

# Fundamentals of Backend Communications and Protocols

Understand what's behind the backend

# Introduction

# Introduction

- Welcome
- Who this course is for?
- Course Outline

# Backend Communication Design Patterns

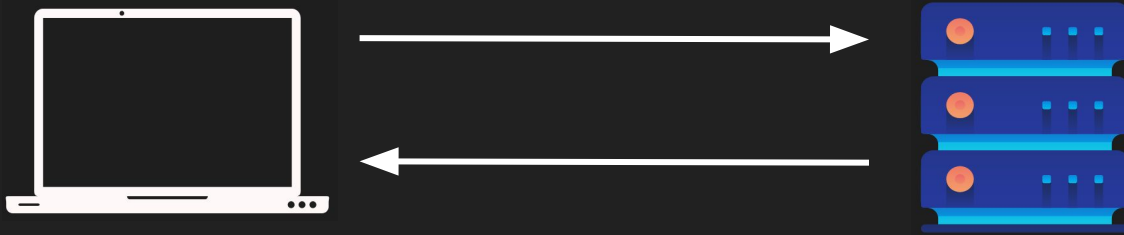
There are handful of ways backends communicate

# Request - Response

Classic, Simple and Everywhere

# Request Response Model

- Client sends a Request
- Server parses the Request
- Server processes the Request
- Server sends a Response
- Client parses the Response and consume



## Where it is used?

- Web, HTTP, DNS, SSH
- RPC (remote procedure call)
- SQL and Database Protocols
- APIs (REST/SOAP/GraphQL)
- Implemented in variations

# Anatomy of a Request / Response

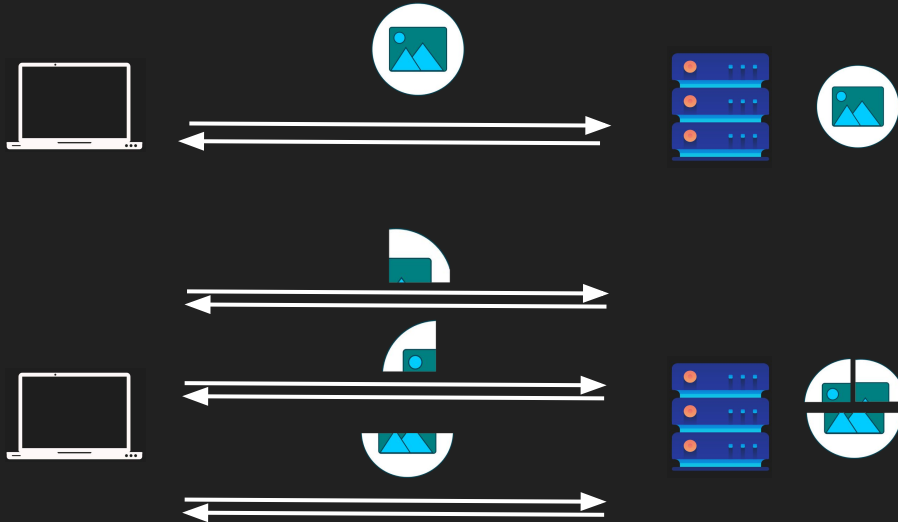
- A request structure is defined by both client and server
- Request has a boundary
- Defined by a protocol and message format
- Same for the response
- E.g. HTTP Request

```
GET / HTTP/1.1  
Headers  
<CRLF>  
BODY
```



# Building an upload image service with request response

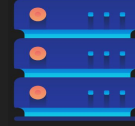
- Send large request with the image (simple)
- Chunk image and send a request per chunk (resumable)



# Doesn't work everywhere

- Notification service
- Chatting application
- Very Long requests
- What if client disconnects?
- We will talk about alternatives

# Request/Response



t0

t2

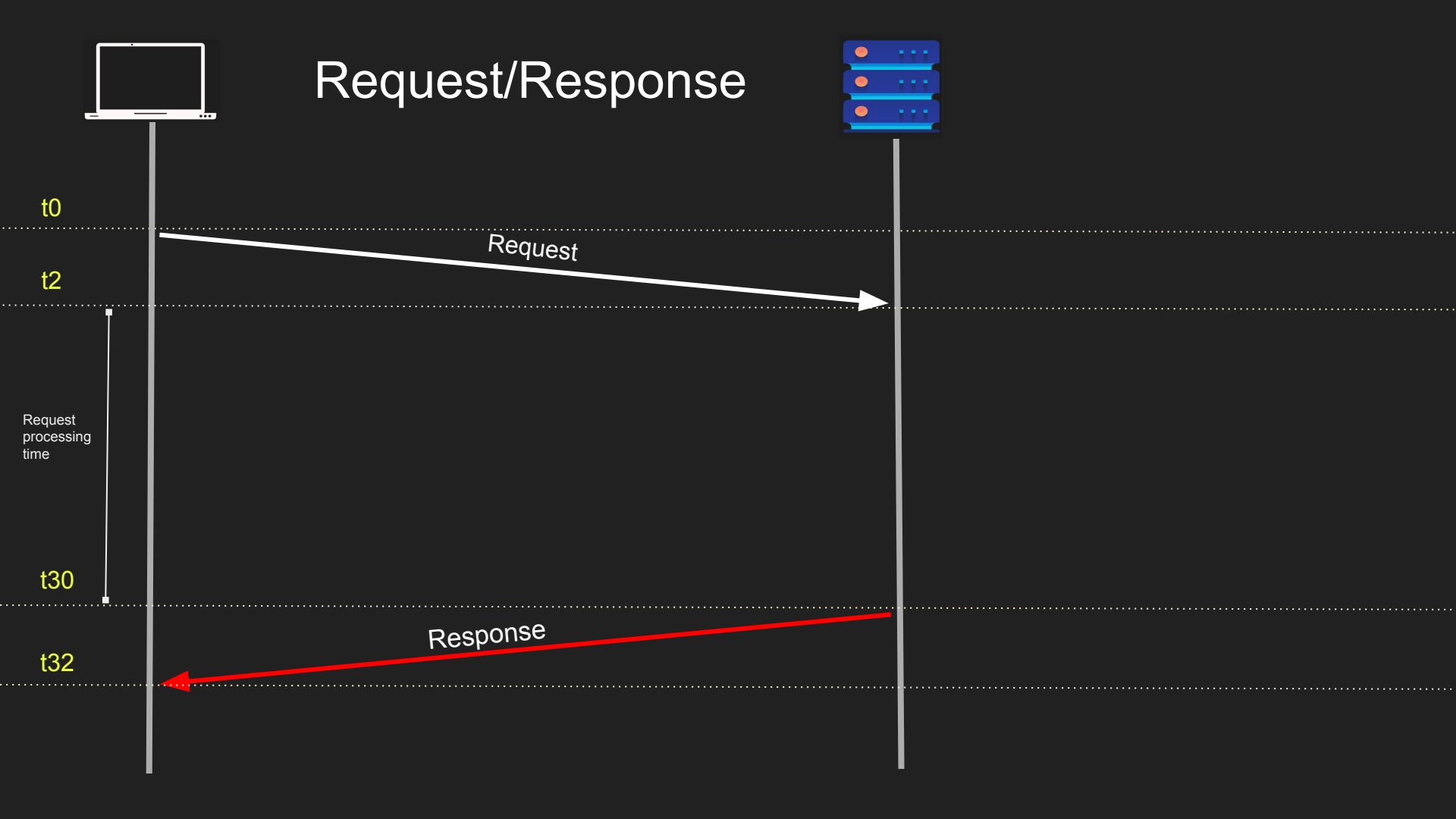
Request  
processing  
time

t30

t32

Request

Response



# Demo Curl

—trace

# Synchronous vs Asynchronous


Can I do work while waiting?

# Synchronous I/O

- Caller sends a request and blocks
- Caller cannot execute any code meanwhile
- Receiver responds, Caller unblocks
- Caller and Receiver are in “sync”

# Example of an OS synchronous I/O

- Program asks OS to read from disk
- Program main thread is taken off of the CPU
- Read completes, program can resume execution



```
1 //Program starts
2 //Program uses CPU to execute stuff
3 doWork();
4 //Program reads from disk
5 //Program can't do anything until file loads
6 readFile("largefile.dat");
7 //program resumes
8 doWork2();
```


# Asynchronous I/O

- Caller sends a request
- Caller can work until it gets a response
- Caller either:
  - Checks if the response is ready (epoll)
  - Receiver calls back when it's done (io\_uring)
  - Spins up a new thread that blocks
- Caller and receiver are not necessary in sync



# Example of an OS asynchronous call (NodeJS)

- Program spins up a secondary thread
- Secondary thread reads from disk, OS blocks it
- Main program still running and executing code
- Thread finish reading and calls back main thread



```
1 //Program starts
2 //Program uses CPU to execute stuff
3 doWork();
4 //Program requests read from disk
5 //Program asks to callback when done
6 //Program moves on to doWork2
7 readFile("largefile.dat", onReadFinished(theFile));
8 //file is probably not read yet
9 //Program happy doing stuff
10 doWork2();
11 //someone just called onReadFinished
12 //processing it.
13 ---->onReadFinished(theFile)
```

# Synchronous vs Asynchronous in Request Response

- Synchronicity is a client property
- Most modern client libraries are asynchronous
- E.g. Clients send an HTTP request and do work



# Synchronous vs Asynchronous in real life

- Just in case it is still confusing
- In Synchronous communication the caller waits for a response from receiver
  - E.g. Asking someone a question in a meeting
- Asynchronous communication the response can come whenever. Caller and receiver can do anything meanwhile
  - email

# Asynchronous workload is everywhere

- Asynchronous Programming (promises/futures)
- Asynchronous backend processing
- Asynchronous commits in postgres
- Asynchronous IO in Linux (epoll, io\_uring)
- Asynchronous replication
- Asynchronous OS fsync (fs cache)

# Demo NodeJS

Blocking call (loop) and fetch

# Push

I want it as soon as possible

# Request/response isn't always ideal

- Client wants real time notification from backend
  - A user just logged in
  - A message is just received
- Push model is good for certain cases

# What is Push?

- Client connects to a server
- Server sends data to the client
- Client doesn't have to request anything
- Protocol must be bidirectional
- Used by RabbitMQ

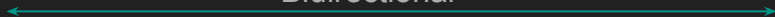




# Push



Bidirectional  
connection

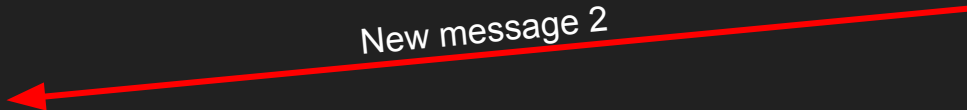


New message 1



\*Backend gets  
message\*

New message 2



# Push Pros and Cons

- Pros
  - Real time
- Cons
  - Clients must be online
  - Clients might not be able to handle
  - Requires a bidirectional protocol
  - Polling is preferred for light clients

# Demo WebSockets

Push notification (users login)

# Short Polling

Request is taking a while, I'll check with you later

# Where request/response isn't ideal

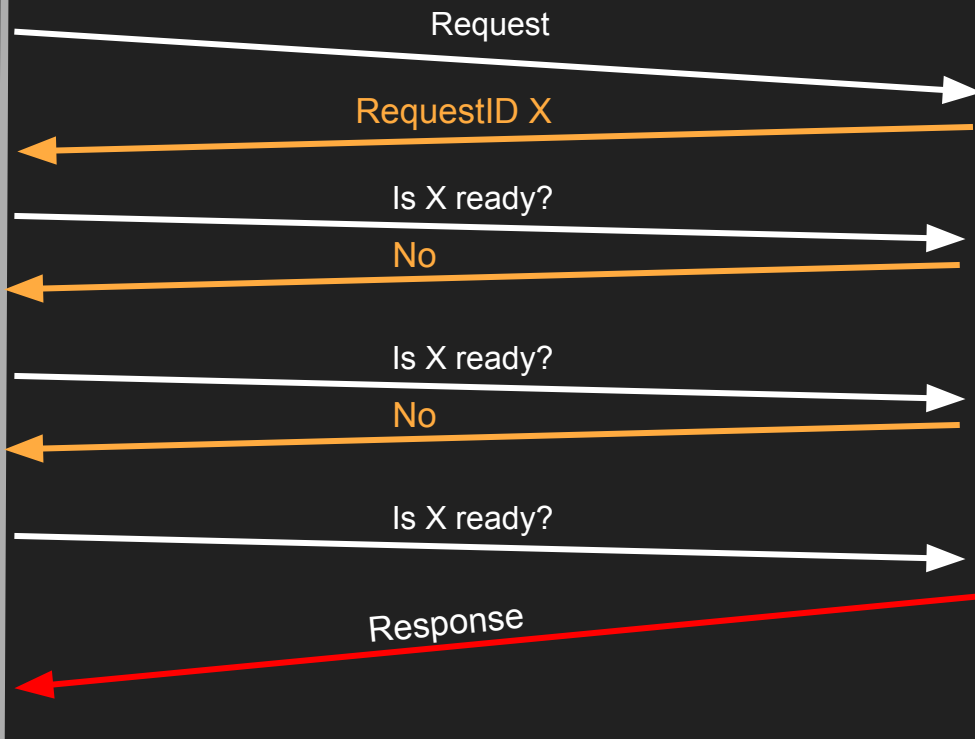
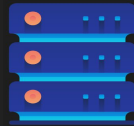
- A request takes long time to process
  - Upload a youtube video
- The backend want to sends notification
  - A user just logged in
- Polling is a good communication style

# What is Short Polling?

- Client sends a request
- Server responds immediately with a handle
- Server continues to process the request
- Client uses that handle to check for status
- Multiple “short” request response as polls



# Short Polling



# Short Polling Pros and Cons

- Pros
  - Simple
  - Good for long running requests
  - Client can disconnect
- Cons
  - Too chatty
  - Network bandwidth
  - Wasted Backend resources



