CSCI-1680
Transport Layer I

Nick DeMarinis

Based partly on lecture notes by Rodrigo Fonseca, David Mazières, Phil Levis, John Jannotti
Administrivia

• IP: due tonight!
  – Look for email today/tomorrow about grading meetings + feedback survey

“Between the time you’ve handed in and the demo meeting, you can continue to making small changes and bug fixes and push them to your git repo”
  – OK: Fixing bugs, code cleanup, README
  – Not OK: Implementing RIP, adding new features
• HW2 is out (finally!): Due Monday, Oct 30
• HW3 will be super short: out Oct 31, due Nov 7
• TCP: Should be out tomorrow
  – Gearup on Monday, Oct 23 6-8pm in CIT316
Today

Light overview of the transport layer and TCP
– Why we need TCP
– What components are involved
– What you will do in the project
Transport layer: the story so far

- Provides support for different applications via ports
- OS provides interface to applications via sockets

⇒ For now: transport layer is part of OS, service provided to apps
Port numbers are part of these headers => OS uses these to map to sockets
Motivation: sending a big file

A problem, in pseudocode:

```bash
$ cp ~/dir/all-my-files.zip ~/some-other-dir

$ scp ~/dir/all-my-files.zip 1.2.3.4:/some-other-dir
```

What are some challenges with implementing the network part?
Motivation: sending a big file

A problem, in pseudocode:

```go
func sender() {
    fd, _ := os.Open("all-my-files.zip")
    conn, _ := net.Dial("1.2.3.4:80")
    buf := ReadTheWholeFile(fd)
    conn.Write(buf)
}

func receiver() {
    conn, err := net.Listen(":80")
    buf := make([]byte, . . .)
    conn.Read(buf)
    fd = os.Open("copy-of-files.zip")
    fd.Write(buf)
}

What are some challenges with implementing the network part?
Motivation: sending a big file

A problem, in pseudocode:

```go
func sender() {
    fd, _ := os.Open("all-my-files.zip")
    conn, _ := net.Dial("1.2.3.4:80")
    buf := ReadTheWholeFile(fd)
    conn.Write(buf)
}
```

