Lec5 – The Transport Layer

50.012 Networks

Jit Biswas

Cohort 1: TT 7&8 (1.409-10)

Cohort 2: TT 24&25 (2.503-4)

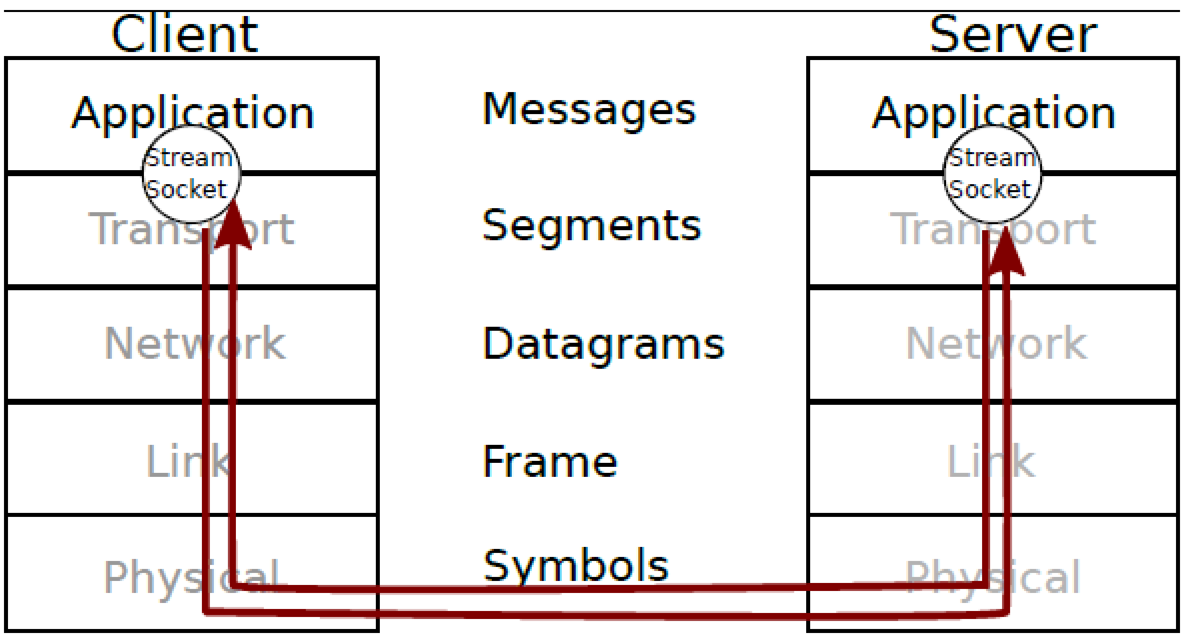
The Transport Layer

Introduction

- Today's lecture: The Transport Layer
  - Transport layer overview
  - The User Datagram Protocol (UDP)
  - Design challenge: design a reliable transport protocol

Overview

- Transport Layer transmits segments and streams
- Application layer Message is converted to segments
- Segments are then passed on to network layer for transmission
- At the receiver side, segments are re-assembled on Transport layer and passed towards Application layer
- Transport layer is very generic: two protocols (UDP and TCP) are used for almost everything!



An analogy

- Hosts – dorms
- Dorm residents – application processes
- Dorm representative – Transport layer
- Mail delivery man – Network layer
- Postal service – reliable delivery of mail to the dorm
- Dorm rep’s service – distribution of mail to every resident

The Transport Layer

- Recall the socket abstraction
- What does the socket permit?

The Transport Layer

- Recall the socket abstraction
- What does the socket permit?
- Sockets allow processes to communicate with each other across hosts
- The Transport layer supports communication between processes
- The Network layer supports communication between hosts

Interfaces to Other Layers

- To the Application layer, abstract datagram or stream socket
  - Logical communication between applications
- To the Network layer, a low-layer socket (aka raw socket)
  - Logical communication between hosts
- The Transport layer provides logical communication between processes
  - Identified by remote and local addresses and protocol used (UDP/TCP)

Ports and Multiplexing

- To address a remote service, the IP address, port number, and transport layer protocol has to be known
- While the IP address identifies the host, the port number (and protocol) identifies the process that is listening
- A process can listen on multiple ports, but usually at most one process listens to each port
- The sender can choose an arbitrary source port

Reserved Port Ranges

- Port numbers are 16 bit long (0-65535)
  - IANA (Internet Assigned Numbers Authority) suggests mapping
- Ports from 0-1023 are system ports, most standard protocols.
- On Linux, only root can create sockets using these
  - Example: 22:SSH, 23: telnet, 80: http, 123:NTP
- Ports from 1024-49151 are registered ports for user applications
  - Example: 8080 for http proxy, or web server run as user
- Port from 49152-65535 are to be used as ephemeral ports
  - Automatically assigned by applications
  - On Linux: 32768-65535

The User Datagram Protocol

UDP

- The User Datagram Protocol (UDP) enables no-frills transport
- It contains the bare minimum: source+destination port, length, checksum
- The UDP header only introduces 8 bytes of overhead to any payload message

