Announcements

• Final Exam on Dec 17\textsuperscript{th} 2 – 5 PM

• Lab 4 Deadline extended to Thursday

• Lab 5 Partnerships: extended to 5 labs with the same lab partner.

• Discussion questions with answers after class.
Reading Quiz
Transport Layer
Today

- Unreliable, unordered service: UDP
- Principles of reliability
- Class of protocols: Automatic Repeat Requests
Moving down a layer!

- Application Layer
- Transport: end-to-end connections, reliability
- Network: routing
- Link (data-link): framing, error detection
- Physical: 1’s and 0’s/bits across a medium (copper, the air, fiber)
Transport Layer perspective

Application is the boss

Transport: executing within the OS kernel

Network: ours to command!
Transport Layer Header

Assigning ports to socket ID

Application: HTTP

Transport: TCP

Network: IP

Link: Ethernet

data
Transport Layer: Runs on end systems

Logical communication between processes
Last Class: Multiplexing/De-Multiplexing

(Simultaneous transmission of two or more signals/messages over a single channel.)

- The network is a shared resource.
  - It does NOT care about your applications, sockets, etc.

- Senders mark segments, in header, with identifier (port)
How many of these services might we provide at the transport layer? Which?

- Reliable transfers
- Error detection
- Error correction
- Bandwidth guarantees
- Latency guarantees
- Encryption
- Message ordering
- Link sharing fairness

A. 4 or fewer  D. 7
B. 5           E. All 8
C. 6
How many of these services might we provide at the transport layer? Which?

- Reliable transfers (T)
- Error detection (U, T)
- Error correction (T)
- Bandwidth guarantees
- Latency guarantees
- Encryption
- Message ordering (T)
- Link sharing fairness (T)

Critical question: Can it be done at the end host?

A. 4 or fewer
B. 5
C. 6
D. 7
E. All 8
TCP sounds great! UDP…meh. Why do we need it?

A. It has good performance characteristics.

B. Sometimes all we need is error detection.

C. We still need to distinguish between sockets.

D. It basically just fills a gap in our layering model.
TCP sounds great! UDP…meh. Why do we need it?

A. It has good performance characteristics.

B. Sometimes all we need is error detection.

C. We still need to distinguish between sockets.

D. It basically just fills a gap in our layering model.
UDP – User Datagram Protocol

• Unreliable, unordered service
• Adds:
  – end points identified by ports
  – multiplexing
  – checksum (error detection)
UDP: User Datagram Protocol [RFC 768]

• “No frills,” “Bare bones” Internet transport protocol
  – RFC 768 (1980)
  – Length of the document?
UDP: User Datagram Protocol [RFC 768]

- “Best effort” service,
- UDP segments may be:
  - Lost
  - Delivered out of order (same as underlying network layer)
How many of the following steps does UDP implement? (which ones?)

1. exchange an initiate handshake (connection setup)
2. break up packet into segments at the source and number them
3. place segments in order at the destination
4. error-checking with checksum
How many of the following steps does UDP implement? (which ones?)

1. exchange an initiate handshake (connection setup)
2. break up packet into segments at the source and number them
3. place segments in order at the destination
4. error-checking with checksum
UDP Segment

<table>
<thead>
<tr>
<th>SrcPort</th>
<th>DstPort</th>
<th>Length</th>
<th>Checksum</th>
<th>Data</th>
</tr>
</thead>
</table>

32 bits
## TCP Segment!

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>Source port number</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number</td>
</tr>
<tr>
<td>sequence number</td>
<td>Sequence number</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number</td>
</tr>
<tr>
<td>head len</td>
<td>Not used</td>
</tr>
<tr>
<td>U</td>
<td>Urgent flag</td>
</tr>
<tr>
<td>A</td>
<td>Acknowledgement flag</td>
</tr>
<tr>
<td>P</td>
<td>Push flag</td>
</tr>
<tr>
<td>R</td>
<td>Reset flag</td>
</tr>
<tr>
<td>S</td>
<td>Syn flag</td>
</tr>
<tr>
<td>F</td>
<td>Syn-ack flag</td>
</tr>
<tr>
<td>receive window</td>
<td>Receive window</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>Urg data pointer</td>
<td>Urgent data pointer</td>
</tr>
<tr>
<td>options (variable length)</td>
<td>Options (variable length)</td>
</tr>
<tr>
<td>application data</td>
<td>Application data</td>
</tr>
<tr>
<td>(variable length)</td>
<td></td>
</tr>
</tbody>
</table>
UDP Segment