# Demo Job Status

Check if a job is done with progress

# Long Polling

Request is taking long, I'll check with you later  
But talk to me only when it's ready

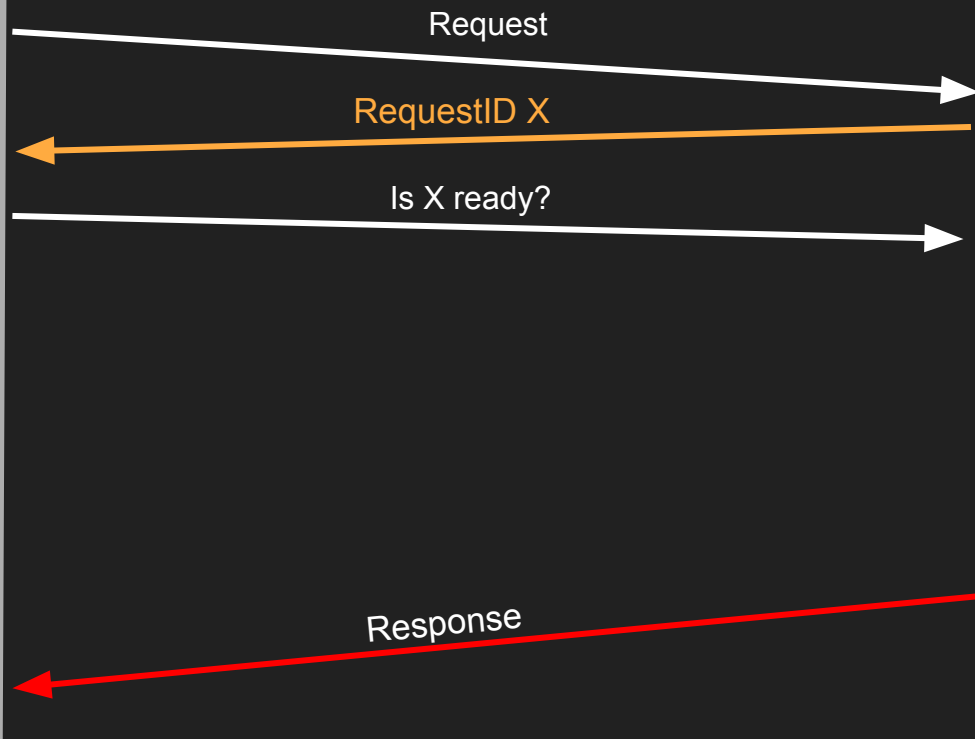
# Where request/response & polling isn't ideal

- A request takes long time to process
  - Upload a youtube video
- The backend want to sends notification
  - A user just logged in
- Short Polling is a good but chatty
- Meet Long polling (Kafka uses it)

# What is Long Polling?

- Client sends a request
- Server responds immediately with a handle
- Server continues to process the request
- Client uses that handle to check for status
- Server DOES not reply until it has the response
- So we got a handle, we can disconnect and we are less chatty
- Some variation has timeouts too

# Long Polling



# Long Polling Pros and Cons

- Pros
  - Less chatty and backend friendly
  - Client can still disconnect
- Cons
  - Not real time

# Demo Job Status

Check if a job is done

# Server Sent Events

One Request, a very very long response



# Limitations of Request/Response

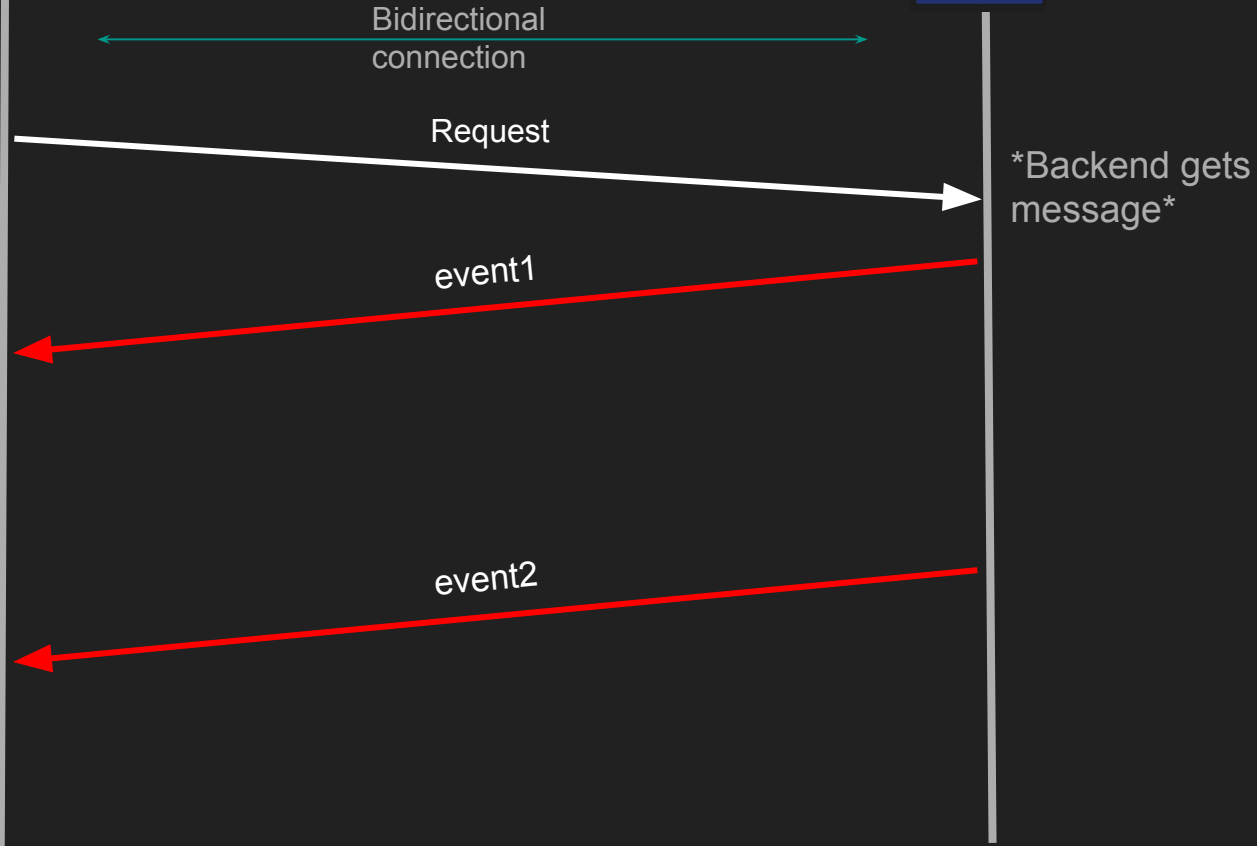
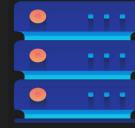
- Vanilla Request/response isn't ideal for notification backend
- Client wants real time notification from backend
  - A user just logged in
  - A message is just received
- Push works but restrictive
- Server Sent Events work with Request/response
- Designed for HTTP

# What is Server Sent Events?

- A response has start and end
- Client sends a request
- Server sends logical events as part of response
- Server never writes the end of the response
- It is still a request but an unending response
- Client parses the streams data looking for events
- Works with request/response (HTTP)



# SSE



# Server Sent Events Pros and Cons

- Pros
  - Real time
  - Compatible with Request/response
- Cons
  - Clients must be online
  - Clients might not be able to handle
  - Polling is preferred for light clients
  - HTTP/1.1 problem (6 connections)

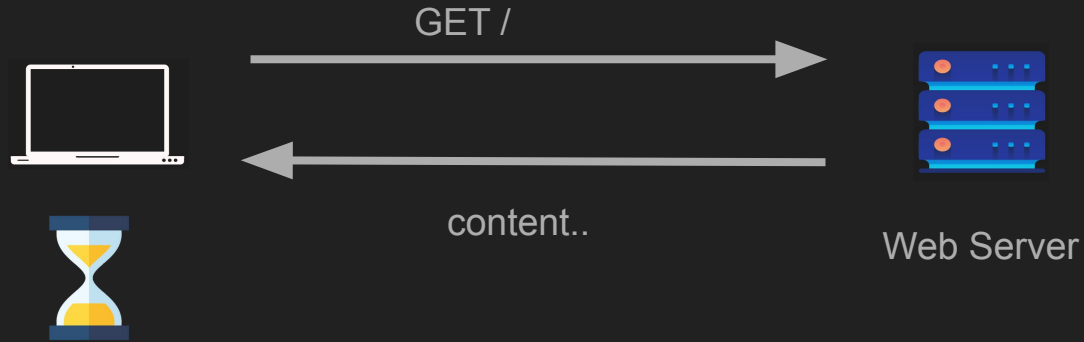
# Demo Job Status

Check if a job is done with progress using SSE

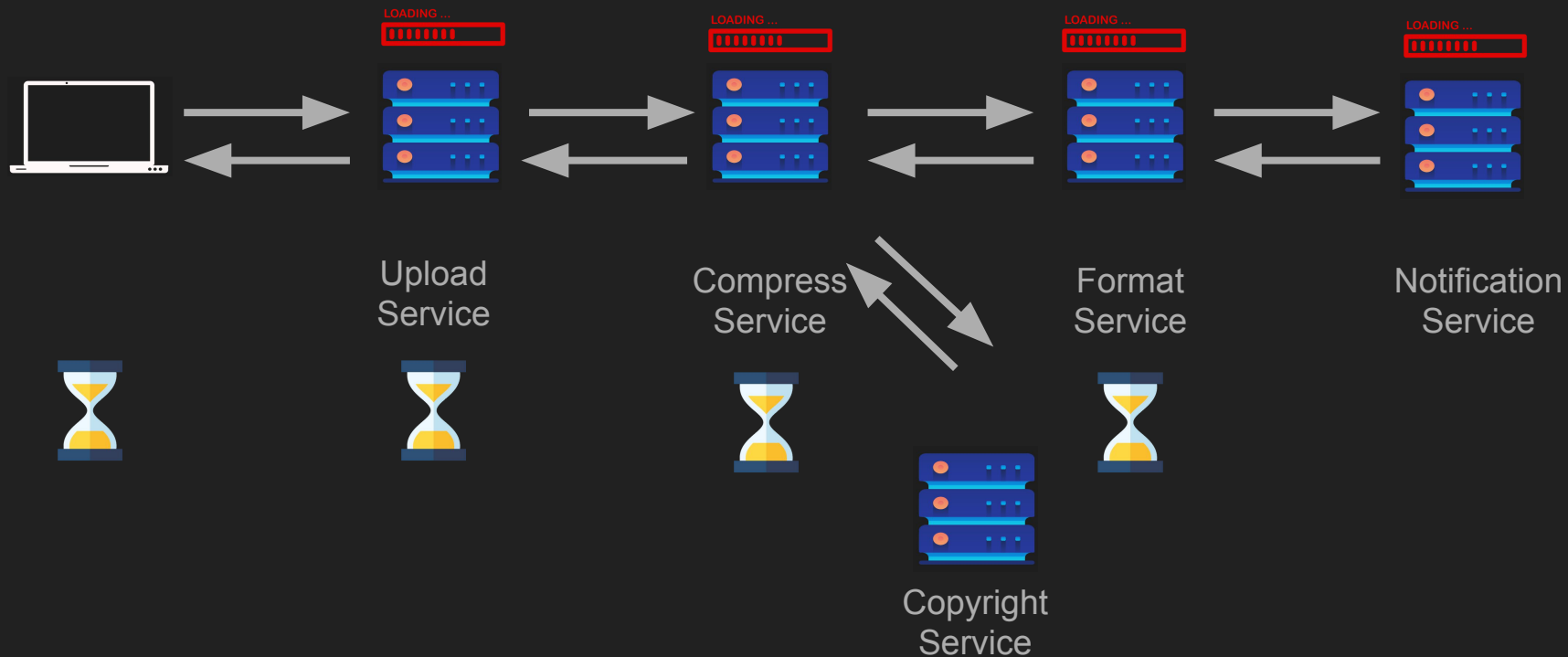
# Publish Subscribe

One publisher many readers

# Request Response



# Where it breaks





# Request/Response pros and cons

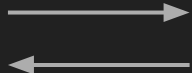
## Pros

- Elegant and Simple
- Scalable

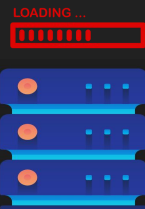
## Cons

- Bad for multiple receivers
- High Coupling
- Client/Server have to be running
- Chaining, circuit breaking

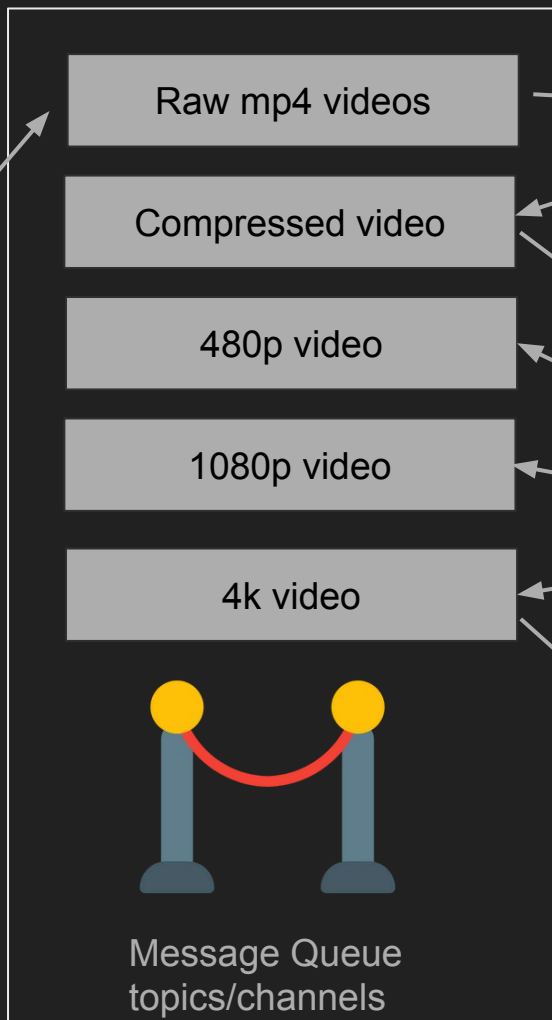
# Pub/Sub



Uploaded!



