CS 241 Honors
Networking and TCP

Ben Kurtovic

University of Illinois Urbana-Champaign

April 11, 2017

Overview

- Regular section focuses on practical networking (what system calls to use, etc.)

Overview

- Regular section focuses on practical networking (what system calls to use, etc.)
    - But how does networking *really* work behind the scenes?

Overview

- Regular section focuses on practical networking (what system calls to use, etc.)
  - But how does networking *really* work behind the scenes?
- Motivation for TCP *(Why do we need it?)*
  - OSI/Internet model (layers of protocols)

Overview

- Regular section focuses on practical networking (what system calls to use, etc.)
    - But how does networking *really* work behind the scenes?
- Motivation for TCP *(Why do we need it?)*
    - OSI/Internet model (layers of protocols)
- Guarantees *(What does it need to do?)*
    - Connection management, reliability, flow control, congestion control

Overview

- Regular section focuses on practical networking (what system calls to use, etc.)
  - But how does networking *really* work behind the scenes?
- Motivation for TCP *(Why do we need it?)*
  - OSI/Internet model (layers of protocols)
- Guarantees *(What does it need to do?)*
  - Connection management, reliability, flow control, congestion control
- Implementation *(How does it do it?)*
  - What's the responsibility of the user, and what's the responsibility of the kernel (or other parts of the network)?

Motivation

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)

OSI Model

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

Source: Wikipedia

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)

Why care about TCP?

We could have multiple classes about any of these layers or any part of them, so why care about TCP in particular?

Why care about TCP?

We could have multiple classes about any of these layers or any part of them, so why care about TCP in particular?

- Often a bottleneck in systems code

Why care about TCP?

We could have multiple classes about any of these layers or any part of them, so why care about TCP in particular?

- Often a bottleneck in systems code
- The OS has a fairly significant role
  - Link layer is more of a hardware problem
  - Internet layer is a router problem
  - Application layer is not a systems problem, it's what the system supports

Why care about TCP?

We could have multiple classes about any of these layers or any part of them, so why care about TCP in particular?

- Often a bottleneck in systems code
- The OS has a fairly significant role
  - Link layer is more of a hardware problem
  - Internet layer is a router problem
  - Application layer is not a systems problem, it's what the system supports
- It's a complex and interesting study of how protocols are designed and evolve

TCP refresher

What we are provided (from IP):

TCP refresher

What we are provided (from IP):

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

TCP refresher

What we are provided (from IP):

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

What we must provide (to applications):

TCP refresher

What we are provided (from IP):

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

What we must provide (to applications):

- A stream-like way of sending data reliably

TCP refresher

What we are provided (from IP):

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

What we must provide (to applications):

- A stream-like way of sending data reliably

More specifically:

TCP refresher

- Connection management
  - How do we start talking, how do we stop talking?
    (last one is surprisingly painful)
  - *Why?*

TCP refresher

- Connection management
  - How do we start talking, how do we stop talking?
    (last one is surprisingly painful)
  - *Why?*
- Reliable transmission
  - What happens if packets are lost?
  - What happens if packets are received *out of order*?
    (How can this happen?)
  - What happens if packets are corrupted?

TCP refresher

- Connection management
  - How do we start talking, how do we stop talking?
    (last one is surprisingly painful)
  - *Why?*
- Reliable transmission
  - What happens if packets are lost?
  - What happens if packets are received *out of order*?
    (How can this happen?)
  - What happens if packets are corrupted?
- Flow control
  - How do we avoid flooding our destination?

TCP refresher

- Connection management
  - How do we start talking, how do we stop talking?
    (last one is surprisingly painful)
  - *Why?*
- Reliable transmission
  - What happens if packets are lost?
  - What happens if packets are received *out of order*?
    (How can this happen?)
  - 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?

Guarantees

Two Generals' Problem

"Let's attack tomorrow"

Two Generals' Problem

"Let's attack tomorrow"

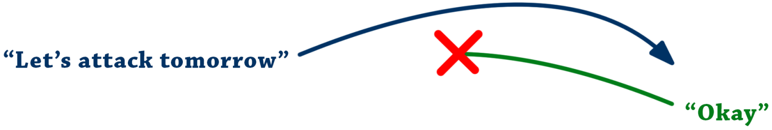
???

Two Generals' Problem

"Let's attack tomorrow"

"Okay"

Two Generals' Problem



???

Two Generals' Problem

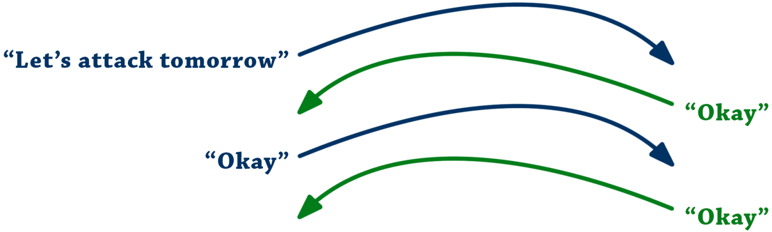
"Let's attack tomorrow" →

← **"Okay"**

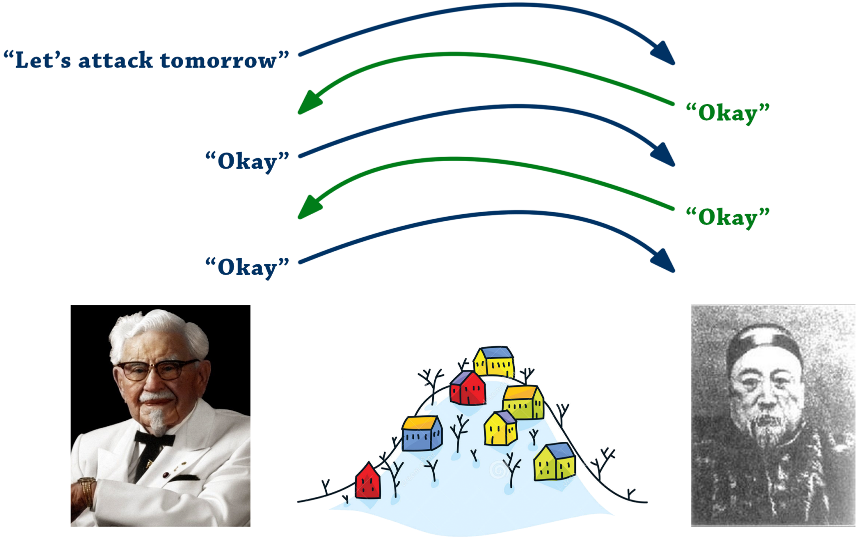
"Okay" →

Two Generals' Problem



Two Generals' Problem



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
- TCP is designed to deal with some degree of uncertainty
- Acknowledgements are necessary for reliable transmission

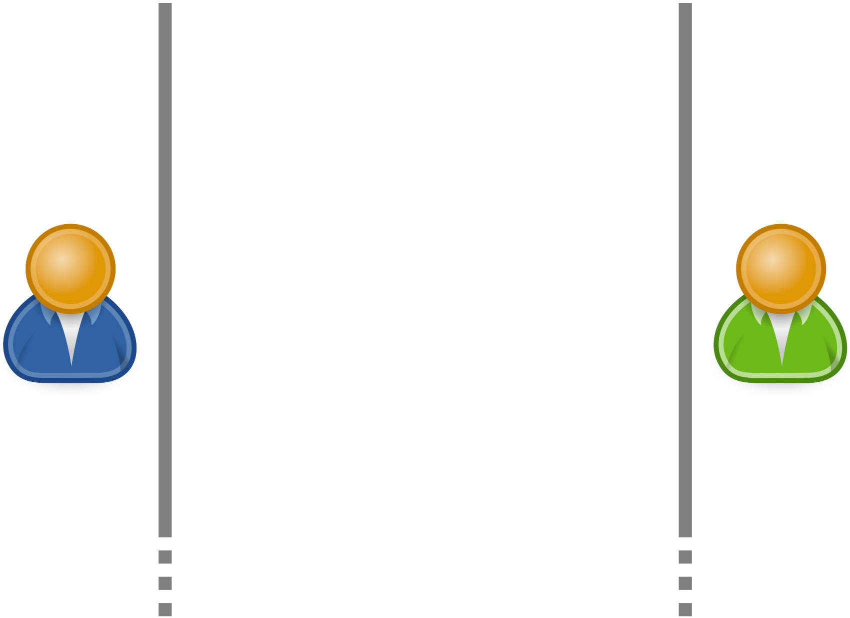
Reliable transmission primer

- 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

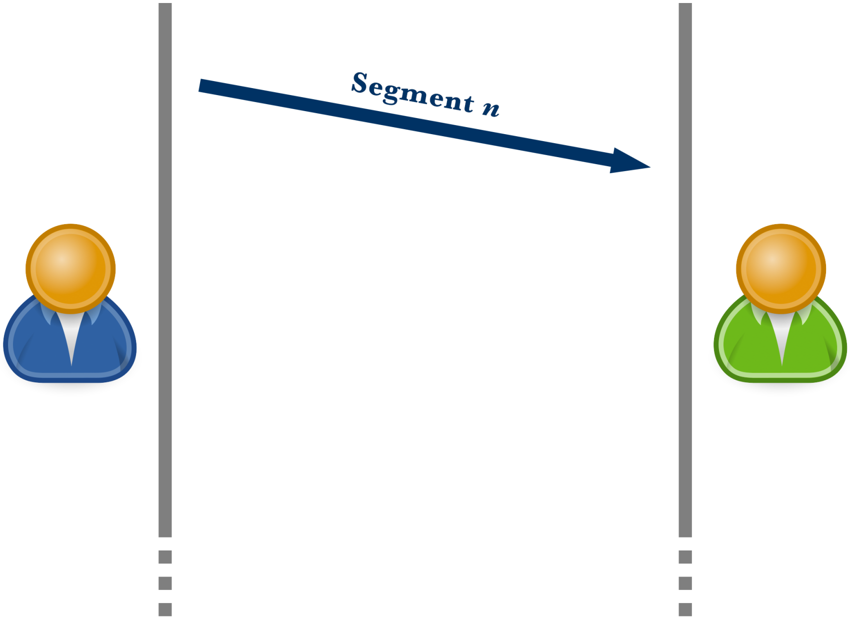
Reliable transmission primer

- 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*)

