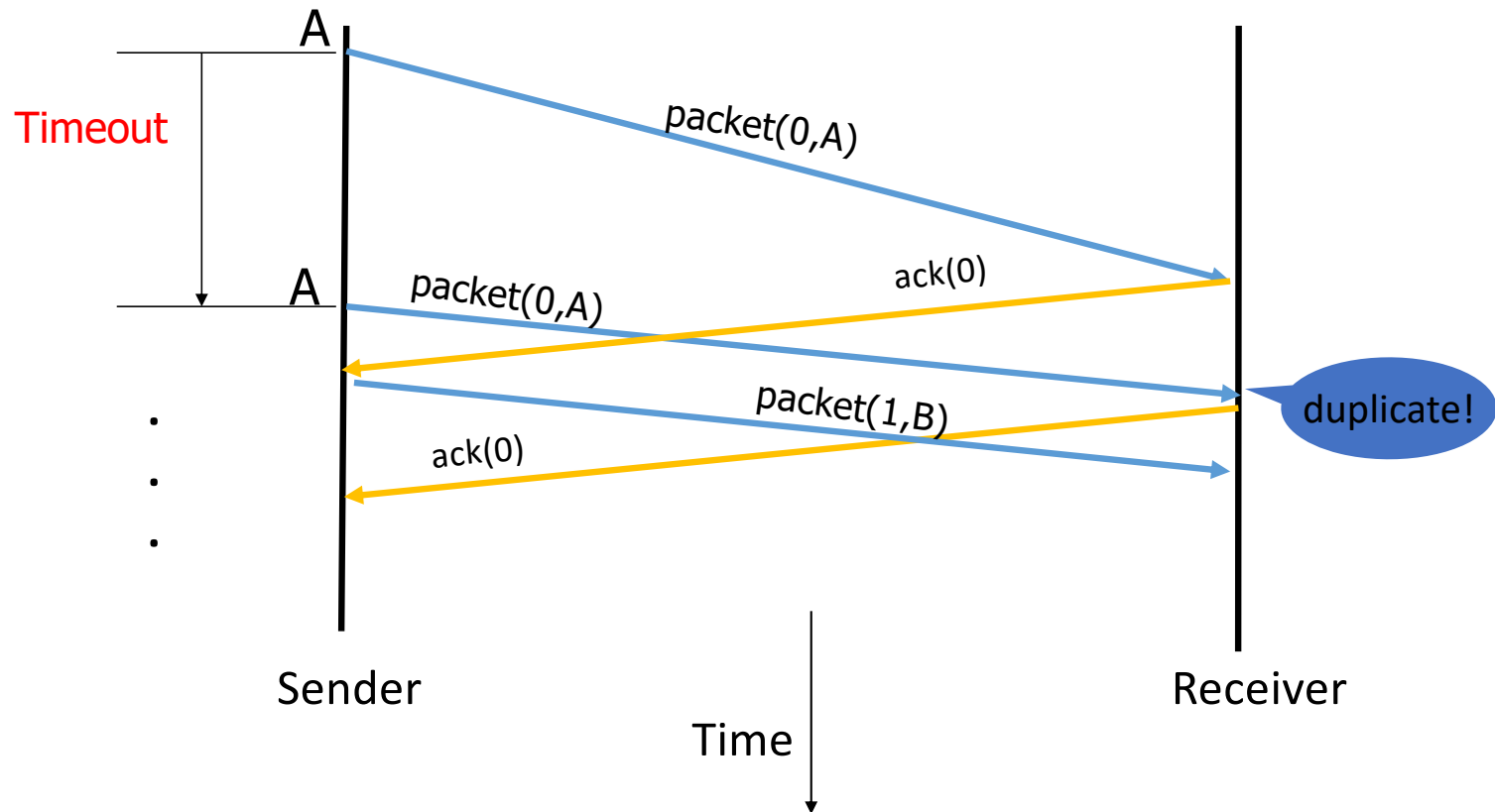
Dealing with Loss: Timeouts

- Can lead to **duplicate** data packets or ACKs, but sequence numbers let us identify these