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SrcPort</td>
<td>DstPort</td>
</tr>
<tr>
<td>Length</td>
<td>Checksum</td>
</tr>
</tbody>
</table>

Data

32 bits
UDP Checksum

- Goal: Detect transmission errors (e.g. flipped bits)
  - Router memory errors
  - Driver bugs
  - Electromagnetic interference
UDP Checksum at the Sender

• Treat the entire segment as 16-bit integer values
• Add them all together (sum)
• Put the 1’s complement in the checksum header field
One’s Compliment

• In bitwise compliment, all of the bits in a binary number are flipped.

• So 1111000011110000 -> 0000111100001111
example: add two 16-bit integers

```
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
```

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result
• Add all the received data together as 16-bit integers
• Add that to the checksum
• If result is not 1111 1111 1111 1111, there are errors!
If our checksum addition yields all ones, are we guaranteed to be error-free?

A. Yes

B. No
If our checksum addition yields all ones, are we guaranteed to be error-free?

A. Yes

B. No: we could have two bit flips that cancel each other out.
UDP Applications

• Latency sensitive
  – Quick request/response (DNS)
  – Network management (SNMP, TFTP)
  – Voice/video chat

• Error correction unnecessary (periodic msgs)

• Communicating with lots of others
What if you want something more reliable than UDP, but faster/not as full featured as TCP?

A. Sorry, you’re out of luck.

B. Write your own transport protocol.

C. Add in the features you want at the application layer.
What if you want something more reliable than UDP, but faster/not as full featured as TCP?

Write your own transport protocol:

i. more control: OS controls scheduling, prioritization

ii. reusability: all application layer protocols can use it – great if you own a datacenter (e.g. Google) and you can run your own environment.
What if you want something more reliable than UDP, but faster/not as full featured as TCP?

Add in the features you want at the application layer.

i. easier than modifying the OS.

ii. write into your application custom features on top of UDP that are specific to your application

iii. don’t overburden the transport layer

Irrespective of which layer we choose, adopting something new is always hard on the Internet!
TCP: send() Blocking

• Recall: With TCP, send() blocks if buffer full.
With TCP, send() blocks if buffer full.

- Does UDP need to block? Should it?

A. Yes, if buffers are full, it should.
B. It doesn’t need to, but it might be useful.
C. No, it does not need to and shouldn’t do so.
With TCP, `send()` blocks if buffer full.

- Does UDP need to block? Should it?
  A. Yes, if buffers are full, it should.
  B. It doesn’t need to, but it might be useful.
  C. No, it does not need to and shouldn’t do so.
Summary

Transport Layer:
• Provides a logical communication between processes/applications
• Packets are called segments at the transport layer
• Transport layer protocol: responsible for adding port numbers (mux/demux segments)

UDP:
• No “frills” protocol
• No state maintained about the packet
• Checksum (1’s complement) over IP + UDP + payload.
  – Can only correct for 1 bit errors.
• Adds port numbers over unreliable network (best effort)
• Applications:
  – Latency sensitive applications: real-time audio, video
  – Communicating with a lot of end-hosts (like DNS)
• UDP Sockets:
  – Do not need to be implemented as blocking system calls for correctness since the only guarantee
    UDP makes is best-effort delivery.
  – However, send/recv can be implemented as blocking system calls depending on the application
Today

- Principles of reliability
  - The Two Generals Problem
- Automatic Repeat Requests
  - Stop and Wait
  - Timeouts and Losses
  - Pipelined Transmission
The Two Generals Problem

- Two army divisions (blue) surround enemy (red)
  - Each division led by a general
  - Both must agree when to simultaneously attack
  - If either side attacks alone, defeat

- Generals can only communicate via messengers
  - Messengers may get captured (unreliable channel)
The Two Generals Problem

- How to coordinate?
  - Send messenger: “Attack at dawn”
  - What if messenger doesn’t make it?
The Two Generals Problem

- How to be sure messenger made it?
  - Send acknowledgment: “I delivered message”
In the “two generals problem”, can the two armies reliably coordinate their attack? (using what we just discussed)

A. Yes (explain how)

B. No (explain why not)
The Two Generals Problem

- Result
  - Can’t create perfect channel out of faulty one
  - *Can only increase probability of success*
Give up? No way!

As humans, we like to face difficult problems.