Source Port 16 bit	Dest Port 16 bit
Fragment length 16 bit	UDP Checksum 16 bit
Data (up to 65,519 Bytes without 8 Byte header)	

UDP protocol segment header + body

Using UDP Sockets

- UDP sockets are simple

For the server:

```
import socket, sys
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server_address = ('localhost', 1234)
message = 'This is the message'
sent = sock.sendto(message, server_address)
print >>sys.stderr, 'sent %s bytes to %s' % (sent,
server_address)
```

Helpful URL -
https://pymotw.com/2/socket/udp.html

Using UDP Sockets

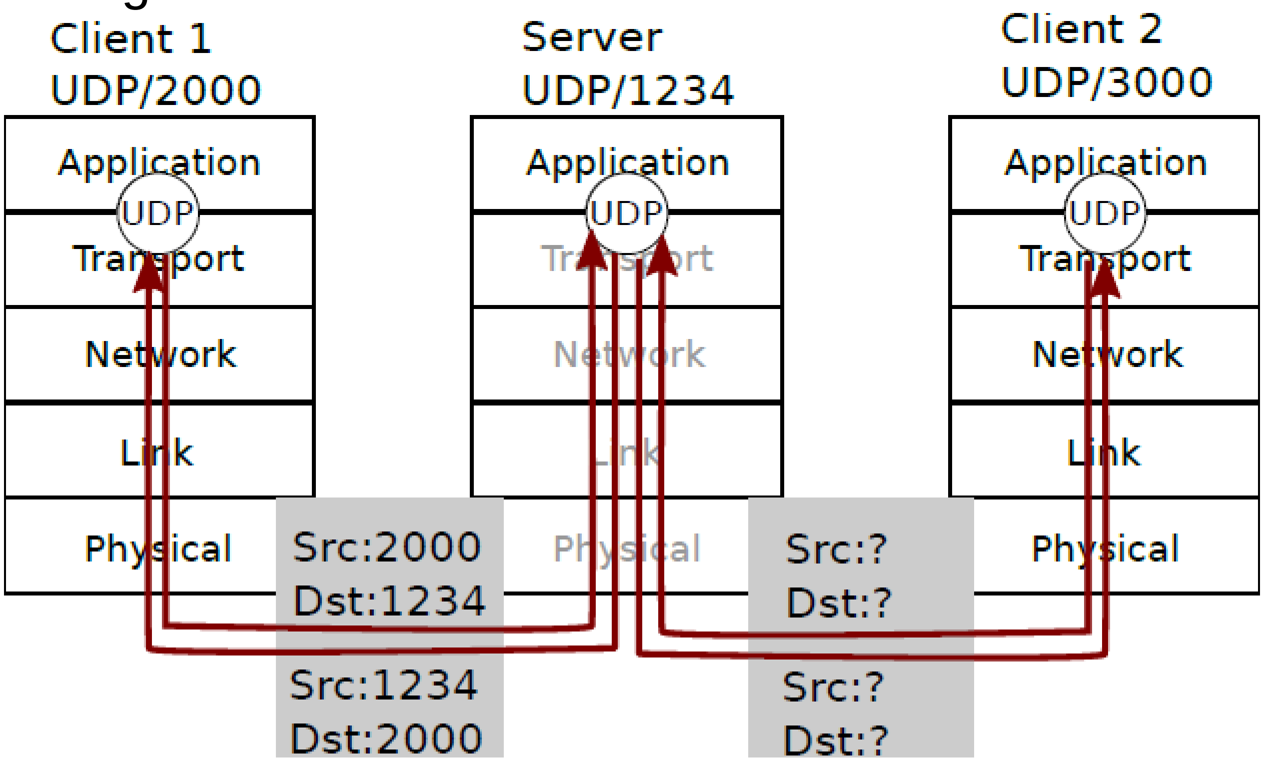
- UDP sockets are simple

For the client:

```
import socket, sys
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind (('localhost', 1234))
data, server = sock.recvfrom(4096)
print >>sys.stderr, 'received "%s" ' % data
sock.close()
```