Dealing with Loss: Timeouts

- Can lead to **duplicate** data packets or ACKs, but sequence numbers let us identify these
- packets may only be **delayed**, not **actually lost**



Complete Reliable Data Transfer Protocol Using the Stop-and-Wait Approach

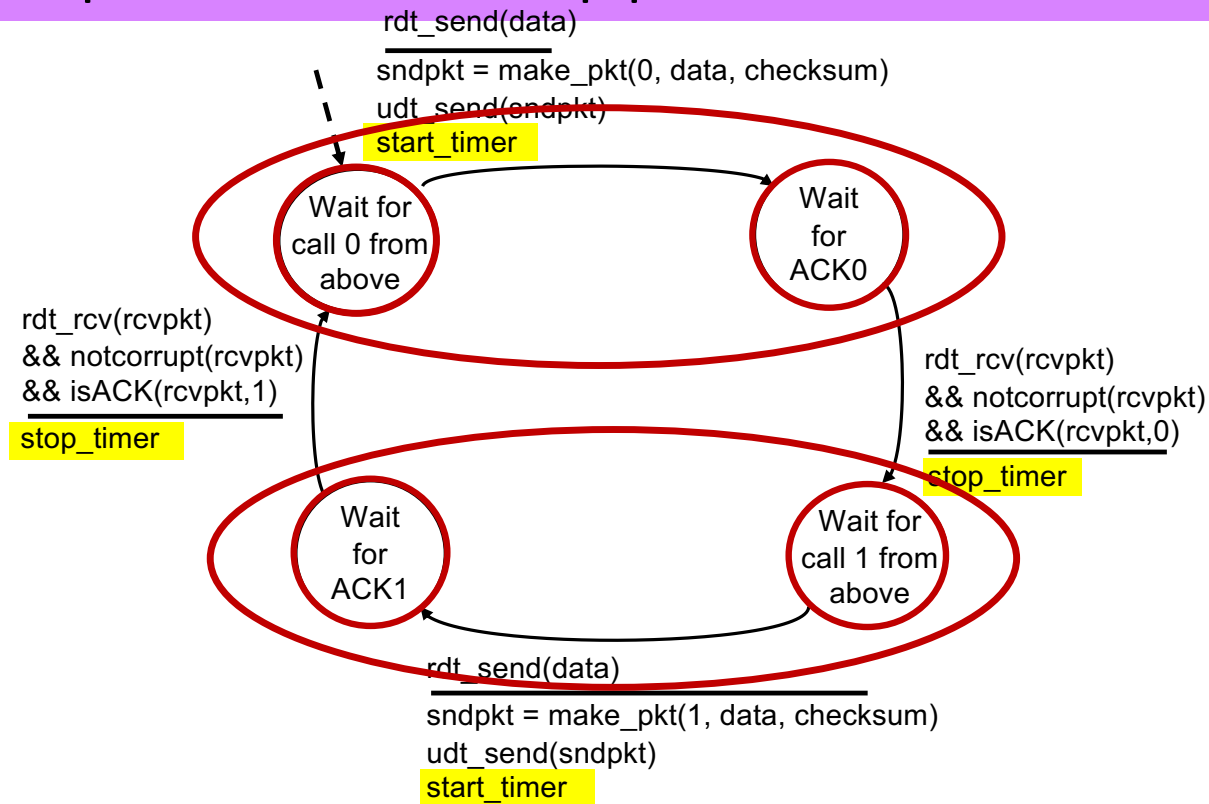
Sender:

- Set seq=0
- Wait for data to send **(1)**
- Create and send packet(seq, data)
- Start timer
- Wait for ACK or Timeout **(2)**
 - If ACK && not corrupt && ACK_seq == seq:
 - Stop timer
 - $seq = (seq+1) \% 2$
 - Wait for more data (1)
 - Else if Timeout:
 - Resend packet
 - Restart timer
 - Wait for ACK (2)

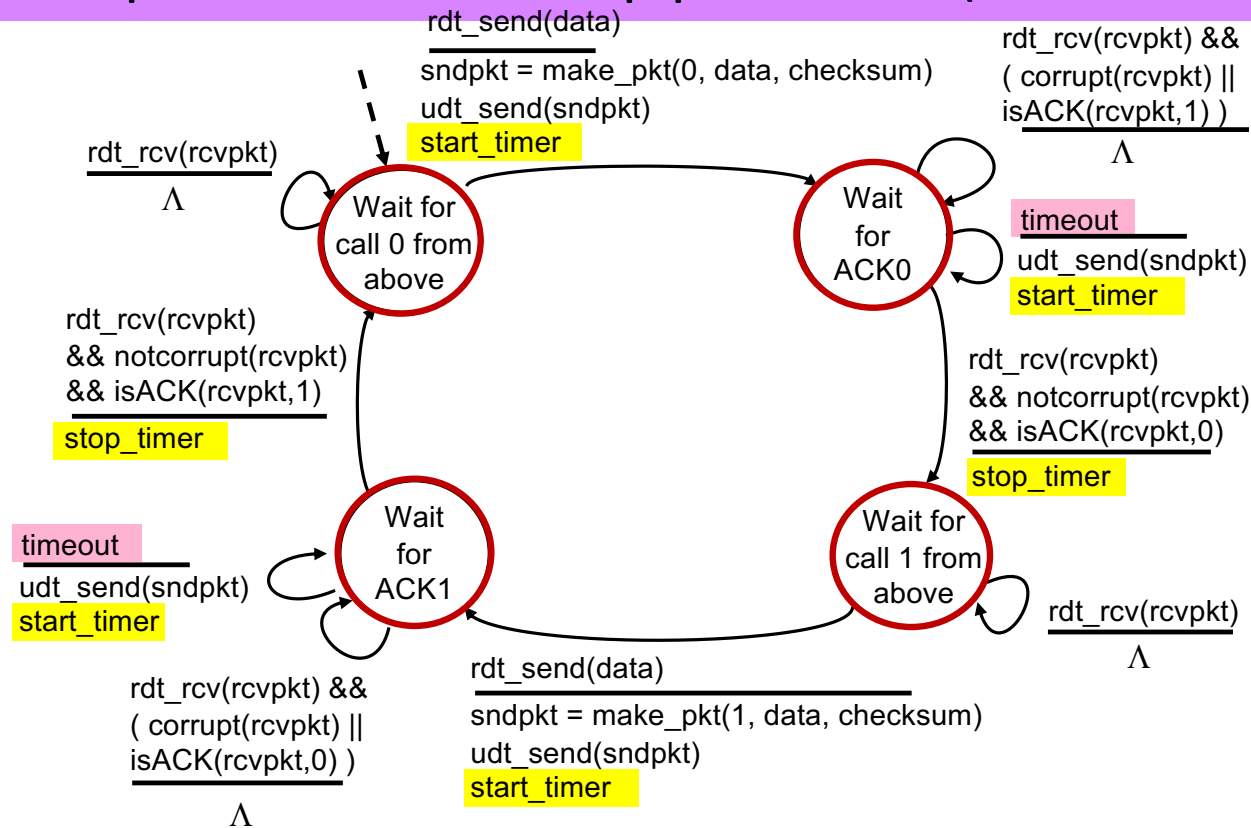
Receiver:

- Set seq=0
- Wait for packet from sender **(1)**
 - If not corrupt:
 - Send ACK(recvd_seq)
 - If recvd_seq == seq:
 - Extract and deliver to application
 - $seq = (seq+1) \% 2$
 - Wait for new packet from sender (1)

