



NTNU – Trondheim
Norwegian University of
Science and Technology

Application Layer – Video and Sockets

TTM4200 – Computer Networks

individual Readiness Assurance Test

- Find the right answer **alone**.
- Choose the answer that fits **best**.
- **One** handwritten A4 page as helping material.

20:00

team Readiness Assurance Test

- Find the right answer **in teams** (**breakout** rooms).
- Choose the answer that fits **best**.
- **One** handwritten A4 page as helping material.
- **Discuss!**

20:00

Feedback Friday

- RAT
 - 70% is 2 points – made an excel mistake
- Unix Commands / Docker
- Cyber Security ;)
- Live Streaming vs. DASH: need to see
- Lab Issues
- UDP vs. TCP: To come
- Bittorent details
 - <https://bittorrent.org/bittorrentecon.pdf>
 - <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5451765>

Feedback Reference Group Meeting

What's working well

- Overall satisfaction is high and there hasn't been much negative feedback
- Lab setup is strongly appreciated, especially:
 - Continuous feedback in Jupyter
 - Small, iterative steps that let them progress incrementally
- Mentimeter is a positive addition, with the expectation set that Mentimeter questions will usually be addressed in the next lecture
- Group exercises in SMASH are appreciated
- Group structure seems to work well
- 20 Minutes for the RATs appreciated

Feedback Reference Group Meeting

Issues / friction points

- Early labs lacked a clear link between preparation material and lab tasks (“missing link” in the first labs)
- RAT questions feel ambiguous
 - Some questions are not clear enough
 - Question for more descriptive iRAT
- Some students feel the reading material didn’t prepare them well for the RATs, because:
 - The technical depth of the readings varied
 - The mapping felt unbalanced (e.g., it was felt that one paragraph out of 20 pages drove multiple RAT questions)
- Fixed group places in R22

Feedback Reference Group Meeting

Action Items

- RATs
 - Tried to align RAT questions better to readings
 - Monitor RAT performance and feedback
- Follow up on group settings via mentimeter
- Mentimeter Q&A
 - will be monitored during lecture best-effort
 - will be followed up in the next lecture in written form
- Random group placing in R22
- Group contract mandatory part to be included in the first lab delivery

Time for a break

we restart at 11:15

Join at [menti.com](https://www.menti.com) | use code 1738 0268

NTNU
Norwegian University of Science and Technology

Instructions

Go to
www.menti.com

Enter the code

1738 0268



Or use QR code



Lab 3

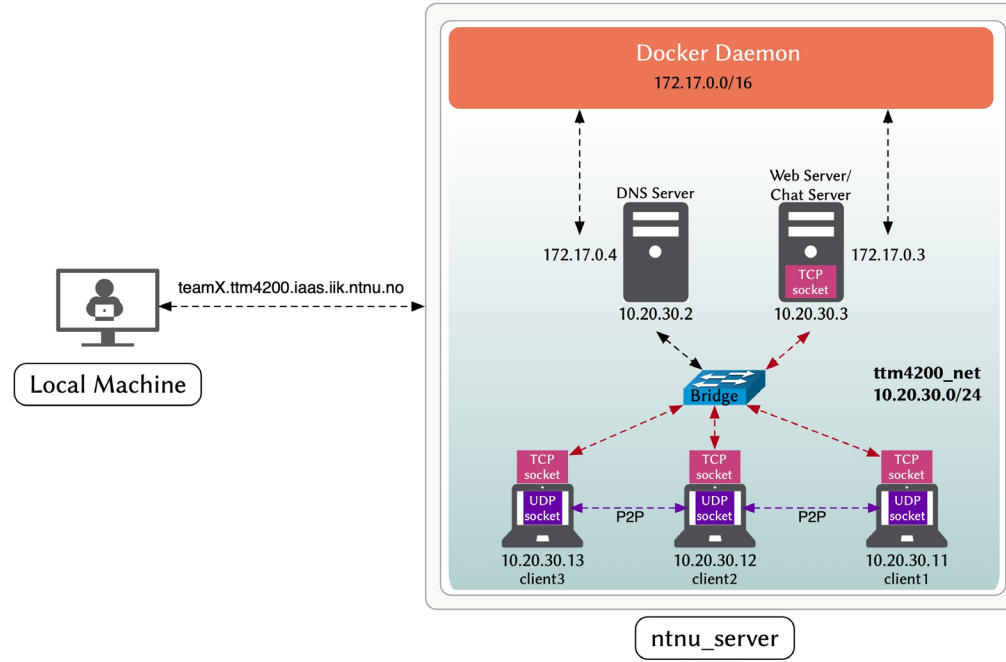
Sockets, socket programming!

- Same topology as in Lab 2
- Make a chatroom with TCP sockets (using a Python socket API)
- Server-client architecture (TCP server, TCP clients)

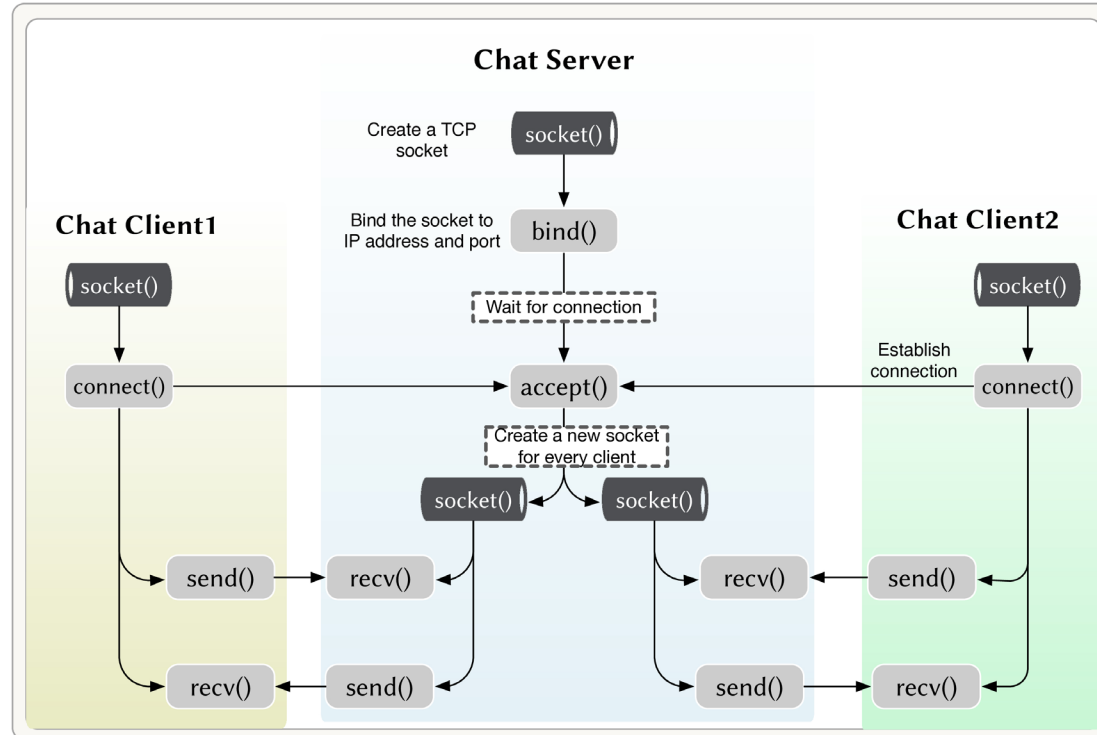
Deliverable 1

- Report tasks, i.e., what to answer and include, will be clearly stated in Lab 4
- You will have to include the code from lab 3 in the report. Just do the lab and follow the milestones, then you are good

Lab 3



Lab 3



Readings for next week

Book pages

- 212-242 (Sec. 3.1, 3.2, 3.3 & 3.4.1)
- Optional
 - 243-256 (Sec. 3.4.2, 3.4.3, 3.4.4, reviewed in class)

Key concepts

Logical communication

Segments vs. Datagrams

Best-effort

Reliable vs. unreliable transfer

Congestion Control

Multiplexing & demultiplexing

Transport layer: overview

Our goal:

- understand principles behind transport layer services:
 - multiplexing, demultiplexing
 - reliable data transfer
 - flow control
 - congestion control
- learn about Internet transport layer protocols:
 - UDP: connectionless transport
 - Next week {
 - TCP: connection-oriented reliable transport
 - TCP congestion control

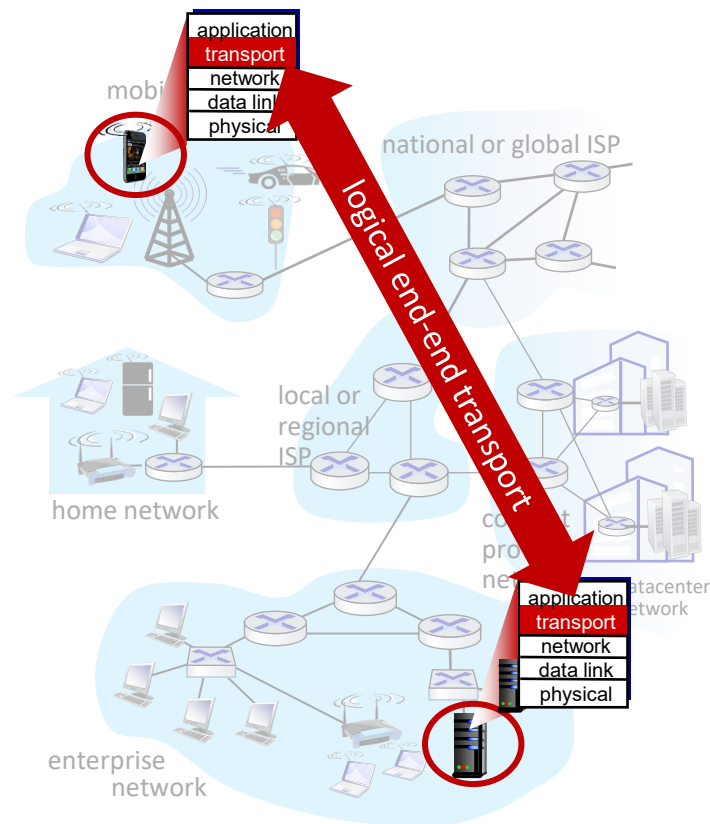
Transport layer: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality



Transport services and protocols

- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
 - sender: breaks application messages into *segments*, passes to network layer
 - receiver: reassembles segments into messages, passes to application layer
- Two principal transport protocols available to Internet applications
 - TCP, UDP



Transport vs. network layer services and protocols

- **network layer:** logical communication between *hosts*
- **transport layer:** logical communication between *processes*
 - relies on, enhances, network layer services

household analogy:

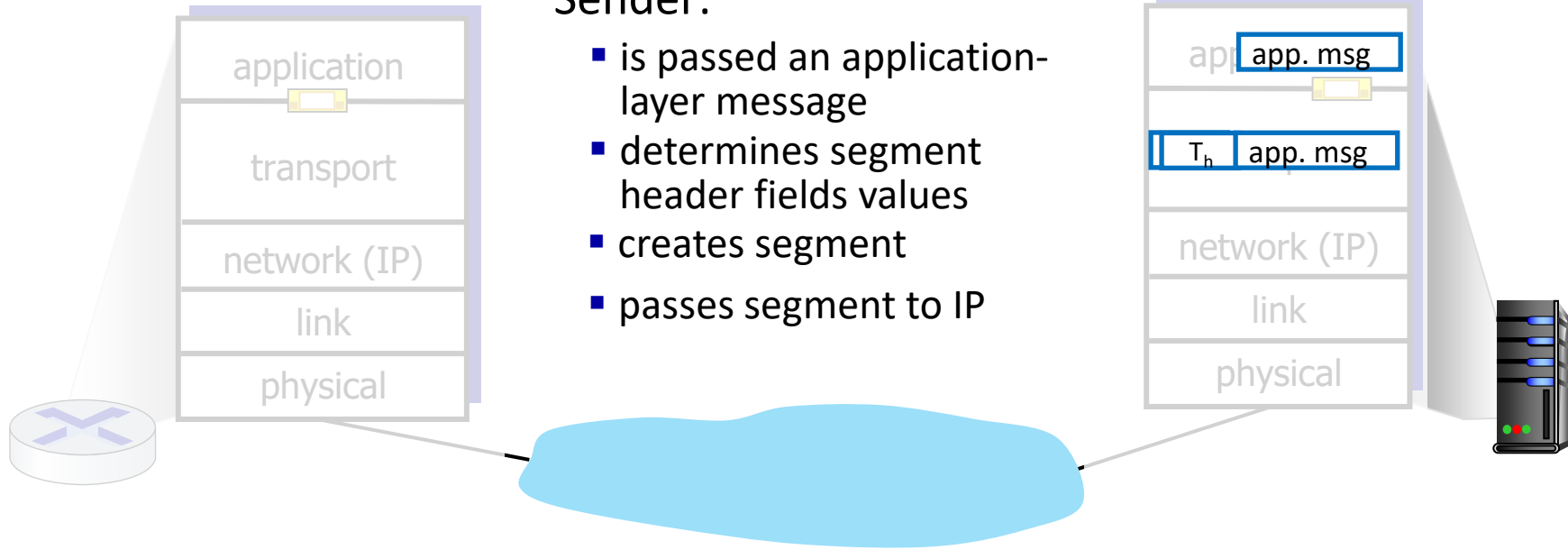
12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Transport Layer Actions

Sender:

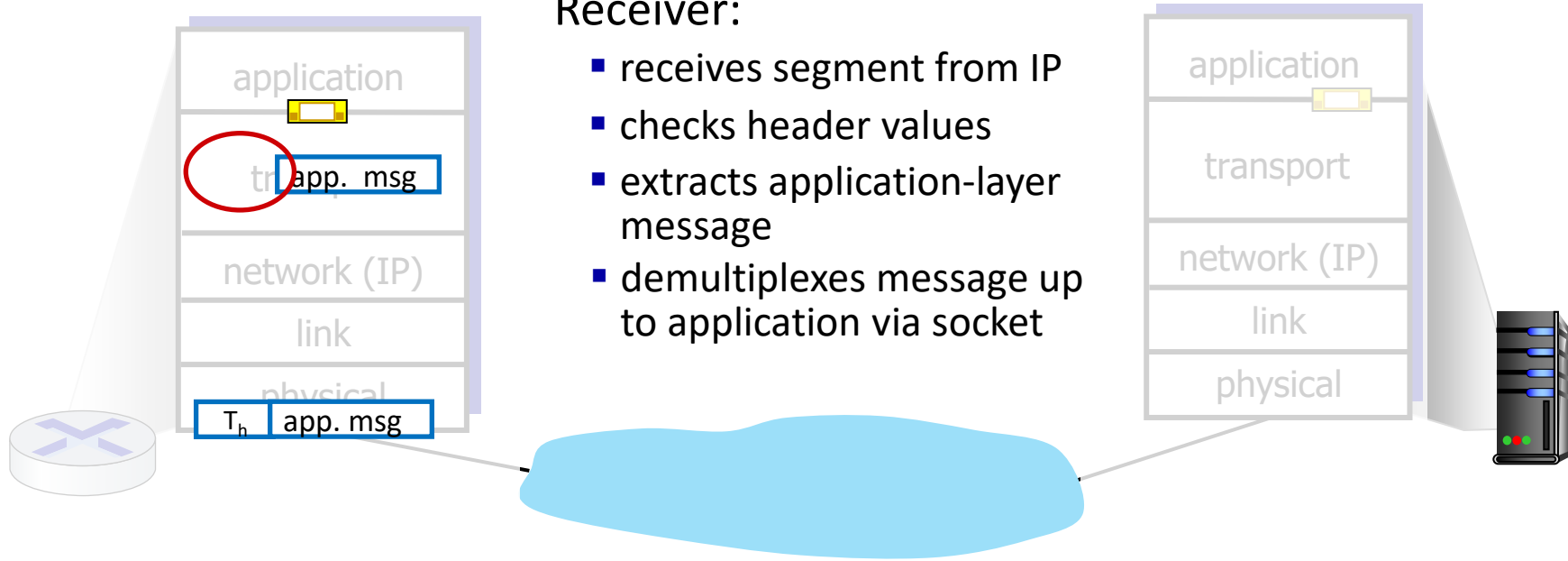
- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP



Transport Layer Actions

Receiver:

- receives segment from IP
- checks header values
- extracts application-layer message
- demultiplexes message up to application via socket



Two principal Internet transport protocols

■ **TCP:** Transmission Control Protocol

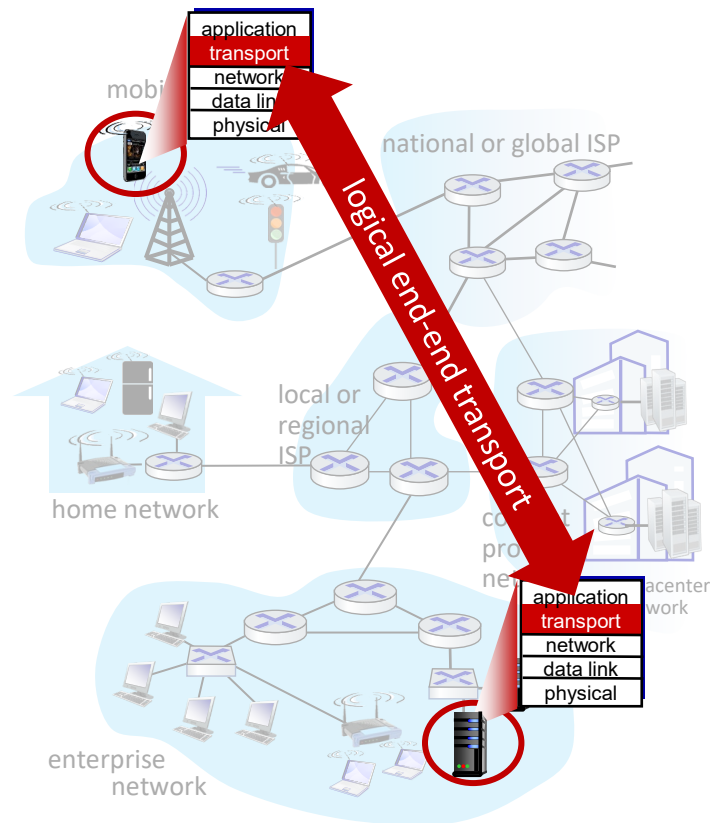
- reliable, in-order delivery
- congestion control
- flow control
- connection setup

■ **UDP:** User Datagram Protocol

- unreliable, unordered delivery
- no-frills extension of "best-effort" IP

■ services not available:

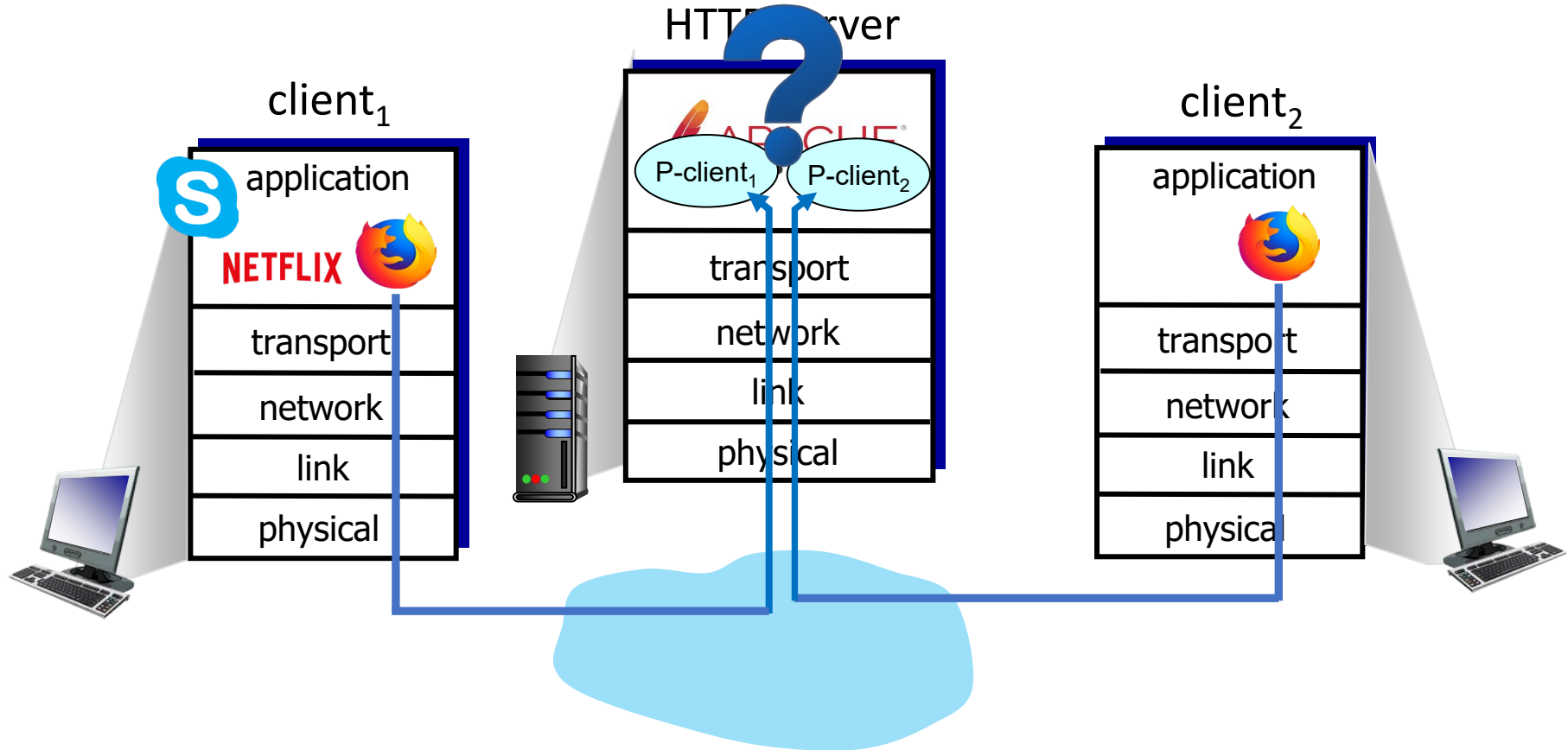
- delay guarantees
- bandwidth guarantees



Chapter 3: roadmap

- Transport-layer services
- **Multiplexing and demultiplexing**
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality





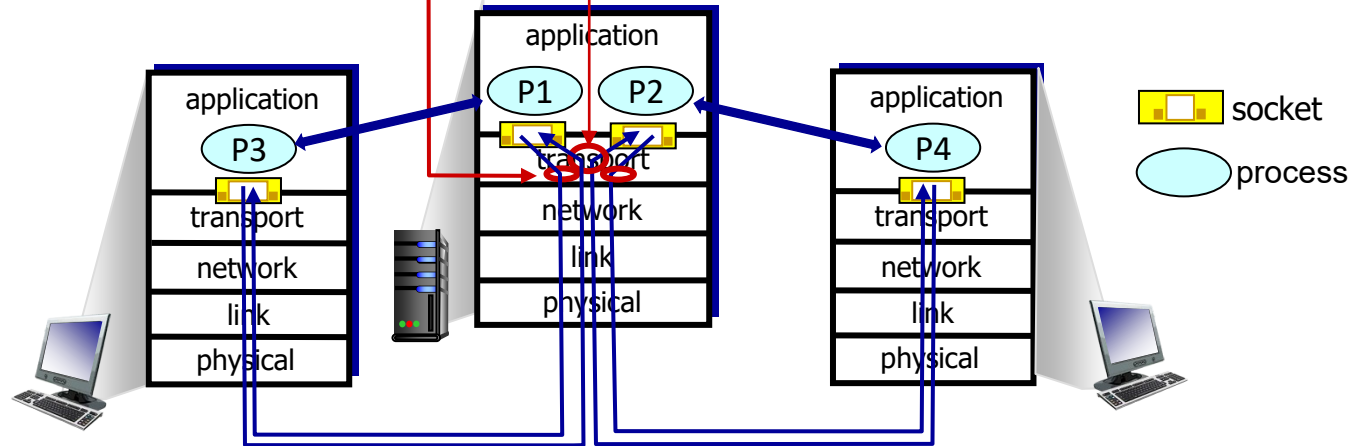
Multiplexing/demultiplexing

multiplexing at sender:

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

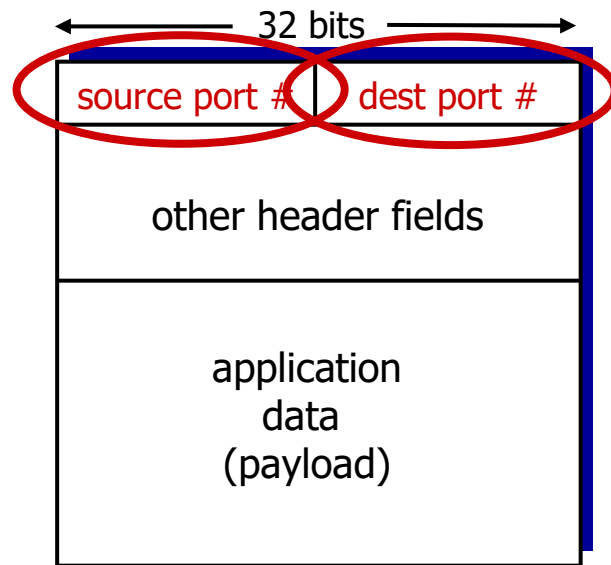
demultiplexing at receiver:

use header info to deliver received segments to correct socket



How demultiplexing works

- host receives IP datagrams
 - each datagram has source IP address, destination IP address
 - each datagram carries one transport-layer segment
 - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

Summary

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using destination port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers
- Multiplexing/demultiplexing happen at *all* layers

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- **Connectionless transport: UDP**
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality



UDP: User Datagram Protocol

- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
 - lost
 - delivered out-of-order to app
- *connectionless*:
 - no handshaking between UDP sender, receiver
 - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
 - UDP can blast away as fast as desired!
 - can function in the face of congestion

UDP: User Datagram Protocol

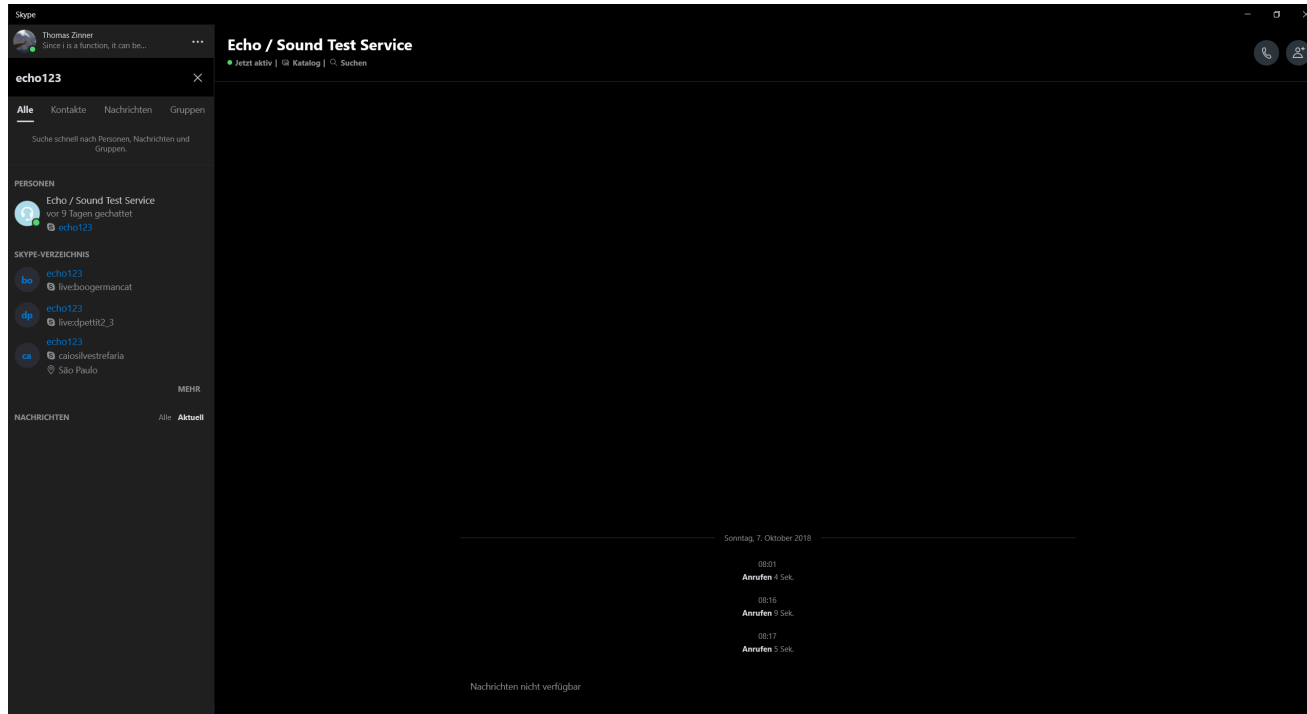
- UDP use:
 - streaming multimedia apps (loss tolerant, rate sensitive)
 - DNS
 - SNMP
 - HTTP/3
- if reliable transfer needed over UDP (e.g., HTTP/3):
 - add needed reliability at application layer
 - add congestion control at application layer