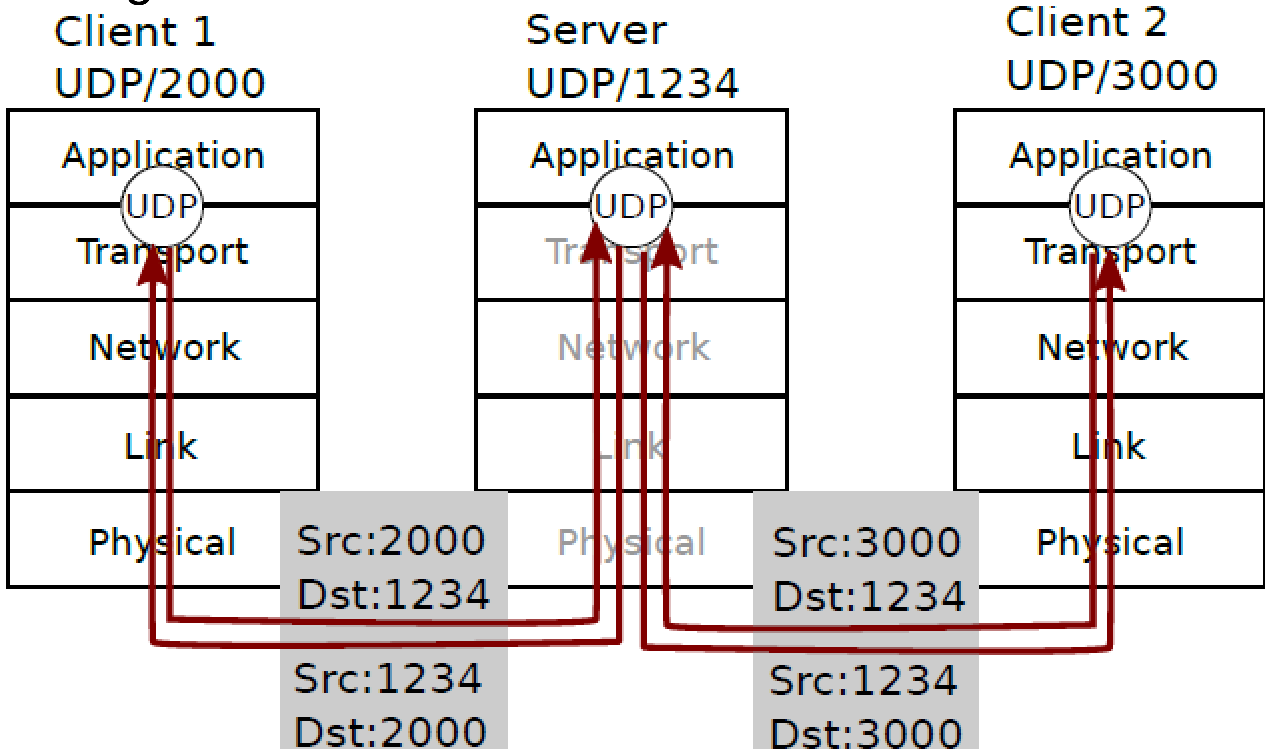
Connection-less Communication

- Consider a UDP server process listening on a socket using port 1234, and handling two clients at the same time.



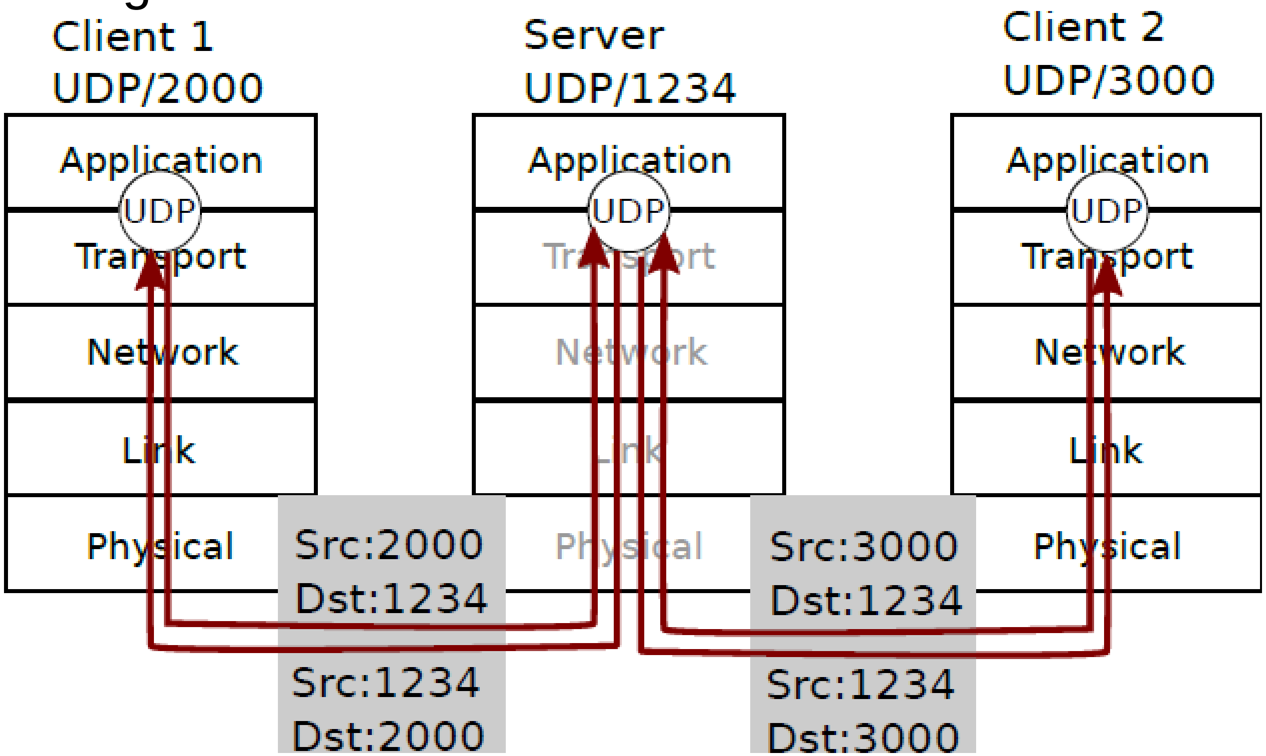
Connection-less Communication

- Consider a UDP server process listening on a socket using port 1234, and handling two clients at the same time.



