CS 241 Honors
TCP

Ben Kurtovic, Jonathan Wexler

University of Illinois Urbana-Champaign

November 8, 2017
What is TCP?
Overview

What is TCP?

```c
int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
connect(sock_fd, r->ai_addr, r->ai_addrlen);
char * buffer = "Here's Some Data!";
write(sock_fd, buffer, strlen(buffer));
close(sock_fd);
```

But how does this actually work?

Ben Kurtovic, Jonathan Wexler (UIUC)
Overview

What is TCP?

```c
int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
connect(sock_fd, r->ai_addr, r->ai_addrlen);
char * buffer = "Here's Some Data!";
write(sock_fd, buffer, strlen(buffer));
close(sock_fd);
```

But how does this actually work?
Internet Layers
OSI model

- Internet is built in layers of protocols
- Defined by what is provided to them (layers below), and what they must provide (layers above)

<table>
<thead>
<tr>
<th>OSI Model</th>
<th>Data unit</th>
<th>Layer</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>7. Application</td>
<td>Network process to application</td>
</tr>
<tr>
<td>Host</td>
<td>Data</td>
<td>6. Presentation</td>
<td>Data representation, encryption and decryption, convert machine dependent data to machine independent data</td>
</tr>
<tr>
<td>Host</td>
<td></td>
<td>5. Session</td>
<td>Interhost communication, managing sessions between applications</td>
</tr>
<tr>
<td>Host</td>
<td>Segments</td>
<td>4. Transport</td>
<td>Reliable delivery of segments between points on a network.</td>
</tr>
<tr>
<td>Media</td>
<td>Packet/Datagram</td>
<td>3. Network</td>
<td>Addressing, routing and (not necessarily reliable) delivery of datagrams between points on a network.</td>
</tr>
<tr>
<td>Media</td>
<td>Bit/Frame</td>
<td>2. Data link</td>
<td>A reliable direct point-to-point data connection.</td>
</tr>
<tr>
<td>Media</td>
<td>Bit</td>
<td>1. Physical</td>
<td>A (not necessarily reliable) direct point-to-point data connection.</td>
</tr>
</tbody>
</table>

Internet model (RFC 1122)

Application layer
Meaningful functionality for the user (e.g. HTTP, FTP, SMTP, SSH) plus common “support” protocols (e.g. DNS, BGP)

Transport layer
Reliable transmission, connection management (TCP), or not (UDP)

Internet layer
Addressing and routing packets through a network, without reliability (IP)

Link layer
Direct connection between hosts, semi-reliable (e.g. Ethernet, Wi-Fi)
TCP refresher

What are we provided? (from IP)

- Ability to send small, discrete packets through the Internet to a specific destination
- Packets are not guaranteed to be delivered
- Packets are not guaranteed to only be delivered once (could be duplicated)
- Packets are not guaranteed to arrive in the same order you send them
TCP refresher

What are we provided? (from IP)

- Ability to send small, discrete packets through the Internet to a specific destination
TCP refresher

What are we provided? (from IP)

- Ability to send small, discrete packets through the Internet to a specific destination
  - Packets are not guaranteed to be delivered
  - Packets are not guaranteed to only be delivered once (could be duplicated)
  - Packets are not guaranteed to arrive in the same order you send them
What are we provided? (from IP)

- Ability to send small, discrete packets through the Internet to a specific destination
  - Packets are not guaranteed to be delivered
  - Packets are not guaranteed to only be delivered once (could be duplicated)
  - Packets are not guaranteed to arrive in the same order you send them

What we must provide (to applications):
TCP refresher

What are we provided? (from IP)

- Ability to send small, discrete packets through the Internet to a specific destination
  - Packets are not guaranteed to be delivered
  - Packets are not guaranteed to only be delivered once (could be duplicated)
  - Packets are not guaranteed to arrive in the same order you send them

What we must provide (to applications):

- A stream-like way of sending data with the guarantee it is delivered
TCP refresher

What are we provided? (from IP)

- Ability to send small, discrete packets through the Internet to a specific destination
  - Packets are not guaranteed to be delivered
  - Packets are not guaranteed to only be delivered once (could be duplicated)
  - Packets are not guaranteed to arrive in the same order you send them

What we must provide (to applications):

- A stream-like way of sending data with the guarantee it is delivered

More specifically:
Connection management

- How do we start talking, how do we stop talking?
TCP refresher

- Connection management
  - How do we start talking, how do we stop talking?