Complete Reliable Data Transfer Protocol Using the Stop-and-Wait Approach



Complete Reliable Data Transfer Protocol Using the Stop-and-Wait Approach (Sender Side)



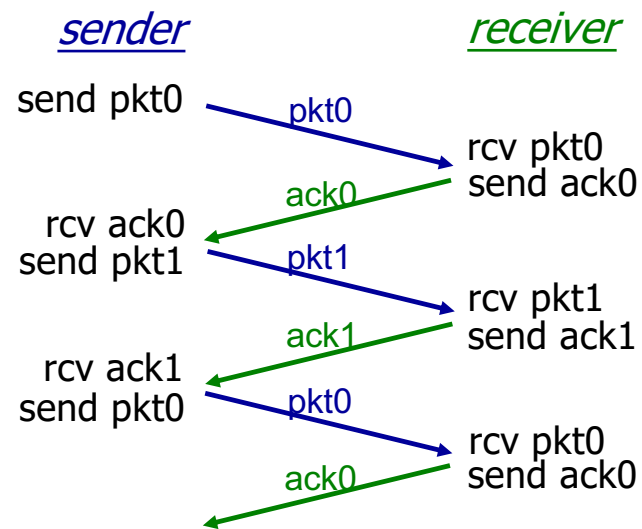
Reliable Data Transfer: Mechanism Summary

- **Checksums**: detect errors
- **ACKs/NAKs**: provide sender with feedback about what has been received
- **Retransmissions**: sender resends lost/corrupted packets
- **Sequence numbers**: identify packets, allow de-duplication
- **Timeouts**: detect (probable) loss, decide when to retransmit

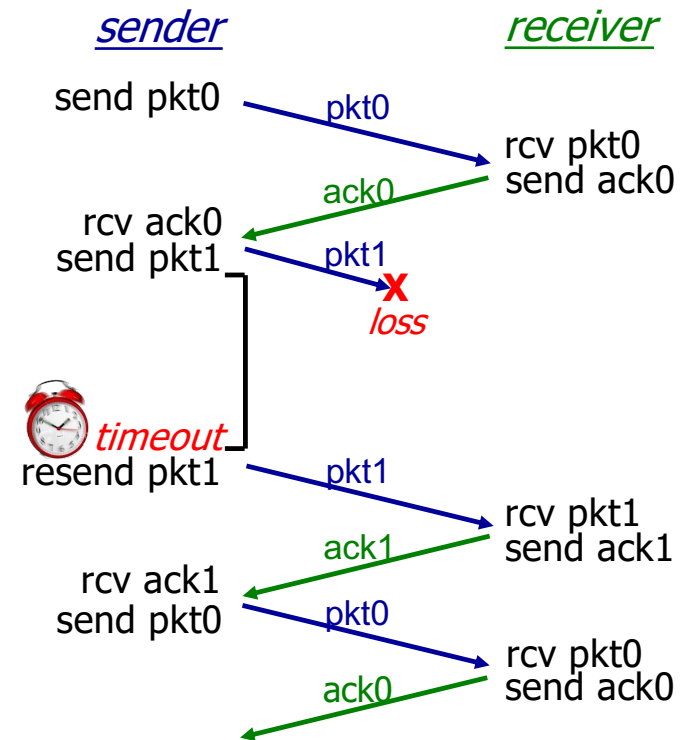
Summary

- Transport layer provides **logical process-to-process communication** (and other optional services) over network layer's best-effort host-to-host communication
- **UDP** offers lightweight **multiplexing/demultiplexing** + **error checking**
- We can implement **reliable data transfer** on top of unreliable network channel that corrupts and loses packets
 - Next time, we'll see how to make this efficient