Summary: UDP

- “no frills” protocol:
 - segments may be lost, delivered out of order
 - best effort service: “send and hope for the best”
- UDP has its plusses:
 - no setup/handshaking needed (no RTT incurred)
 - can function when network service is compromised
 - helps with reliability (checksum)
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

Examples: UDP vs. TCP

Voice over IP



Voice over IP

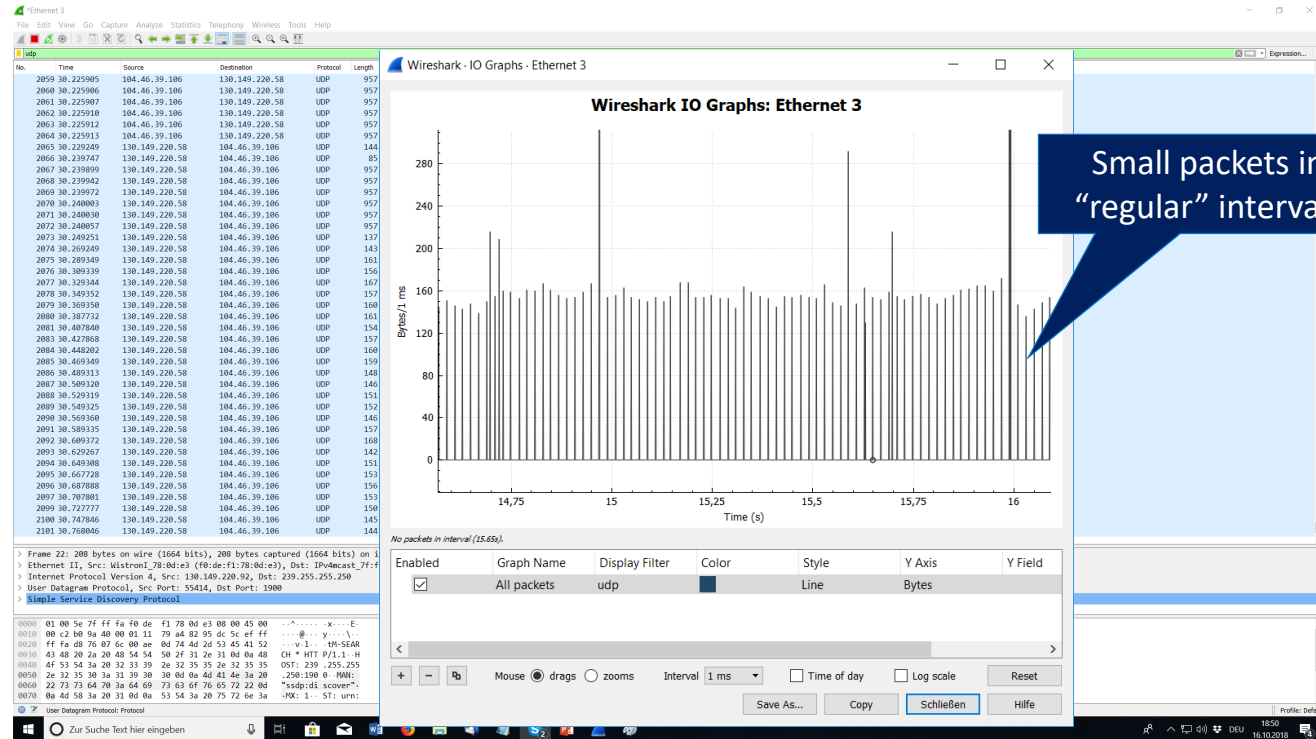
The image shows a Wireshark packet capture of Voice over IP traffic. The main display area shows a list of 2101 packets, all of which are UDP packets from 130.149.220.58 to 144.45.72.23888. The packet details pane shows the structure of a UDP packet, including the Ethernet II header, Internet Protocol Version 4 header, and User Datagram Protocol header. The packet bytes pane shows the raw data of the packet, including the Ethernet II header, Internet Protocol Version 4 header, and User Datagram Protocol header.

No.	Time	Source	Destination	Protocol	Length	Info
2059	30.225905	104.46.39.106	130.149.220.58	UDP	957	23888 → 4572 Len=915
2060	30.225906	104.46.39.106	130.149.220.58	UDP	957	23888 → 4572 Len=915
2061	30.225907	104.46.39.106	130.149.220.58	UDP	957	23888 → 4572 Len=915
2062	30.225910	104.46.39.106	130.149.220.58	UDP	957	23888 → 4572 Len=915
2063	30.225912	104.46.39.106	130.149.220.58	UDP	957	23888 → 4572 Len=915
2064	30.225913	104.46.39.106	130.149.220.58	UDP	957	23888 → 4572 Len=915
2065	30.225920	130.149.220.58	104.45.72.23888	UDP	144	4572 → 23888 Len=182
2066	30.230747	130.149.220.58	104.45.72.23888	UDP	85	4572 → 23888 Len=43
2067	30.230899	130.149.220.58	104.45.72.23888	UDP	957	4572 → 23888 Len=915
2068	30.230942	130.149.220.58	104.45.72.23888	UDP	957	4572 → 23888 Len=915
2069	30.230972	130.149.220.58	104.45.72.23888	UDP	957	4572 → 23888 Len=915
2070	30.240003	130.149.220.58	104.45.72.23888	UDP	957	4572 → 23888 Len=915
2071	30.240030	130.149.220.58	104.45.72.23888	UDP	957	4572 → 23888 Len=915
2072	30.240057	130.149.220.58	104.45.72.23888	UDP	957	4572 → 23888 Len=915
2073	30.240251	130.149.220.58	104.45.72.23888	UDP	137	4572 → 23888 Len=95
2074	30.269249	130.149.220.58	104.45.72.23888	UDP	143	4572 → 23888 Len=101
2075	30.289349	130.149.220.58	104.45.72.23888	UDP	161	4572 → 23888 Len=119
2076	30.309339	130.149.220.58	104.45.72.23888	UDP	156	4572 → 23888 Len=114
2077	30.329344	130.149.220.58	104.45.72.23888	UDP	167	4572 → 23888 Len=125
2078	30.349352	130.149.220.58	104.45.72.23888	UDP	157	4572 → 23888 Len=115
2079	30.369350	130.149.220.58	104.45.72.23888	UDP	168	4572 → 23888 Len=118
2080	30.387732	130.149.220.58	104.45.72.23888	UDP	161	4572 → 23888 Len=119
2081	30.407840	130.149.220.58	104.45.72.23888	UDP	154	4572 → 23888 Len=112
2083	30.427868	130.149.220.58	104.45.72.23888	UDP	157	4572 → 23888 Len=115
2084	30.448202	130.149.220.58	104.45.72.23888	UDP	160	4572 → 23888 Len=118
2085	30.469349	130.149.220.58	104.45.72.23888	UDP	159	4572 → 23888 Len=117
2086	30.489313	130.149.220.58	104.45.72.23888	UDP	148	4572 → 23888 Len=106
2087	30.509320	130.149.220.58	104.45.72.23888	UDP	146	4572 → 23888 Len=104
2088	30.529319	130.149.220.58	104.45.72.23888	UDP	151	4572 → 23888 Len=109
2089	30.549325	130.149.220.58	104.45.72.23888	UDP	152	4572 → 23888 Len=110
2090	30.569360	130.149.220.58	104.45.72.23888	UDP	146	4572 → 23888 Len=104
2091	30.589335	130.149.220.58	104.45.72.23888	UDP	157	4572 → 23888 Len=115
2092	30.609372	130.149.220.58	104.45.72.23888	UDP	168	4572 → 23888 Len=126
2093	30.629267	130.149.220.58	104.45.72.23888	UDP	142	4572 → 23888 Len=100
2094	30.649308	130.149.220.58	104.45.72.23888	UDP	151	4572 → 23888 Len=109
2095	30.667728	130.149.220.58	104.45.72.23888	UDP	153	4572 → 23888 Len=111
2096	30.687888	130.149.220.58	104.45.72.23888	UDP	156	4572 → 23888 Len=114
2097	30.707901	130.149.220.58	104.45.72.23888	UDP	153	4572 → 23888 Len=111
2099	30.727777	130.149.220.58	104.45.72.23888	UDP	150	4572 → 23888 Len=108
2100	30.747846	130.149.220.58	104.45.72.23888	UDP	145	4572 → 23888 Len=103
2101	30.768046	130.149.220.58	104.45.72.23888	UDP	144	4572 → 23888 Len=102

Frame 22: 208 bytes on wire (1664 bits), 208 bytes captured (1664 bits) on interface 0
Ethernet II, Src: Wistron1_78:0d:e3 (f0:de:f1:78:0d:e3), Dst: IPv4mcast_7f:ff:fa (01:00:5e:7f:ff:fa)
Internet Protocol Version 4, Src: 130.149.220.92, Dst: 239.255.255.250
User Datagram Protocol, Src Port: 55416, Dst Port: 1900
Simple Service Discovery Protocol

0000 01 00 5e 7f ff fa f0 de f1 78 0d e3 00 00 45 00 ...x...E
0010 00 c2 b0 9a 40 00 01 11 79 a4 82 95 dc 5c ff ff ...@...y...
0020 ff fa d8 76 07 dc 0e 0e 0d 74 dd 2d 53 45 41 52 ...v1...TH-SEAR
0030 41 48 20 2a 20 40 45 54 5a 50 71 31 26 31 60 40 ...H *HT 1/1,1 - H
0040 4f 53 54 3a 20 32 33 39 2e 31 35 25 2e 32 35 35 ...OST: 239.255.255
0050 2e 32 35 30 3a 31 39 30 30 00 0a 4d 41 4e 3a 20 ...250:190 0 -HW:
0060 22 73 73 64 70 3a 04 09 03 6f 76 65 72 22 64 ..."ssdp:190:65"
0070 0a 4d 58 3a 20 31 0d 0a 53 54 3a 20 75 72 6e 3a ...H: 1 - ST: urn:

Voice over IP



VoD Streaming



https://www.youtube.com/watch?v=xQ_IQS3VKjA&t

VoD Streaming

Capturing from Ethernet 3

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter: <Ctrl> F Expression...

No.	Time	Source	Destination	Protocol	Length	Info
29340	49.964955	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=33986548 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29341	49.964997	130.149.220.58	74.125.11.89	TCP	54	52668 → 443 [ACK] Seq=33886 Ack=33987980 Win=23616 Len=0
29342	49.965210	74.125.11.89	130.149.220.58	TLSv1.2	1486	Application Data [TCP segment of a reassembled PDU]
29343	49.965212	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=33990412 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29344	49.965213	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=33990844 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29345	49.965214	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=33992276 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29346	49.965216	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=33993788 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29347	49.965217	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=33995140 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29348	49.965218	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=33996572 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29349	49.965220	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=33998004 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29350	49.965221	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=33999436 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29351	49.965222	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34000868 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29352	49.965264	130.149.220.58	74.125.11.89	TCP	54	52668 → 443 [ACK] Seq=33886 Ack=34002300 Win=23616 Len=0
29353	49.965385	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34002300 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29354	49.965387	74.125.11.89	130.149.220.58	TLSv1.2	1486	Application Data [TCP segment of a reassembled PDU]
29355	49.965388	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34003104 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29356	49.965389	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34004536 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29357	49.965390	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34005968 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29358	49.965391	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34007400 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29359	49.965392	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34008832 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29360	49.965393	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34010264 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29361	49.965395	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34011696 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29362	49.965396	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34013128 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29363	49.965440	130.149.220.58	74.125.11.89	TCP	54	52668 → 443 [ACK] Seq=33886 Ack=34014560 Win=23616 Len=0
29364	49.965900	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34016020 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29365	49.965902	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34017452 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29366	49.965903	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34018884 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29367	49.965904	74.125.11.89	130.149.220.58	TLSv1.2	1486	Application Data [TCP segment of a reassembled PDU]
29368	49.965905	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34020316 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29369	49.965907	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34021748 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29370	49.965908	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34023180 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29371	49.965909	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34024612 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29372	49.965910	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34026044 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29373	49.965911	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34027476 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29374	49.965954	130.149.220.58	74.125.11.89	TCP	54	52668 → 443 [ACK] Seq=33886 Ack=34028908 Win=23616 Len=0
29375	49.966792	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34030340 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29376	49.966793	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34031772 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29377	49.966794	74.125.11.89	130.149.220.58	TCP	1486	443 → 52668 [ACK] Seq=34033204 Ack=33886 Win=873 Len=1432 [TCP segment of a reassembled PDU]
29378	49.966795	74.125.11.89	130.149.220.58	TLSv1.2	1404	Application Data
29379	49.966798	130.149.220.58	74.125.11.89	TCP	54	52668 → 443 [ACK] Seq=33886 Ack=34034640 Win=23616 Len=0
29380	50.176701	HueltetP_bc:01:33	Broadcast	ARP	60	who has 130.149.220.15? Tell 130.149.220.8