- Reliable transmission
  - What happens if packets are lost?
  - What happens if packets are received out of order?
  - What happens if packets are corrupted?
TCP refresher

- **Connection management**
  - How do we start talking, how do we stop talking?

- **Reliable transmission**
  - What happens if packets are lost?
  - What happens if packets are received *out of order*?
  - What happens if packets are corrupted?

- **Flow control**
  - How do we avoid flooding our destination?
Connection management
- How do we start talking, how do we stop talking?

Reliable transmission
- What happens if packets are lost?
- What happens if packets are received *out of order*?
- What happens if packets are corrupted?

Flow control
- How do we avoid flooding our destination?

Congestion control
- How do we avoid flooding *the network*?
- How can we play fairly with other users?
Reliable Transmission
Two Generals’ Problem

First, a quick thought experiment!

- General Tso and Colonel Sanders are on opposite sides of a hill
- They need to agree on a time to attack, and will only attack at that time if they’re sure the other one will too
- They can’t be sure that their messengers won’t get lost or captured before they reach the other

Can they agree to attack?
Two Generals’ Problem
“Let’s attack tomorrow”
Two Generals’ Problem

“Let’s attack tomorrow”
Two Generals’ Problem

“Let’s attack tomorrow”

“Okay”
Two Generals’ Problem

“Let’s attack tomorrow”

“Okay”

???
Two Generals’ Problem

“Let’s attack tomorrow”

“Okay”

“Okay”

“Okay”
Two Generals’ Problem

“Let’s attack tomorrow”

“Okay”

“Okay”

“Okay”
Moral of the story

- Two Generals’ Problem is proven unsolvable: there is no general solution to ensure both sides communicating over an unreliable link can agree on something.
- So, we can’t design an algorithm where both the sender and the receiver know that they agree with each other.
- TCP is designed to deal with some degree of uncertainty.
  - Acknowledgements are necessary for reliable transmission.
All we get from the user is a sequence of bytes (every time they call `write/send`).

Message gets broken up into *segments* up to size MSS:
- Maximum segment size: based on how much the network layer can transmit at once (IP packet fragmentation is possible, though very undesirable).

We need some way to know which segments were received.
Reliable transmission primer

- All we get from the user is a sequence of bytes (every time they call `write/send`)
- Message gets broken up into *segments* up to size MSS
  - Maximum segment size: based on how much the network layer can transmit at once (IP packet fragmentation is possible, though very undesirable)
- We need some way to know which segments were received
- Solution: number the bytes (*sequence number*)
Reliable transmission: Stop-and-wait
Reliable transmission: Stop-and-wait
Reliable transmission: Stop-and-wait

Segment $n$

Acknowledge $n$
Reliable transmission: Stop-and-wait

Segment $n$

Acknowledge $n$

Segment $n+1$
Reliable transmission: Stop-and-wait

Segment $n$

Acknowledge $n$

Segment $n+1$

Acknowledge $n+1$
Reliable transmission: Stop-and-wait
Reliable transmission: Stop-and-wait
Reliable transmission: Stop-and-wait

Segment $n$

Timeout $n$
Reliable transmission: Stop-and-wait

Segment $n$

Timeout $n$

Segment $n$

Acknowledge $n$
Reliable transmission: Stop-and-wait

Segment $n$

Timeout $n$
Reliable transmission: Stop-and-wait

Timeout $n$

Segment $n$

Acknowledge $n$
Reliable transmission: Stop-and-wait
Reliable transmission: Stop-and-wait
Reliable transmission: Stop-and-wait

- Legitimate way of sending data reliably
- Ensure each segment is received before sending the next one
- Sequence number ensures data is kept in order
Reliable transmission: Stop-and-wait

- Legitimate way of sending data reliably
- Ensure each segment is received before sending the next one
- Sequence number ensures data is kept in order
- Any Problems?