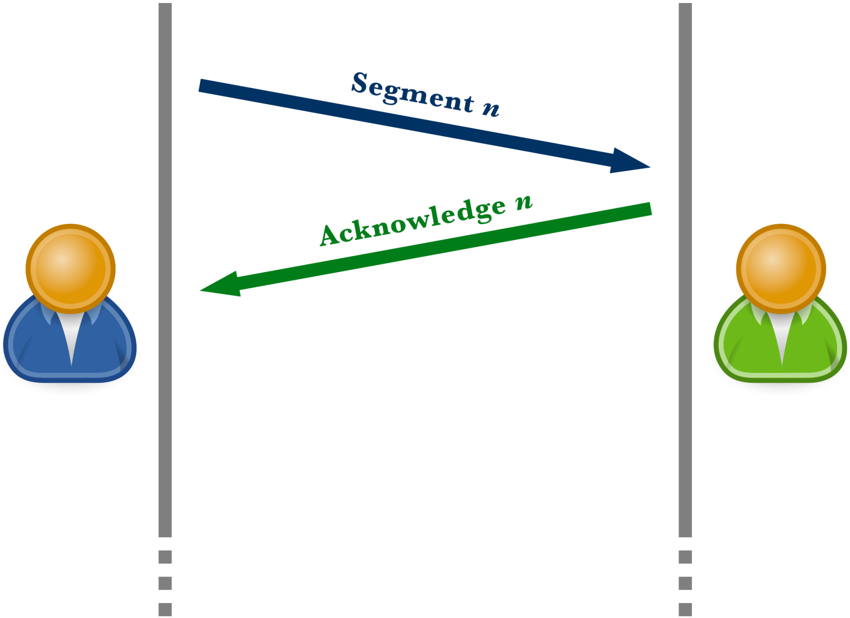
Reliable transmission: Stop-and-wait



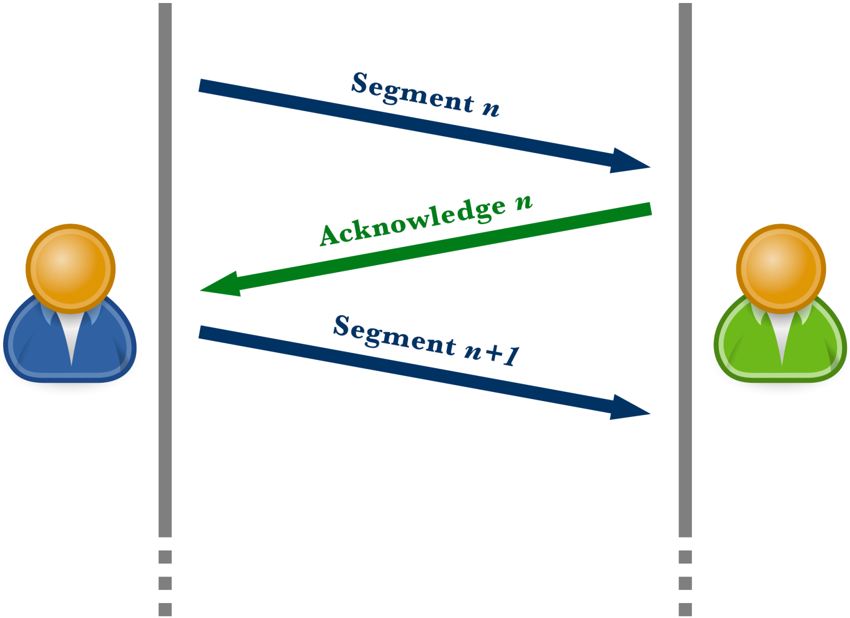
Reliable transmission: Stop-and-wait



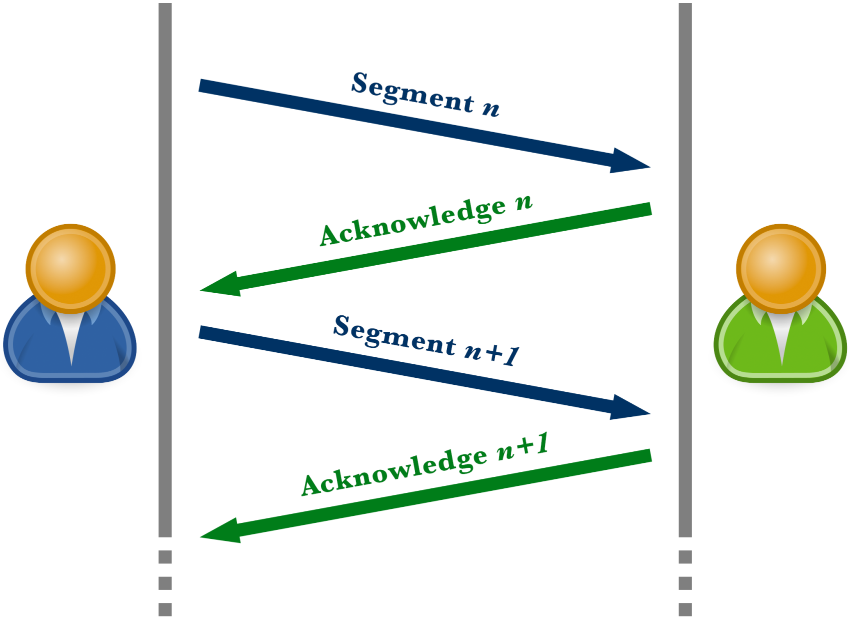
Reliable transmission: Stop-and-wait



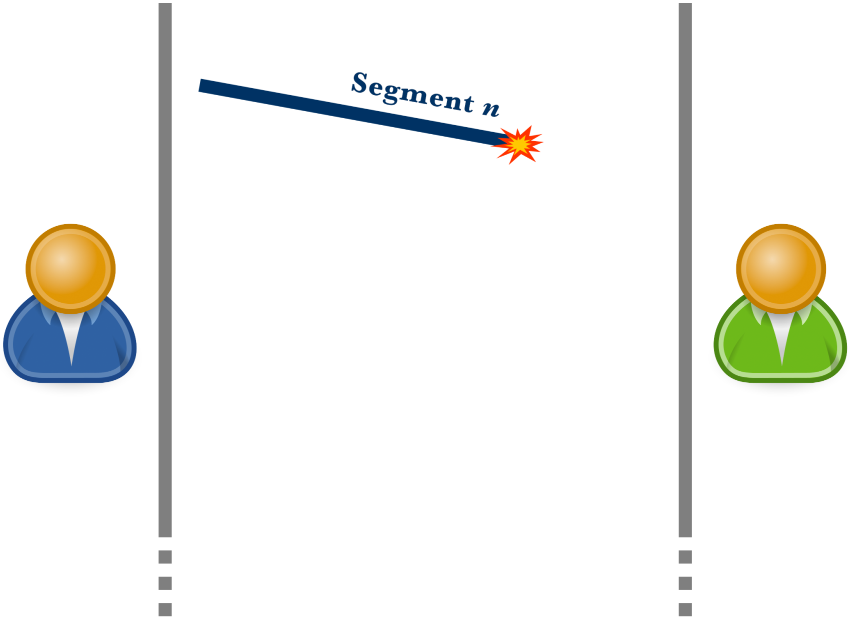
Reliable transmission: Stop-and-wait



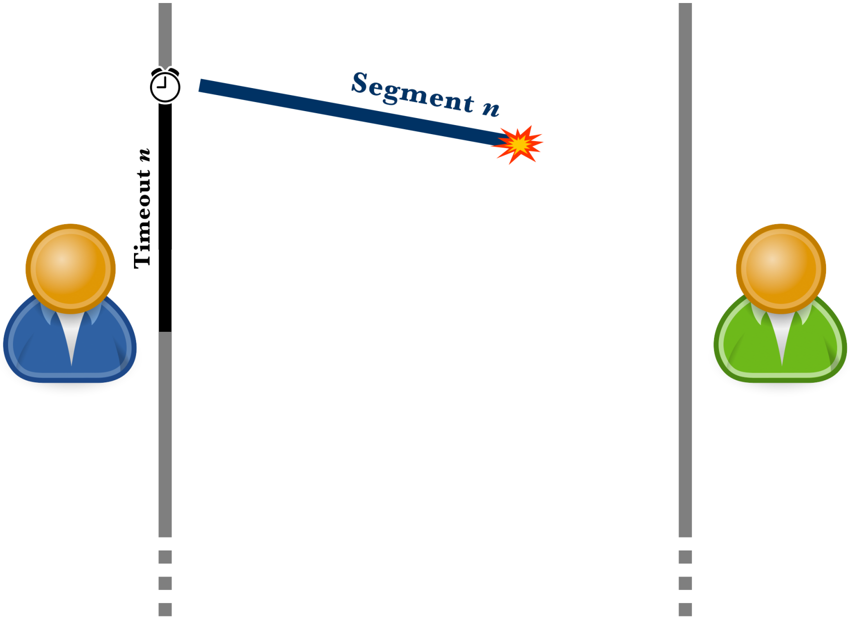
Reliable transmission: Stop-and-wait



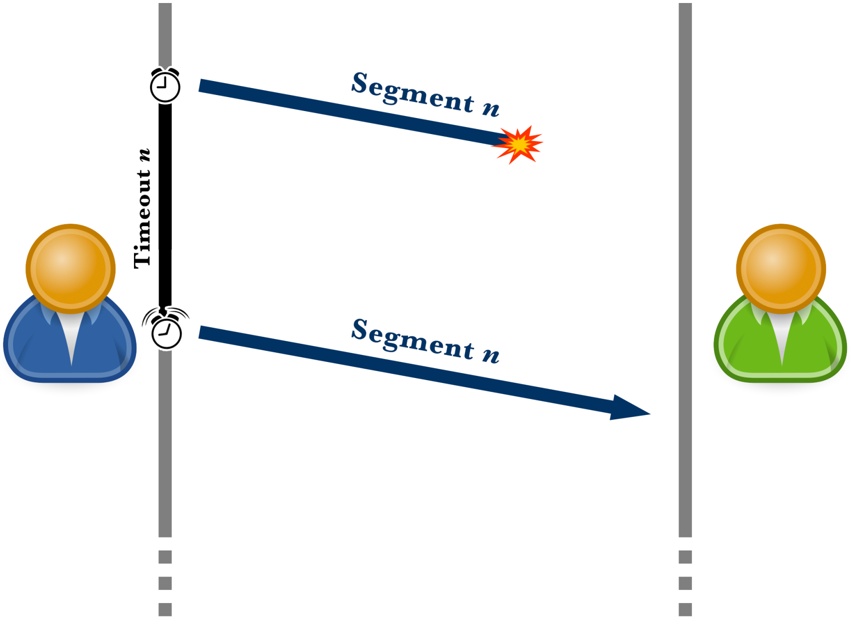
Reliable transmission: Stop-and-wait



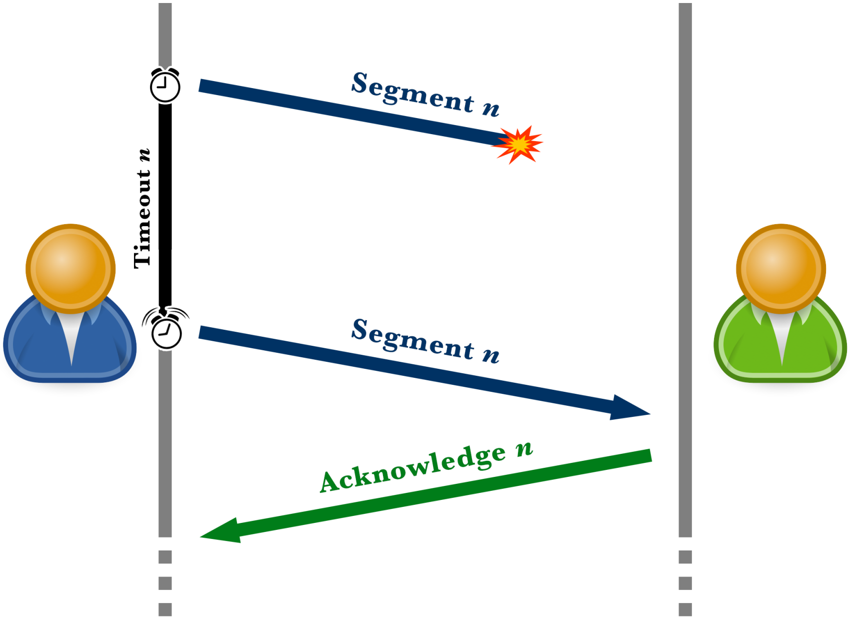
Reliable transmission: Stop-and-wait



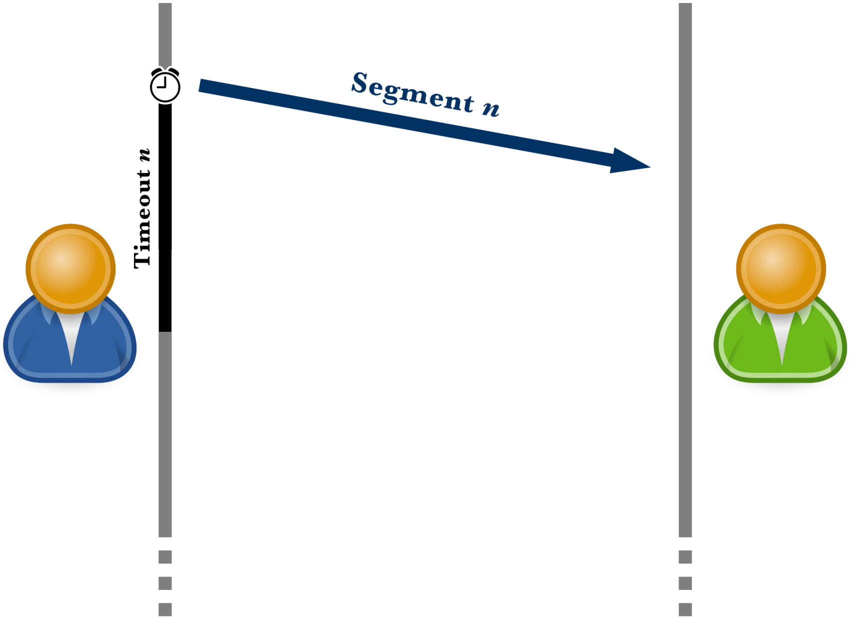
Reliable transmission: Stop-and-wait



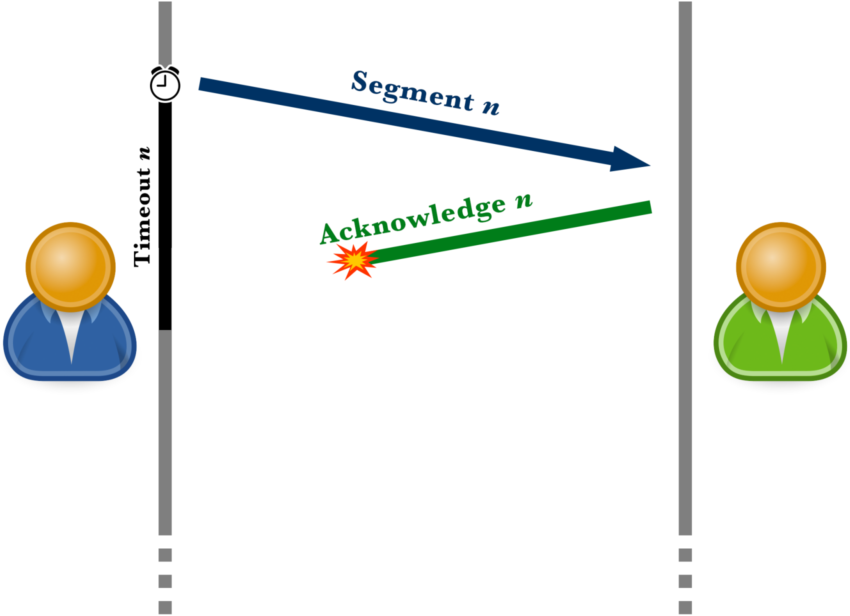
Reliable transmission: Stop-and-wait



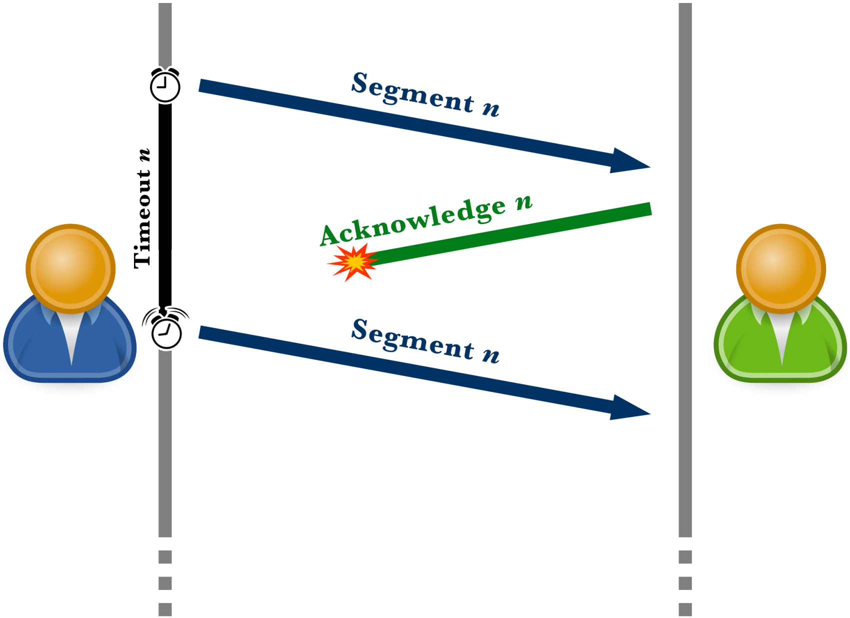
Reliable transmission: Stop-and-wait



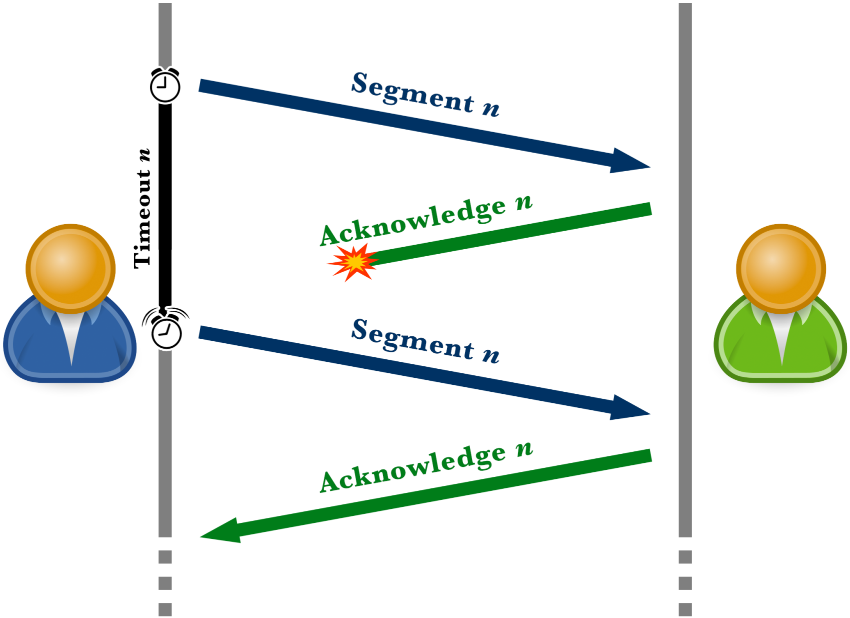
Reliable transmission: Stop-and-wait



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
- Inefficient!
    - Each segment bounded by `MSS` (≥ 536 bytes)
    - Wait at least 1 RTT per segment; typical RTT between 10 and 200 miliseconds
    - Suppose it's 100ms: we can only send $536\text{B}/100\text{ms} \approx 5\text{KB/s}$!

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
- Inefficient!
  - Each segment bounded by `MSS` (≥ 536 bytes)
  - Wait at least 1 RTT per segment; typical RTT between 10 and 200 miliseconds
  - Suppose it's 100ms: we can only send $536\mathrm{B}/100\mathrm{ms} \approx 5\mathrm{KB/s}$!
- Clearly we can do better
  - But before that...

Connection management

How do we start a connection?

- "Three-way handshake" between client and server

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`)

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, we can talk. 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, we can talk. Here's my initial sequence number. (`accept`)
3. `[Client]`: OK.

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, we can talk. Here's my initial sequence number. (`accept`)
3. `[Client]`: OK.

Questions:

- What's the initial sequence number?

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, we can talk. Here's my initial sequence number. (`accept`)
3. `[Client]`: OK.

Questions:

- What's the initial sequence number? 0?