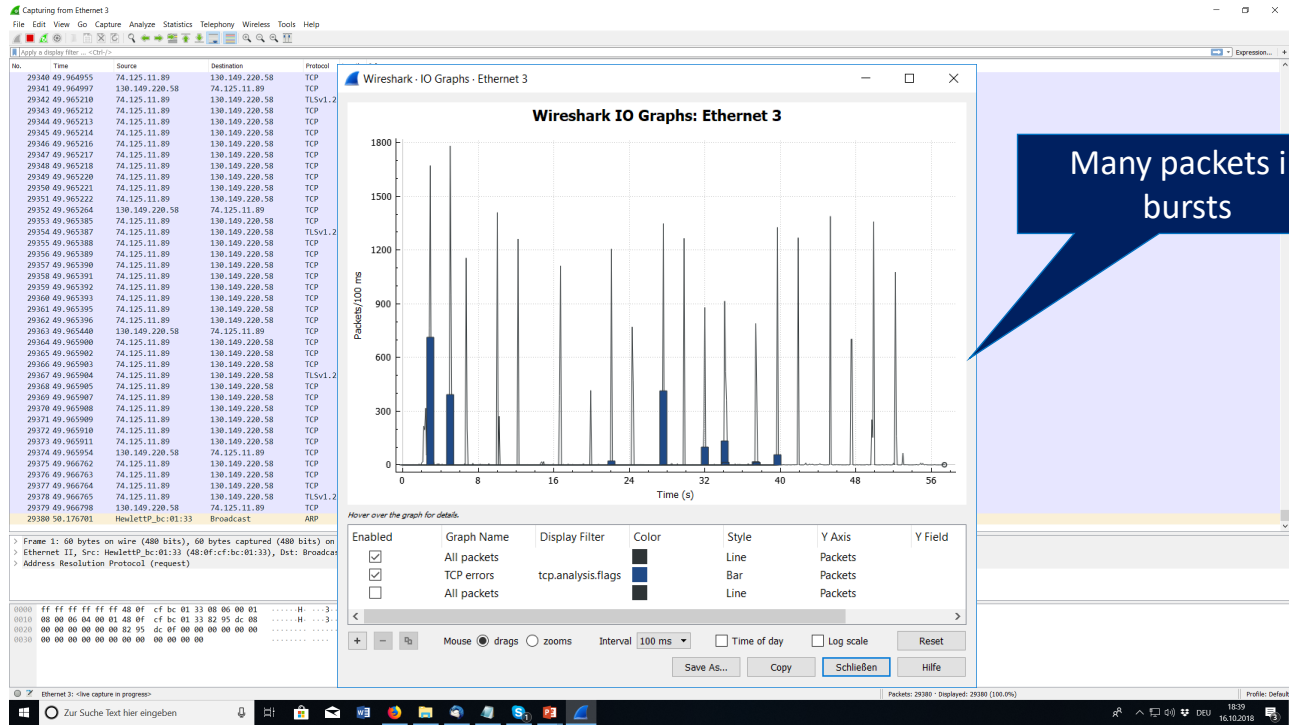
Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0
Ethernet II, Src: HueltetP_bc:01:33 (48:0f:cf:bc:01:33), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
Address Resolution Protocol (request)

0000 ff ff ff ff ff ff 48 0f cf bc 01 33 00 00 00 01H...3...
0010 00 00 00 00 00 00 48 0f cf bc 01 33 02 00 00 00H...3...
0020 00 00 00 00 00 00 82 95 dc 0f 00 00 00 00 00
0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Ethernet 3: <live capture in progress> Packet: 29380 - Displayed: 29380 (100.0%) Profile: Default

Zur Suche Text hier eingeben

VoD Streaming



Enough for today

we continue Thursday at 10:15

Join at menti.com | use code **5217 6469**

Instructions

Go to
www.menti.com

Enter the code

5217 6469



Or use QR code



Lab 3

- Incredible advancement in the labs
- You are doing very well!



> We have a good interaction in the tRAT group



> We have a good interaction in the lab groups



> We have a good interaction with the TAs



> RATs contribute to my learning



> Group discussions (tRAT, Exercises) contribute to my learning



> Labs contribute to my learning



RATS

Question 1: What is a primary purpose of a Content Delivery Network (CDN)?

Encrypt web traffic end-to-end (This is the best answer.)

2.0%

Reduce latency by serving content from servers closer to users

96.0%

Replace TCP with UDP for faster downloads

0.0%

Eliminate the need for HTTP

2.0%

iRAT

Question 3: In a Content Distribution Network (CDN), the best Cluster Selection strategy is:

to perform real-time measurements between clusters and clients. (This is the best answer.)

66.0%

always to select the geographically closest cluster, based on the client's local DNS server (LDNS) IP address.

24.0%

to use DASH with different bitrate versions of the video file.

6.0%

always to select the geographically closest cluster, based on the client's IP address.

4.0%



Question 4: Socket programming:

allows implementing both client-server and P2P systems. (This is the best answer.)

66.0%

only allows implementing client-server systems.

4.0%

only allows implementing P2P systems.

0.0%

allows implementing both client-server and P2P systems in the transport layer.

30.0%

Question 5: Creating a server application using UDP sockets requires:

a server socket bound to a given port. (This is the best answer.)

58.0%

a server socket bound to a given port for each received connection.

14.0%

a connection socket for receiving data and a server socket for sending data.

10.0%

a connection socket per used connection, for both sending and receiving data.

18.0%

iRAT

Question 7: In a client application using TCP sockets:

a single client socket is needed to connect to the server. (This is the best answer.)

28.0%

a client socket needs to be bound to a port before the TCP connection setup.

40.0%

a single client socket is needed to connect to a local port.

0.0%

a client socket needs to first establish a TCP connection before contacting the server.

32.0%

iRAT

Question 9: Assume a \textbf{pure} peer-to-peer application where TCP is being used. How many sockets will be used \textbf{in total} if there are 3 clients, each connected to one another (full mesh)? \textit{Hint: each client will receive and create independent connections, no connection reuse should be considered}

15 (This is the best answer.)

2.0%

6

66.0%

9

24.0%

3

8.0%



Question 9 - Discussion

- Question 9 was tricky, but _in my opinion_ solvable based on the readings
 - Required recapturing, understanding and application to a specific context
- Survey via Mentimeter
 - Was question 9 ok?



iRAT

Question 10: When a client creates an SMTP connection to the server "mail.ntnu.no", what happens first on the client's side?

The creation of a client TCP socket. (This is the best answer.)

50.0%

A DNS query to resolve "mail.ntnu.no".

24.0%

The creation of a server socket.

8.0%

A DNS query to resolve the root, top-level and authoritative servers for "mail.ntnu.no".

18.0%

Sockets vs Ports vs Connections

- IP address = which machine
- Port (0–65535) = which app/service on that machine
- Socket = the OS “handle/endpoint” your program uses to send/receive
- TCP connection = a stateful conversation identified by (src IP:port, dst IP:port, protocol)
- UDP = datagrams; no handshake/state like TCP (but you can still “connect” a UDP socket)
- `bind(local_ip, local_port)` = “use this local address/port for receiving and as my source endpoint”

Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- **Principles of reliable data transfer**
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

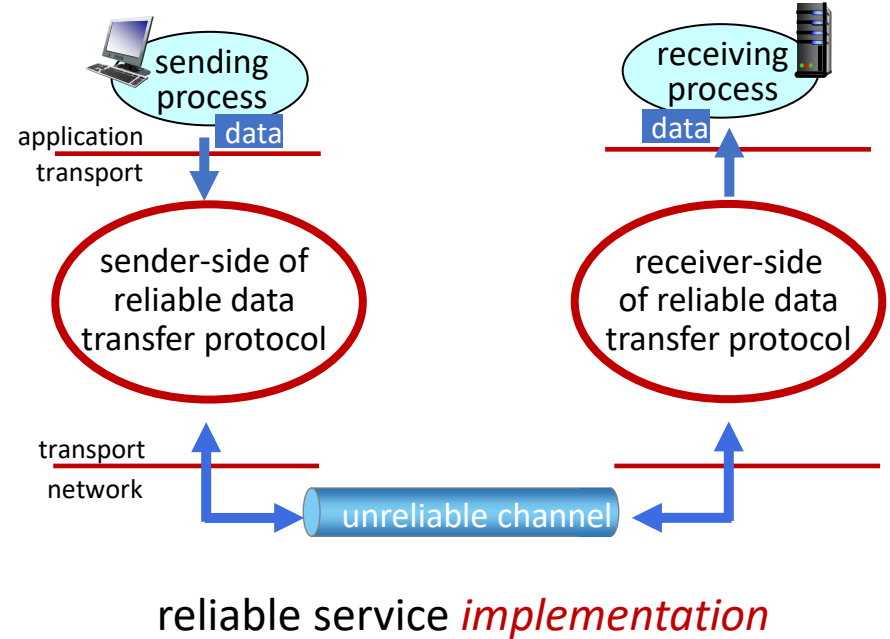
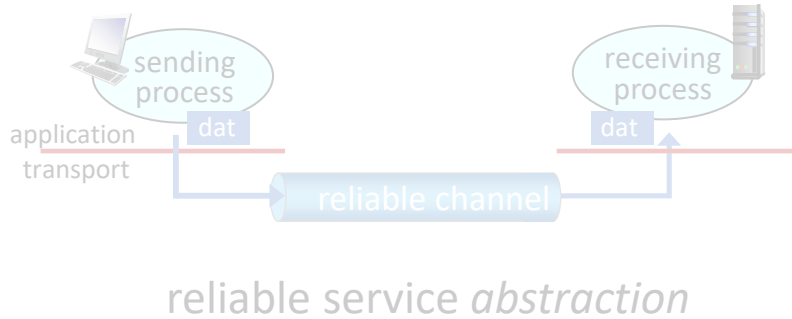


Principles of reliable data transfer



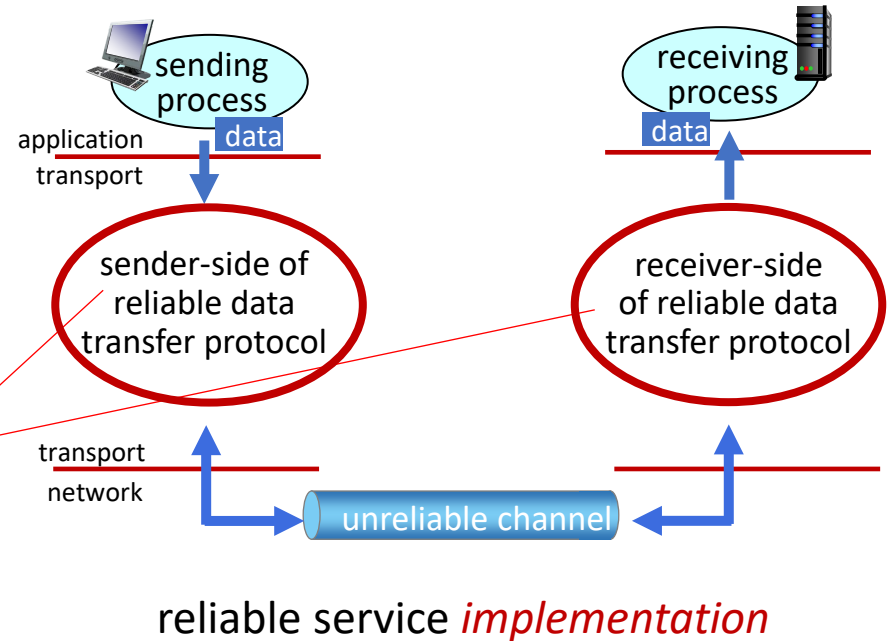
reliable service *abstraction*

Principles of reliable data transfer



Principles of reliable data transfer

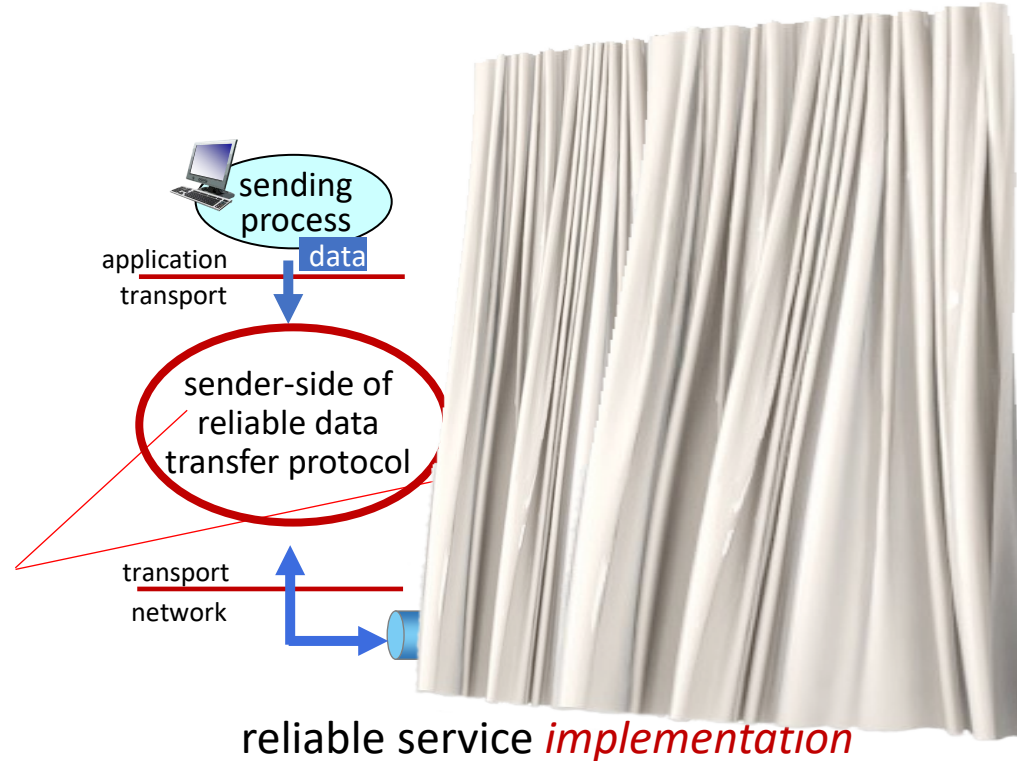
Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)