Each packet is bounded by the maximum transmission unit size (MTU)
This is π \approx 1500 bytes on Ethernet, for example
We'll need to wait the Round Trip Time (RTT) to get the acknowledgement before sending the next packet
This is π \approx 100 ms for trans-US
This means we can only send 1500B / 1sec = 15KB/s

(It would take 36 hours to download an average movie)

Clearly we can do better
Send packets concurrently!
Reliable transmission: Stop-and-wait

- Legitimate way of sending data reliably
- Ensure each segment is received before sending the next one
- Sequence number ensures data is kept in order
- Any Problems? Slow!

Each packet is bounded by the maximum transmission unit size (MTU) which is $\pi 1500$ bytes on Ethernet, for example. We'll need to wait the Round Trip Time (RTT) to get the acknowledgement before sending the next packet, which is $\pi 100$ ms for trans-US. This means we can only send $\frac{1500}{1}$ bytes per second, or 15KB/s! (It would take 36 hours to download an average movie.) Clearly we can do better — send packets concurrently!
Reliable transmission: Stop-and-wait

- Legitimate way of sending data reliably
- Ensure each segment is received before sending the next one
- Sequence number ensures data is kept in order
- Any Problems? Slow!
  - Each packet is bounded by the maximum transmission unit size (MTU)
  - This is $\approx 1500$ bytes on Ethernet, for example
  - We’ll need to wait the Round Trip Time (RTT) to get the acknowledgement before sending the next packet
  - This is $\approx 100\text{ms}$ for trans-US
  - This means we can only send $1500B/0.1\text{sec} = 15\text{KB/s}$!
  - (It would take 36 hours to download an average movie)
Reliable transmission: Stop-and-wait

- Legitimate way of sending data reliably
- Ensure each segment is received before sending the next one
- Sequence number ensures data is kept in order
- Any Problems? Slow!
  - Each packet is bounded by the maximum transmission unit size (MTU)
  - This is \( \approx 1500 \) bytes on Ethernet, for example
  - We’ll need to wait the Round Trip Time (RTT) to get the acknowledgement before sending the next packet
  - This is \( \approx 100\text{ms} \) for trans-US
  - This means we can only send \( 1500\text{B}/.1\text{sec} = 15\text{KB/s}! \)
  - (It would take 36 hours to download an average movie)
- Clearly we can do better
  - Send packets concurrently!
How TCP Works

- In each segment, we’ll have a sequence number and a acknowledgement number
  - Sequence number holds what byte number I’m currently sending
  - Acknowledgement number holds what the next byte I’m expecting to receive is
- If we don’t get an acknowledgement number greater than the sequence number of a segment plus the length of the data in that segment within some amount of time after sending it, resend the segment!
How TCP Works

1. Send some data in separate segment. Keep a copy of all of these segment in a queue with a timer for each one.
2. When we receive an acknowledgement number, remove all packets with a sequence number plus data length less than the acknowledgement number from the queue.
3. If a segment on the queue’s timer expires, resend that segment.
Reliable transmission

seqnum=0xAA000000
seqnum=0xAA000200
seqnum=0xAA000400
Reliable transmission

seqnum=0xAA000000
seqnum=0xAA000200
seqnum=0xAA000400  acknum=0xAA000200
seqnum=0xAA000400  acknum=0xAA000400
seqnum=0xAA000600  acknum=0xAA000600
Reliable transmission

seqnum=0xAA000000
seqnum=0xAA000200
seqnum=0xAA000400
Reliable transmission

segnum=0xAA000000
segnum=0xAA000200
segnum=0xAA000400  acknum=0xAA000200
segnum=0xAA000400  acknum=0xAA000400
Reliable transmission

\[\text{seqnum}=0\timesAA000400\]

\[\text{seqnum}=0\timesAA000200\]

\[\text{seqnum}=0\timesAA000000\]

\[\text{acknum}=0\timesAA000200\]

\[\text{acknum}=0\timesAA000400\]
Reliable transmission

- seqnum=0xAA000000
- seqnum=0xAA000200 (red)
- seqnum=0xAA000400
- acknum=0xAA000200
- acknum=0xAA000200
Reliable transmission