```go
func receiver() {
    conn, err := net.Listen(":80")
    buf := make([]byte, . . .)
    conn.Read(buf)
    fd = os.Open("copy-of-files.zip")
    fd.Write(buf)
}
```

⇒ How do we get data from A->B, reliably?
How does the transport layer help us do this?
UDP: User Datagram Protocol

Send a message between ports... and nothing else

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

Data
UDP: What could possibly go wrong?

Map of the Internet, 2021 (via BGP)
OPTE project
Problem: Reliability

Packets could...

• Dropped packets
• Duplicate packets
• Packets arrive out of order

Multiple hops and paths => Lots of opportunities for failure!
=> TCP has mechanisms to deal with this
Also: **performance** challenges

- *Hosts* have different (and unknown!) resources

- *Network* has unknown resources
  
  => Varying RTT, link bandwidth
Also: performance challenges

- **Hosts** have different (and unknown!) resources
  
  => *Flow control*: how much data can we send to receiver?

- **Network** has unknown resources
  
  => Varying RTT, link bandwidth
  
  => *Congestion control*: must not overload network
Also: **performance** challenges

- **Hosts** have different (and unknown!) resources
  => *Flow control*: how much data can we send to receiver?

- **Network** has unknown resources
  => Varying RTT, link bandwidth
  => *Congestion control*: must not overload network

---

Two performance goals:
1. Must not overwhelm receiver, or network (**critical**!!)
2. Maximize throughput => best performance
So how does it work?
TCP: the big picture
TCP – Transmission Control Protocol

• Service model: “reliable, connection oriented, full duplex ordered byte stream”
• Flow control: If one end stops reading, writes at other eventually stop/fail
• Congestion control: Keeps sender from overloading the network
TCP: Key features

• Initially: RFC 793 (1981) (+ many others now)

• Creates concept of **connections** between two endpoints
  => Each connection has its own state
TCP: Key features

- Initially: RFC 793 (1981) (+ many others now)

- Creates concept of **connections** between two endpoints
  
  =>$> $ Each connection has its own state

- **End-to-end protocol**
  
  -- Minimal assumptions on the network
  
  -- All mechanisms run on the end points (ie, not routers)
TCP: Key features

• Initially: RFC 793 (1981) (+ many others now)

• Creates concept of connections between two endpoints
  => Each connection has its own state

• End-to-end protocol
  – Minimal assumptions on the network
  – All mechanisms run on the end points (ie, not routers)

Why is this important?
TCP Header

<table>
<thead>
<tr>
<th>Data Offset</th>
<th>Reserved</th>
<th>U</th>
<th>A</th>
<th>C</th>
<th>P</th>
<th>S</th>
<th>T</th>
<th>R</th>
<th>S</th>
<th>Y</th>
<th>N</th>
<th>FIN</th>
<th>Window Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>URG</td>
<td>ACK</td>
<td>PSH</td>
<td>RST</td>
<td>SYN</td>
<td>FIN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Source Port
- Destination Port
- Sequence Number
- Acknowledgement Number
- Checksum
- Urgent Pointer
- Options
- Data

20 Bytes
Important Header Fields

• **Ports: multiplexing**
• **Sequence number**
  – Where segment is in the stream (in `bytes`)
• **Acknowledgment Number**
  – Next expected sequence number
• **Window**
  – How much data you’re willing to receive
• **Flags**...
Important Header Fields: Flags

- SYN:
- ACK:
- FIN:

- RST: reset connection (used for errors)
- PSH: push data to the application immediately
- URG: whether there is urgent data
Important Header Fields: Flags

- **SYN**: establishes connection ("synchronize")
- **ACK**: this segment ACKs some data (all packets except first)
- **FIN**: close connection (gracefully)
- **RST**: reset connection (used for errors)
- **PSH**: push data to the application immediately
- **URG**: whether there is urgent data
Less important header fields

- **Checksum**: Very weak, like IP
  - Has weird semantics ("pseudo header"), more on this later…

- Data Offset: used to indicate TCP options (mostly unused)
- Urgent Pointer
TCP Standards: The Many RFCs

- RFC793 (Original)
- RFC1122 (Some corrections)
- RFC5681 (Congestion control)
- RFC7414 (Roadmap to TCP RFCs)
- Various Errata...
TCP Standards: The Many RFCs

**RFC documents** [edit]

- RFC 793 – TCP v4
- RFC 1122 – includes some error corrections for TCP
- RFC 1323 – TCP Extensions for High Performance [Obsoleted by RFC 7323]
- RFC 1948 – Defending Against Sequence Number Attacks
- RFC 2018 – TCP Selective Acknowledgment Options
- RFC 5681 – TCP Congestion Control
- RFC 6247 – Moving the Undeployed TCP Extensions RFC 1072, 1106, 1110, 1145, 1146, 137
- RFC 6298 – Computing TCP’s Retransmission Timer
- RFC 6824 – TCP Extensions for Multipath Operation with Multiple Addresses
- RFC 7323 – TCP Extensions for High Performance