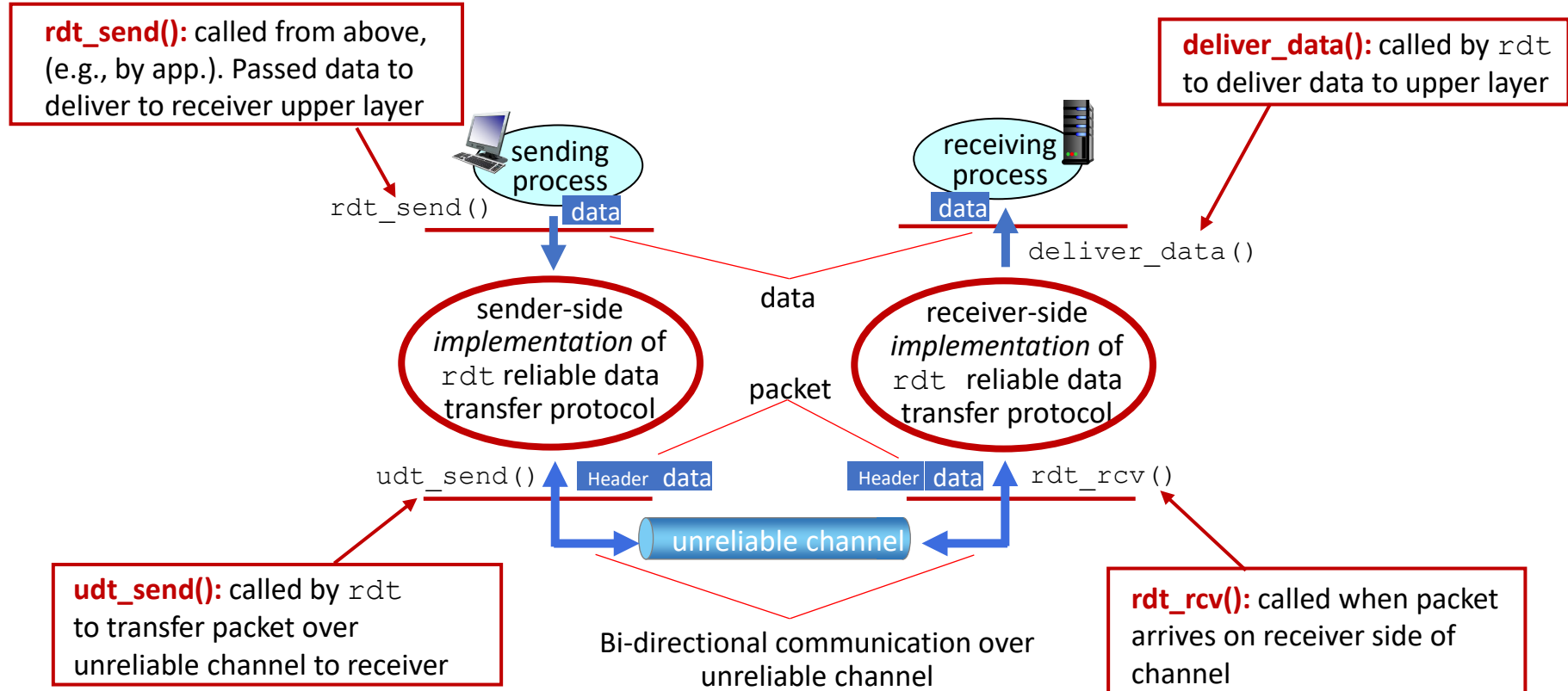
Principles of reliable data transfer

Sender, receiver do *not* know the “state” of each other, e.g., was a message received?

- unless communicated via a message



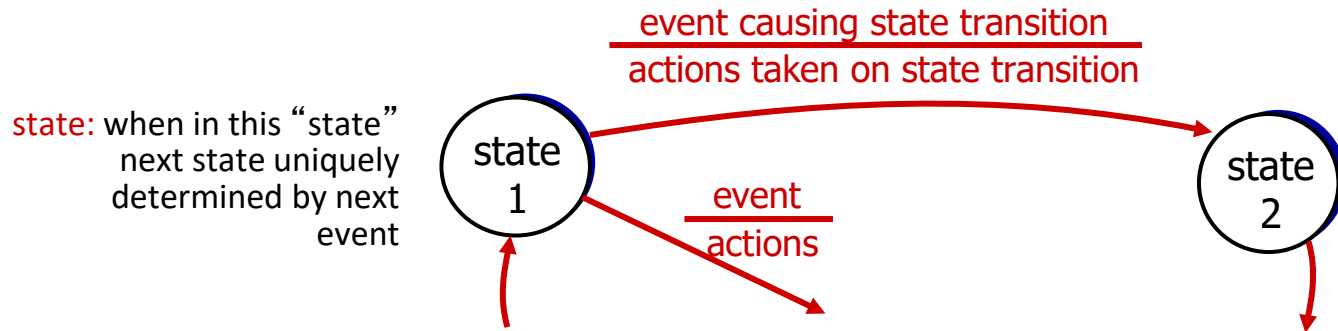
Reliable data transfer protocol (rdt): interfaces



Reliable data transfer: getting started

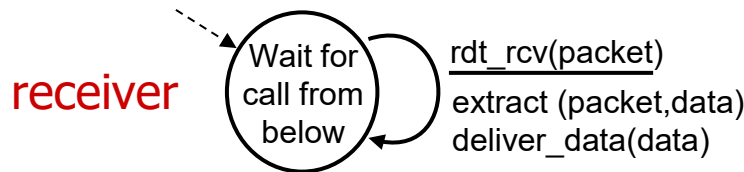
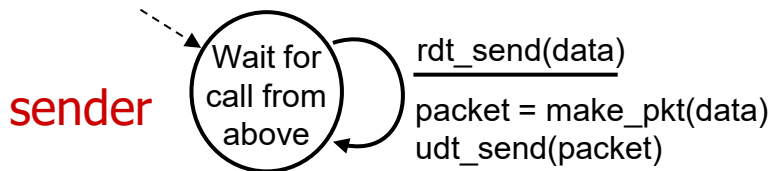
We will:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
 - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender, receiver



rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
 - no bit errors
 - no loss of packets
- *separate* FSMs for sender, receiver:
 - sender sends data into underlying channel
 - receiver reads data from underlying channel



rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
 - checksum (e.g., Internet checksum) to detect bit errors
- *the* question: how to recover from errors?

How do humans recover from “errors” during conversation?

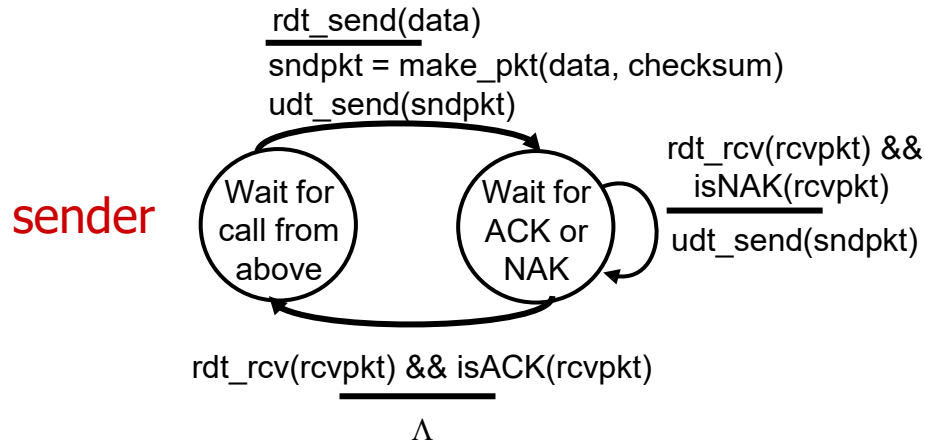
rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
 - checksum to detect bit errors
- *the* question: how to recover from errors?
 - *acknowledgements (ACKs)*: receiver explicitly tells sender that pkt received OK
 - *negative acknowledgements (NAKs)*: receiver explicitly tells sender that pkt had errors
 - sender *retransmits* pkt on receipt of NAK

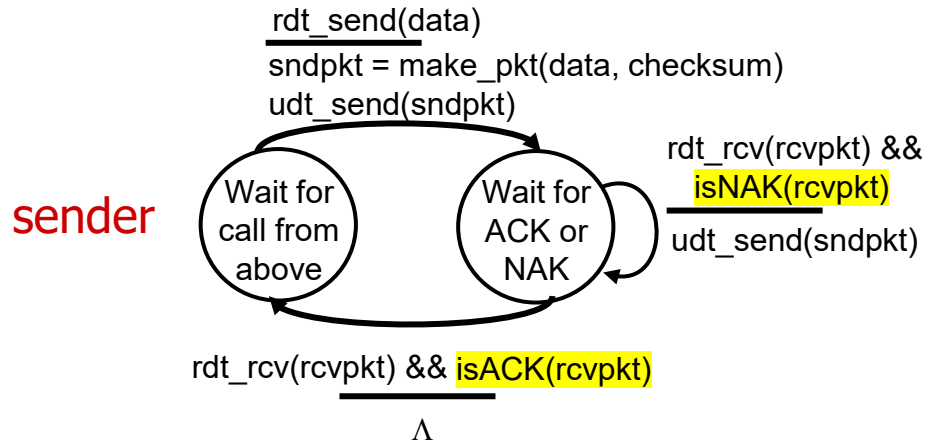
stop and wait

sender sends one packet, then waits for receiver response

rdt2.0: FSM specifications



rdt2.0: FSM specification

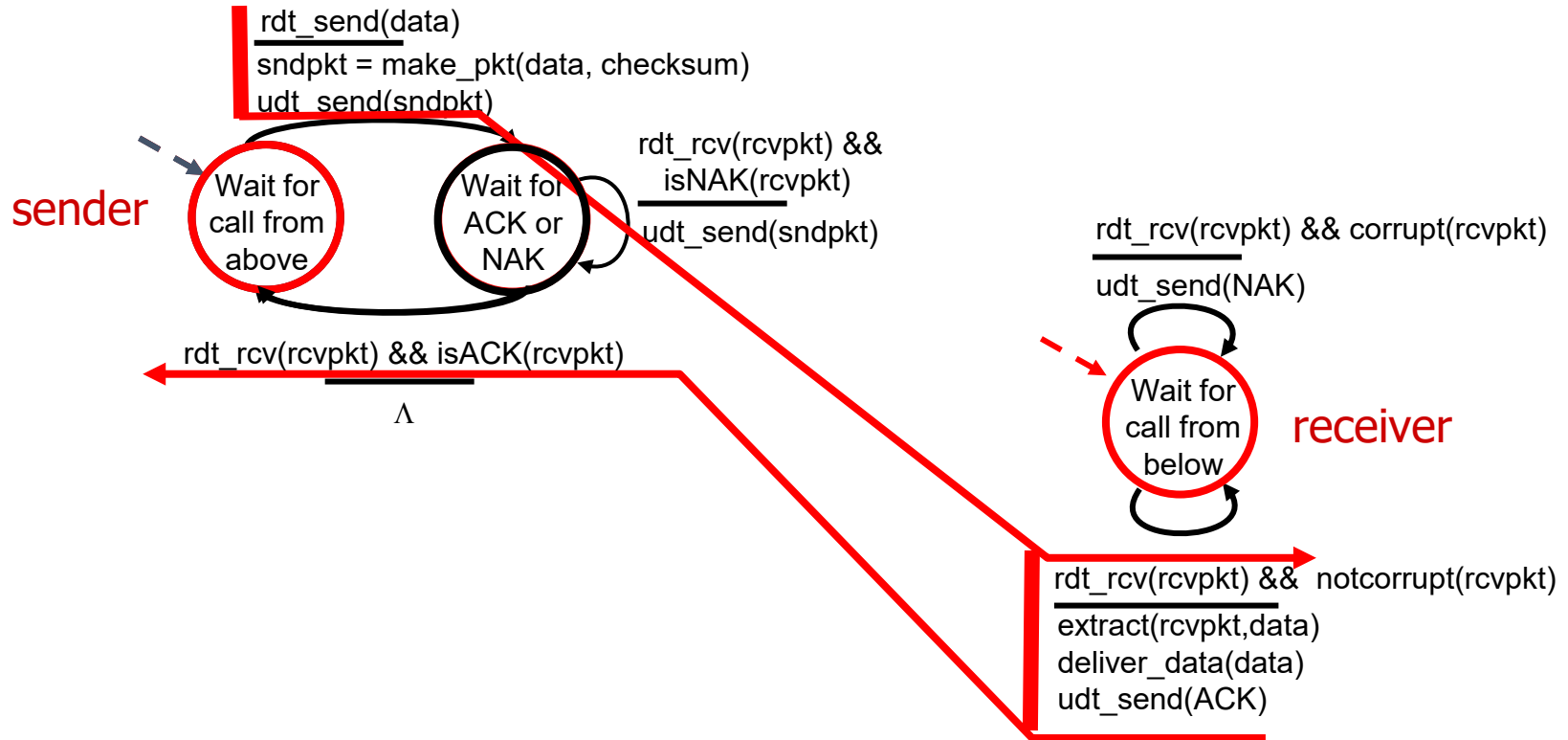


Note: “state” of receiver (did the receiver get my message correctly?) isn’t known to sender unless somehow communicated from receiver to sender