- seqnum=0xAA000000
- seqnum=0xAA000200
- seqnum=0xAA000400  acknum=0xAA000200
- acknum=0xAA000200

- seqnum=0xAA000200
Reliable transmission

seqnum=0xAA000000
seqnum=0xAA000200
seqnum=0xAA000400
acknum=0xAA000200

acknum=0xAA000200

seqnum=0xAA000200
acknum=0xAA000600
Reliable transmission

seqnum=0xAA000000
seqnum=0xAA000200
seqnum=0xAA000400
seqnum=0xAA000600
Reliable transmission

seqnum=0xAA000000
seqnum=0xAA000200
seqnum=0xAA000400
seqnum=0xAA000600  acknum=0xAA000000
acknum=0xAA000000
acknum=0xAA000000
acknum=0xAA000000
Reliable transmission

seqnum=0xAA000000
seqnum=0xAA000200
seqnum=0xAA000400
acknum=0xAA000000
seqnum=0xAA000600
acknum=0xAA000000
acknum=0xAA000000
seqnum=0xAA000000
Reliable transmission

segnum=0xAA000000
segnum=0xAA000200
segnum=0xAA000400
segnum=0xAA000600  acknum=0xAA000000
segnum=0xAA000800  acknum=0xAA000000
segnum=0xAA000000
segnum=0xAA000000  acknum=0xAA000000
segnum=0xAA000000  acknum=0xAA000000
segnum=0xAA000000  acknum=0xAA000000
segnum=0xAA000000  acknum=0xAA000000
segnum=0xAA000000  acknum=0xAA000000
Connection Management
Connection management

How do we start a connection?

- "Three-way handshake" between client and server
How do we start a connection?

- "Three-way handshake" between client and server

0 [Server]: I’m ready to talk to people! (listen)
Connection management

How do we start a connection?

- "Three-way handshake" between client and server

0  [Server]: I’m ready to talk to people! (listen)

1  [Client]: Hi, can we talk? Here’s my initial sequence number. (connect)
Connection management

How do we start a connection?

- "Three-way handshake" between client and server

0 [Server]: I’m ready to talk to people! (listen)

1 [Client]: Hi, can we talk? Here’s my initial sequence number. (connect)

2 [Server]: OK, I acknowledge your sequence number. Here’s my initial sequence number. (accept)
Connection management

How do we start a connection?

- "Three-way handshake" between client and server

0  [Server]: I’m ready to talk to people! (listen)

1  [Client]: Hi, can we talk? Here’s my initial sequence number. (connect)

2  [Server]: OK, I acknowledge your sequence number. Here’s my initial sequence number. (accept)

3  [Client]: OK, I acknowledge your sequence number.

Questions:
- What's the initial sequence number? 0?
- No, it's random. Why?
- What happens if a message is lost?
How do we start a connection?

- "Three-way handshake" between client and server

0. [Server]: I’m ready to talk to people! (listen)

1. [Client]: Hi, can we talk? Here’s my initial sequence number. (connect)

2. [Server]: OK, I acknowledge your sequence number. Here’s my initial sequence number. (accept)

3. [Client]: OK, I acknowledge your sequence number.

Questions:

- What’s the initial sequence number?
How do we start a connection?

- "Three-way handshake" between client and server

0 [Server]: I’m ready to talk to people! (listen)

1 [Client]: Hi, can we talk? Here’s my initial sequence number. (connect)

2 [Server]: OK, I acknowledge your sequence number. Here’s my initial sequence number. (accept)

3 [Client]: OK, I acknowledge your sequence number.

Questions:

- What’s the initial sequence number? 0?
How do we start a connection?

- "Three-way handshake" between client and server

0  [Server]: I’m ready to talk to people! (listen)

1  [Client]: Hi, can we talk? Here’s my initial sequence number. (connect)

2  [Server]: OK, I acknowledge your sequence number. Here’s my initial sequence number. (accept)

3  [Client]: OK, I acknowledge your sequence number.

Questions:

- What’s the initial sequence number? 0? No, it’s random. Why?
Connection management

How do we start a connection?