Connection-less Communication

- Consider a UDP server process listening on a socket using port 1234, and handling two clients at the same time.



- Transport layer passes datagrams to process using dest. port
- UDP Applications can then de-multiplex based on src IP/port

Connection-oriented demux

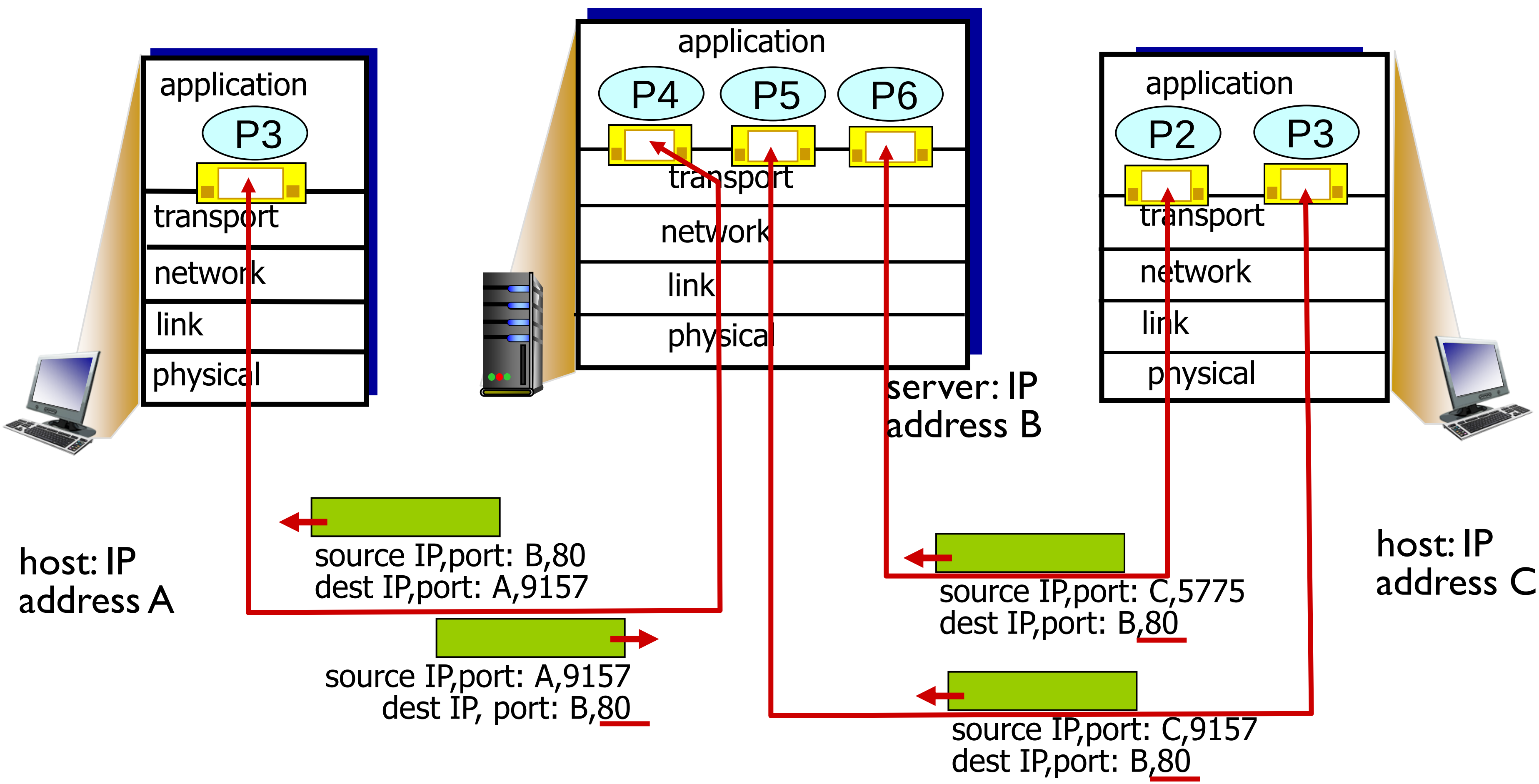
SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket

- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

From K&R: Chapter 3 slides

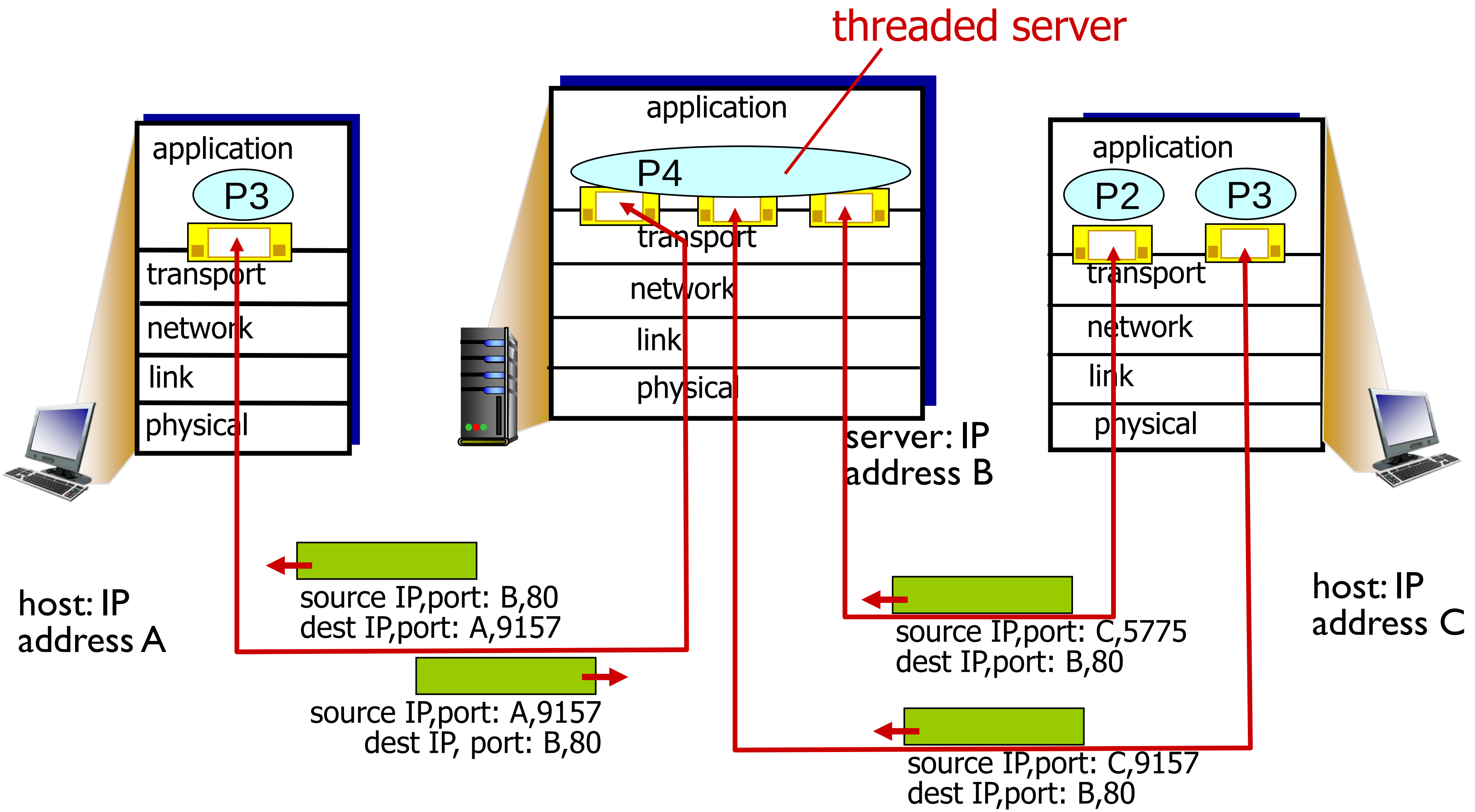
Connection-oriented demux: example



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

From K&R: Chapter 3 slides

Connection-oriented demux: example



From K&R: Chapter 3 slides

Back to UDP: Broadcasting

- Imagine you want to distribute same data to many receivers
  - Example: Status messages, Video streaming
- UDP is a good choice for this, why?

Back to UDP: Broadcasting

- Imagine you want to distribute same data to many receivers
  - Example: Status messages, Video streaming
- UDP is a good choice for this, why?

- UDP is connectionless - the sender does not need to perform handshakes or similar with receiver
- Connection-based protocols cannot easily (or at all) be used for broadcasting

UDP Checksums

- UDP Checksums detect transmission errors within a segment
- Checksum is computed as following:
  - UDP segment is interpreted as 16-bit unsigned integers array
    - Including header (apart from checksum field)
  - All values are then added together, a carry is wrapped around
    - Checksum = ones' complement of that sum (XOR with 0xFFFF)
- Example, lets assume we have 10 fields of 16bit/4hex each

```
4500 + 0030 + 4422 + 4000 + 8006 + 0000 + 8c7c + 19ac
+ ae24 + 1e2b = 2BBCF
```

- Wrap around the carry (2): `2+BBCF=BBD1`
- Compute the complement: `BBD1 XOR FFFF=442E`
- Why like this exactly? Was easy to implement 40 years ago

What else could we want?

- Any ideas for problems that could happen?
- What else would be nice to have on the transport layer?

What else could we want?

- Any ideas for problems that could happen?
- What else would be nice to have on the transport layer?