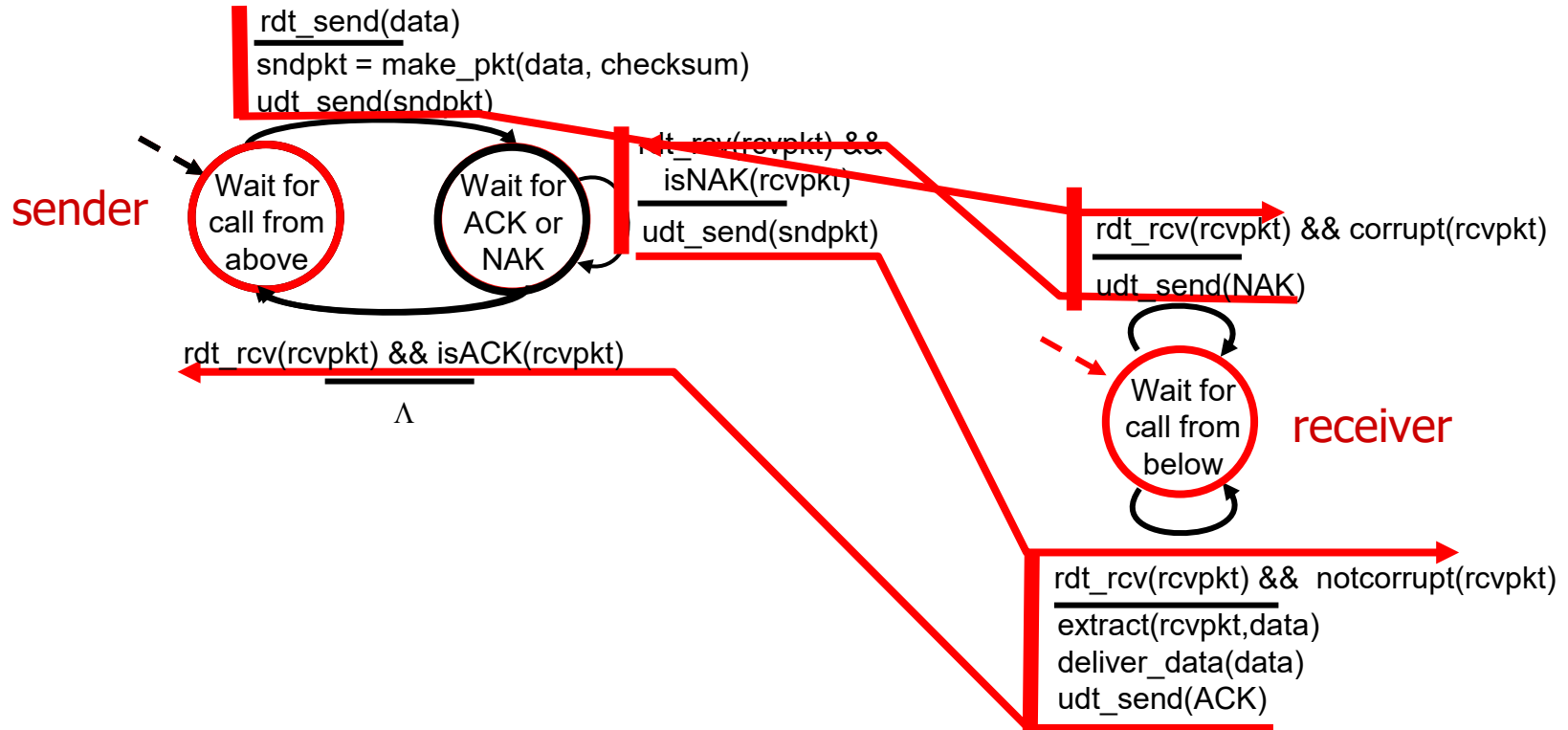
- that’s why we need a protocol!



rdt2.0: operation with no errors



rdt2.0: corrupted packet scenario



rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

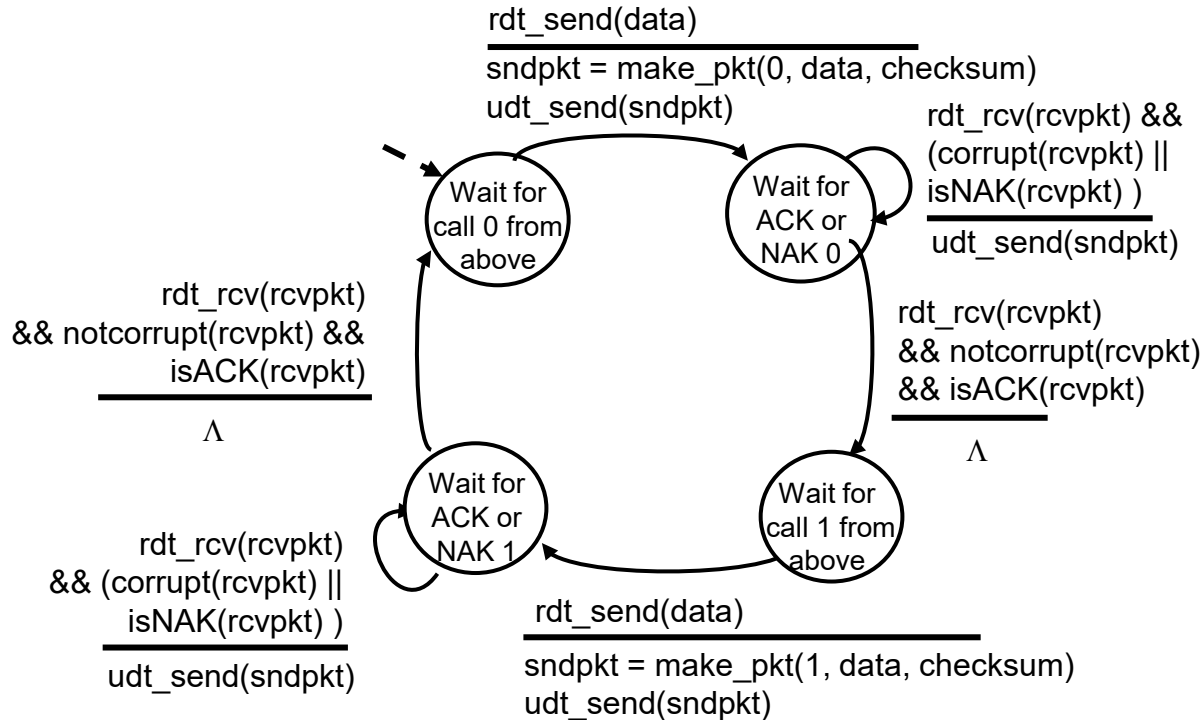
handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

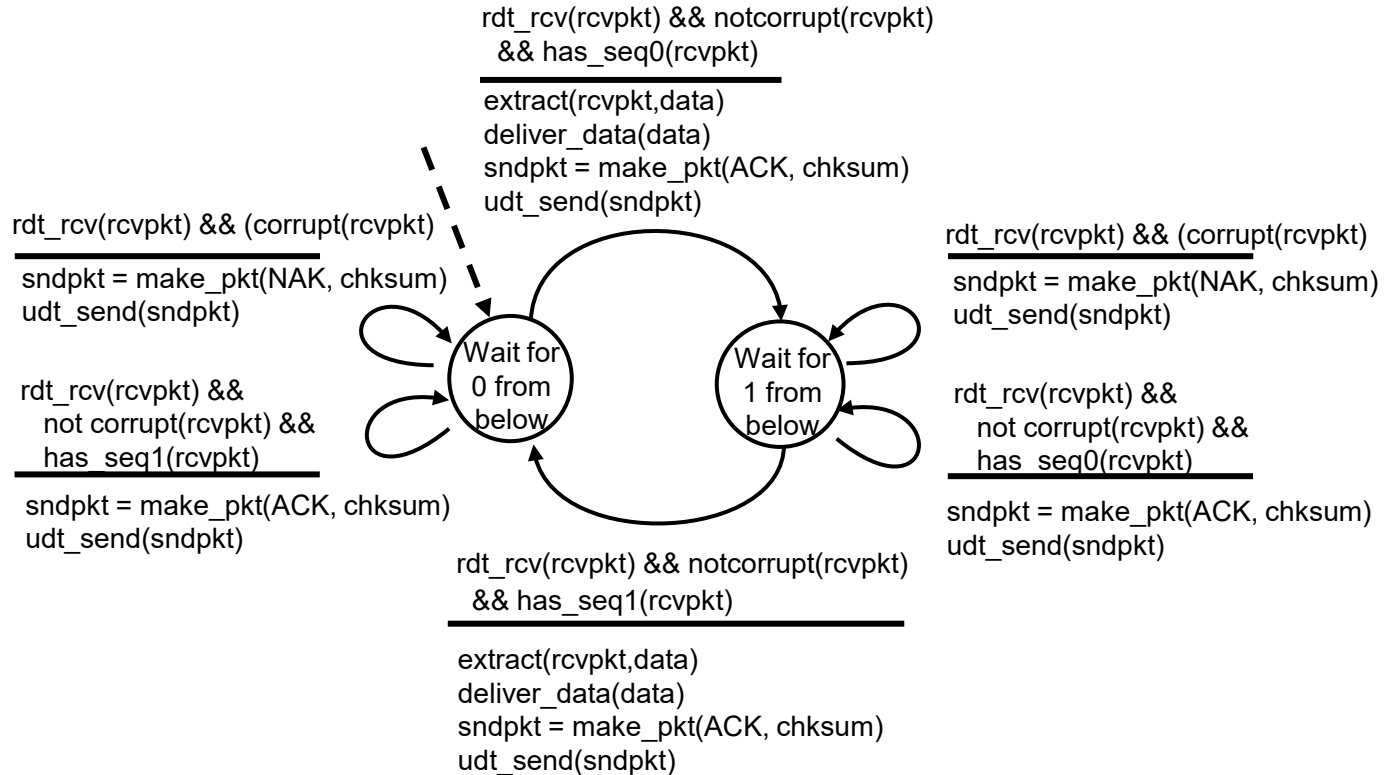
stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: sender, handling garbled ACK/NAKs



rdt2.1: receiver, handling garbled ACK/NAKs



Time for a break

we restart at 11:15

Join at [menti.com](https://www.menti.com) | use code 5217 6469

NTNU
Knowledge for the next order

Instructions

Go to
www.menti.com

Enter the code

5217 6469



Or use QR code



rdt2.1: discussion

sender:

- seq # added to pkt
- two seq. #s (0,1) will suffice.
Why?
- must check if received ACK/NAK corrupted
- twice as many states
 - state must "remember" whether "expected" pkt should have seq # of 0 or 1

receiver:

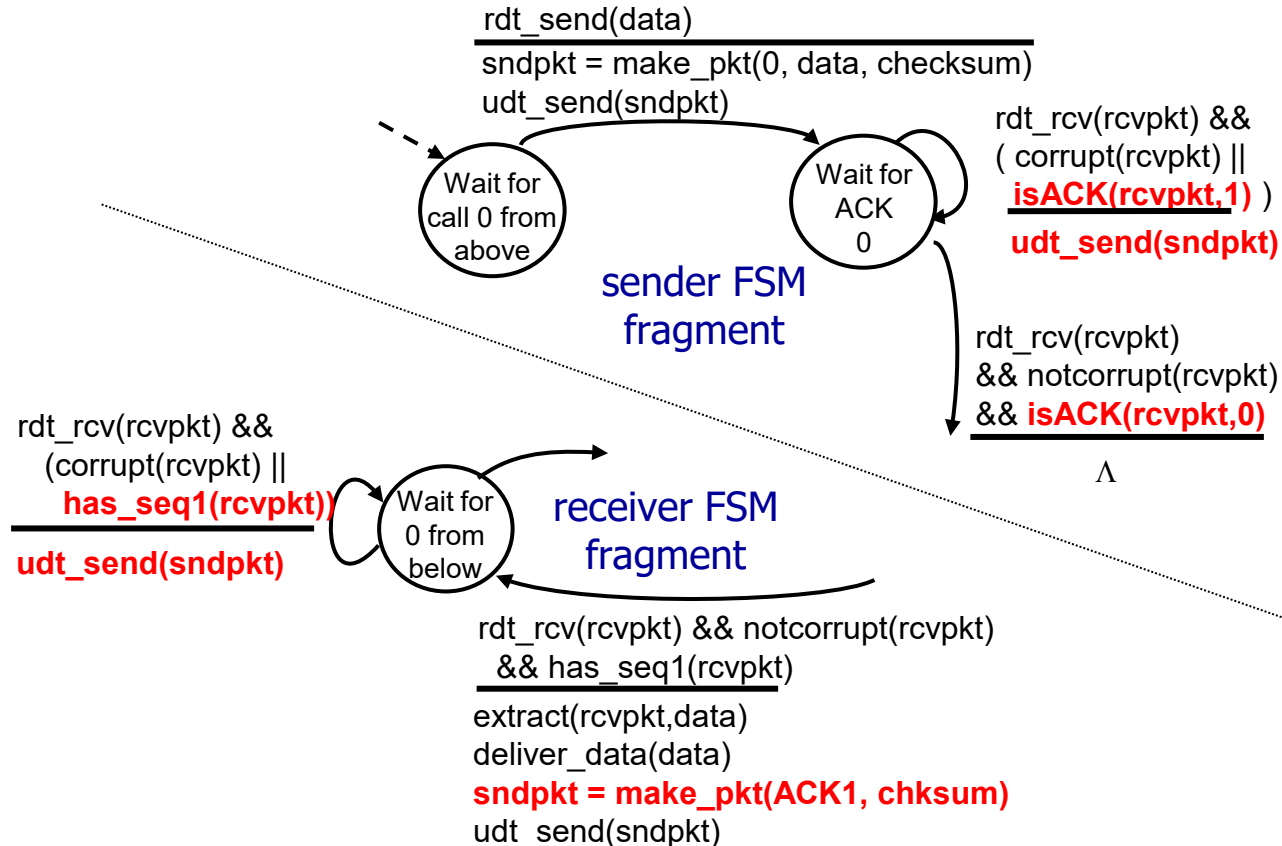
- must check if received packet is duplicate
 - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
 - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK:
retransmit current pkt

As we will see, TCP uses this approach to be NAK-free

rdt2.2: sender, receiver fragments



Time for a break

we restart at 11:15

Join at [menti.com](https://www.menti.com) | use code 5217 6469

NTNU
Knowledge for an open world

Instructions

Go to
www.menti.com

Enter the code

5217 6469



Or use QR code



rdt3.0: channels with errors *and* loss

New channel assumption: underlying channel can also *lose* packets (data, ACKs)

- checksum, sequence #s, ACKs, retransmissions will be of help ... but not quite enough

Q: How do *humans* handle lost sender-to-receiver words in conversation?