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, we can talk. Here's my initial sequence number. (`accept`)
3. `[Client]`: OK.

Questions:

- What's the initial sequence number? 0? No, it's random. Why?

Connection management

How do we start a connection?

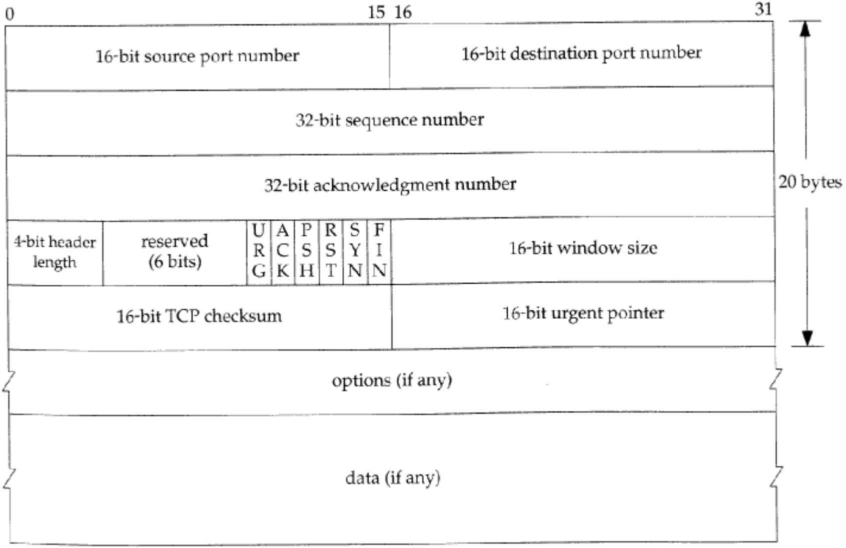
- "Three-way handshake" between client and server

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, we can talk. Here's my initial sequence number. (`accept`)
3. `[Client]`: OK.

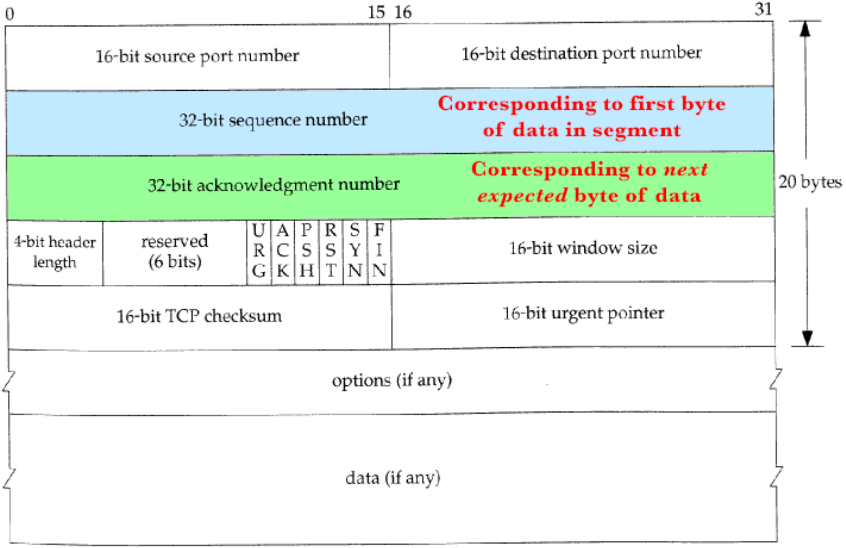
Questions:

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

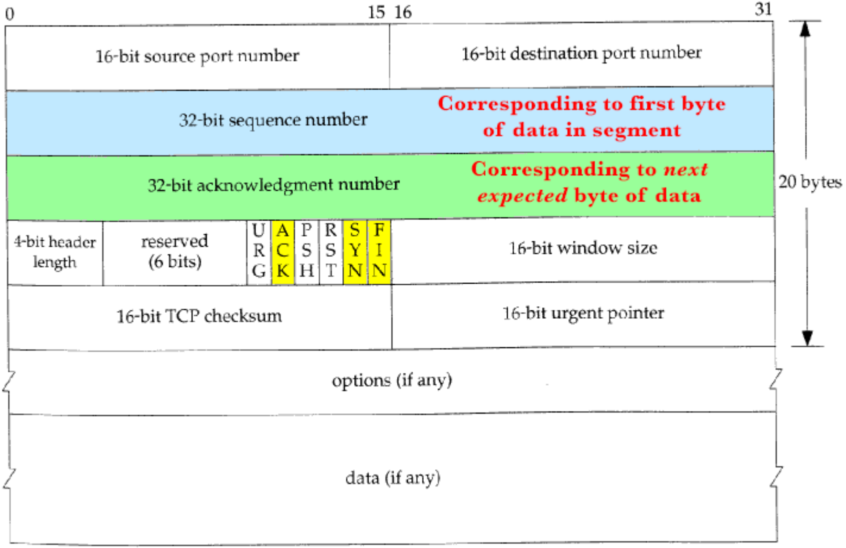
TCP segment anatomy



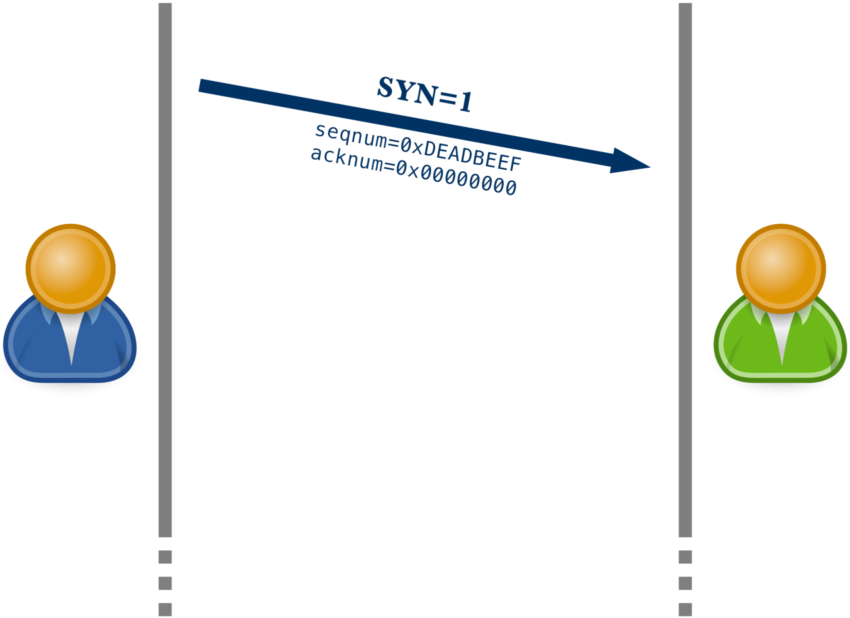
TCP segment anatomy



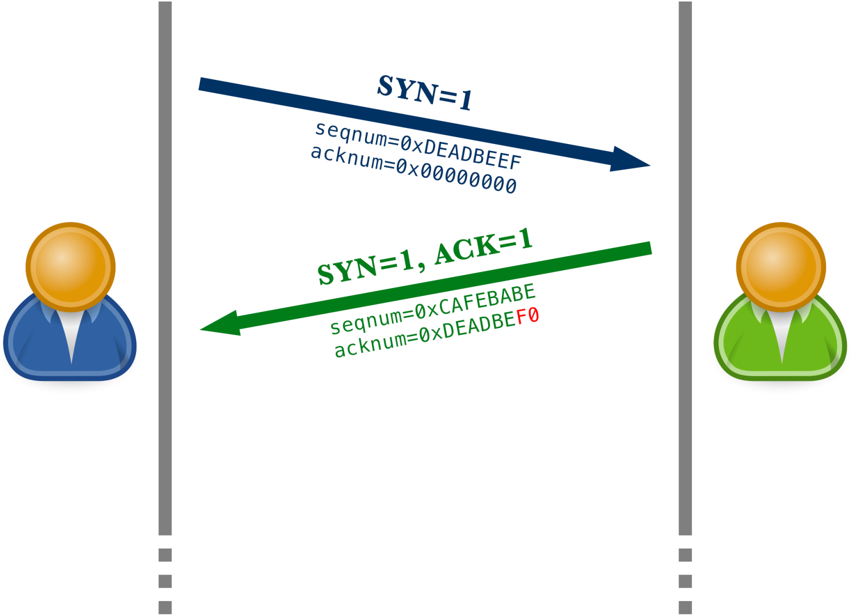
TCP segment anatomy



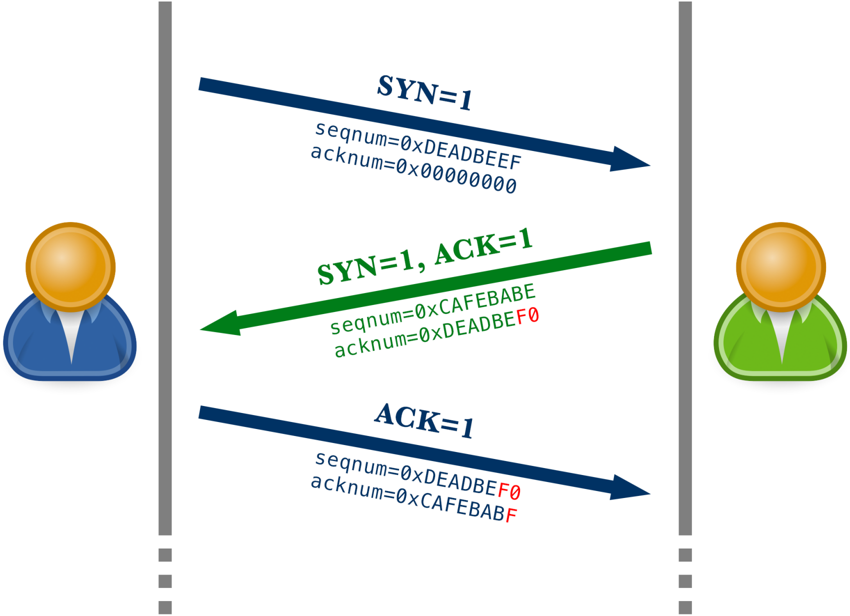
TCP connection handshake



TCP connection handshake



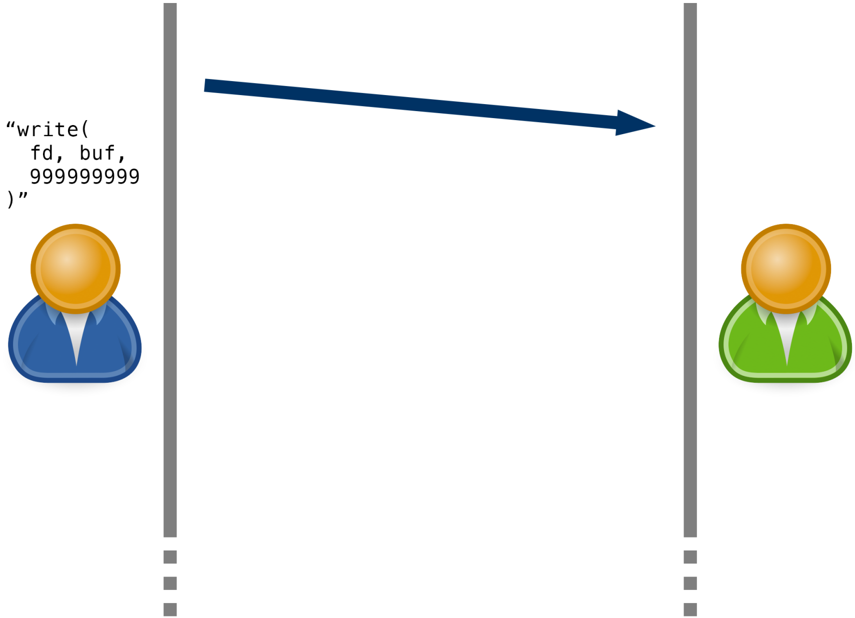
TCP connection handshake



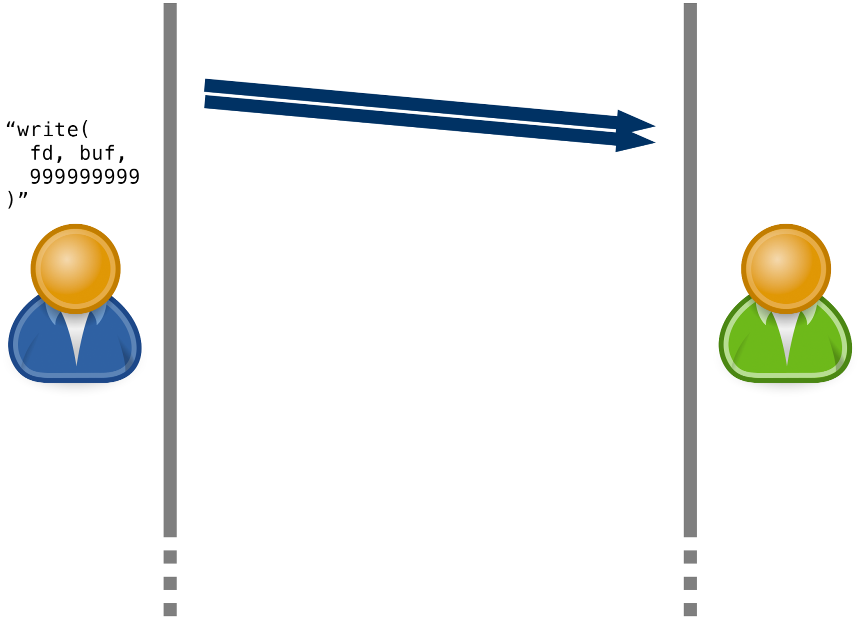
Now back to reliable transmission!

- How do we deal with the speed problem from earlier?

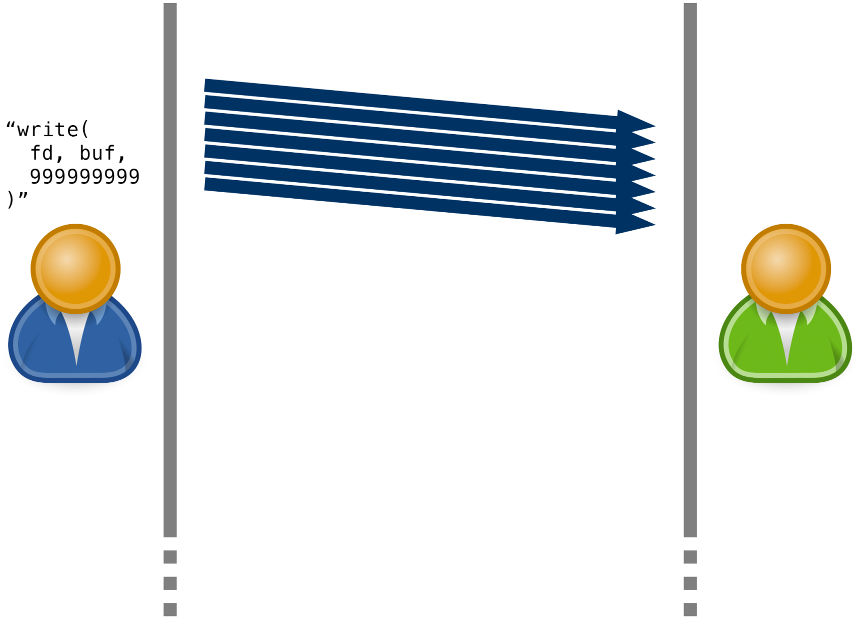
Reliable transmission: Don't try this at home



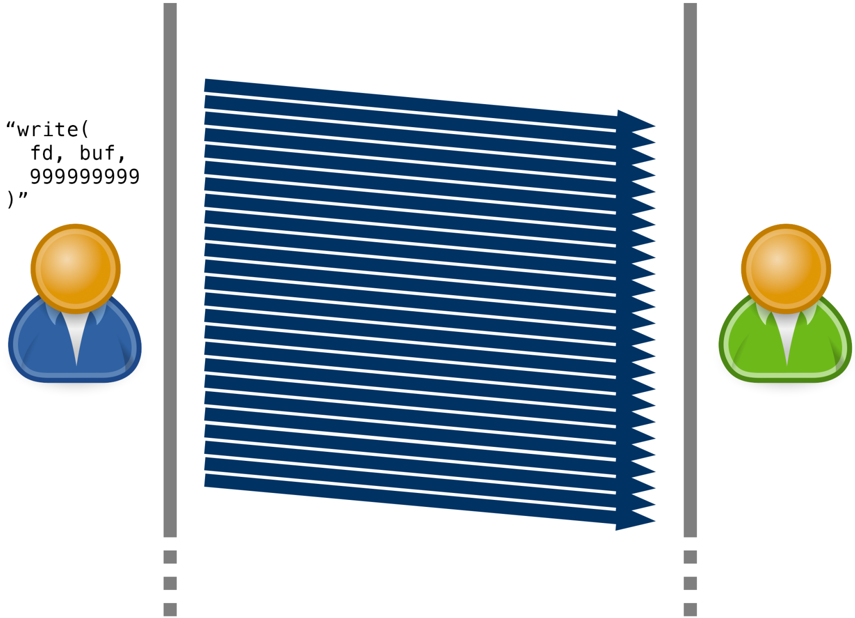
Reliable transmission: Don't try this at home



Reliable transmission: Don't try this at home



Reliable transmission: Don't try this at home



Reliable transmission: Challenges

What's wrong with this?

Reliable transmission: Challenges

What's wrong with this?

- Separate timeout for each segment or one timeout for all?
  - Go-Back-N vs. Selective Repeat

Reliable transmission: Challenges

What's wrong with this?

- Separate timeout for each segment or one timeout for all?
  - Go-Back-N vs. Selective Repeat
- Flood the recipient
  - All the bytes must be stored in a buffer in the OS somewhere
  - What if the OS doesn't want to store 100 MB in memory before a program decides to call `read`?

Reliable transmission: Challenges

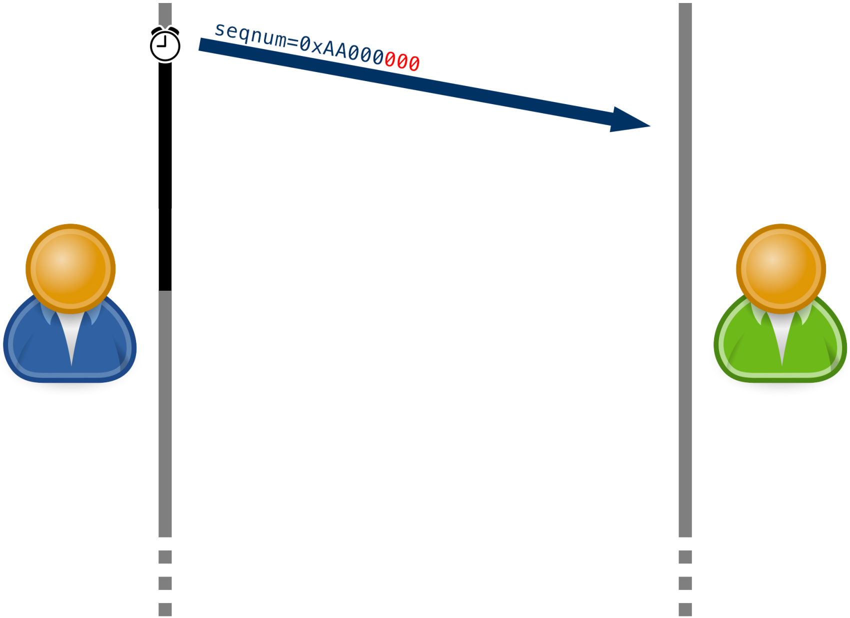
What's wrong with this?