For now:

- We need to recover from corrupted segments somehow
- We might want to detect incorrect sequence of segments

Risk of Loss and Re-ordering

- How big is the risk of re-ordering and loss in practice?
  - It depends on the network path and load
  - Following numbers are not representative
- Experimental values can be found here:

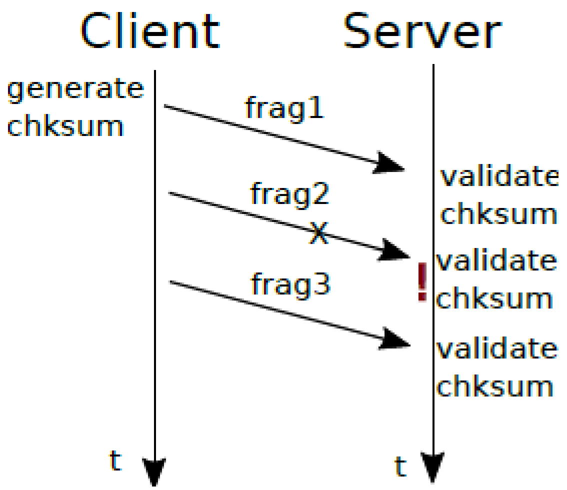
`http://openmymind.net/How-Unreliable-Is-UDP/`

- Excerpt reliability
  - Reliability was >98.5%
  - Little influence of length
- Excerpt Ordering
  - A lot of packets were out-of-order (arriving after successor packet)
  - When discarding those packets: only 50% of packets received!

Design your own Reliable Transport Protocol

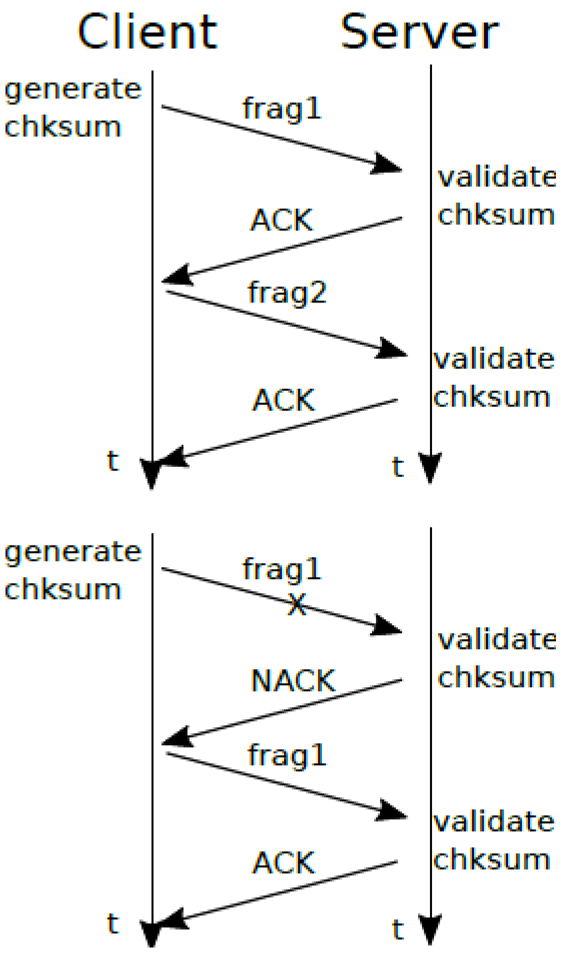
Design Challenge 1

- Design your own protocol like UDP, with additional features
- Lets consider the following scenario:
  - About 10% of all segments will have corrupted data content
  - How to detect this/ recover from this?
- Your task 1: design a reliable transmission protocol (RTP) to recover from transmission errors



RTPv1: ACKs/NAKs

- Use negative acknowledgments (NAK):
  - Sender sends segments one by one
  - Receiver will receive segment (no lost segments for now)
  - Receiver will validate checksum
    - If checksum OK, receiver sends ACK
    - If checksum is not OK, rec. sends NAK
  - If sender receives ACK, he will send next segment
  - If sender receives NAK, he will re-send last segment



ACKs, NAKs

- How do the ACKs and NAKs look like?
- For the moment, they could just have 33 bit:
  - 16 source port
  - 16 destination port
  - 1 bit ACK/NAK flag (0=acknowledged, 1=not acknowledged)
  - In practice, length is going to be multiple of 8bit, why?
- Do we need anything more? Why is short better?
- Alternative: explicit ACKs
  - Advantages/Disadvantages?

Corruption of ACK / NAK

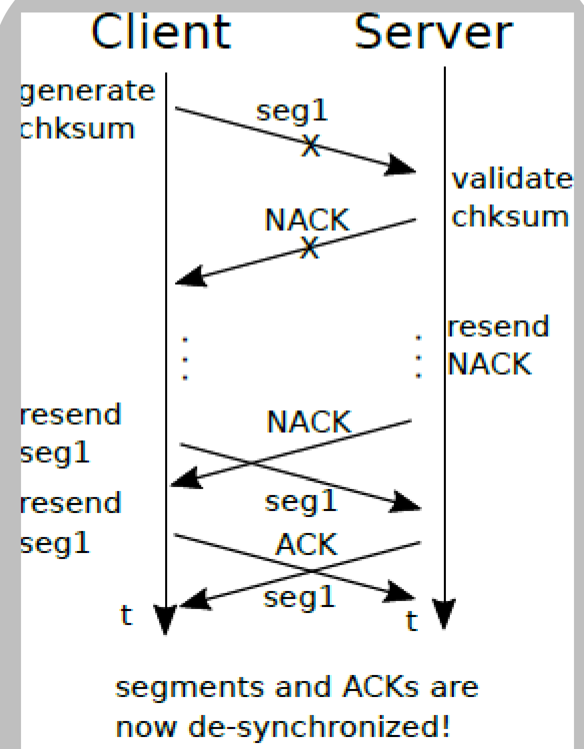
- How to handle corruption of ACK / NAK?