Stop-and-Wait: Example Executions

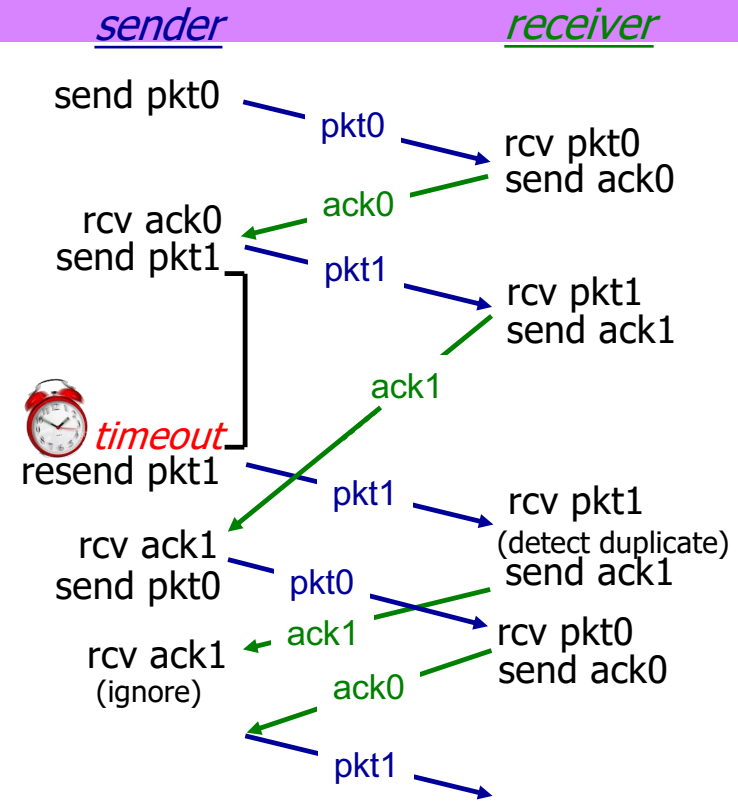
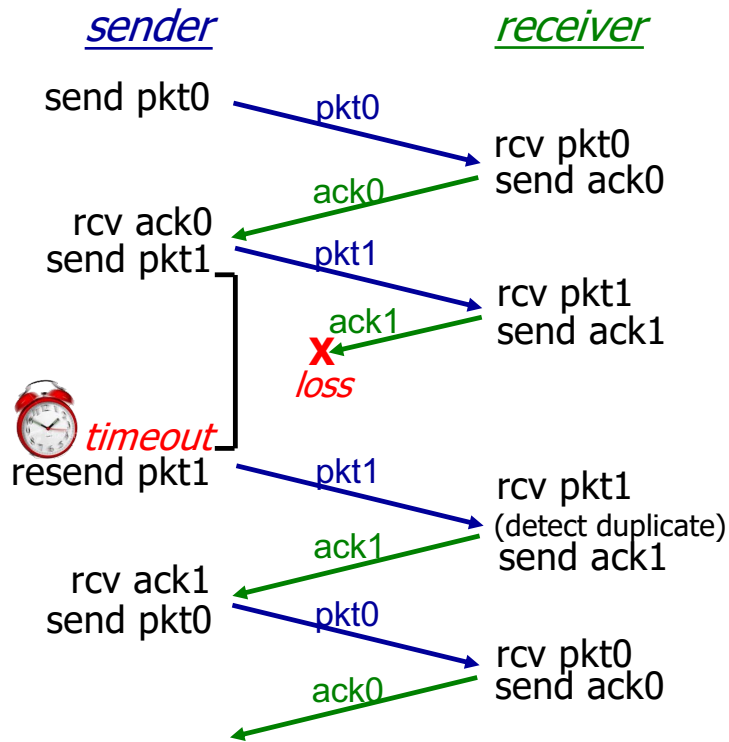


(a) no loss



(b) packet loss

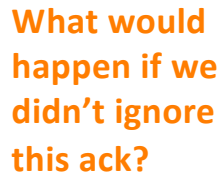
Stop-and-Wait: Example Executions



Complete Reliable Data Transfer Protocol Using the Stop-and-Wait Approach

- **Why don't we use duplicate ACKs as evidence of loss?**
- In the preliminary versions of this protocol that we discussed Monday, we said the receiver could send a NAK or duplicate ACK after receiving a corrupted message to prompt the sender to retransmit. The final version relies on timeouts only. **Why?**
- i.e. Why not retransmit immediately in the case where ACK seq doesn't match our expected seq?