- "Three-way handshake" between client and server

0] [Server]: I’m ready to talk to people! (listen)
1] [Client]: Hi, can we talk? Here’s my initial sequence number. (connect)
2] [Server]: OK, I acknowledge your sequence number. Here’s my initial sequence number. (accept)
3] [Client]: OK, I acknowledge your sequence number.

Questions:

- What’s the initial sequence number? 0? No, it’s random. Why?
- What happens if a message is lost?
TCP segment anatomy

- 16-bit source port number
- 16-bit destination port number
- 32-bit sequence number
- 32-bit acknowledgment number
- 4-bit header length
- reserved (6 bits)
- URG
- ACK
- PSH
- RST
- SYN
- FIN
- 16-bit window size
- 16-bit TCP checksum
- 16-bit urgent pointer
- options (if any)
- data (if any)

20 bytes
TCP segment anatomy

- 16-bit source port number
- 16-bit destination port number
- 32-bit sequence number (Corresponding to first byte of data in segment)
- 32-bit acknowledgment number (Corresponding to next expected byte of data)
- 4-bit header length
  - reserved (6 bits)
  - URG, ACK, PSH, RST, SYN, FIN
- 16-bit window size
- 16-bit TCP checksum
- 16-bit urgent pointer
- Options (if any)
- Data (if any)
TCP segment anatomy

- 16-bit source port number
- 16-bit destination port number
- 32-bit sequence number
  
  Corresponding to first byte of data in segment
- 32-bit acknowledgment number
  
  Corresponding to next expected byte of data
- 4-bit header length
  
  reserved (6 bits)
- URG
- ACK
- PSH
- RST
- SYN
- FIN
- 16-bit window size
- 16-bit TCP checksum
- 16-bit urgent pointer
- options (if any)
- data (if any)
TCP connection handshake

SYN=1
seqnum=0xDEADBEEF
acknum=0x00000000
TCP connection handshake

SYN=1
seqnum=0xDEADBEEF
acknum=0x00000000

SYN=1, ACK=1
seqnum=0xCAFEBABE
acknum=0xDEADBEEF0
TCP connection handshake

1. SYN=1
   - seqnum=0xDEADBEEF
   - acknum=0x00000000

2. SYN=1, ACK=1
   - seqnum=0xCAFEBABE
   - acknum=0xDEADBEEF

3. ACK=1
   - seqnum=0xDEADBEEF
   - acknum=0xCAFEBABE
Now back to fast transmission!

- How do we know how many packets we can send at once?
"write(fd, buf, 9999999999)"
"write(fd, buf, 9999999999)"
"write(fd, buf, 999999999)"
"write(fd, buf, 9999999999)"
Reliable transmission: Challenges

What’s wrong with this?

Too much data at once!

What if our client has a smaller buffer than we do?

Solution: Have our client tell us how much they want!
Reliable transmission: Challenges

What’s wrong with this?

- Too much data at once!
- What if our client has a smaller buffer than we do?
- Solution: Have our client tell us how much they want!
Flow control

The TCP header consists of the following fields:

- 16-bit source port number
- 16-bit destination port number
- 32-bit sequence number
- 32-bit acknowledgment number
- 4-bit header length (reserved for 6 bits)
- URG, ACK, PSH, RST, SYN,_FIN
- 16-bit window size
- 16-bit TCP checksum
- 16-bit urgent pointer
- Options (if any)
- Data (if any)
Flow control

- **Receiver window**: number of bytes sender of segment is willing to receive
- By default: goes up to $2^{16} - 1$, corresponds to size of buffer in OS

Still restricts throughput, but not nearly as much as stop-and-wait

Max $2^{16}$ bytes per RTT

⇡ 640KB/s assuming RTT = 100ms
Flow control

- **Receiver window**: number of bytes sender of segment is willing to receive
- By default: goes up to $2^{16} - 1$, corresponds to size of buffer in OS
- Still restricts throughput, but not nearly as much as stop-and-wait
  - Max $2^{16} - 1$ bytes per RTT $\approx 640\text{KB/s}$ assuming RTT $= 100\text{ms}$
Reliable transmission: error checking

The diagram illustrates the structure of a TCP header, including fields such as:
- 16-bit source port number
- 16-bit destination port number
- 32-bit sequence number
- 32-bit acknowledgment number
- 4-bit header length
- 16-bit window size
- 16-bit TCP checksum
- Options (if any)
- Data (if any)
16-bit checksum is just the one's complement sum of all 16-bit words in the segment (including the header), then one's complemented.