- Separate timeout for each segment or one timeout for all?
  - Go-Back-N vs. Selective Repeat
- Flood the recipient
  - All the bytes must be stored in a buffer in the OS somewhere
  - What if the OS doesn't want to store 100 MB in memory before a program decides to call `read`?
- Flood the network

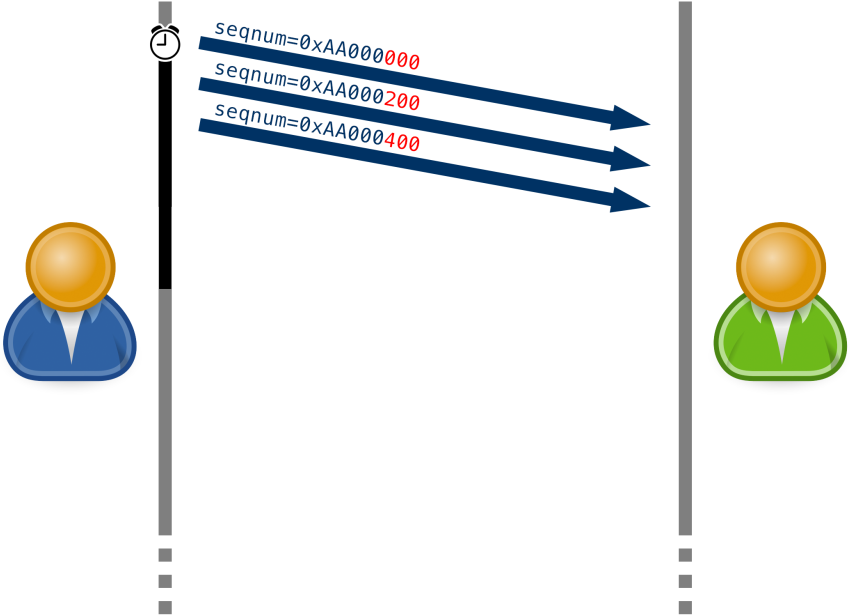
Reliable transmission in TCP

- Timer starts for oldest un-acknowledged segment
- On timeout, resend *only* that segment
    - Hopefully subsequent segments were buffered in the recipient
- Also resend on three duplicate acknowledgements