Corruption of ACK / NAK

- How to handle corruption of ACK / NAK?
- Why is retransmission of (N)AK bad?

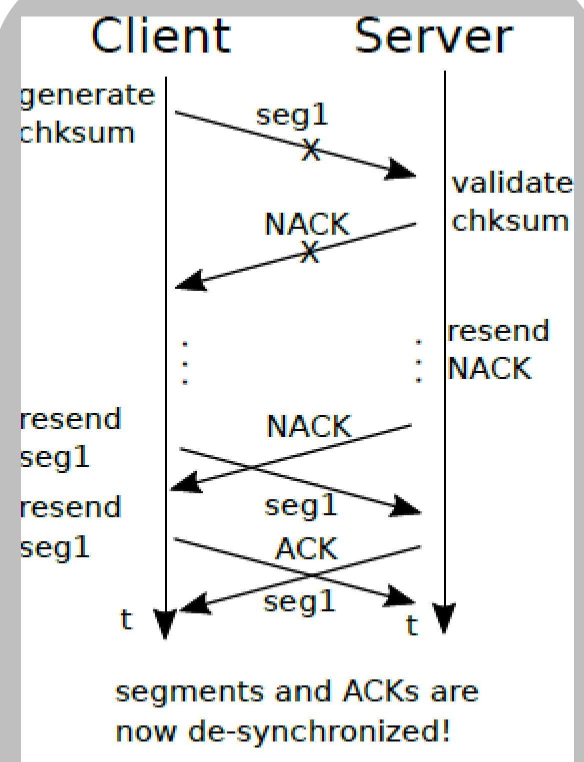
Corruption of ACK / NAK

- How to handle corruption of ACK / NAK?
- Why is retransmission of (N)AK bad?
- Let’s assume there is a timeout, after which the sender re-sends ACK/NAK
  - What happens if timeout is too short?



Corruption of ACK / NAK

- How to handle corruption of ACK / NAK?
- Why is retransmission of (N)AK bad?
- Let's assume there is a timeout, after which the sender re-sends ACK/NAK
  - What happens if timeout is too short?
- NAK will be received for next packet

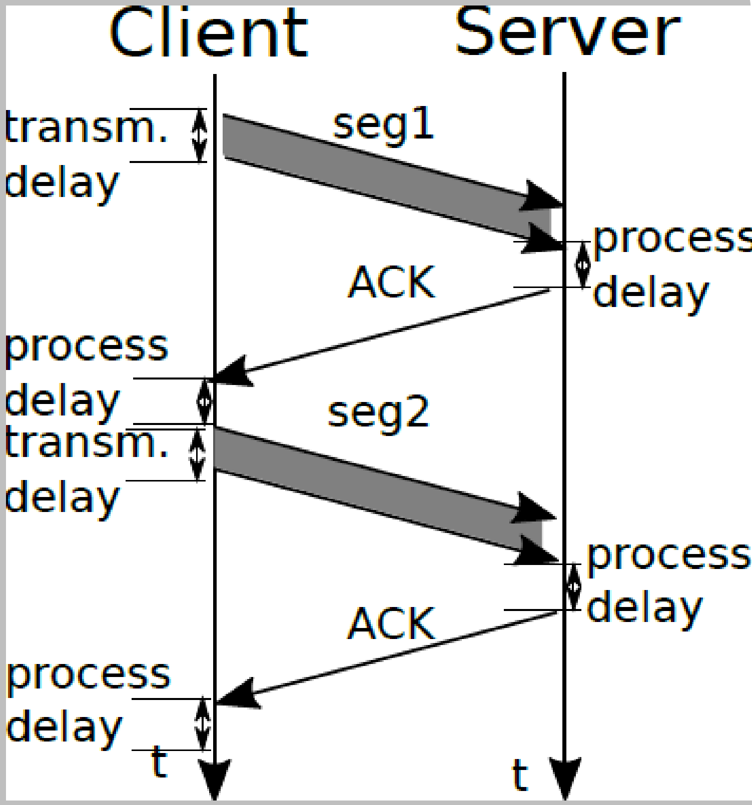


Idle waiting time

- So far, each datagram is ACK'ed
- Transmitting message and getting reply takes time
  - 2x latency + jitter
  - Timeout needs to be at least that time
- This leads to long waiting times
- How does this affect our throughput?

Link Utilization

- Lets assume we use ACKs + timeout
- Need ACK before next segment
- Lets assume 1 Gbps connection
  - No queuing delay for now
- Transm. delay of 1kB: $\frac{8000}{10^9} = 8\mu s$
  - ACK trans. delay negligible
- Assume propagation delay 1 ms
- Data/ACK processing delay: 1 ms
- $\rightarrow$ 4ms + 0:008ms per segment
- Effective transmission speed: 250 segments/s or 250kB/s!