If checksum fails in receiver, just discard packet, like we didn’t get it.

Commonly will also have stronger checksums at the link layer (e.g. for Ethernet, Wi-Fi), and possibly also at the application layer.

Why not just rely on the TCP checksum?
Congestion control

How do we avoid flooding the network?
How do we avoid flooding the network?

- Network has a maximum amount of data (*capacity*) we can push through it at one time (based on bandwidth of wires, load of intermediate routers, etc.)
- Dynamic and somewhat unpredictable → need to adapt quickly
Congestion control

How do we avoid flooding the network?

- Network has a maximum amount of data \((capacity)\) we can push through it at one time (based on bandwidth of wires, load of intermediate routers, etc.)
- Dynamic and somewhat unpredictable \(\rightarrow\) need to adapt quickly
- Solution: introduce a *congestion window*
  - While the receiver window tells you how much the recipient is willing to receive, the congestion window tells you how much you are able to send
  - How much you actually send is the smaller of these two (roughly)
Congestion control

How do we avoid flooding the network?

- Network has a maximum amount of data (*capacity*) we can push through it at one time (based on bandwidth of wires, load of intermediate routers, etc.)
- Dynamic and somewhat unpredictable → need to adapt quickly
- Solution: introduce a *congestion window*
  - While the receiver window tells you how much the recipient is willing to receive, the congestion window tells you how much you are able to send
  - How much you actually send is the smaller of these two (roughly)
- Arguably the most complicated part of TCP: dozens of variants exist and is an ongoing area of research
Congestion control: AIMD

Basic sketch:

- Start congestion window at a small value (1 MSS)
- Keep increasing the window periodically until a loss occurs—this means we are sending too much, so decrease it and try again
  - Additive increase: Increase window at a linear rate
  - Multiplicative decrease: Decrease window at an exponential rate
- AIMD ensures fairness between multiple connections!

TCP Sawtooth, red curve represents the network capacity
Congestion control: Stages

- Three stages (TCP Reno):
  - Slow start: Exponential increase until loss or threshold $ssthresh$ is reached
  - Congestion avoidance: Linear increase until loss
  - Fast recovery: If loss is due to duplicate ACKs, cut window in half and increase linearly
  - If loss due to timeout, drop down to slow start
Connection termination

- *Four-way handshake (FIN/ACK, FIN/ACK)*
- Both sides can close independently

```
Initiator      Receiver

ESTABLISHED
connection

active close
FIN_WAIT_1

FIN_WAIT_2

TIME_WAIT

CLOSED

FIN

ACK

FIN

ACK

CLOSED

CLOSED

ESTABLISHED
connection

CLOSE_WAIT
passive close

LAST_ACK
```
Connection termination: TIME_WAIT

- Lasts for 2 MSL (maximum segment lifetime), \( \approx 2 \) mins
- Prevents delayed/out-of-order packets from being picked up by a subsequent connection (rare)
- Gives enough time for last ACK to be received and resent if necessary
Connection termination: TIME_WAIT

- Lasts for 2 MSL (maximum segment lifetime), \( \approx 2 \) mins
- Prevents delayed/out-of-order packets from being picked up by a subsequent connection (rare)
- Gives enough time for last ACK to be received and resent if necessary
- Prevents errors and data loss!
- Don’t use SO_REUSEADDR except for debugging!
Implementation (things to know)
Transmission Control Block: stores TCP parameters (including receiver window) in operating system

Processing new data and sending out ACKs happens asynchronously in OS, not when you call read/write

Thus, packet segmentation is not reliable

One write call may be received through multiple read calls, or vice versa

Except on localhost...
Transmission Control Block: stores TCP parameters (including receiver window) in operating system

Processing new data and sending out ACKs happens asynchronously in OS, not when you call read/write

Thus, packet segmentation is not reliable

One write call may be received through multiple read calls, or vice versa

- Except on localhost...