- RFC 7414 – A Roadmap for TCP Specification Documents
- RFC 9293 – Transmission Control Protocol (TCP)
<table>
<thead>
<tr>
<th>STD: 7</th>
<th>Internet Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request for Comments: 9293</td>
<td>W. Eddy, Ed.</td>
</tr>
<tr>
<td>Obsoletes: 793, 879, 2873, 6093, 6429, 6528, 6691</td>
<td>MTI Systems</td>
</tr>
<tr>
<td>Updates: 1011, 1122, 5961</td>
<td>August 2022</td>
</tr>
<tr>
<td>Category: Standards Track</td>
<td></td>
</tr>
<tr>
<td>ISSN: 2070–1721</td>
<td></td>
</tr>
</tbody>
</table>
Establishing a Connection

**Goals**
- Contact the other side (or error)
- Both sides agree on initial sequence numbers
Establishing a Connection

- Three-way handshake
  - Two sides agree on respective initial sequence nums
- If no one is listening on port: OS may send RST
- If server is overloaded: ignore SYN
- If no SYN-ACK: retry, timeout
Summary of TCP States

Connection Establishment

Passive open
- SYN_RCVD
  - SYN/SYN + ACK
  - SYN/SYN + ACK
  - Send/SYN
- LISTEN
  - SYN_SENT
    - SYN/SYN + ACK
    - SYN + ACK/ACK
- ESTABLISHED
  - ACK
  - FIN/ACK
- CLOSE_WAIT
  - FIN/ACK
  - CLOSE
- LAST_ACK
  - FIN/ACK
- CLOSED
  - TIME_WAIT
  - FIN/ACK
  - FIN

Active open
- SYN
  - SYN/SYN + ACK
  - FIN/ACK

Passive close:
- Can still send!

Active close:
- Can still receive

Timeout after two segment lifetimes
TCP State Diagram

**CLOSED**
- (Start)
- LISTEN/-
- LISTEN
- SYN
- SYN/SYN+ACK
- SYN/SYN+ACK (simultaneous open)
- SYN+ACK/ACK
- SYN+ACK
- SYN/SYN+ACK
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
- SYN/SENT
- SYN
- CONNECT/SYN (Step 1 of the 3-way-handshake)
- CLOSE/-
- CLOSE/-
- RST/-
- SYN/SYN+ACK
- DATA EXCHANGE OCCURS
- ESTABLISHED (Step 3 of the 3-way-handshake)
Sequence numbers

How to pick the initial sequence number?

• Protocols based on relative sequence numbers based on starting value
• Why not start at 0?

• RFC9293, Sec 3.4.1: Procedure for picking ISN, based on timer and cryptographic hash
  => For project, just pick a random integer :)}
Relative Sequence Numbering

- Ethernet II, Src: Apple_cd:6a:23 (c8:89:f3:cd:6a:23), Dst: IntelCor_63:c4:45 (0)
  - Transmission Control Protocol, Src Port: 49719, Dst Port: 22, Seq: 0, Len: 0
    - Source Port: 49719
    - Destination Port: 22
    - [Stream index: 8]
      [Conversation completeness: Complete, WITH_DATA (31)]
      [TCP Segment Len: 0]
    - Sequence Number: 0 (relative sequence number)
  - Sequence Number (raw): 200828645
    - [Next Sequence Number: 1 (relative sequence number)]
    - Acknowledgment Number: 0
    - Acknowledgment number (raw): 0
    - 1011 .... = Header Length: 44 bytes (11)
  - Flags: 0x002 (SYN)
How do we tell two connections apart?

=> Port numbers
   – 5-tuple (proto., source IP, source port, dest IP, dest port) => 1 Connection
   – Kernel maintains socket table: maps (5-tuple) => Socket

• If a 5-tuple is reused => new ISN, so sequence numbers likely out of range from past connection
Netstat

deeemer@vesta ~/Development % netstat -an
Active Internet connections (including servers)

<table>
<thead>
<tr>
<th>Proto</th>
<th>Recv-Q</th>
<th>Send-Q</th>
<th>Local Address</th>
<th>Foreign Address</th>
<th>(state)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>10.3.146.161.51094</td>
<td>104.16.248.249.443</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>10.3.146.161.51076</td>
<td>172.66.43.67.443</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp6</td>
<td>0</td>
<td>0</td>
<td>2620:6e:6000:900.51074</td>
<td>2606:4700:3108::443</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>10.3.146.161.51065</td>
<td>35.82.230.35.443</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>10.3.146.161.51055</td>
<td>162.159.136.234.443</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>10.3.146.161.51038</td>
<td>17.57.147.5.5223</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp6</td>
<td>0</td>
<td>0</td>
<td>*.*51036</td>
<td><em>.</em></td>
<td>LISTEN</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>*.*51036</td>
<td><em>.</em></td>
<td>LISTEN</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>127.0.0.1.14500</td>
<td><em>.</em></td>
<td>LISTEN</td>
</tr>
</tbody>
</table>
Keeping state: the TCB

State for a TCP connection kept in Transmission Control Buffer (TCB)

- Keeps initial sequence numbers, connection state, send/recv buffers, status of unACK’d segments, ...

- When to allocate?
  - Server: listening on a connection*
  - Client: Initiating a connection (sending a SYN)
  - Server: accepting a new connection (receiving SYN)
Recall: the socket table

- Each connection has an associated TCB in the kernel
- For each packet, kernel maps the 5-tuple (tcp/udp, local IP, local port, remote IP, remote port) => socket
- Depending on socket type, socket contains TCB