Upload Service



Raw mp4 videos

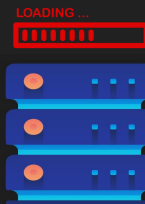
Compressed video

480p video

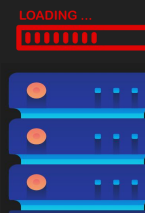
1080p video

4k video

Message Queue  
topics/channels



Compress  
Service



Format  
Service



Notification  
Service

# Pub/Sub pros and cons

## Pros

- Scales w/ multiple receivers
- Great for microservices
- Loose Coupling
- Works while clients not running

## Cons

- Message delivery issues (Two general problem)
- Complexity
- Network saturation

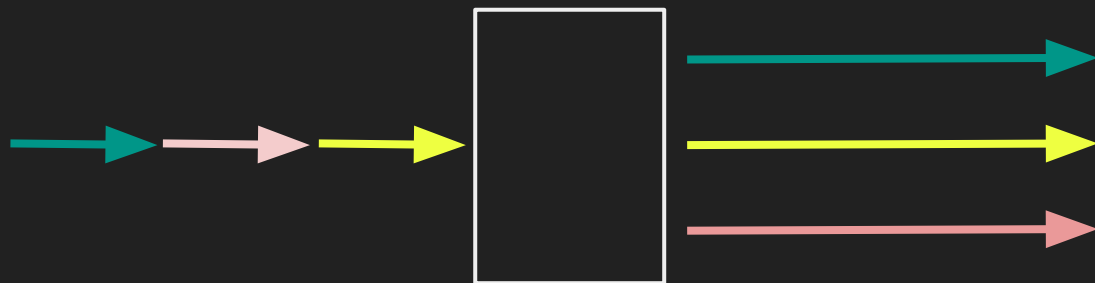
# RabbitMQ Demo

Publish a job and have worker pick them up

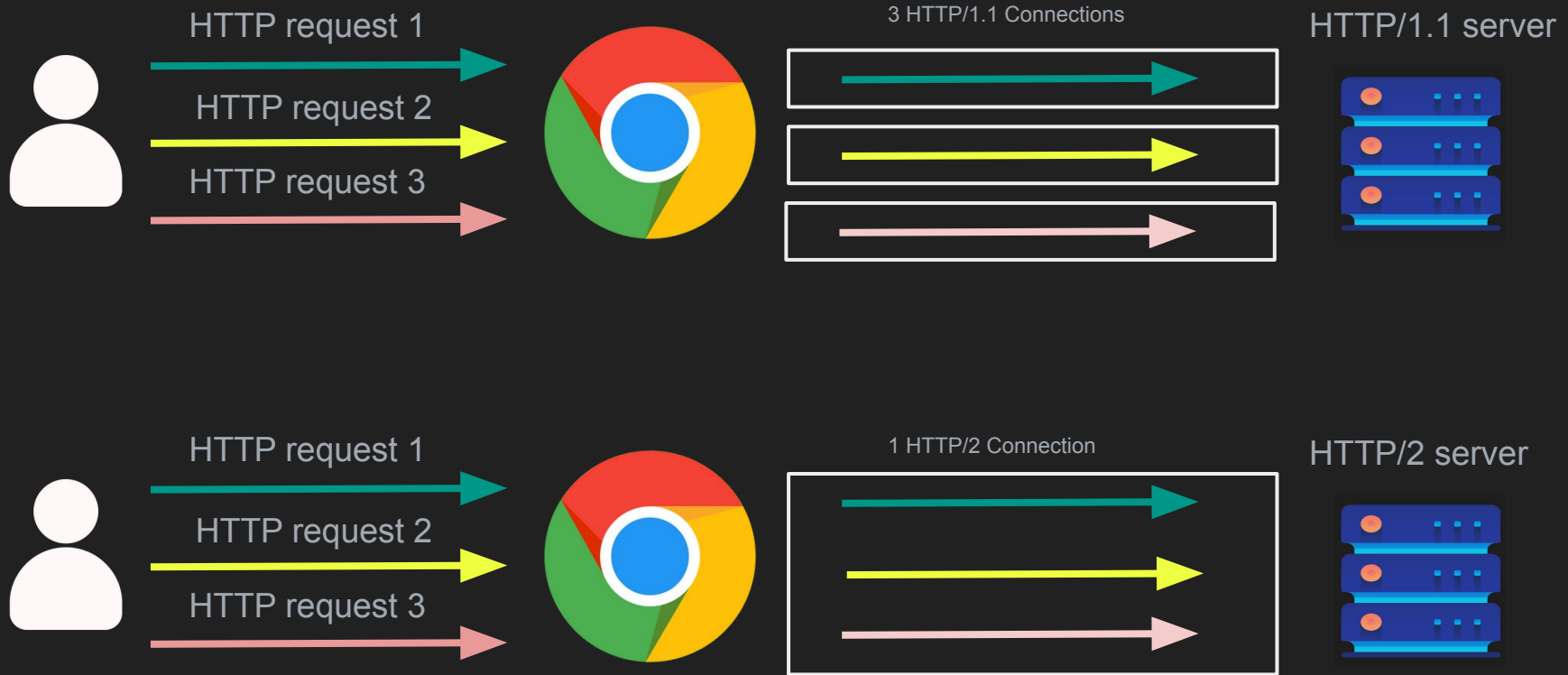
# Multiplexing vs Demultiplexing

HTTP/2, QUIC,  
Connection Pool, MPTCP

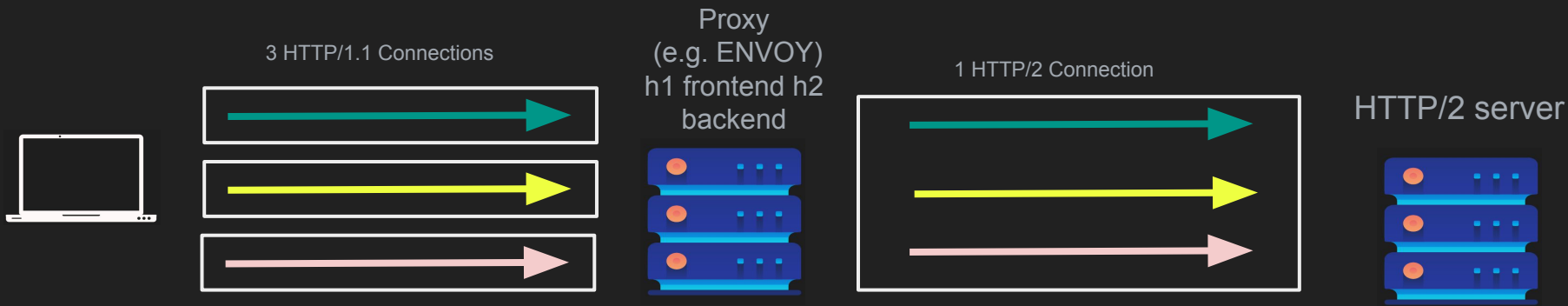
# Multiplexing vs Demultiplexing



# Multiplexing example HTTP/2



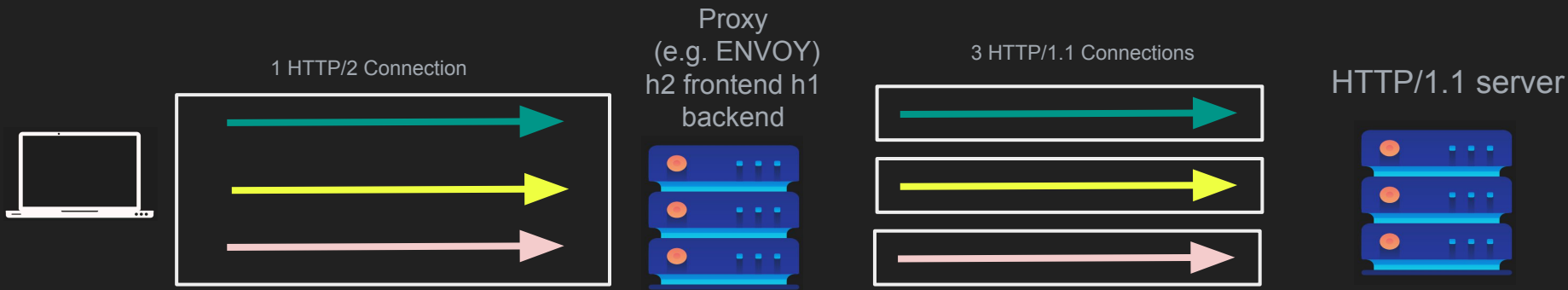
# Multiplexing HTTP/2 on the Backend



More throughput  
High backend resources (CPU for h2)

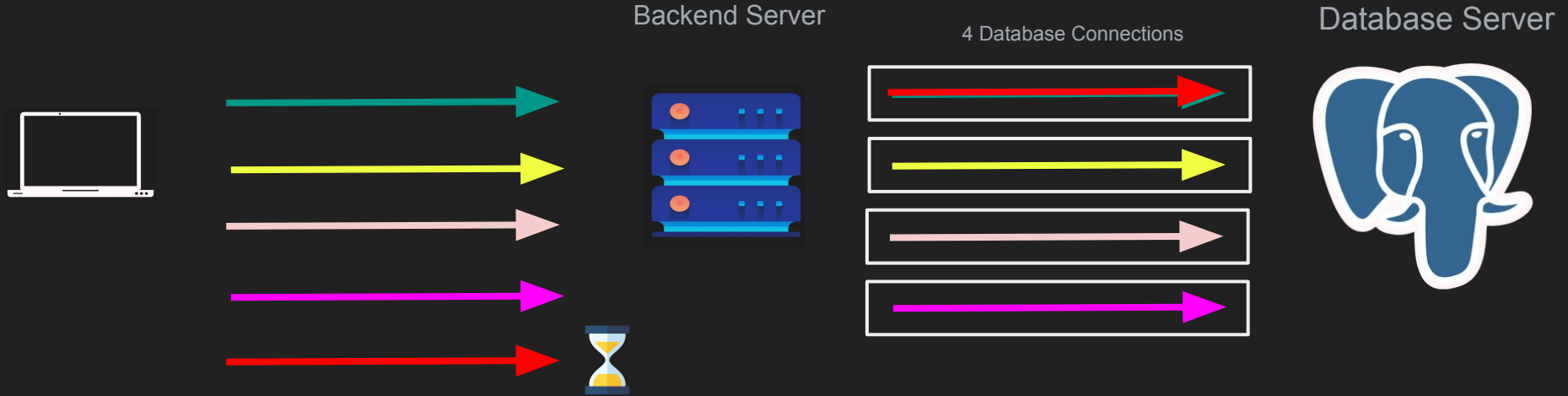


# Demultiplexing HTTP/1.1 on the Backend

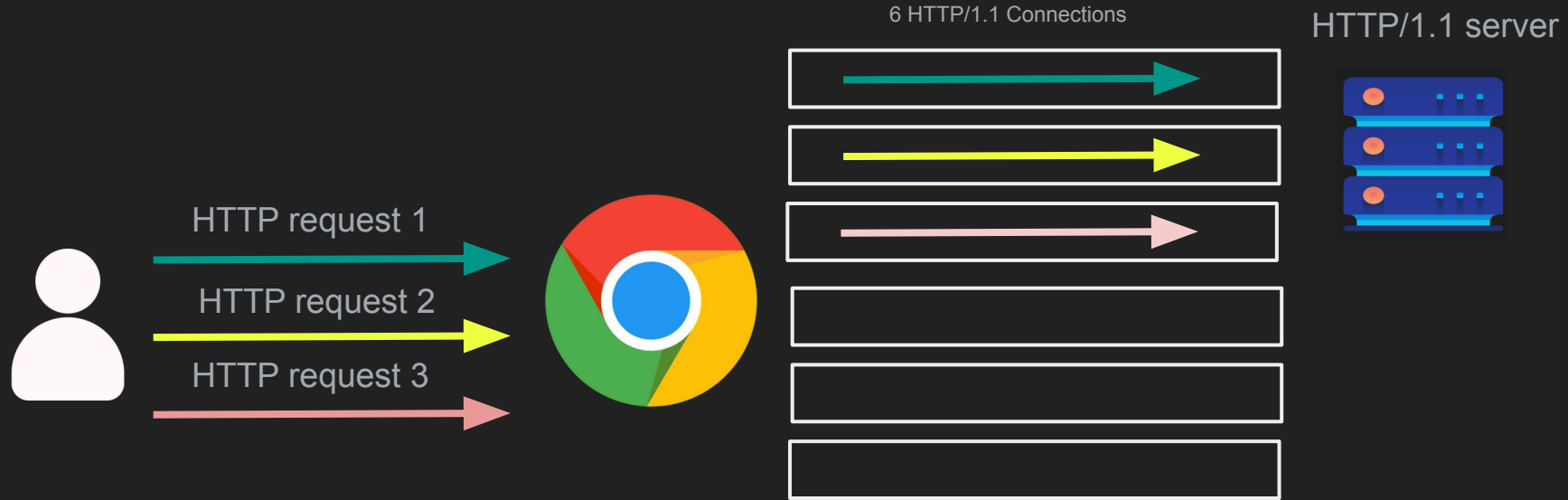


less throughput  
low backend resources (simple h1)

# Connection Pooling



# Browser demultiplexing in HTTP/1.1



Note: Chrome allows up to 6 connections per domain, user's requests are demultiplexed in the 6 connections

# Browser connection pool Demo

Up to 6 connections in HTTP/1.1  
After that we are blocked

# Stateless vs Stateful

Is state stored in the backend?