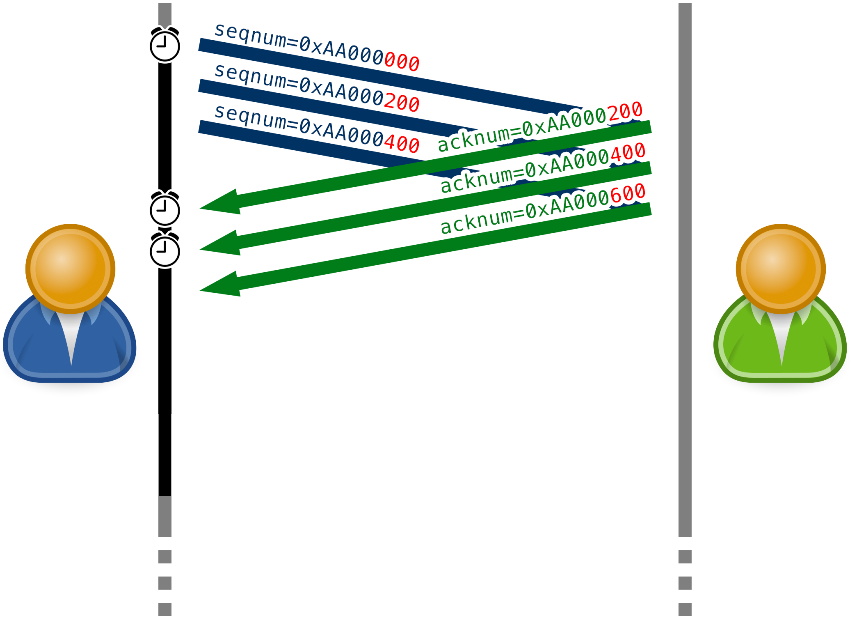
Reliable transmission



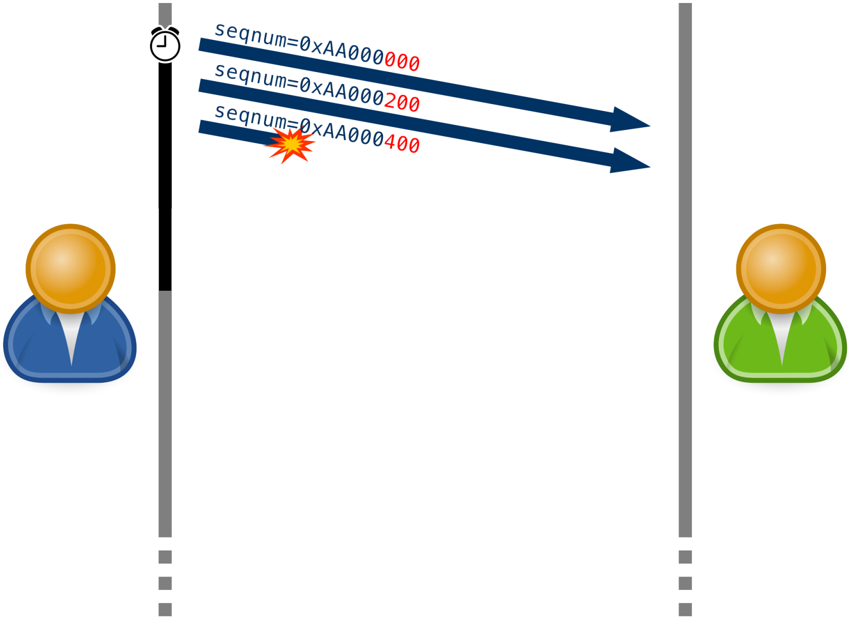
Reliable transmission



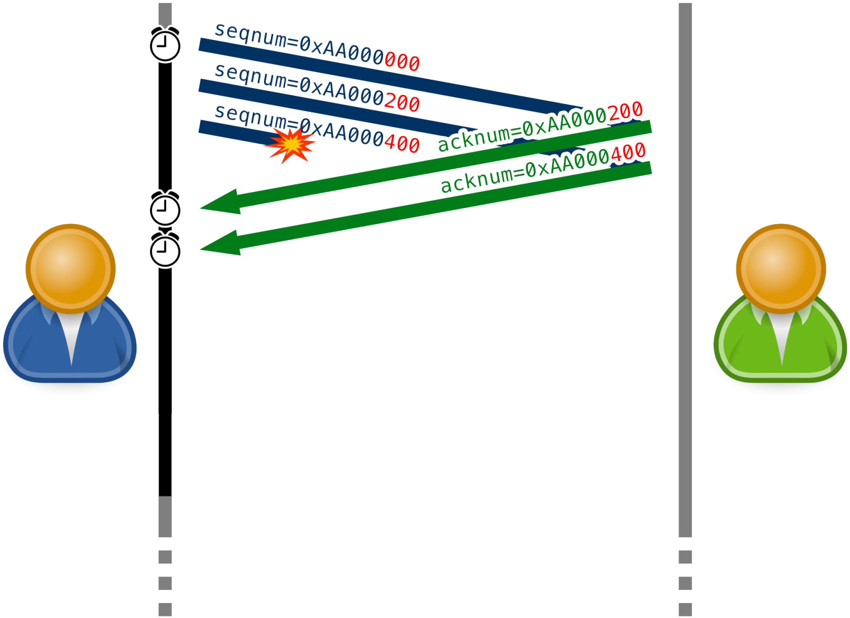
Reliable transmission



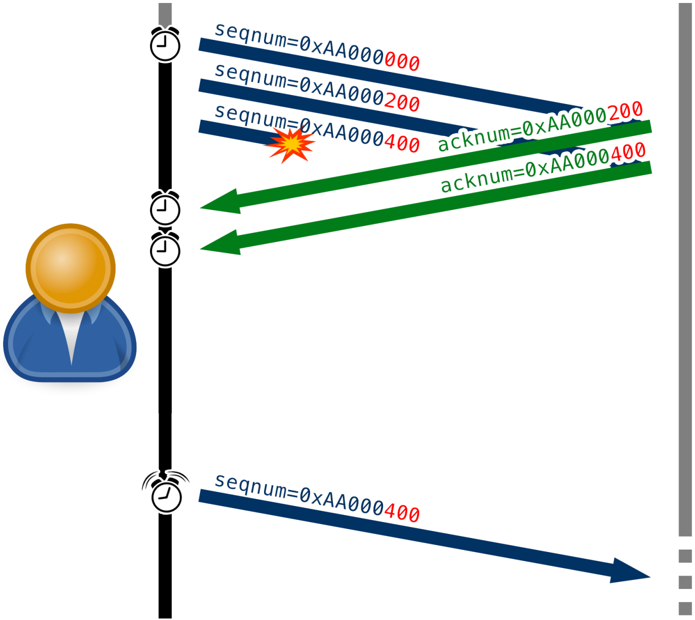
Reliable transmission



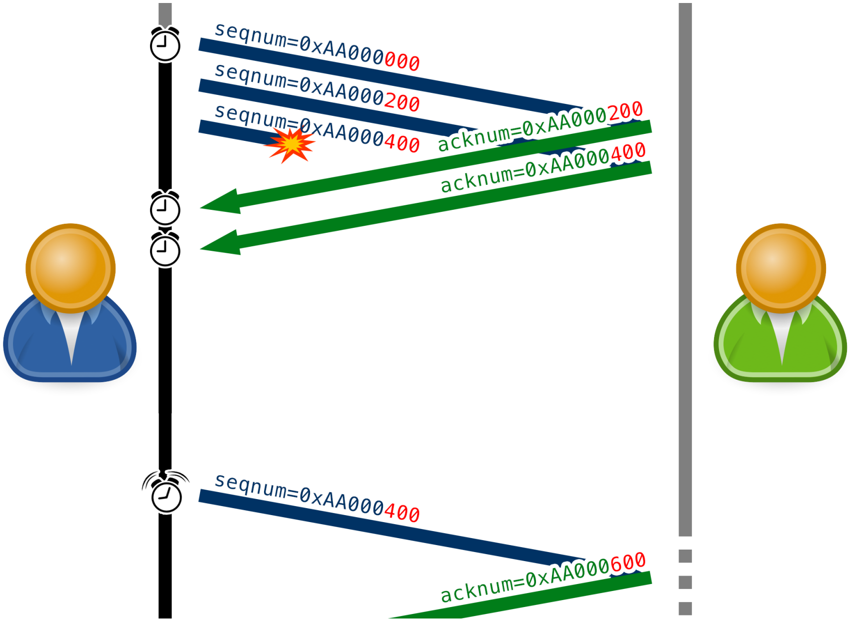
Reliable transmission



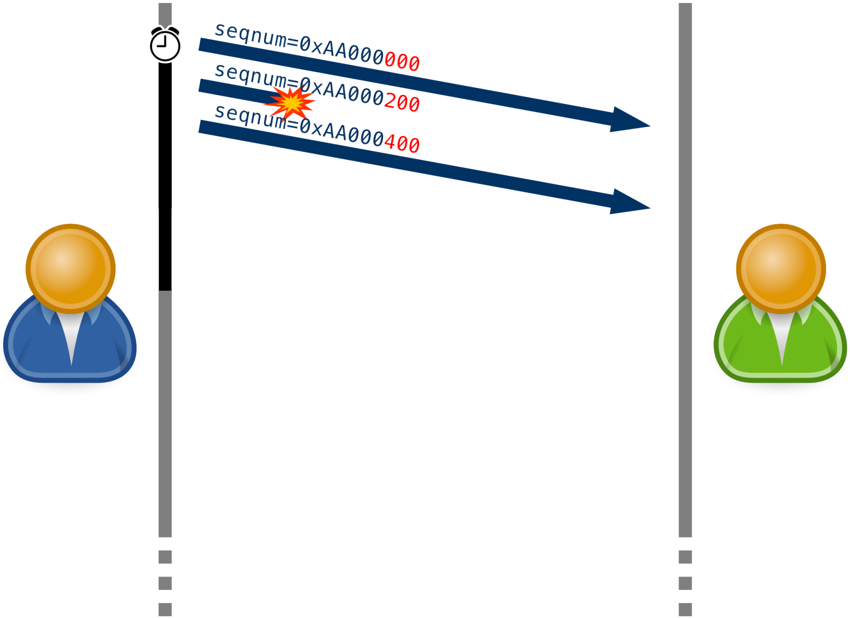
Reliable transmission



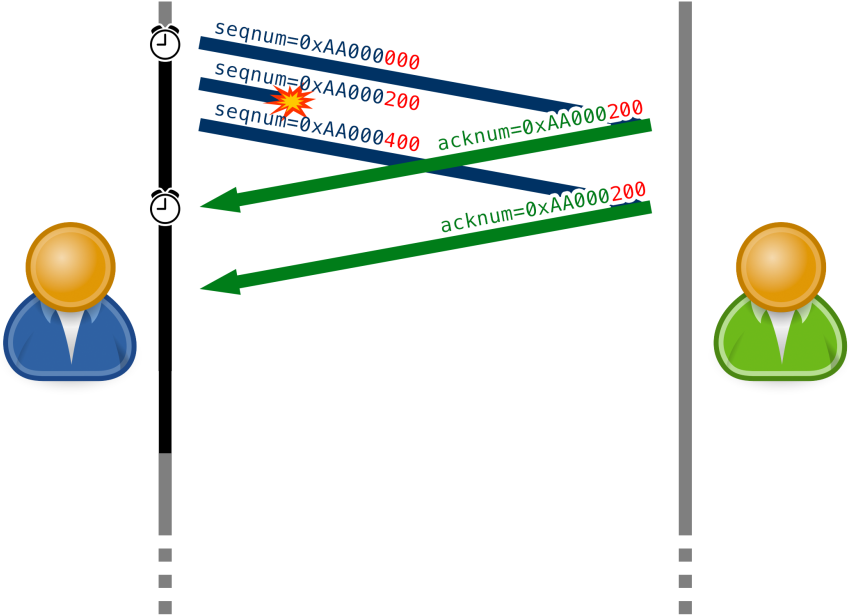
Reliable transmission



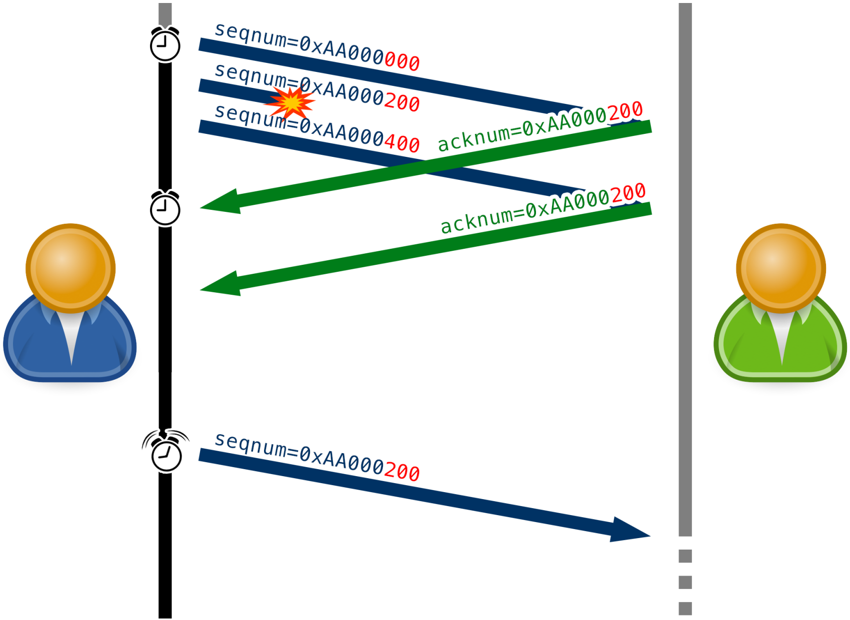
Reliable transmission



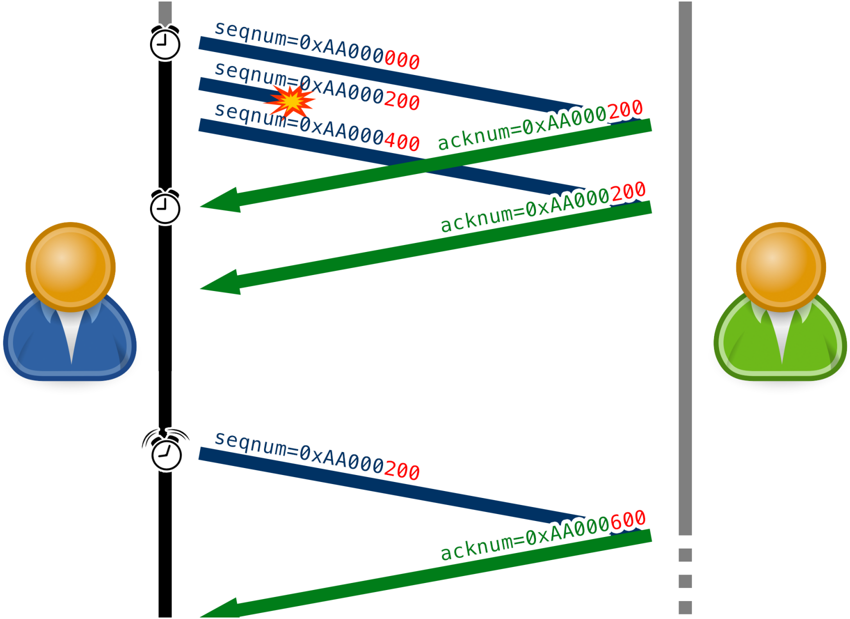
Reliable transmission



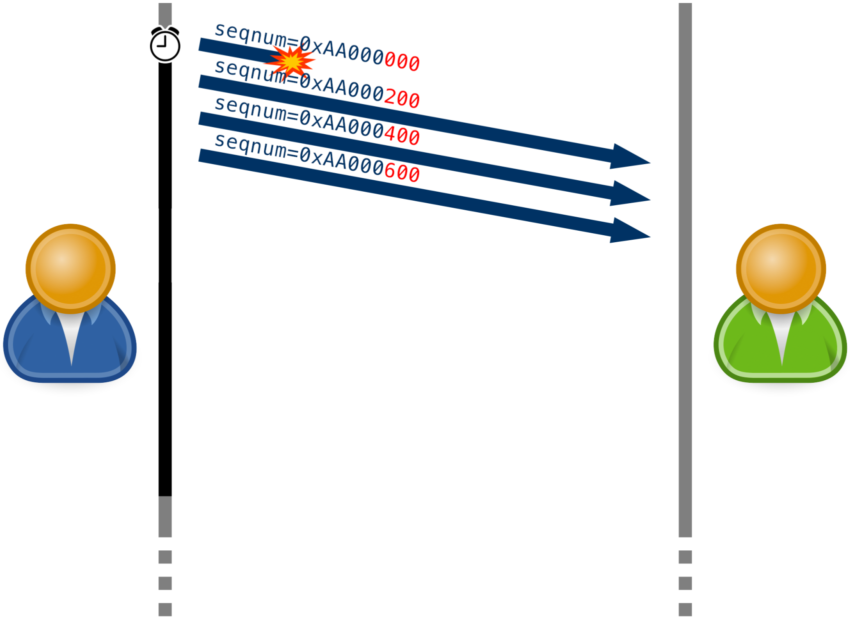
Reliable transmission



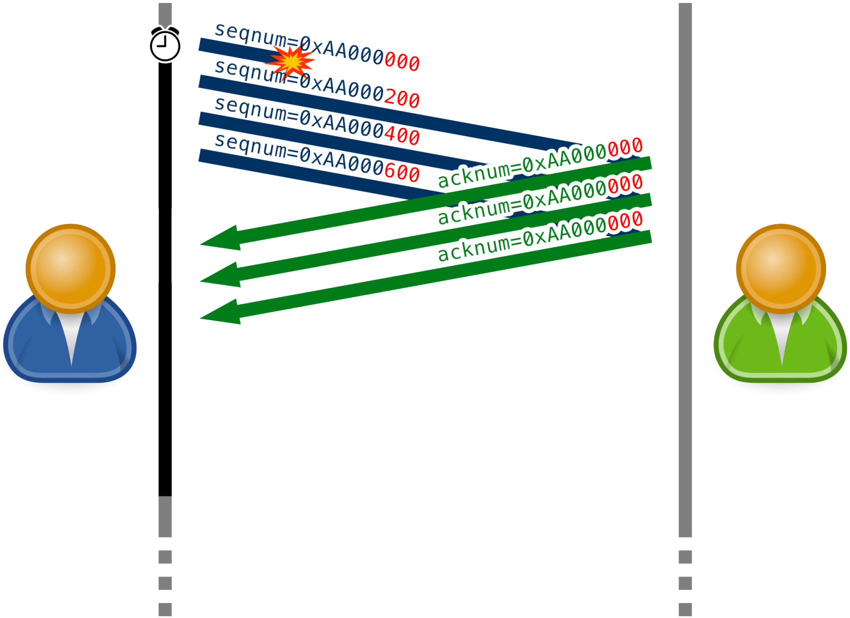
Reliable transmission



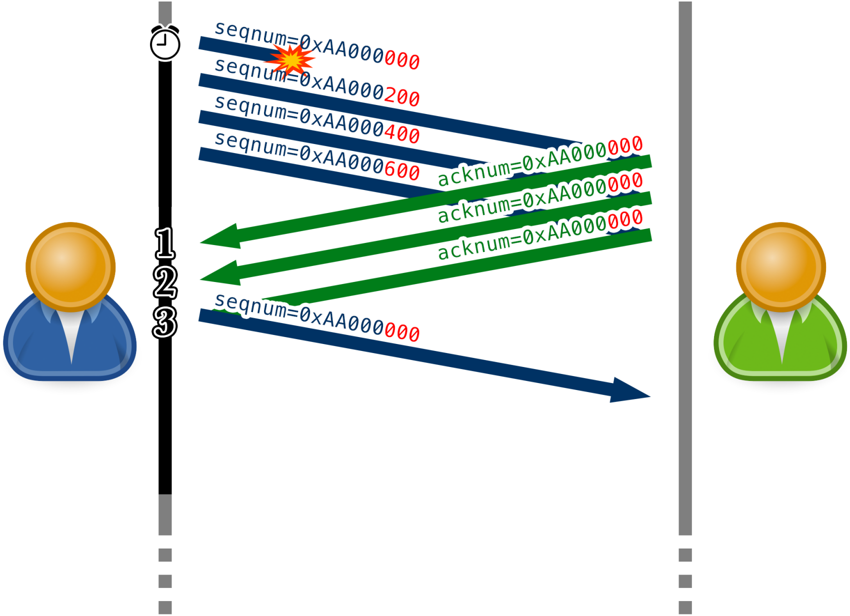
Reliable transmission



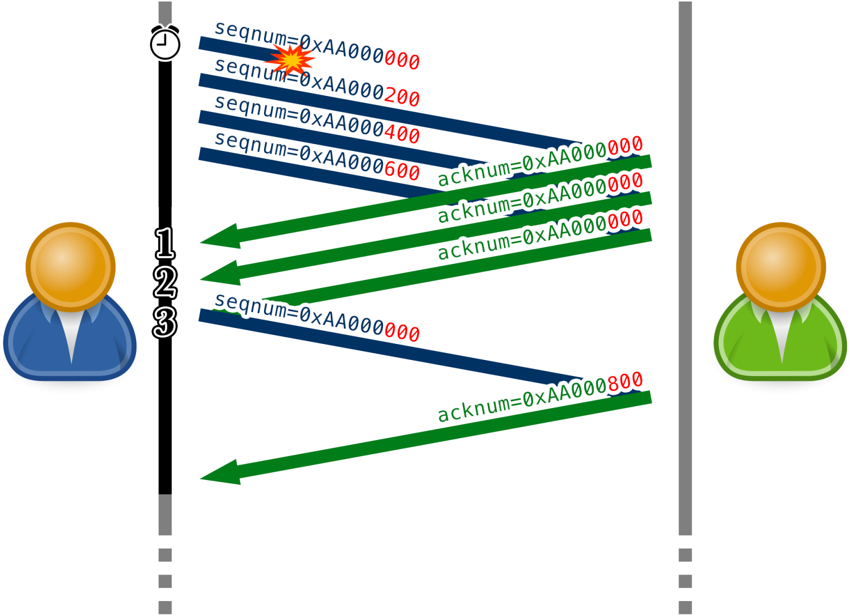
Reliable transmission



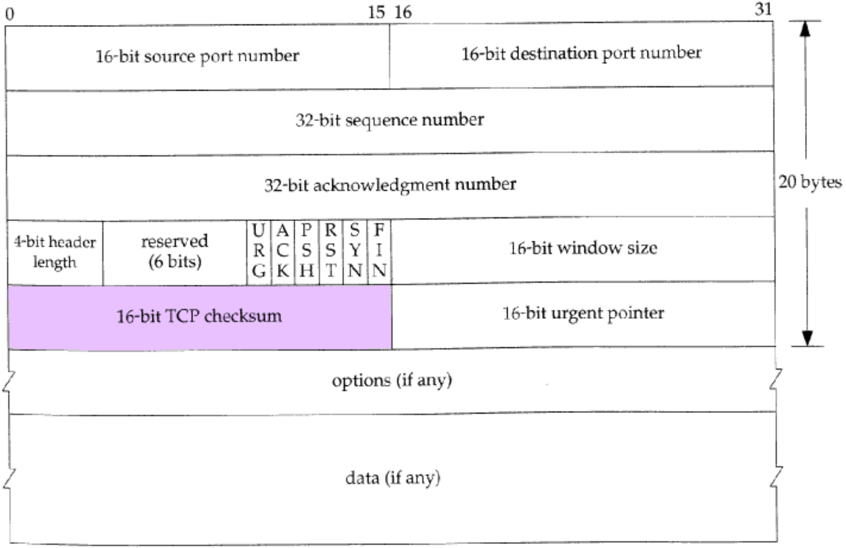
Reliable transmission



Reliable transmission



Reliable transmission: error checking



Reliable transmission: error checking

- 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?

Flow control

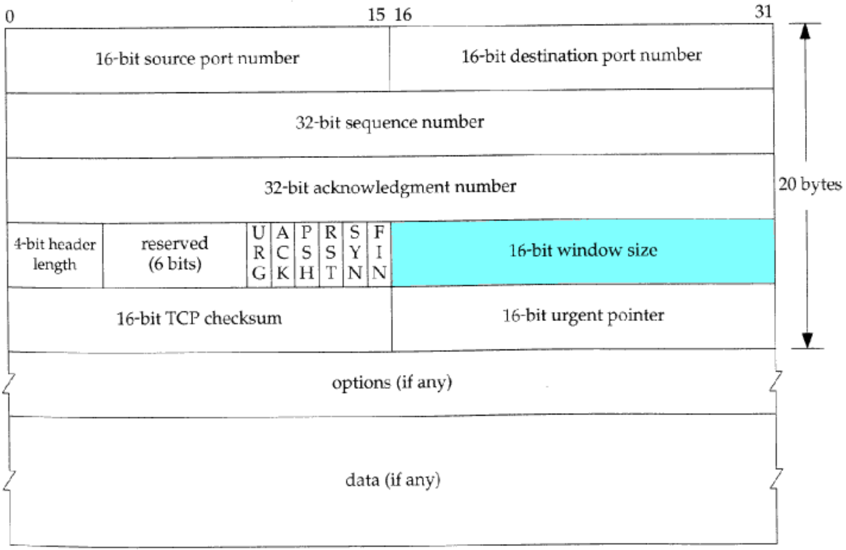
How do we avoid flooding the recipient if they aren't ready to receive yet?

Flow control

How do we avoid flooding the recipient if they aren't ready to receive yet?

- *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

Flow control

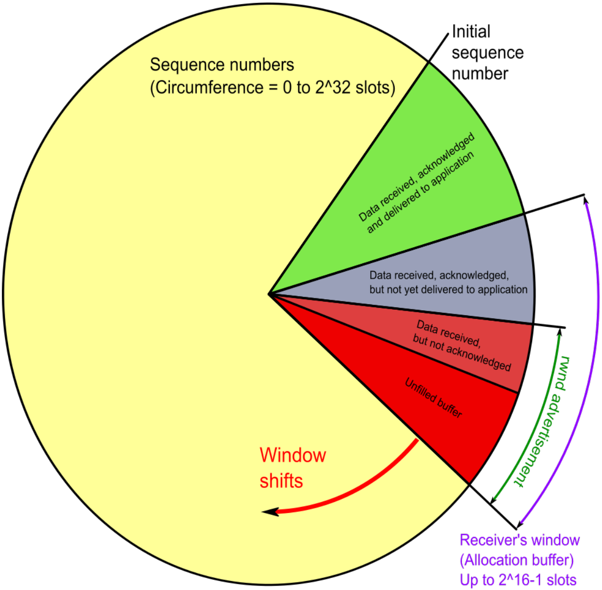


Flow control

How do we avoid flooding the recipient if they aren't ready to receive yet?