rdt3.0: channels with errors *and* loss

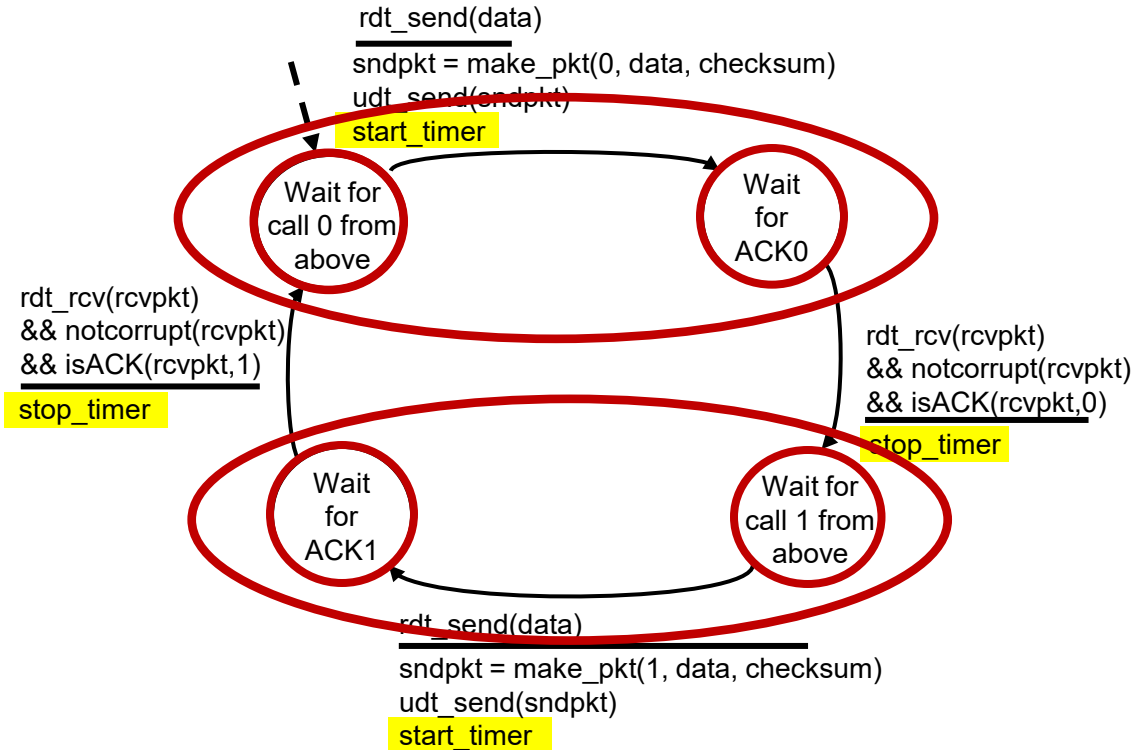
Approach: sender waits “reasonable” amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
 - retransmission will be duplicate, but seq #s already handles this!
 - receiver must specify seq # of packet being ACKed
- use countdown timer to interrupt after “reasonable” amount of time

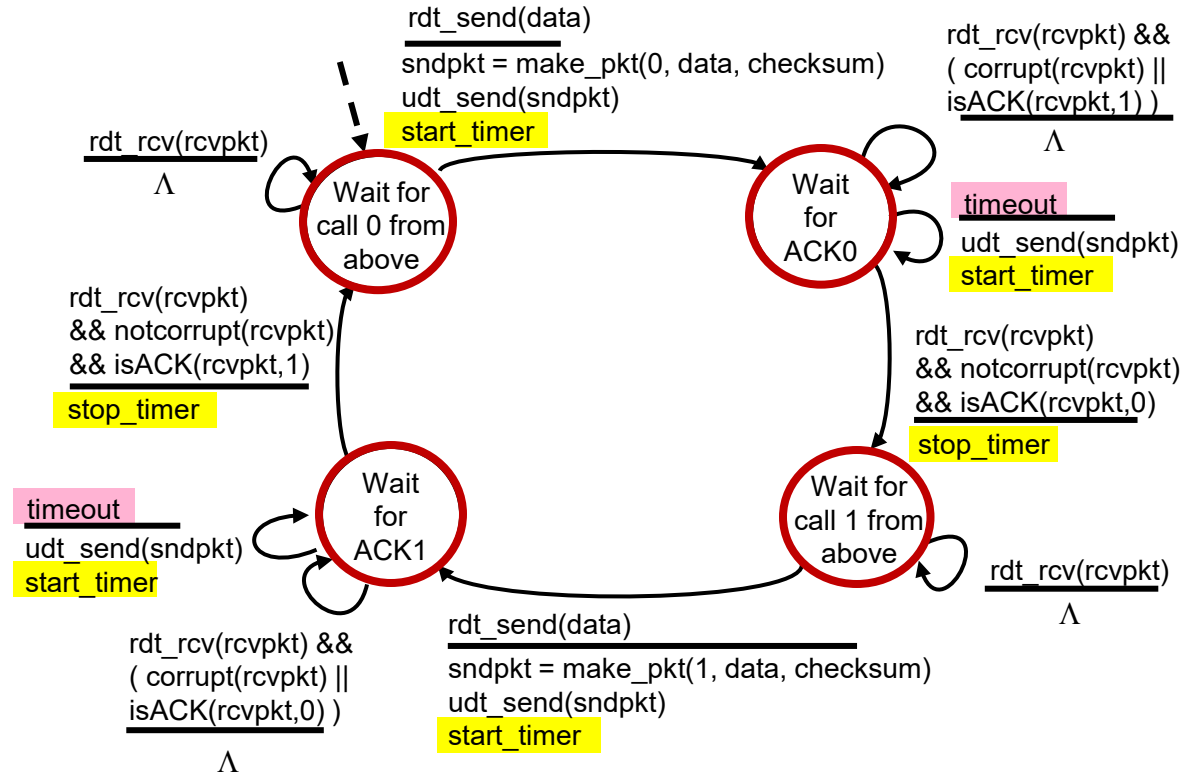


timeout

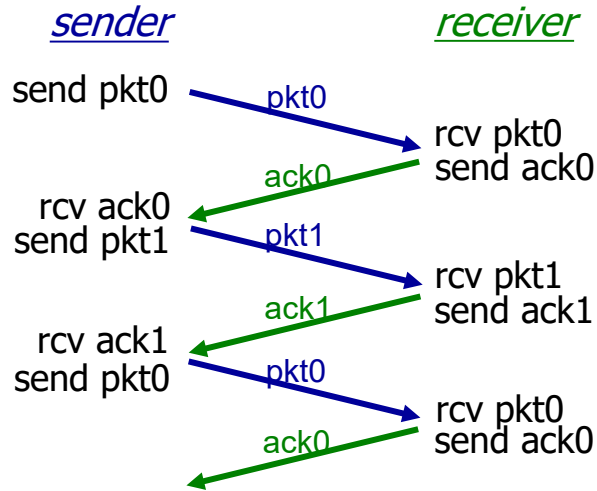
rdt3.0 sender



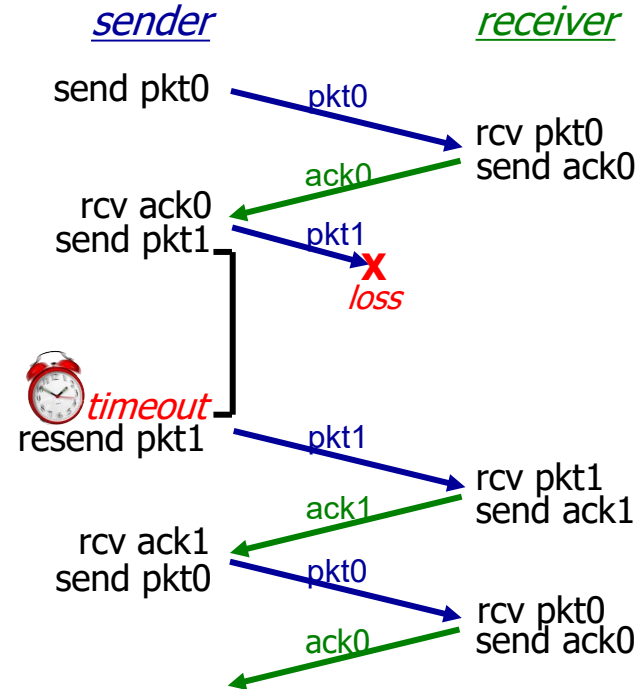
rdt3.0 sender



rdt3.0 in action

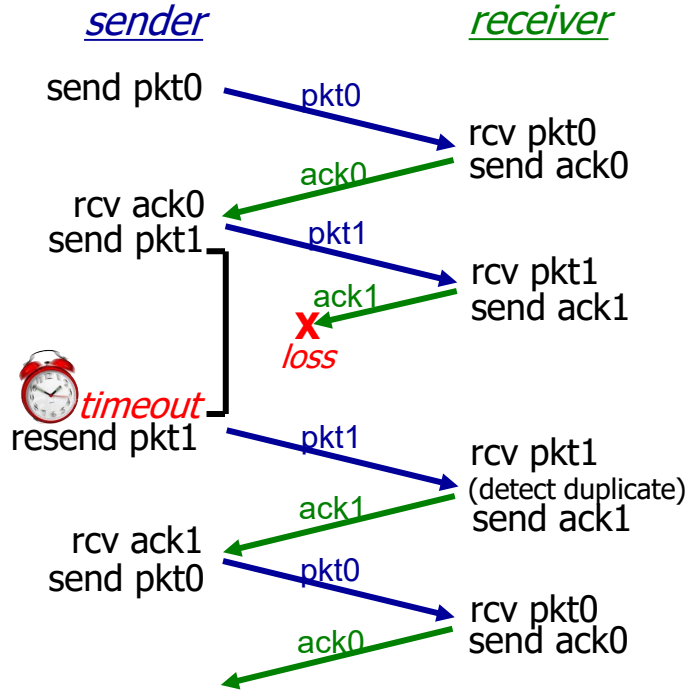


(a) no loss

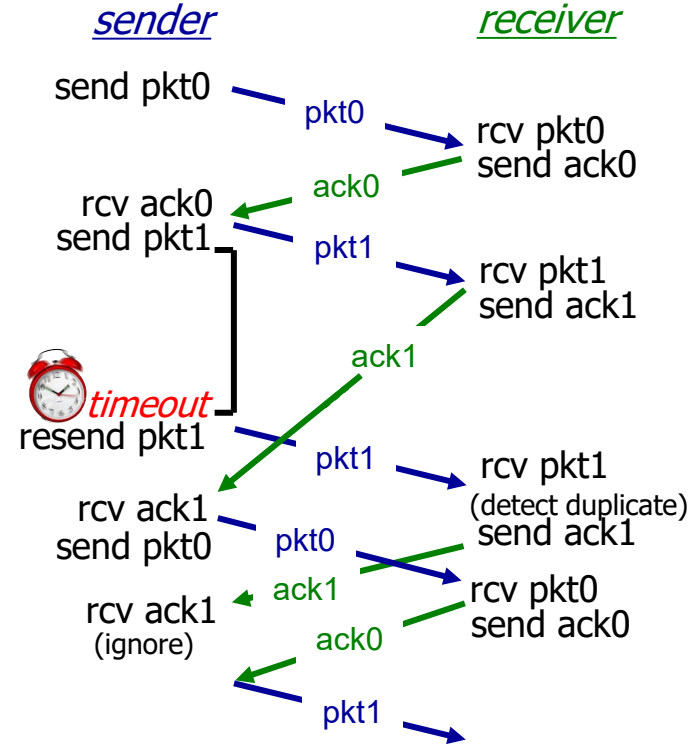


(b) packet loss

rdt3.0 in action



(c) ACK loss



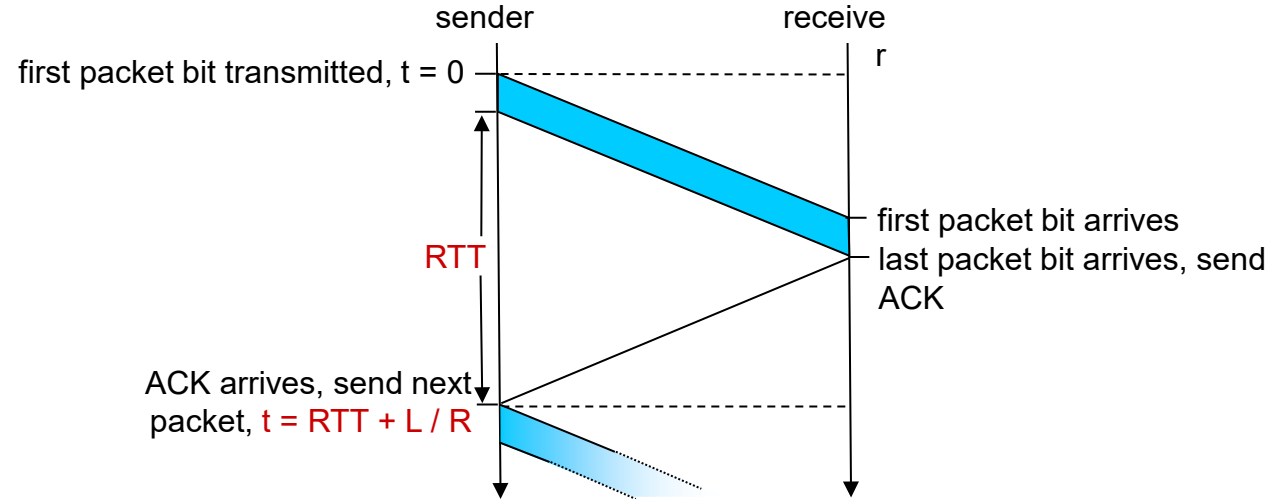
(d) premature timeout/ delayed ACK

Performance of rdt3.0 (stop-and-wait)

- U_{sender} : *utilization* – fraction of time sender busy sending
- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
 - time to transmit packet into channel:

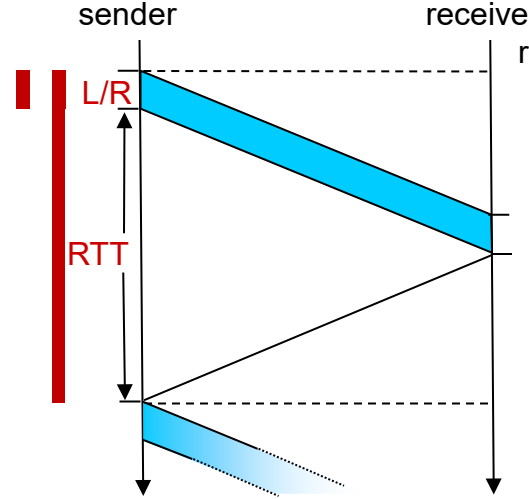
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

rdt3.0: stop-and-wait operation



rdt3.0: stop-and-wait operation

$$\begin{aligned}U_{\text{sender}} &= \frac{L / R}{RTT + L / R} \\&= \frac{.008}{30.008} \\&= 0.00027\end{aligned}$$