# Stateful vs Stateless backend

- Stateful
  - Stores state about clients in its memory
  - Depends on the information being there
- Stateless
  - Client is responsible to “transfer the state” with every request
  - May store but can safely lose it

# Stateless Backends

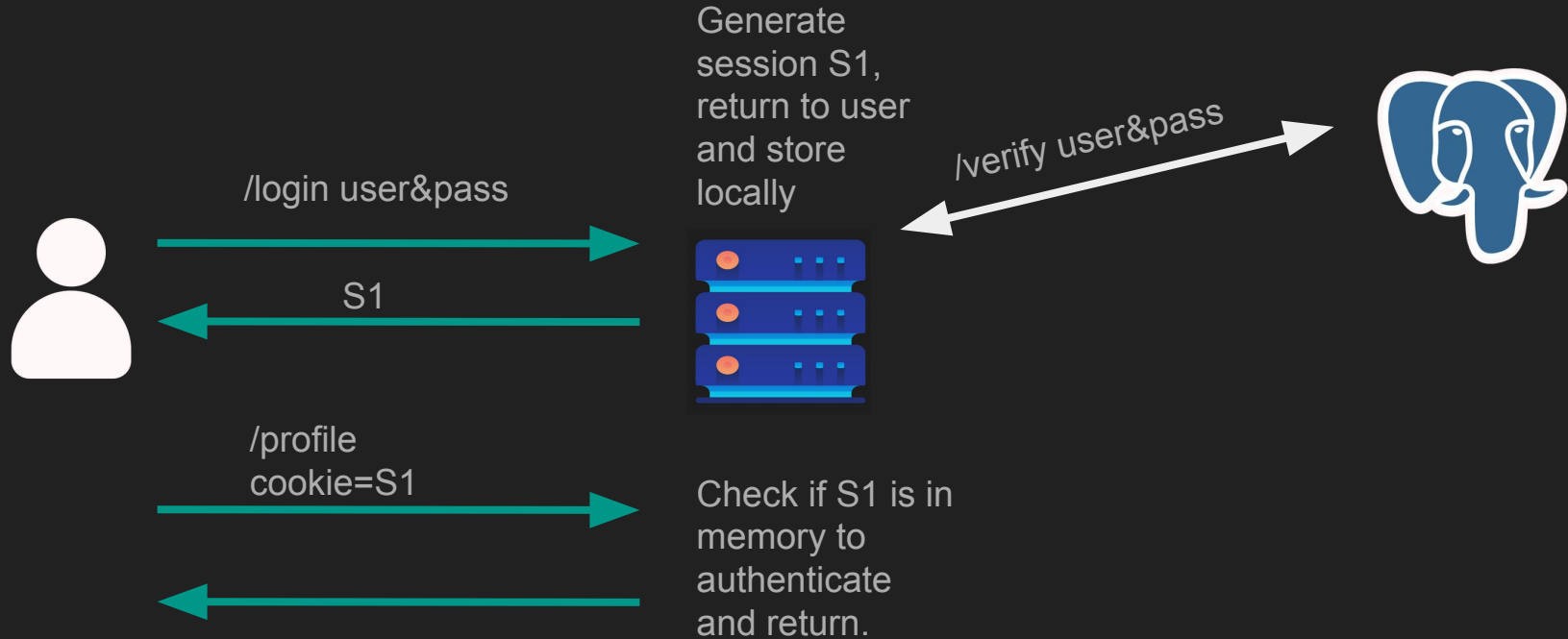
- Stateless backends can still store data somewhere else
- Can you restart the backend during idle time while the client workflow continues to work?

# What makes a backend stateless?

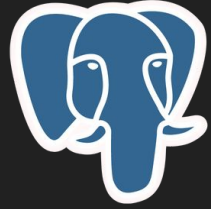
- Stateless backends can store state somewhere else (database)
- The backend remain stateless but the system is stateful
- Can you restart the backend during idle time and the client workflow continue to work?



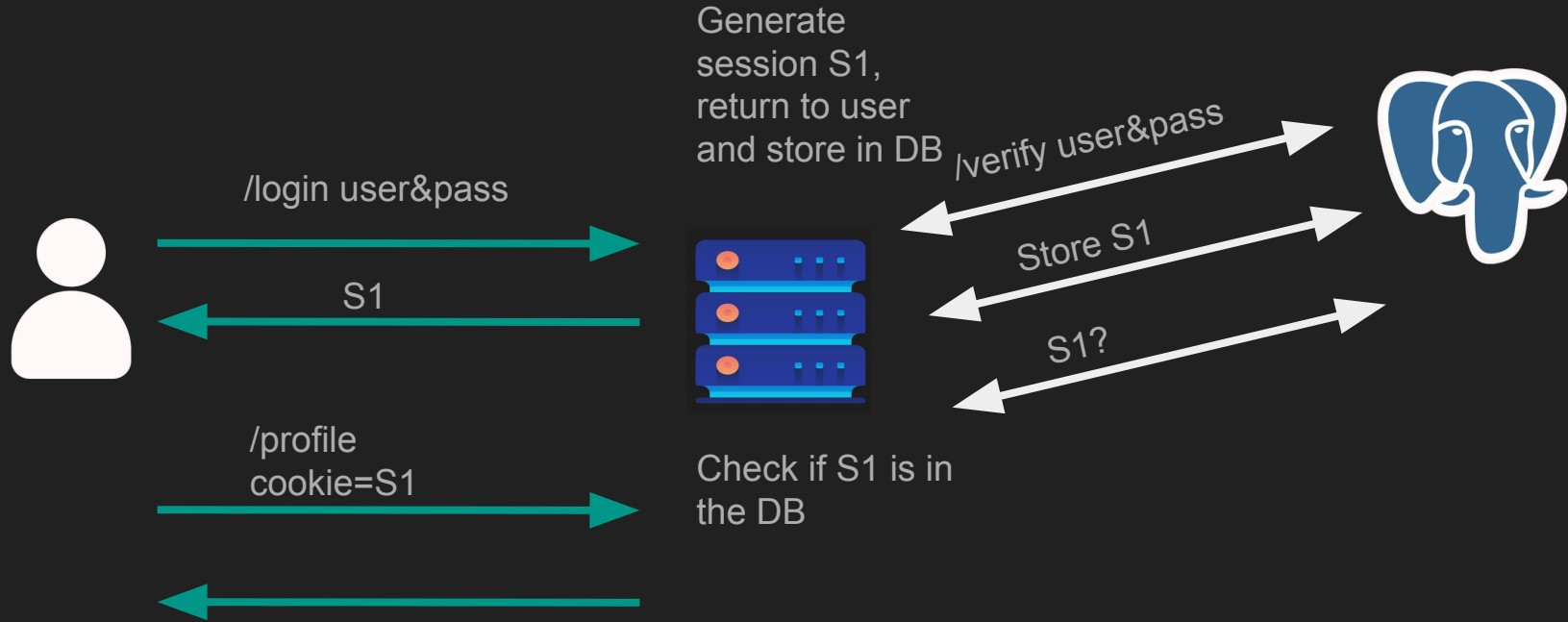
# Stateful backend



# Stateful backend where it breaks



# Stateless backend



Backend App is stateless but entire system still stateful (database dies we lose everything)

# Stateless vs Stateful protocols

- The Protocols can be designed to store state
- TCP is stateful
  - Sequences, Connection file descriptor
- UDP is stateless
  - DNS send queryID in UDP to identify queries
  - QUIC sends connectionID to identify connection

# Stateless vs Stateful protocols

- You can build a stateless protocol on top of a stateful one and vice versa
- HTTP on top of TCP
- If TCP breaks, HTTP blindly create another one
- QUIC on top UDP

# Complete Stateless System

- Stateless Systems are rare
- State is carried with every request
- A backend service that relies completely on the input
  - Check if input param is a prime number
- JWT (JSON Web Token)
- Definitions go nowhere

# HTTP vs TCP Demo

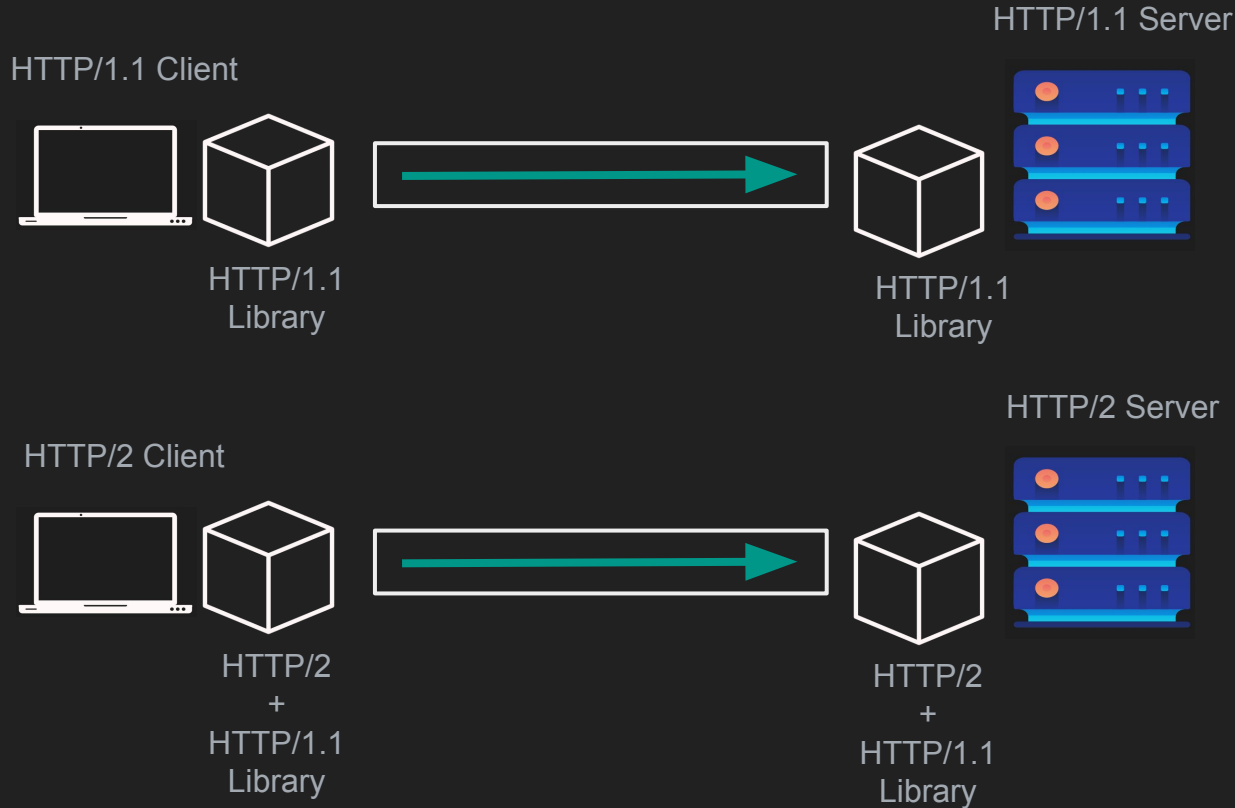
Restart a TCP server vs Restart an HTTP Server