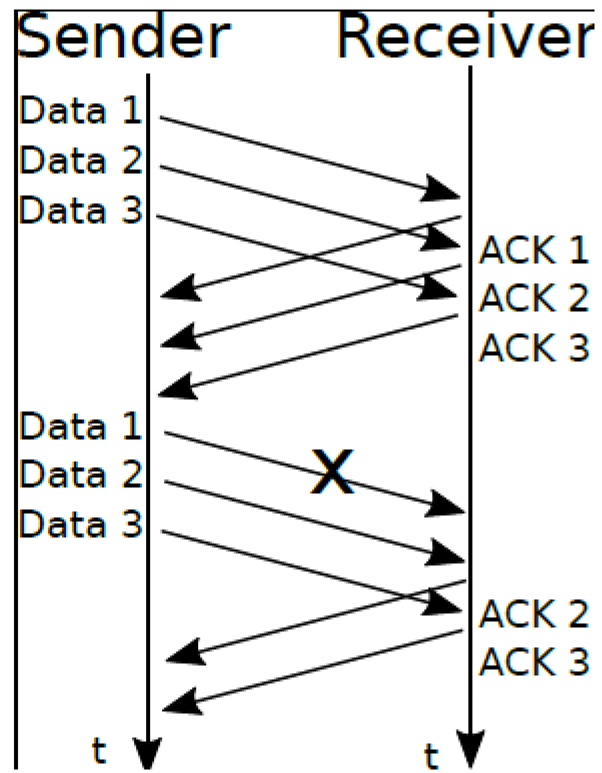
Design Challenge part 2

- Main problems of RTPv1
  - Bad throughput/idle waiting
  - Use of NAKs might lead to missed errors
- Based on my RTPv1 or your own protocol:
  - Design a protocol that
    - Is more efficient
    - Tolerates loss of complete segments/ACKs/NAKs
  - Hint: do you have some idea on how to enable re-transmissions?

RTPv2: Pipelining

Idea to improve throughput:

- Use explicit ACKs, with timeout at sender
- Send multiple segments while waiting for ACKs
- Each segment contains segment ID
- Each ACK contains segment ID
- How big should the segment number be?
  - Trade-off size vs. number of segments in pipeline
  - TCP uses 32 bit segment IDs (counting bytes, not #datagrams)



RTPv2: Retransmission

- Segment number will be used in ACKs
  - This allows us to re-send ACKs
  - The sender can correlate them to exact segments sent
- If many ACKs are outstanding, sender could start sending less data. . .
- How much overhead introduced through segment IDs

Efficiency/Overhead

- Lets define the effective rate of a protocol
  - R=|m|/|f|
  - With |m| length of payload, |f| length of segment
- UDP has low overhead, only adds 8 Byte
- For a 56k Byte long payload:
  - R=0.999. . .
- Bit errors will reduce efficiency
  - If we cannot recover from one error, R=0!

Re-ordering

- In real life, segments could also arrive in wrong order
- Segment IDs allow re-ordering of segments
- Advantage if IDs can be tied together (sequence)
- This can then also naturally handle re-sent segments

Cumulative ACKs

- Transmission schemes can be enhanced with cumulative ACKs
- Instead of sending individual ACKs, send one ACK for several segments
- For example: If receiver successfully received 10 sequential segments, ACK'ing the last shows that all were received

Go-Back-N vs Selective repeat

Two approaches to transmission pipelining

- Selective Repeat
  - Up to n packets can be un-ACK'ed by sender
  - Each packet is ACK'ed individually (also out-of-order ones, are buffered)
  - Each sent packet has its own timer for timeout
- Go-Back-N
  - Sender can have up to n un-ACK'ed segments in pipeline
  - Receiver only sends cumulative ACK
    - Out-of-order segments are not ACK'ed
  - Sender has timeout for oldest un-ACK'ed segments
    - When that timer runs out, all un-ACK'ed segments are re-transmitted

Activity 5 – Reliable Transport Protocol

Open the following URL and choose the Go back N animation applet. Answer the questions below:

https://wps.pearsoned.com/ecs_kurose_compnetw_6/216/55463/14198702.cw/index.html

1. Send packets till the window is full. How many could you send?
2. Pause and kill the first packet
3. Observe what happens to the remaining packets. Are they acknowledged?
4. Now reset and send a full window. Kill the first acknowledgement. Does the window advance? Have all packets been successfully acknowledged? Does the receiver have to acknowledge each and every packet?
5. Now repeat the same actions on the selective repeat animation. Were there unnecessary retransmissions? Did you have to wait?
6. Briefly compare both approaches
7. Submit on eDimension

Conclusion

- The Transport layer provides logical communication between processes
  - Identified by remote and local IP address, port numbers, and protocol used
- Ports allow to multiplex to listening process on receiver side
- UDP is one of the possible protocols - it is very minimalistic
- We started to design our Reliable Data Transport protocol to improve UDP
- Implementing re-transmission schemes is non-trivial - more on that in L6