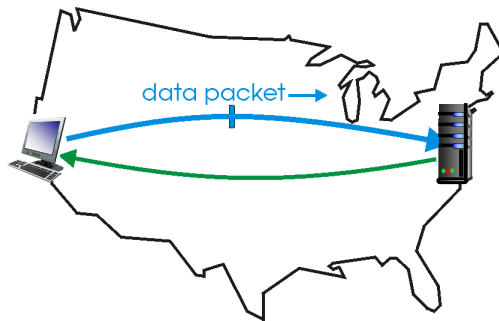


- rdt 3.0 protocol performance stinks! 💩
- Protocol limits performance of underlying infrastructure (channel)

rdt3.0: pipelined protocols operation

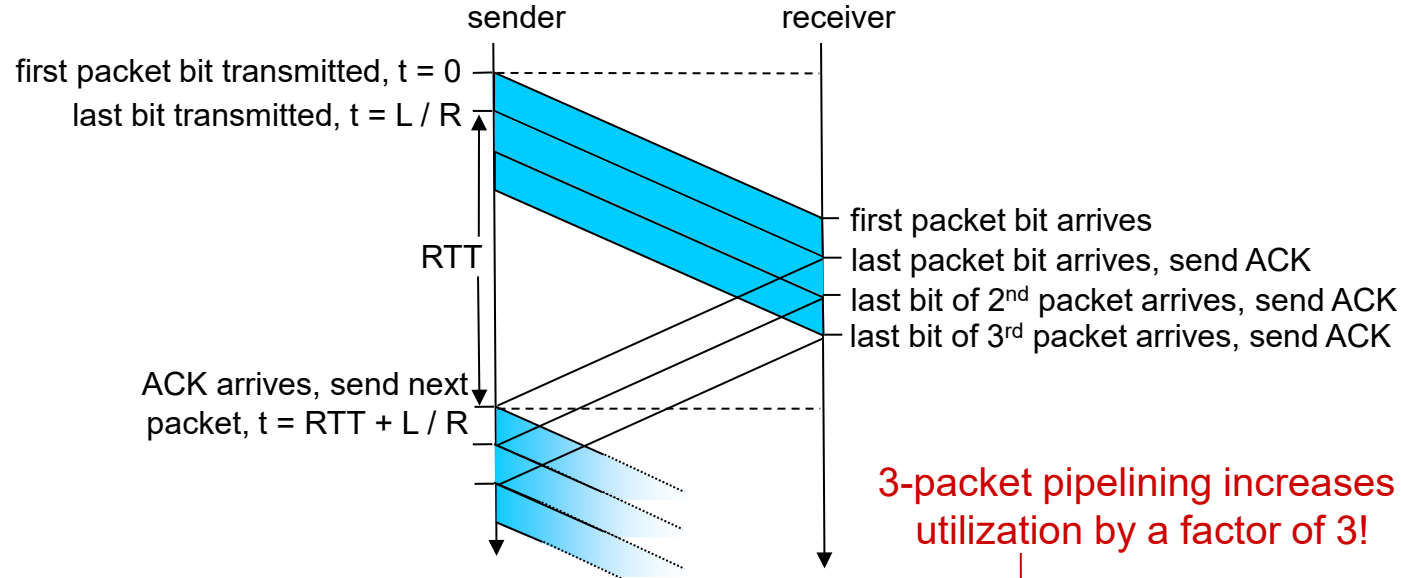
pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

Pipelining: increased utilization

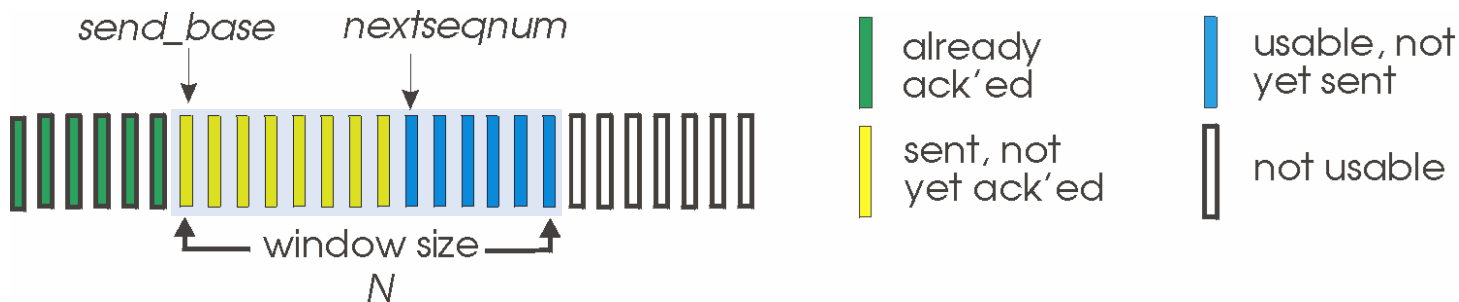


3-packet pipelining increases utilization by a factor of 3!

$$U_{\text{sender}} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Go-Back-N: sender

- sender: "window" of up to N , consecutive transmitted but unACKed pkts
 - k -bit seq # in pkt header

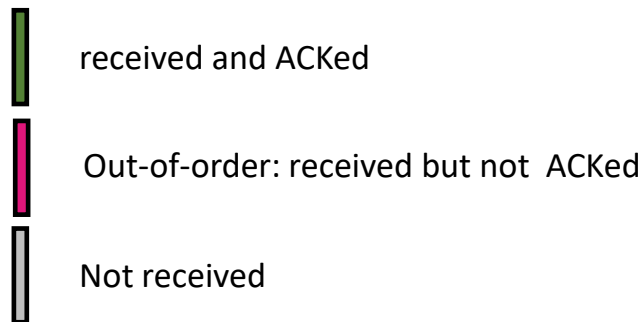
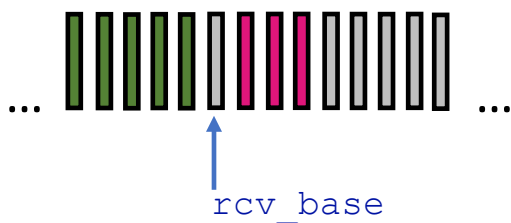


- **cumulative ACK:** $ACK(n)$: ACKs all packets up to, including seq # n
 - on receiving $ACK(n)$: move window forward to begin at $n+1$
- timer for oldest in-flight packet
- $timeout(n)$: retransmit packet n and all higher seq # packets in window

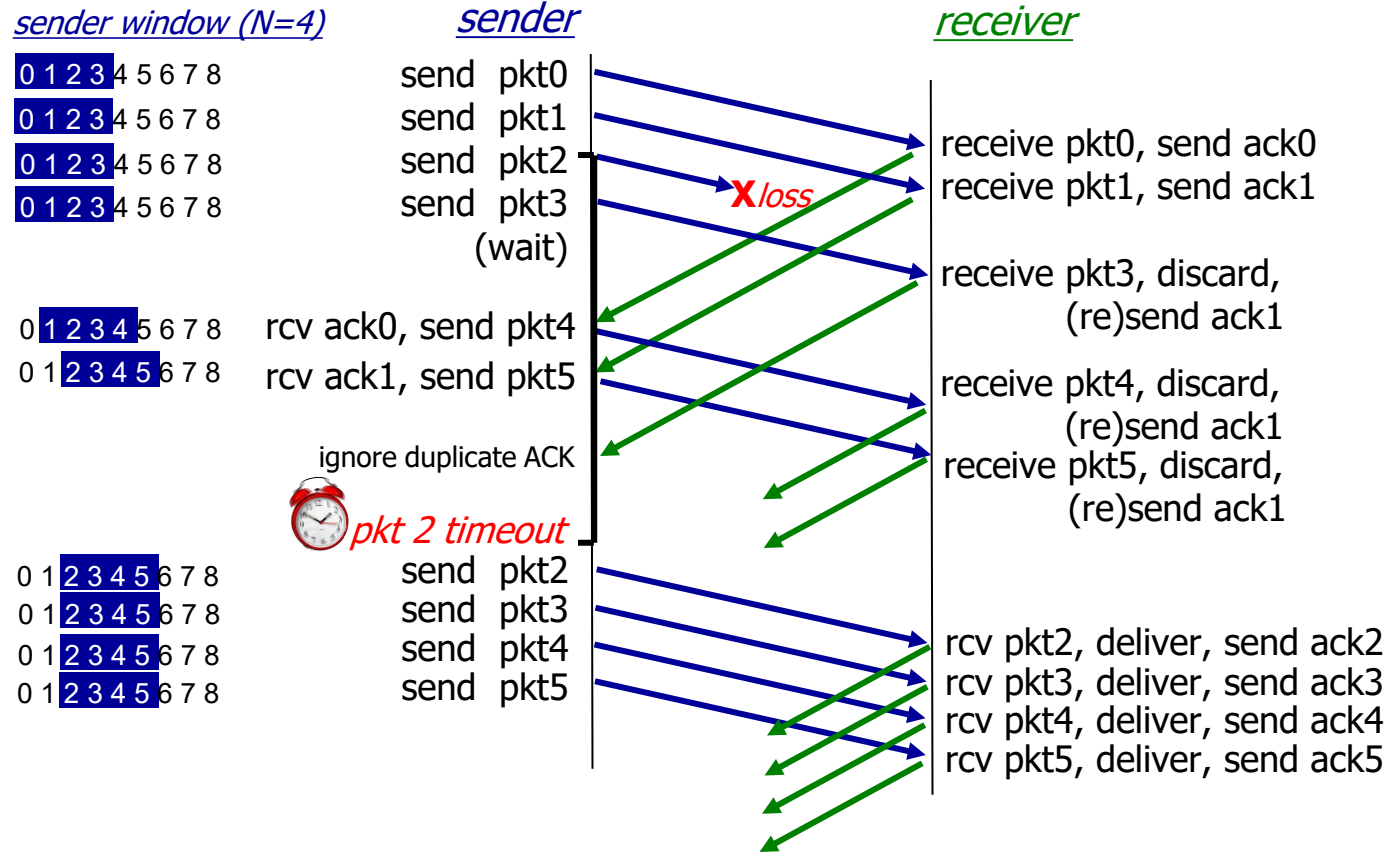
Go-Back-N: receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
 - may generate duplicate ACKs
 - need only remember `rcv_base`
- on receipt of out-of-order packet:
 - can discard (don't buffer) or buffer: an implementation decision
 - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:



Go-Back-N in action



Selective repeat

- receiver *individually* acknowledges all correctly received packets
 - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender times-out/retransmits individually for unACKed packets
 - sender maintains timer for each unACKed pkt
- sender window
 - N consecutive seq #s
 - limits seq #s of sent, unACKed packets

Readings for next week

Book pages

- 212-242 (Sec. 3.1, 3.2, 3.3 & 3.4.1)
- Optional
 - 243-256 (Sec. 3.4.2, 3.4.3, 3.4.4, reviewed in class)

Key concepts

Logical communication

Segments vs. Datagrams

Best-effort

Reliable vs. unreliable transfer

Congestion Control

Multiplexing & demultiplexing

Problem 22 (pg. 203) ← not solved in class

Consider distributing a file of $F = 10$ Gbit to N peers. The server has an upload rate of $u_s = 1$ Gbit/s and each peer has a download rate of $d_i = 200$ Mbit/s and an upload rate of u . For $N = 10, 100$ and 1000 and $u = 2$ Mbit/s, 10 Mbit/s and 100 Mbit/s, prepare a table giving the minimum distribution time for each of the combinations of N and u for both client-server distribution and P2P distribution.

Solving Problem 22 (pg. 203)

- For calculating the minimum distribution time for client-server distribution, we use the following formula:
 - $D_{cs} > \max\{NF/u_s, F/d_{\min}\}$

N		10	100	1000
u	2 Mbit/s	100s	1000s	10000s
	10 Mbit/s	100s	1000s	10000s
	100 Mbit/s	100s	1000s	10000s

Solving Problem 22 (pg. 203) – continued

- For calculating the minimum distribution time for P2P distribution, we use the following formula:
 - $D_{P2P} > \max\{F/u_s, F/d_{\min}, NF/(u_s + \sum_{i=1}^N u_i)\}$

N		10	100	1000
u	2 Mbit/s	98s	83s	50s
	10 Mbit/s	90s	50s	50s
	100 Mbit/s	50	50s	50s

Problem 26 (pg. 204) ← not solved in class

Suppose Bob joins a BitTorrent torrent but he does not want to upload any data to any other peers (so-called free-rider).

- a) Alice who has been using BitTorrent tells Bob that he cannot receive a complete copy of the file that is shared by the swarm. Is Alice correct or not? Why?
- b) Charlie claims that Alice is wrong and that he has even been using a collection of multiple computers (with distinct IP addresses) in the computer lab in his department to make his downloads faster, using some additional coordination scripting. What could his script have done?

Solving Problem 26 *(pg. 204)*

- Alice is incorrect. As long as there are enough peers staying in the swarm for a long enough time. Bob can always receive data through optimistic unchoking by other peers.
- Charlie can run a client on each host, let each client “free-ride,” and combine the collected chunks from the different hosts into a single file. He can even write a small scheduling script to make the different hosts ask for different chunks of the file. This is actually a kind of Sybil attack in P2P networks.

R 27 (*based on pg. 198*) ← not solved in class

For a typical client-server application over TCP why must the server program be executed before the client program?

For a client-server application over UDP why may the client program be executed before the server program?

What happens if data is sent in each case?

Solving R 27 *(based on pg. 198)*

- a) For the TCP application, as soon as the client is executed, it attempts to initiate a TCP connection with the server. If the TCP server is not running, then the client will fail to make a connection.
- b) For the UDP application, the client does not initiate connections (or attempt to communicate with the UDP server) immediately upon execution.
- c) We cannot send data in TCP because the connection will fail. In UDP the data will be sent but not received by anyone.

Enough for today

Please do the weekly feedback

See you next week!

Join at [menti.com](https://www.menti.com) | use code 5217 6469

NTNU
Knowledge for an better world

Instructions

Go to
www.menti.com

Enter the code

5217 6469



Or use QR code