# Sidecar Pattern

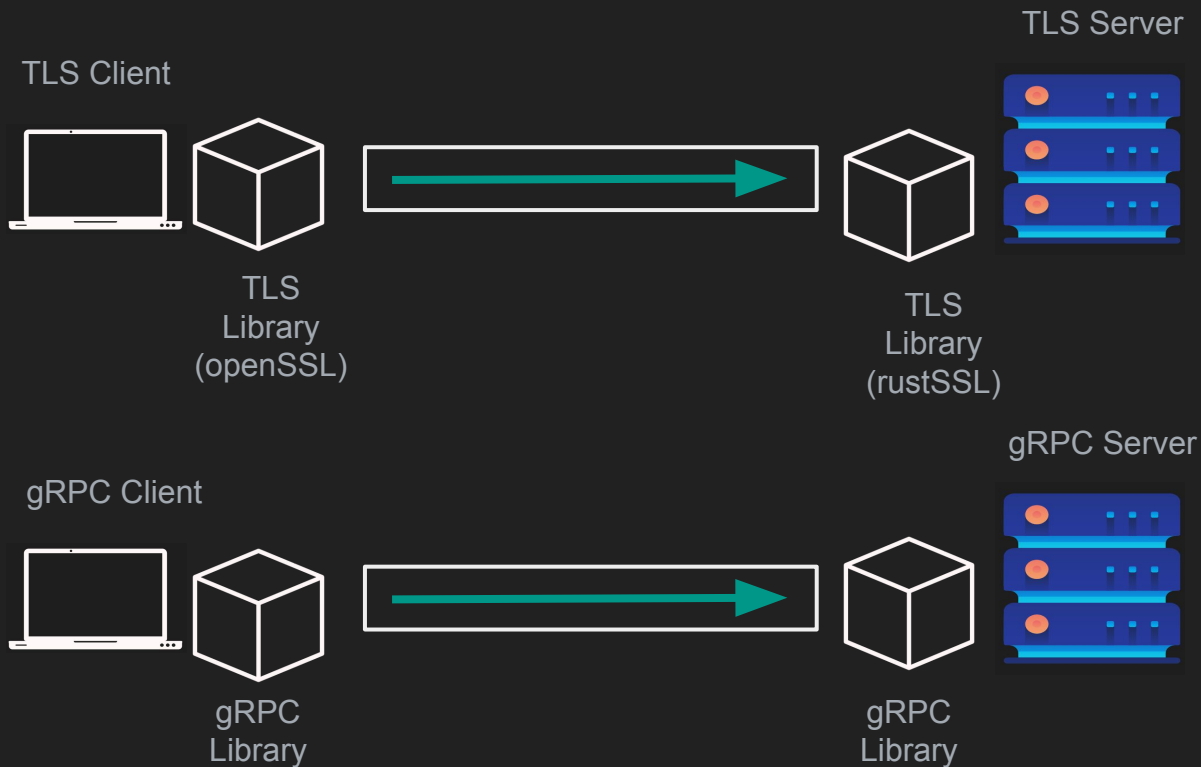
Thick clients, Thicker backends



# Every protocol requires a library



# Every protocol requires a library

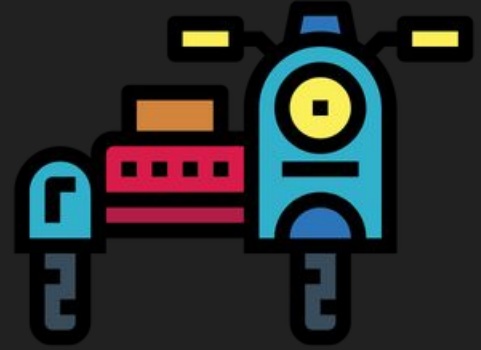


# Changing the library is hard

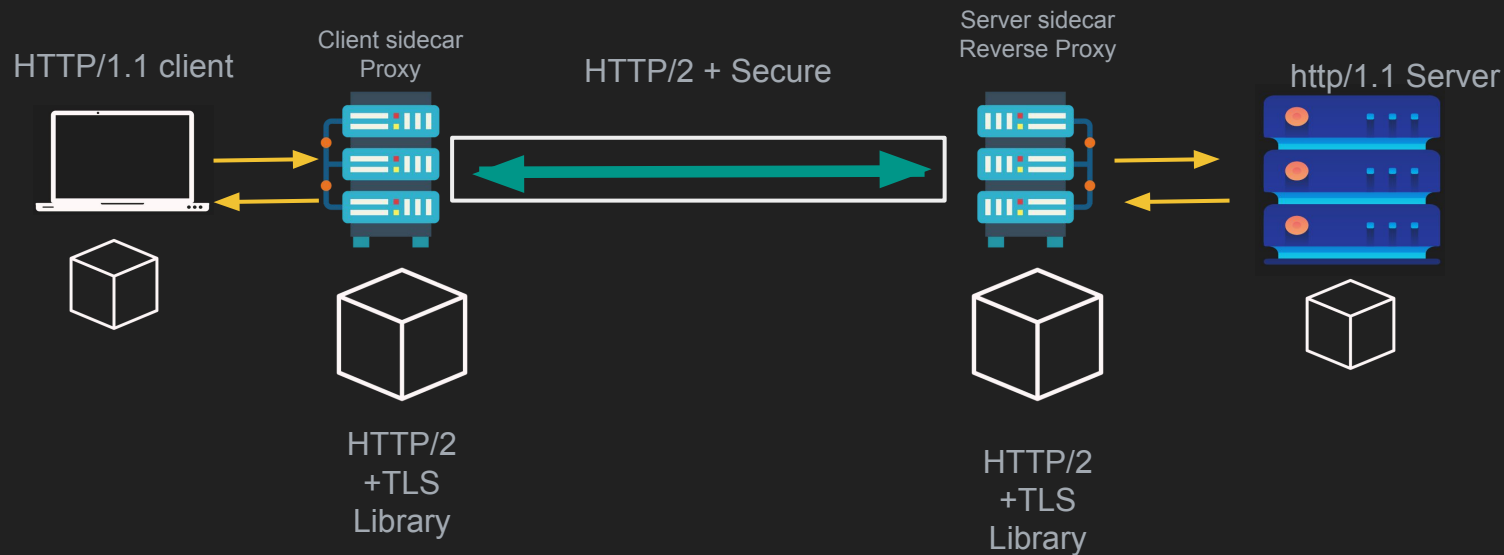
- Once you use the library your app is entrenched
- App & Library “should” be same language
- Changing the library require retesting
- Breaking changes Backward compatibility
- Adding features to the library is hard
- Microservices suffer

# When if we delegate communication?

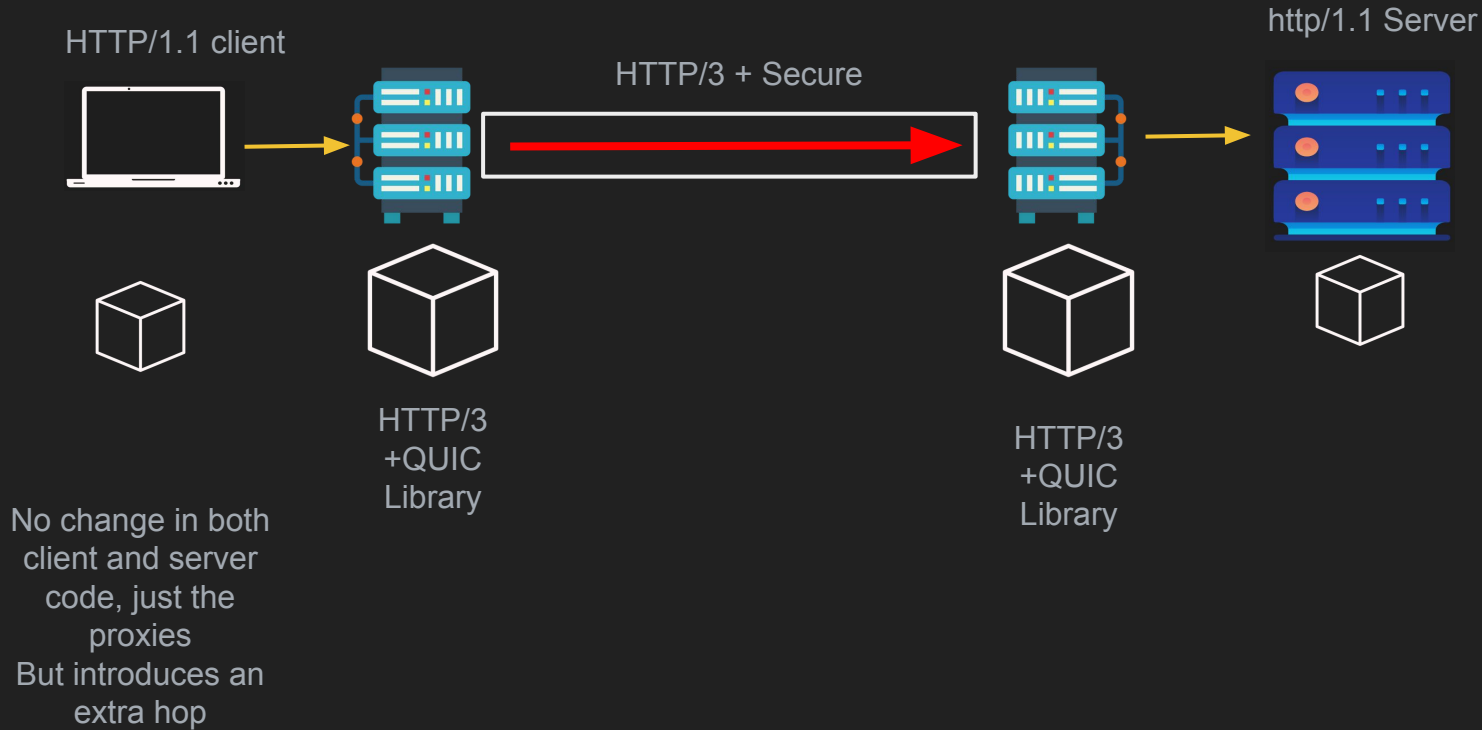
- Proxy communicate instead
- Proxy has the rich library
- Client has thin library (e.g. h1)
- Meet Sidecar pattern
- Each client must have a sidecar proxy



# Sidecar Design Pattern



# Sidecar Design Pattern - Upgrade to HTTP/3!



# Sidecar Examples

- Service Mesh Proxies
  - Linkerd, Istio, Envoy
- Sidecar Proxy Container
- Must be Layer 7 Proxy

# Pros & Cons of Sidecar proxy

## Pros

- Language agnostic (polyglot)
- Protocol upgrade
- Security
- Tracing and Monitoring
- Service Discovery
- Caching

## Cons

- Complexity
- Latency



# Protocols

How the client and backend communicate

# Protocol Properties

What to take into account when designing a protocol?

# What is a protocol?

- A system that allows two parties to communicate
- A protocol is designed with a set of properties
- Depending on the purpose of the protocol
- TCP, UDP, HTTP, gRPC, FTP

# Protocol properties

- Data format
  - Text based (plain text, JSON, XML)
  - Binary (protobuf, RESP, h2, h3)
- Transfer mode
  - Message based (UDP, HTTP)
  - Stream (TCP, WebRTC)
- Addressing system
  - DNS name, IP, MAC
- Directionality
  - Bidirectional (TCP)
  - Unidirectional (HTTP)
  - Full/Half duplex

# Protocol properties

- State
  - Stateful (TCP, gRPC, apache thrift)
  - Stateless (UDP, HTTP)
- Routing
  - Proxies, Gateways
- Flow & Congestion control
  - TCP (Flow & Congestion)
  - UDP (No control)
- Error management
  - Error code
  - Retries and timeouts

# OSI Model

Open Systems Interconnection model

# Why do we need a communication model?

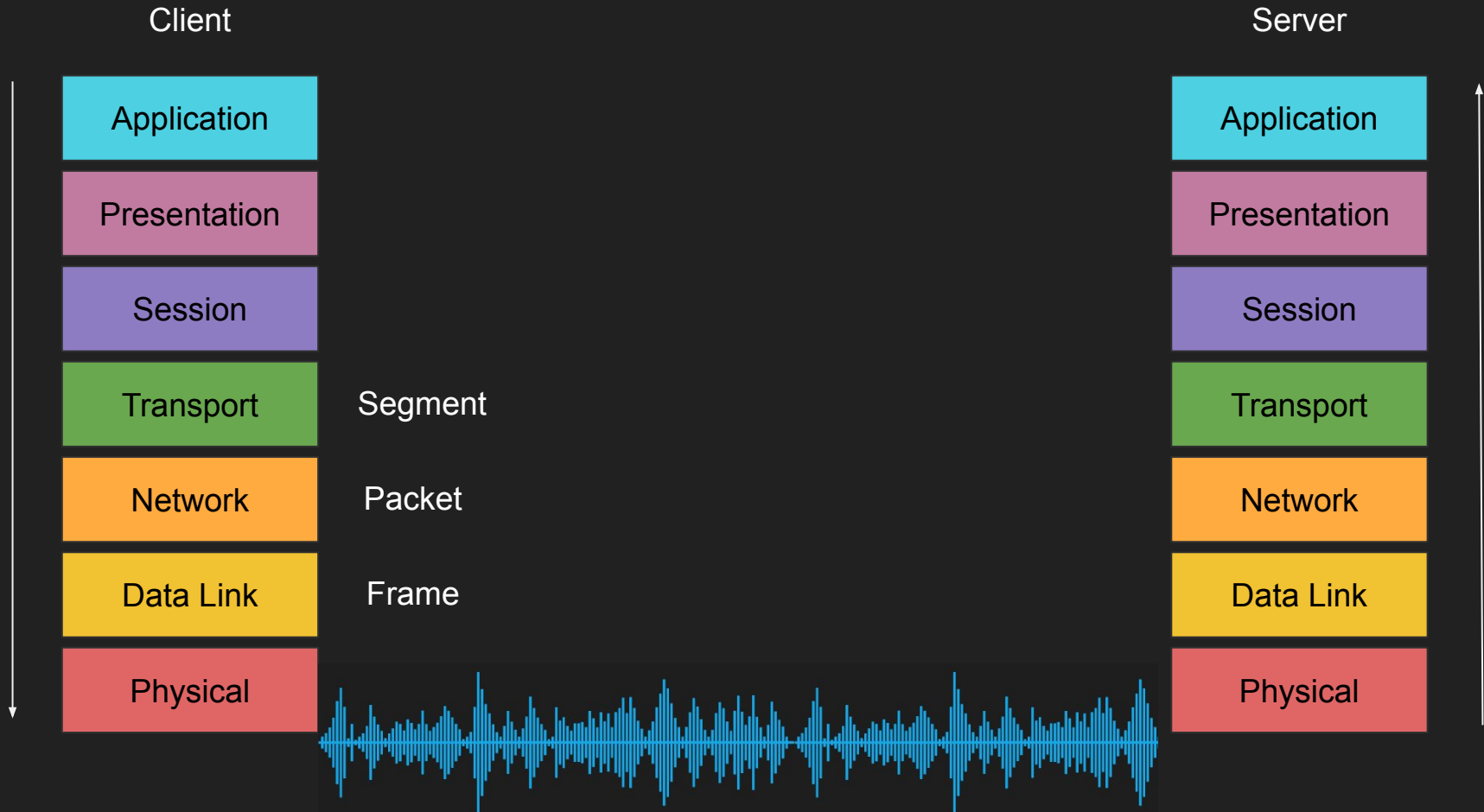
- Agnostic applications
  - App doesn't need to know network medium
  - Otherwise we need an App for WIFI, ethernet vs LTE vs fiber
- Network Equipment Management
  - Without a standard model, upgrading network equipments becomes difficult
- Decoupled Innovation
  - Innovations can be done in each layer separately without affecting the rest of the models

# What is the OSI Model?

- 7 Layers each describe a specific networking component
- Layer 7 - Application - HTTP/FTP/gRPC
- Layer 6 - Presentation - Encoding, Serialization
- Layer 5 - Session - Connection establishment, TLS
- Layer 4 - Transport - UDP/TCP
- Layer 3 - Network - IP
- Layer 2 - Data link - Frames, Mac address Ethernet
- Layer 1 - Physical - Electric signals, fiber or radio waves