```
deemer@vesta ~ % netstat -anl
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address Foreign Address (state)
tcp4 0 0 172.17.48.121.56915 192.168.1.58.7000 SYN_SENT
tcp4 0 0 172.17.48.121.56908 142.250.80.35.443 ESTABLISHED
tcp4 0 0 172.17.48.121.56887 13.225.231.50.80 ESTABLISHED
...
tcp4 0 0 *.22 *.* LISTEN
```
Two “types” of sockets:

- “Normal” sockets
- Listen sockets

<table>
<thead>
<tr>
<th>Proto</th>
<th>Recv-Q</th>
<th>Send-Q</th>
<th>Local Address</th>
<th>Foreign Address</th>
<th>(state)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>172.17.48.121.56915</td>
<td>192.168.1.58.7000</td>
<td>SYN_SENT</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>172.17.48.121.56908</td>
<td>142.250.80.35.443</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>172.17.48.121.56887</td>
<td>13.225.231.50.80</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td><em>.</em></td>
<td><em>.</em></td>
<td>LISTEN</td>
</tr>
</tbody>
</table>
“Normal” sockets

- Connection between two specific endpoints
- Can send/recv data

Listen sockets

- Created by receiver to accept new connections
- When a client connects, client info gets queued by kernel
- When server process calls accept(), a new (“normal”) socket is created between the server and that client
NOTA BENE: This diagram is only a summary and must not be taken as the total specification. Many details are not included.
SYN flooding

What happens if you send a huge number of SYN packets?
A hacky solution: SYN cookies

• Don’t allocate TCB on first SYN
• Encode some state inside the initial sequence number that goes back to the client (in the SYN+ACK)
• What gets encoded?
  – Coarse timestamp
  – Hash of connection IP/port
  – Other stuff (implementation dependent)
• Better ideas?
Next class

• Sending data over TCP
Connection Termination

• FIN bit says no more data to send
  – Caused by close or shutdown
  – Both sides must send FIN to close a connection
• Typical close
The IPv4 Header

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>4 bits</td>
</tr>
<tr>
<td>IHL</td>
<td>4 bits</td>
</tr>
<tr>
<td>TOS</td>
<td>8 bits</td>
</tr>
<tr>
<td>Total length</td>
<td>16 bits</td>
</tr>
<tr>
<td>Identification</td>
<td>20 bits</td>
</tr>
<tr>
<td>Flags</td>
<td>8 bits</td>
</tr>
<tr>
<td>Fragment offset</td>
<td>16 bits</td>
</tr>
<tr>
<td>TTL</td>
<td>8 bits</td>
</tr>
<tr>
<td>Protocol</td>
<td>8 bits</td>
</tr>
<tr>
<td>Header checksum</td>
<td>8 bits</td>
</tr>
<tr>
<td>Source address</td>
<td>40 bits</td>
</tr>
<tr>
<td>Destination address</td>
<td>40 bits</td>
</tr>
<tr>
<td>Options</td>
<td>0-40 bytes</td>
</tr>
<tr>
<td>Data</td>
<td>Up to 65536 bytes</td>
</tr>
</tbody>
</table>

Defined by RFC 791
RFC (Request for Comment): defines network standard