- *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 $\approx 640\mathrm{KB/s}$ assuming RTT = 100ms

Sliding window



Congestion control

How do we avoid flooding the network?

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

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

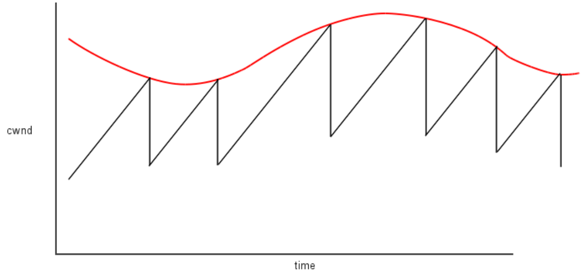
How do we avoid flooding the network?

- 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:

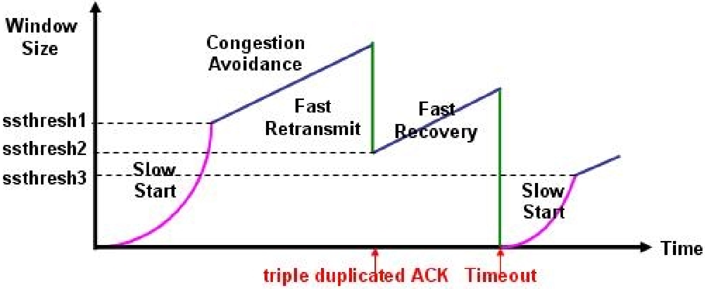
- Start congestion window at a small value (1 MSS)
- Keep increasing the window periodically until a loss occurs—this means we are sending too much, so decrease it and try again
  - *Additive increase*: Increase window at a linear rate
  - *Multiplicative decrease*: Decrease window at an exponential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



TCP Sawtooth, red curve represents the network capacity

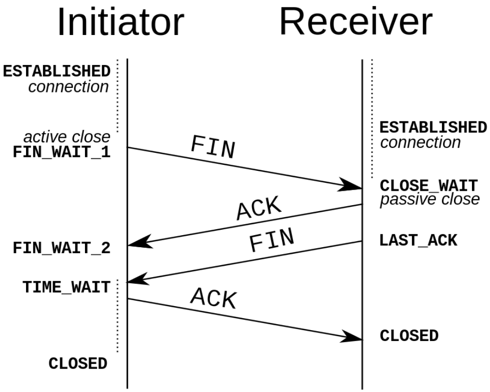
Congestion control: Stages

- 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



Connection termination: `TIME_WAIT`

- Lasts for 2 MSL (maximum segment lifetime), ≈ 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), ≈ 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)

Operating system's role

- *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`...

Operating system's role

- *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

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)

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`

Learn more

- Take **CS/ECE 438**: Communication Networks