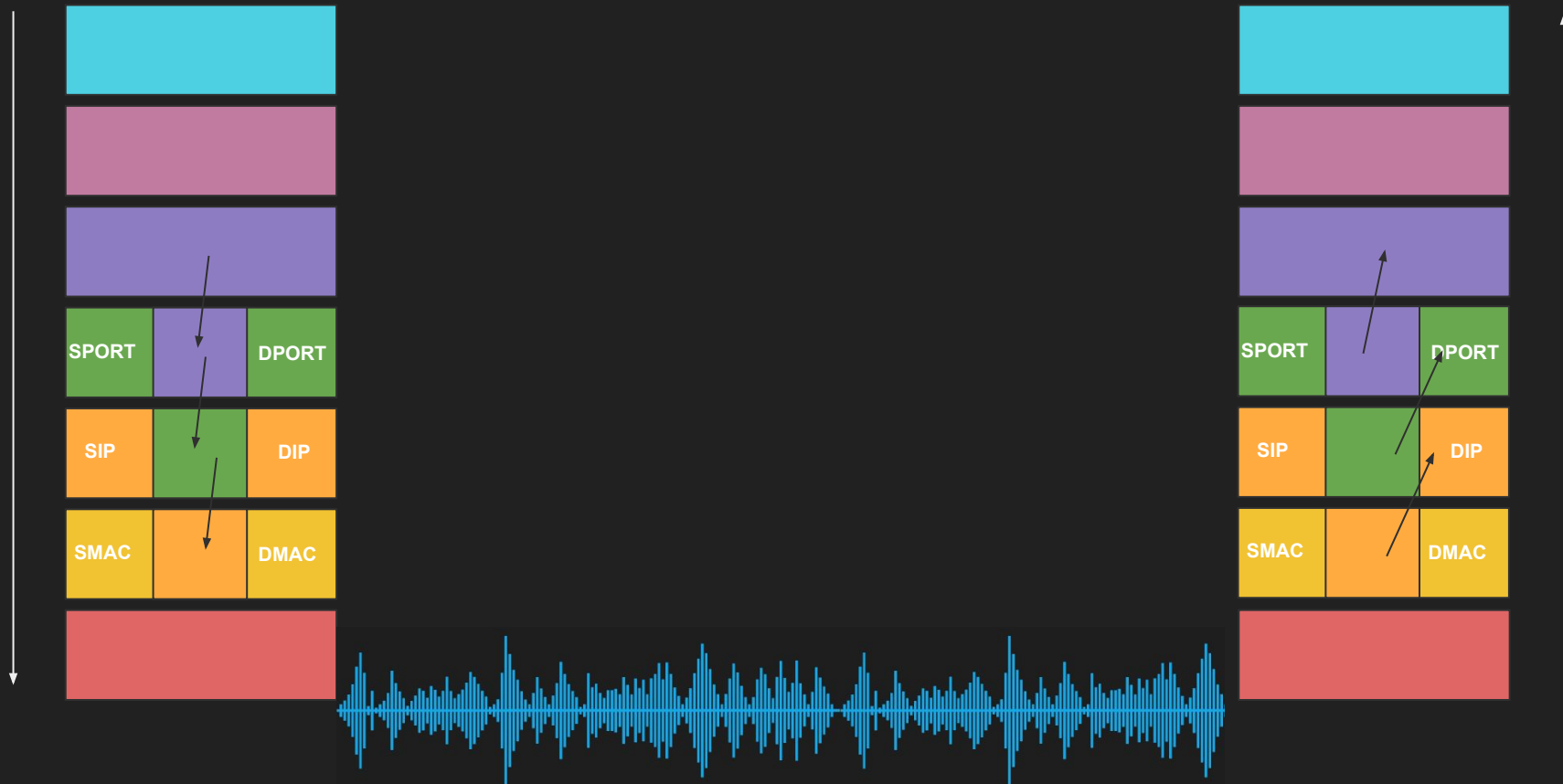


Client sends an HTTPS POST request

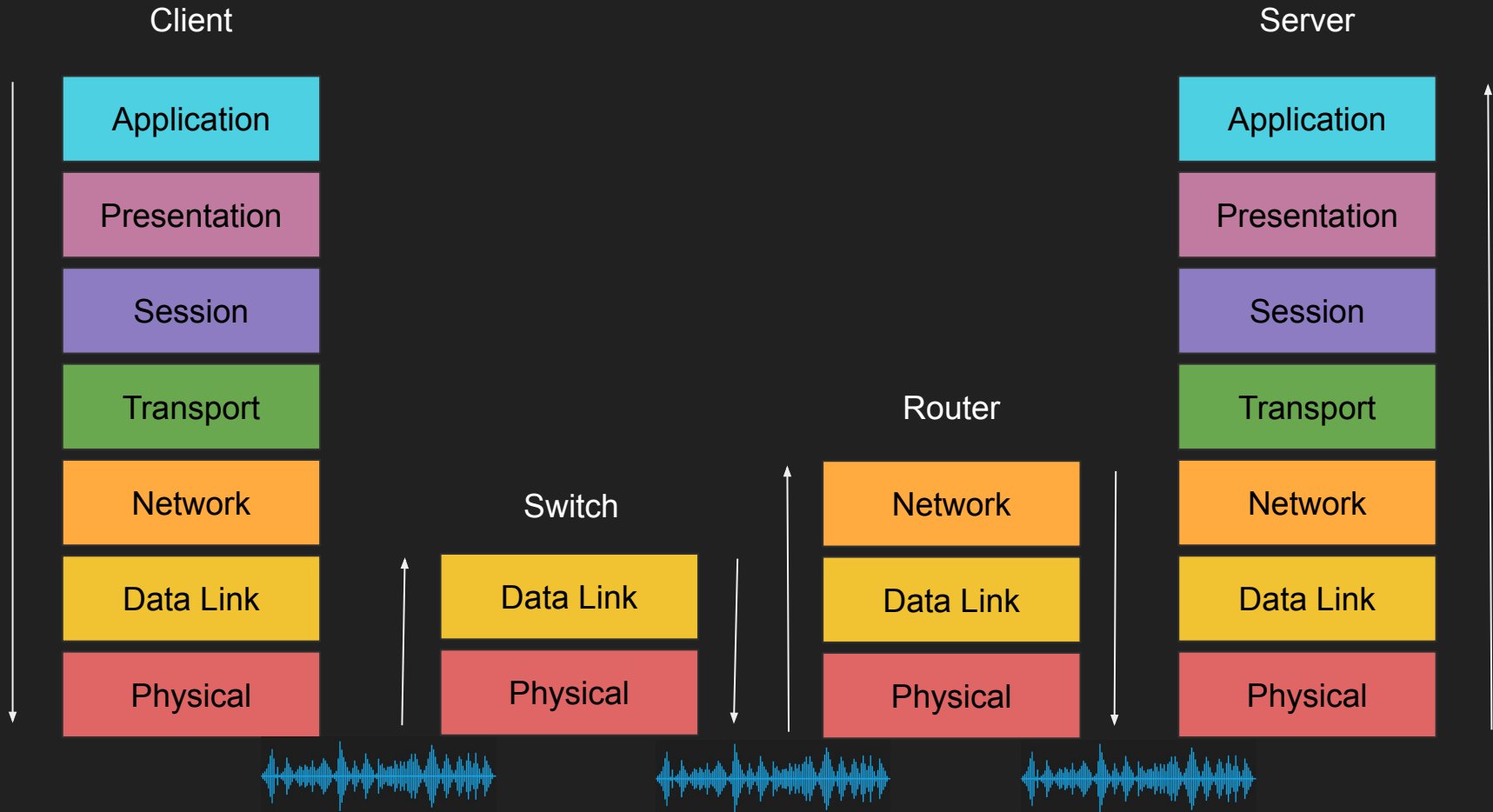


Client

Server



# Across networks

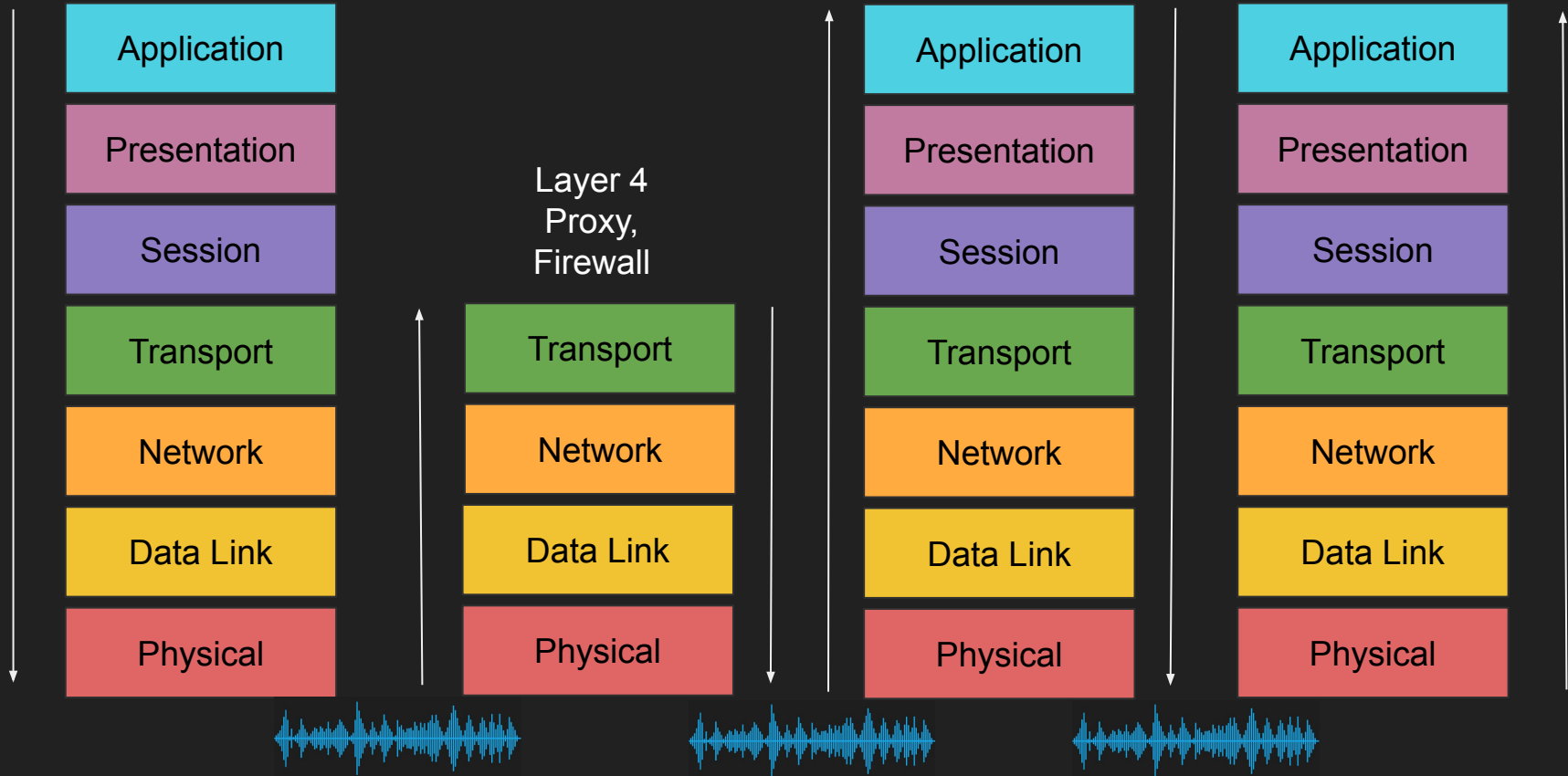


Across networks

Client

Layer 7 Load  
Balancer/CDN

Backend  
Server



# The shortcomings of the OSI Model

- OSI Model has too many layers which can be hard to comprehend
- Hard to argue about which layer does what
- Simpler to deal with Layers 5-6-7 as just one layer, application
- TCP/IP Model does just that

# TCP/IP Model

- Much simpler than OSI just 4 layers
- Application (Layer 5, 6 and 7)
- Transport (Layer 4)
- Internet (Layer 3)
- Data link (Layer 2)
- Physical layer is not officially covered in the model

# OSI Model Summary

- Why do we need a communication model?
- What is the OSI Model?
- Example
- Each device in the network doesn't have to map the entire 7 layers
- TCP/IP is simpler model

# UDP

User Datagram Protocol



# UDP

- Stands for User Datagram Protocol
- Message Based Layer 4 protocol
- Ability to address processes in a host using ports
- Simple protocol to send and receive messages
- Prior communication not required (double edge sword)
- Stateless no knowledge is stored on the host
- 8 byte header Datagram

# UDP Use cases

- Video streaming
- VPN
- DNS
- WebRTC



# Multiplexing and demultiplexing

- IP target hosts only
- Hosts run many apps each with different requirements
- Ports now identify the “app” or “process”
- Sender multiplexes all its apps into UDP
- Receiver demultiplex UDP datagrams to each app

App1-port 5555  
App2-port 7712  
App3-port 2222



10.0.0.1

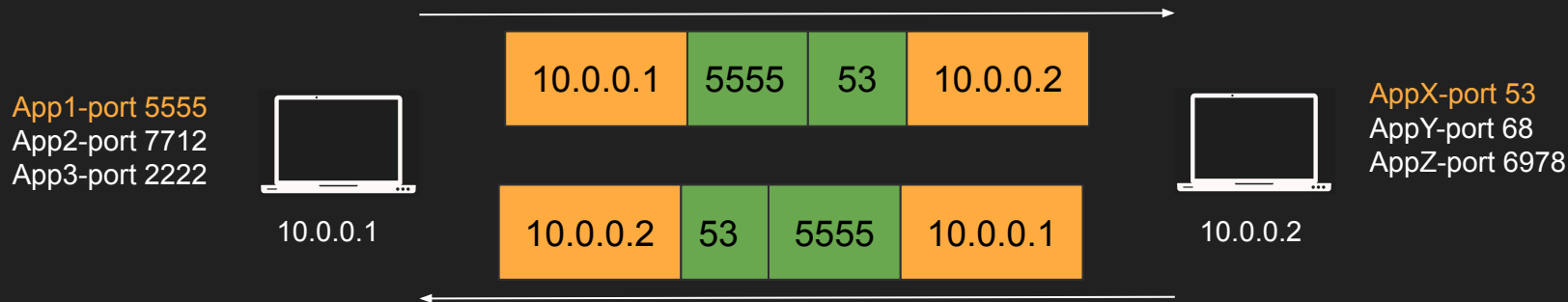


10.0.0.2

AppX-port 53  
AppY-port 68  
AppZ-port 6978

# Source and Destination Port

- App1 on 10.0.0.1 sends data to AppX on 10.0.0.2
- Destination Port = 53
- AppX responds back to App1
- We need Source Port so we know how to send back data
- Source Port = 5555