Stop-and-Wait RDT Operation: Premature Timeouts



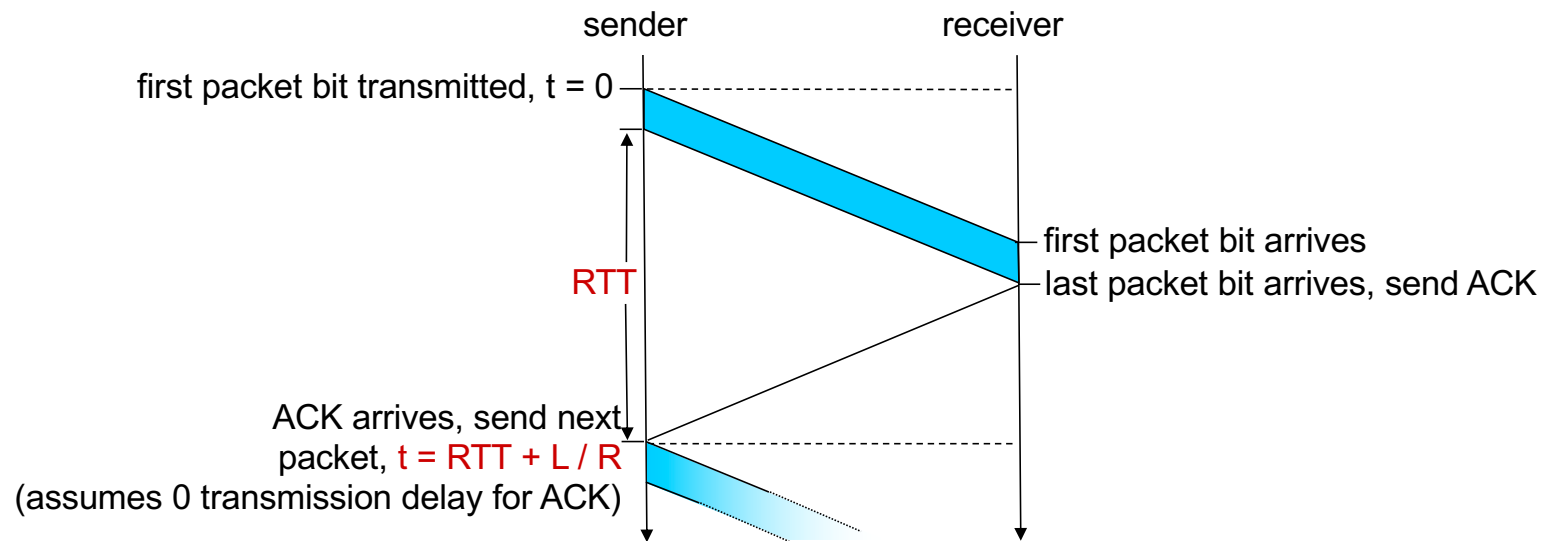
- If** the pkt was really lost,
retransmitting on duplicate ack lets
us respond faster.
- But**, if not, generates a lot of extra
traffic (without additional protocol
changes)
- And**, sender cannot tell which case
it's in

Analyzing Stop-and-Wait

- Our protocol is finally correct...but is it good?