• We can’t control oceans, but we can build canals
• We can’t fly, but we’ve landed on the moon
• We just need engineering!

What can possibly go wrong....
Engineering

• Concerns
  – Message corruption
  – Message duplication
  – Message loss
  – Message reordering
  – Performance

• Our toolbox
  – Checksums
  – Timeouts
  – Acks & Nacks
  – Sequence numbering
  – Pipelining
Engineering

• Concerns
  – Message corruption
  – Message duplication
  – Message loss
  – Message reordering
  – Performance

• Our toolbox
  – Checksums
  – Timeouts
  – Acks & Nacks
  – Sequence numbering
  – Pipelining

We use these to build Automatic Repeat Request (ARQ) protocols.

(We’ll briefly talk about alternatives at the end.)
Automatic Repeat Request (ARQ)

• Intuitively, ARQ protocols act like you would when using a cell phone with bad reception.
  – Sender: Didn’t hear a response? Speak again.

• Refer to book for building state machines.
  – We’ll look at TCP’s states soon
ARQ Broad Classifications

1. Stop-and-wait
Stop and Wait

We have:
• a sender
• a receiver
• time: represented by downwards arrow
Stop and Wait

Sender sends data and waits till they get the response message from the receiver.

Buffer data, and don’t send till response received
Stop and Wait

- Up next: concrete problems and mechanisms to solve them.
- These mechanisms will build upon each other
- Questions?

Slide 55
Corruption?

- Error detection mechanism: checksum
  - Data good – receiver sends back ACK
  - Data corrupt – receiver sends back NACK
Could we do this with just ACKs or just NACKs?

Error detection mechanism: checksum

• Data good – receiver sends back ACK
• Data corrupt – receiver sends back NACK

A. No, we need them both.
B. Yes, we could do without one of them, but we’d need some other mechanism.
C. Yes, we could get by without one of them.
Could we do this with just ACKs or just NACKs?

- **With only ACK**, we could get by with a timeout.
- **With only NACK**, we couldn’t advance (no good).

A. No, we need them both.
B. Yes, we could do without one of them, but we’d need some other mechanism.
C. Yes, we could get by without one of them.
Timeouts and Losses

- Sender starts a clock. If no response, retry.
• Sender starts a clock. If no response, retry.
Timeouts and Losses

- Sender starts a clock. If no response, retry.

- Probably not a great idea for handling corruption, but it works.

Slide 61
Timeouts and Losses

• Timeouts help us handle message losses too!
• Timeouts help us handle message losses too!
Adding timeouts might create new problems for us to worry about. How many? Examples?

A. No new problems (why not?)
B. One new problem (which is..)
C. Two new problems (which are..)
D. More than two new problems (which are..)
Adding timeouts might create new problems for us to worry about. How many? Examples?

A. No new problems (why not?)
B. One new problem (which is..)
C. Two new problems (which are..)
D. More than two new problems (which are..)
**Sequence Numbering**

**Sender**
- Add a monotonically increasing label to each msg

**Receiver**
- Ignore messages with numbers we’ve seen before
- When pipelining (a few slides from now)
  - Detect gaps in the sequence (e.g., 1,2,4,5)
What is our link utilization with a stop-and-wait protocol?

System parameters:
- Link rate: 8 Mbps (one megabyte per second)
- RTT: 100 milliseconds
- Segment size: 1024 bytes

A. < 0.1 %
B. ≈ 0.1 %
C. ≈ 1 %
D. 1-10 %
E. > 10 %
What is our link utilization with a stop-and-wait protocol?

System parameters:
Link rate: 8 Mbps (one megabyte per second)
RTT: 100 milliseconds
Segment size: 1024 bytes

A. < 0.1 %
B. ≈ 0.1 %
C. ≈ 1 %
D. 1-10 %
E. > 10 %

Big Problem: Performance is determined by RTT, not channel capacity!
Keep multiple segments “in flight”

- Allows sender to make efficient use of the link
- Sequence numbers ensure receiver can distinguish segments
- We’ll talk about “how many” next time (windowing).
Keep multiple segments “in flight”

- Allows sender to make efficient use of the link
- Sequence numbers ensure receiver can distinguish segments
- We’ll talk about “how many” next time (windowing).
What should the sender do here?

A. Start sending all data again from 0.
B. Start sending all data again from 2.
C. Resend just 2, then continue with 4 afterwards.

What information does the sender need to make that decision?

What is required by either party to keep track?