# UDP Pros

- Simple protocol
- Header size is small so datagrams are small
- Uses less bandwidth
- Stateless
- Consumes less memory (no state stored in the server/client)
- Low latency - no handshake , order, retransmission or guaranteed delivery

# UDP Cons

- No acknowledgement
- No guarantee delivery
- Connection-less - anyone can send data without prior knowledge
- No flow control
- No congestion control
- No ordered packets
- Security - can be easily spoofed

# Summary

- UDP is a simple message based layer 4 protocol
- Uses ports to address processes
- Stateless
- Pros & Cons

# TCP

Transmission Control Protocol



# TCP

- Stands for Transmission Control Protocol
- Stream based Layer 4 protocol
- Ability to address processes in a host using ports
- “Controls” the transmission unlike UDP which is a firehose
- Connection
- Requires handshake
- 20 bytes headers Segment (can go to 60)
- Stateful

# TCP Use cases

- Reliable communication
- Remote shell
- Database connections
- Web communications
- Any bidirectional communication



# TCP Connection

- Connection is a Layer 5 (session)
- Connection is an agreement between client and server
- Must create a connection to send data
- Connection is identified by 4 properties
  - SourceIP-SourcePort
  - DestinationIP-DestinationPort

# TCP Connection

- Can't send data outside of a connection
- Sometimes called socket or file descriptor
- Requires a 3-way TCP handshake
- Segments are sequenced and ordered
- Segments are acknowledged
- Lost segments are retransmitted

# Multiplexing and demultiplexing

- IP target hosts only
- Hosts run many apps each with different requirements
- Ports now identify the “app” or “process”
- Sender multiplexes all its apps into TCP connections
- Receiver demultiplex TCP segments to each app based on connection pairs

App1-port 5555  
App2-port 7712  
App3-port 2222



10.0.0.1



AppX-port 53  
AppY-port 68  
AppZ-port 6978

10.0.0.2

# Connection Establishment

- App1 on 10.0.0.1 want to send data to AppX on 10.0.0.2
- App1 sends SYN to AppX to synchronous sequence numbers
- AppX sends SYN/ACK to synchronous its sequence number
- App1 ACKs AppX SYN.
- Three way handshake

App1-port 5555  
App2-port 7712  
App3-port 2222



10.0.0.1

10.0.0.1:5555:  
10.0.0.2:22

File descriptor

10.0.0.1 5555 SYN 22 10.0.0.2

10.0.0.2 22 SYN/ACK 5555 10.0.0.1

10.0.0.1 5555 ACK 22 10.0.0.2



10.0.0.2

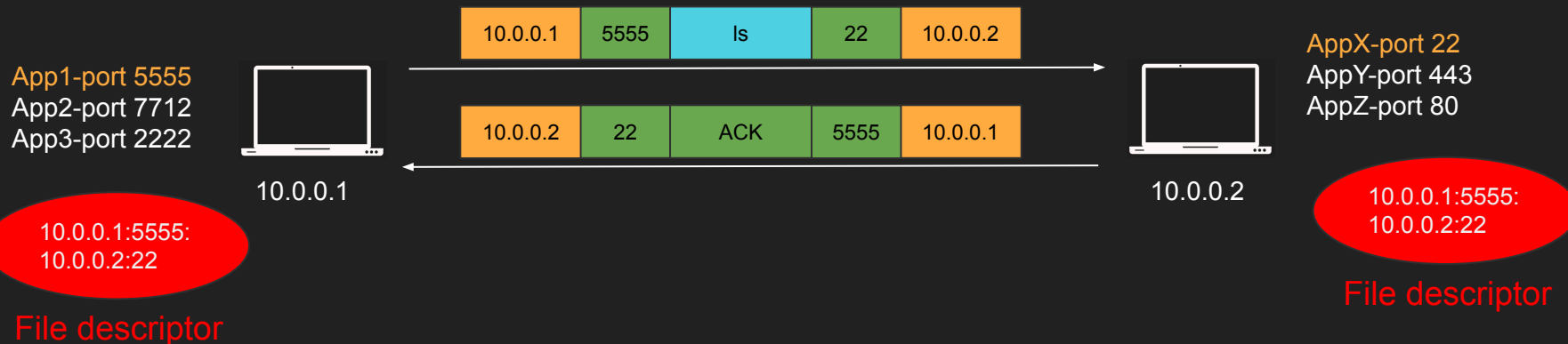
AppX-port 22  
AppY-port 443  
AppZ-port 80

10.0.0.1:5555:  
10.0.0.2:22

File descriptor

# Sending data

- App1 sends data to AppX
- App1 encapsulate the data in a segment and send it
- AppX acknowledges the segment
- Hint: Can App1 send new segment before ack of old segment arrives?



# Acknowledgment

- App1 sends segment 1,2 and 3 to AppX
- AppX acknowledge all of them with a single ACK 3

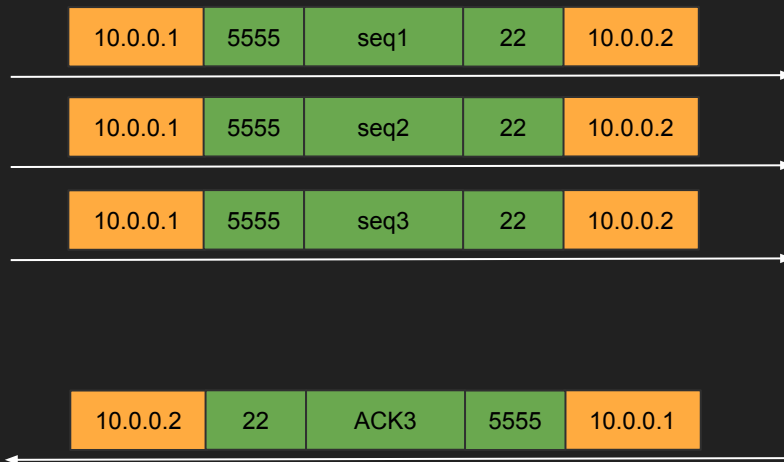
App1-port 5555  
App2-port 7712  
App3-port 2222



10.0.0.1

10.0.0.1:5555:  
10.0.0.2:22

File descriptor



AppX-port 22  
AppY-port 443  
AppZ-port 80



10.0.0.2

10.0.0.1:5555:  
10.0.0.2:22

File descriptor



# Lost data

- App1 sends segment 1,2 and 3 to AppX
- Seg 3 is lost, AppX acknowledge 3
- App1 resend Seq 3

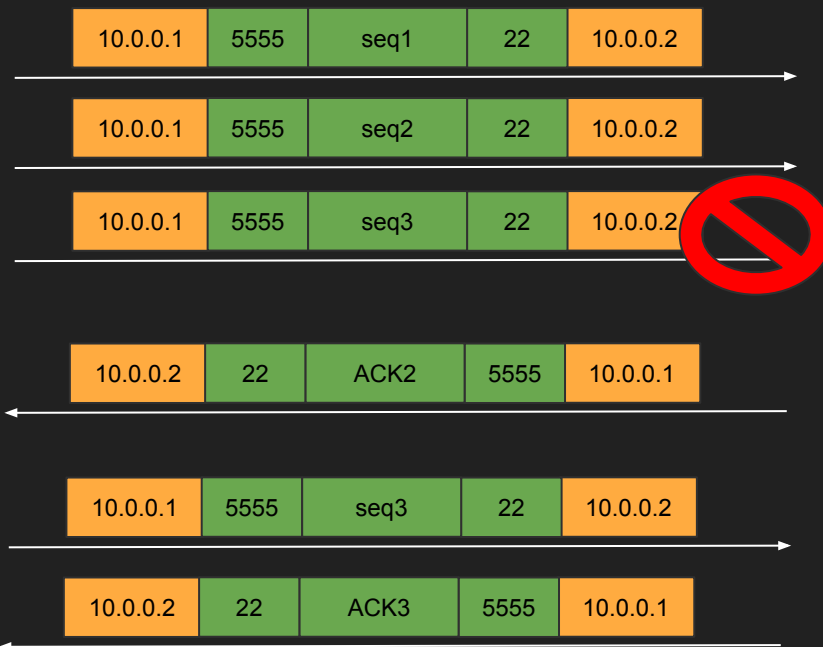
App1-port 5555  
App2-port 7712  
App3-port 2222



10.0.0.1

10.0.0.1:5555:  
10.0.0.2:22

File descriptor



10.0.0.2

AppX-port 22  
AppY-port 443  
AppZ-port 80

10.0.0.1:5555:  
10.0.0.2:22

File descriptor

# Closing Connection

- App1 wants to close the connection
- App1 sends FIN, AppX ACK
- AppX sends FIN, App1 ACK
- Four way handshake

App1-port 5555  
App2-port 7712  
App3-port 2222



10.0.0.1

10.0.0.1:5555:  
10.0.0.2:22

File descriptor

10.0.0.1 5555 FIN 22 10.0.0.2

10.0.0.2 22 ACK 5555 10.0.0.1

10.0.0.2 22 FIN 5555 10.0.0.1

10.0.0.1 5555 ACK 22 10.0.0.2



10.0.0.2

AppX-port 22  
AppY-port 443  
AppZ-port 80

10.0.0.1:5555:  
10.0.0.2:22

File descriptor

# TCP Pros

- Guarantee delivery
- No one can send data without prior knowledge
- Flow Control and Congestion Control
- Ordered Packets no corruption or app level work
- Secure and can't be easily spoofed

# TCP Cons

- Large header overhead compared to UDP
- More bandwidth
- Stateful - consumes memory on server and client
- Considered high latency for certain workloads (Slow start/ congestion/ acks)
- Does too much at a low level (hence QUIC)
  - Single connection to send multiple streams of data (HTTP requests)
  - Stream 1 has nothing to do with Stream 2
  - Both Stream 1 and Stream 2 packets must arrive
- TCP Meltdown
  - Not a good candidate for VPN

# Summary

- Stands for Transmission Control Protocol
- Layer 4 protocol
- “Controls” the transmission unlike UDP which is a firehose
- Introduces Connection concept
- Retransmission, acknowledgement, guaranteed delivery
- Stateful, connection has a state
- Pros & Cons

# TLS

Transport Layer Security

# TLS

- Vanilla HTTP
- HTTPS
- TLS 1.2 Handshake
- Diffie Hellman
- TLS 1.3 Improvements

# HTTP

open



GET /



Headers+  
index.html



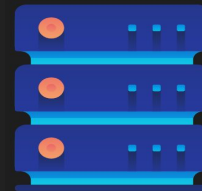
<html>...

close



....

80



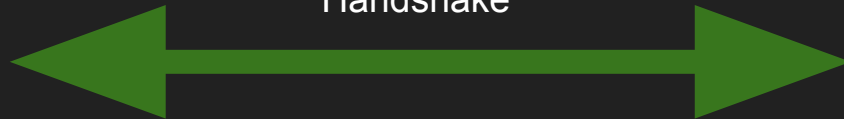


# HTTPS

open



Handshake

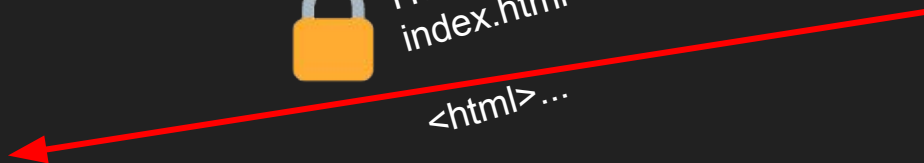


GET /



Headers+  
index.html

<html>...



close

443



....

# Why TLS

- We encrypt with symmetric key algorithms
- We need to exchange the symmetric key
- Key exchange uses asymmetric key (PKI)
- Authenticate the server
- Extensions (SNI, preshared, 0RTT)

# TLS1.2

open



close

Client hello

Server hello (cert)

Change cipher, fin

Change cipher, fin

GET /

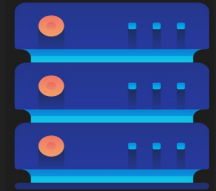
Headers+  
index.html

<html>...  
....