Analyzing Stop-and-Wait

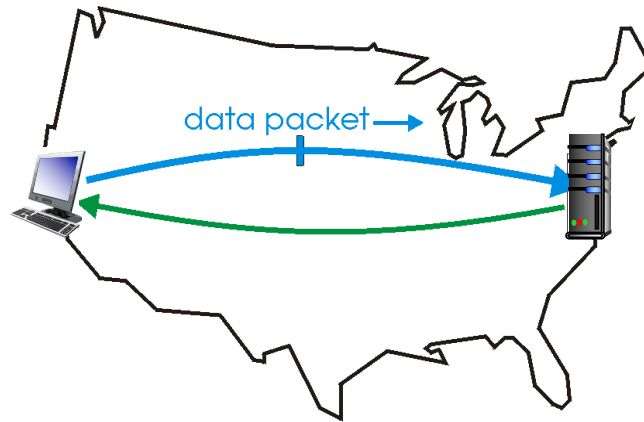
- Our protocol is finally correct...but is it good?



Analyzing Stop-and-Wait

- Our protocol is finally correct...but is it good?

RTT ~60ms
Bandwidth ~1Gbps
Segment size ~1500bytes



(a) a stop-and-wait protocol in operation

Transmission delay:

$$(1500 \times 8) / (10^9) = 12 \text{ microsec}$$

Assuming negligible transmission delay for ACK and no queuing/processing, we wait **60.012 ms** to send next segment

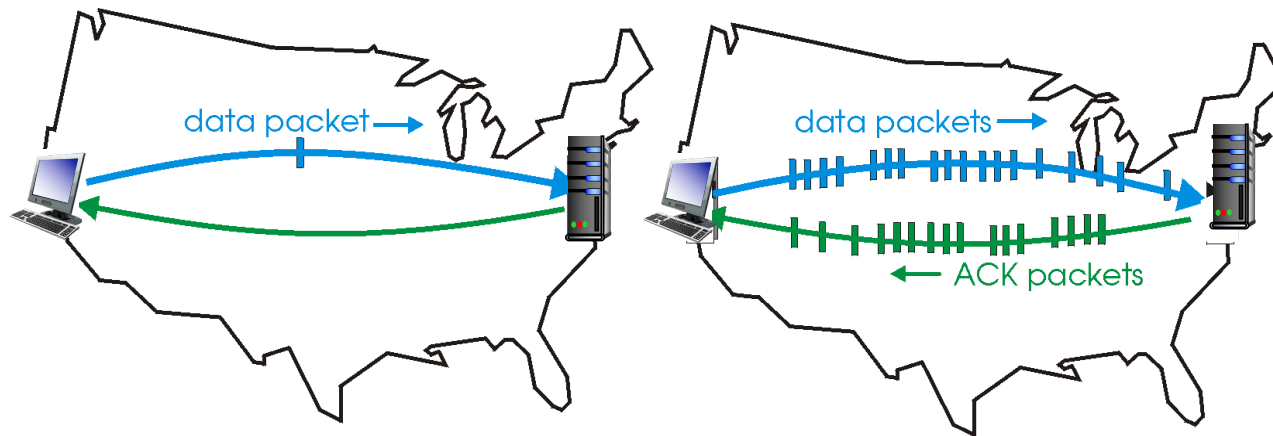
How long would it take to transfer a 1Gb file at this rate?

Improving Performance

- **How can we improve our protocol to get better performance?**
 - Think back to HTTP discussion...what strategies did we use there?

Improving Performance

- **How can we improve our protocol to get better performance?**
 - **Pipelining** – instead of stopping and waiting after **every** packet, we can send **multiple** packets to “fill the pipe” before waiting for acknowledgment



(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

Pipelined Reliable Data Transfer

- Instead of only 1 unacknowledged packet, sender can have **up to N unacknowledged packets** at a time
 - N packets “in-flight”
 - Requires more than 1-bit sequence number
- Sender needs to **buffer** sent but not-yet-acknowledged packets so they can be retransmitted as needed
- Sender buffer typically operates as a “**sliding window**”