This is why application-layer protocols have sizes and headers/footers

Remember, it’s a stream
TCP handshake takes $\frac{3}{2}$ round trips

Congestion control takes several round trips to fully warm up

If you're using SSL, it's even more (2 extra round trips)
Overhead

- TCP handshake takes $\frac{3}{2}$ round trips
- Congestion control takes several round trips to fully warm up
- If you're using SSL, it's even more (2 extra round trips)
- Reusing existing connections is very desirable (compare HTTP/1.0 with HTTP/1.1)
Overhead

- TCP handshake takes $\frac{3}{2}$ round trips
- Congestion control takes several round trips to fully warm up
- If you're using SSL, it's even more (2 extra round trips)
- Reusing existing connections is very desirable (compare HTTP/1.0 with HTTP/1.1)
- 100% network utilization is impossible (congestion sawtooth peaks around 75%)
  - Use UDP if you want to be ridiculous or greedy—but good luck actually receiving everything
Window scaling

- 64 KB receiver window is too small for many modern networks (long/fat pipes)
Window scaling

- 64 KB receiver window is too small for many modern networks (long/fat pipes)
- Can ”scale” window when establishing a connection, up to 1 GB
- Requires that we allocate a buffer that large in the OS somewhere
- Can be tuned in operating system
  (/proc/sys/net/ipv4/tcp_window_scaling)
- Well-tuned networks can be up to ten times faster!
Nagle’s algorithm

- Reduces overhead of sending many small packets in a short time
  - Say you call `write` ten times at once, each writing 1 byte
  - Old TCP: sends 10 packets (each of size 41 bytes = 410 bytes)
  - Nagle’s algorithm: accumulate writes into one packet (50 bytes)
Nagle’s algorithm

- Reduces overhead of sending many small packets in a short time
  - Say you call `write` ten times at once, each writing 1 byte
  - Old TCP: sends 10 packets (each of size 41 bytes = 410 bytes)
  - Nagle’s algorithm: accumulate writes into one packet (50 bytes)
- Good for many situations, but becomes problematic if you don’t want a delay (e.g. typing interactively)
- Disable with sock option `TCP_NODELAY`
TCP Security
SYN Flood

[Diagram of SYN flood attack]
SYN Flood

107.5.78.142:56000, Seq=2435, Waiting for ACK...

ACK-SYN
SYN Flood
SYN Flood

126.247.142.92:29955, Seq=3213, Waiting for ACK...
17.167.8.197:36068, Seq=8874, Waiting for ACK...
112.118.51.246:13683, Seq=6442, Waiting for ACK...
39.46.47.180:12569, Seq=8031, Waiting for ACK...
92.95.229.217:33139, Seq=9503, Waiting for ACK...
94.128.158.238:25025, Seq=2837, Waiting for ACK...
192.17.41.168:29119, Seq=3985, Waiting for ACK...
54.63.181.93:24224, Seq=4560, Waiting for ACK...
69.221.172.246:23816, Seq=2950, Waiting for ACK...
51.53.241.144:13764, Seq=7801, Waiting for ACK...
...
... ACK-SYN
TCP Veto

Dear Betty
I love CS 241!
Best,
Alice

Seq No: 0
Data Len: 11
Data: “Dear Betty

Seq No: 11
Data Len: 15
Data: “I love CS 241

Seq No: 26
Data Len: 11
Data: “Best,

Alice”
Eavesdropper sees packet and sends another one with the same sequence number and data length

Seq No: 11
Data Len: 15
Data: “ECE391 > CS241\n”

Seq No: 11
Data Len: 15
Data: “I love CS241\n”
TCP Veto

Seq No: 0
Data Len: 11
Data: “Dear Betty\n”

Dear Betty
ECE391 > CS241
Best,
Alice

Seq No: 11
Data Len: 15
Data: “ECE391 > CS241\n”

The “Real” packet gets ignored as a duplicate!

Seq No: 26
Data Len: 11
Data: “Best,\nAlice”
Take **CS/ECE 438**: Communication Networks