RSA Public key



RSA Private key



# Diffie Hellman

Private  $x$



+

Public  $g, n$



=



Symmetric key

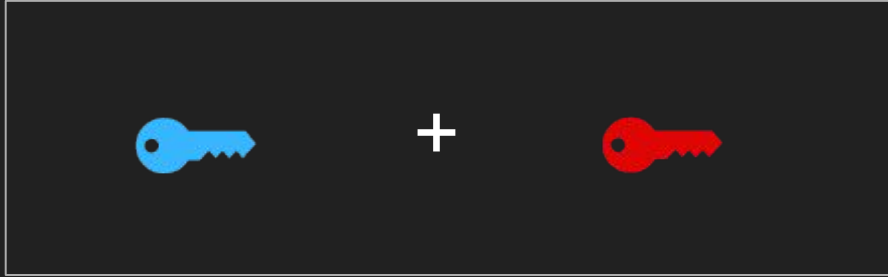
+

Private  $y$

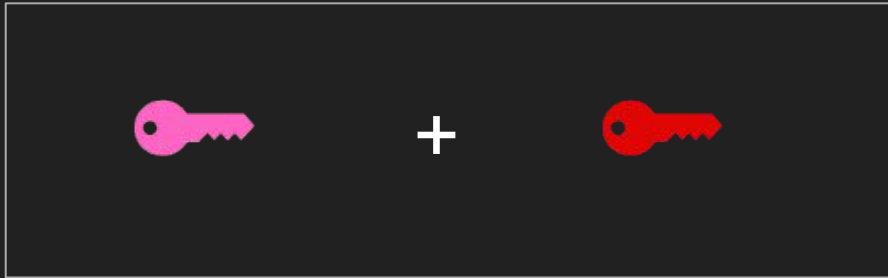


# Diffie Hellman

Public/  
Unbreakable  
/can be shared  
 $g^x \% n$



Public/  
Unbreakable  
/can be shared  
 $g^y \% n$



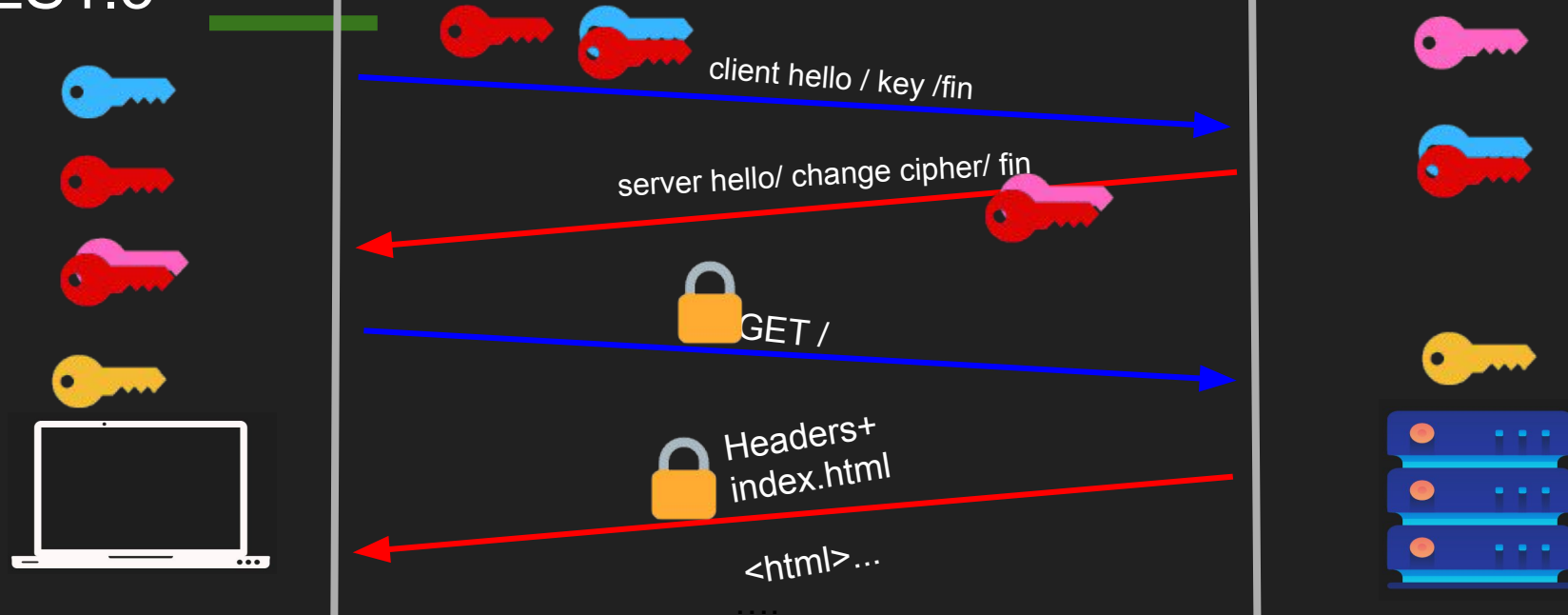
$$(g^x \% n)^y = g^{xy} \% n$$
$$(g^y \% n)^x = g^{xy} \% n$$

# TLS Extensions

- SNI
  - Server Name Indication
- ALPN
  - Application Layer Protocol Negotiation
- Pre-shared key
  - 0RTT
- ECH
  - Encrypted Client Hello

# TLS1.3

open



close

$$(g^{x \% n})^y = g^{xy \% n}$$
$$(g^{y \% n})^x = g^{xy \% n}$$

# TLS Summary

- Vanilla HTTP
- HTTPS
- TLS 1.2 Handshake (two round trips)
- Diffie Hellman
- TLS 1.3 Improvements (one round trip can be zero)



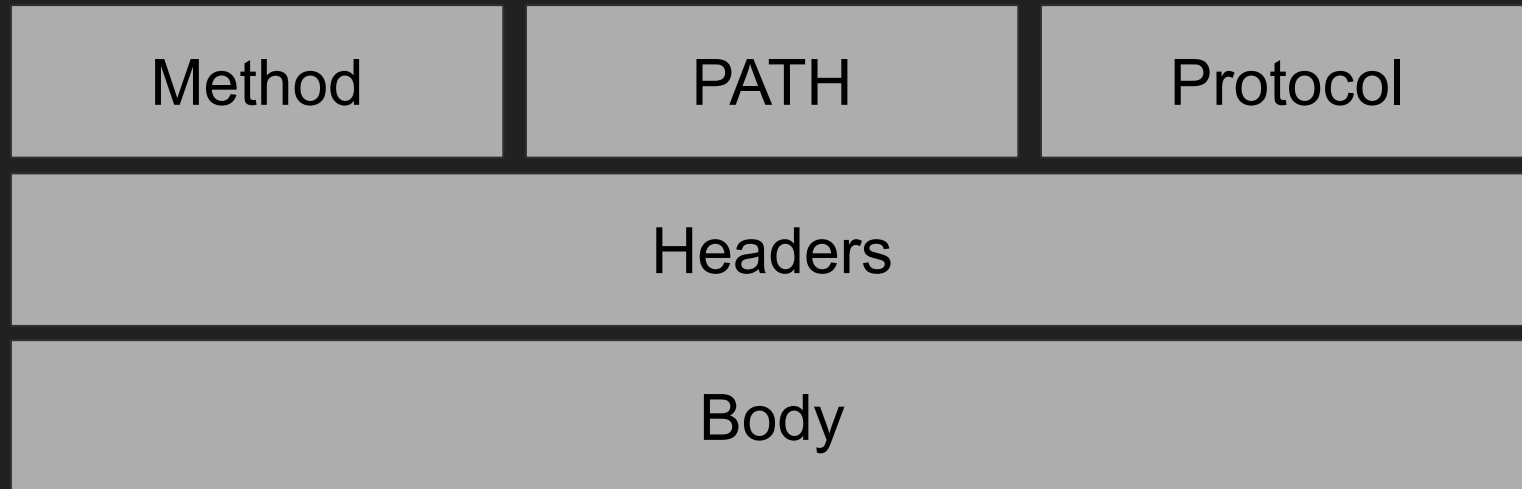
# HTTP/1.1

Simple web protocol lasts decades

# Client / Server

- (Client) Browser, python or javascript app, or any app that makes HTTP request
- (Server) HTTP Web Server, e.g. IIS, Apache Tomcat, NodeJS, Python Tornado

# HTTP Request



# HTTP Request

```
curl -v http://husseinnasser.com/about
```

```
> GET /about HTTP/1.1  
> Host: husseinnasser.com  
> User-Agent: curl/7.79.1  
> Accept: */*
```

# HTTP Response

Protocol	Code	Code Text
Headers		
Body		

# HTTP Response

```
< HTTP/2 301
< location: https://www.husseinnasser.com/about
< date: Wed, 26 Oct 2022 17:10:59 GMT
< content-type: text/html; charset=UTF-8
< server: ghs
< content-length: 232
< x-xss-protection: 0
< x-frame-options: SAMEORIGIN
<
<HTML><HEAD><meta http-equiv="content-type" content
<TITLE>301 Moved</TITLE></HEAD><BODY>
```

# HTTP

open



GET /



Headers+  
index.html

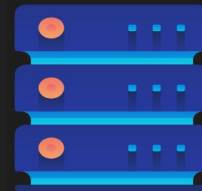
<html>...



close



80



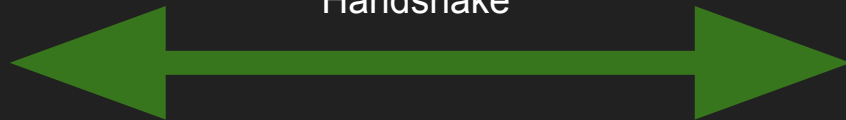
....

# HTTPS

open



Handshake

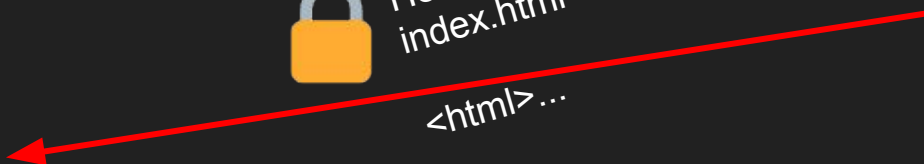


GET /



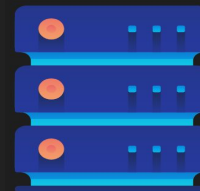
Headers+  
index.html

<html>...



close

443



....



# HTTP 1.0



open

close

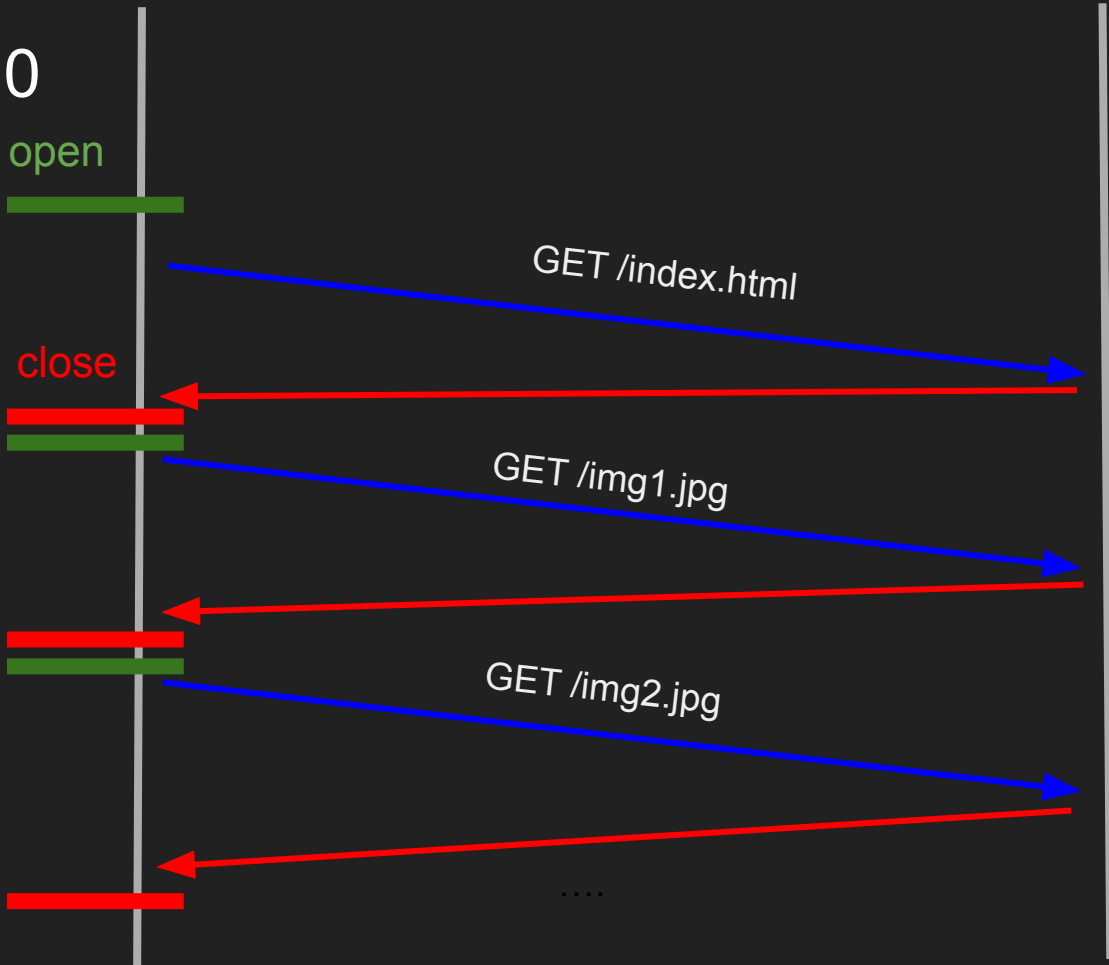
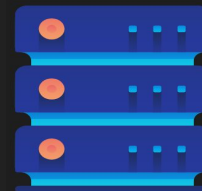
GET /index.html

GET /img1.jpg

GET /img2.jpg

....

80



# HTTP 1.0

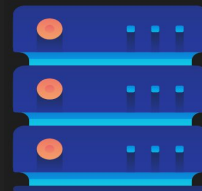
- New TCP connection with each request
- Slow
- Buffering (transfer-encoding:chunked didn't exist)
- No multi-homed websites (HOST header)

# HTTP 1.1

open



80



GET /index.html

GET /img1.jpg

GET /img2.jpg

close

....

# HTTP 1.1 Pipelining



open

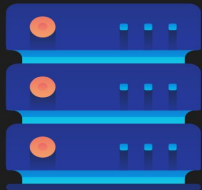


GET /index.html

GET /img1.jpg

GET /img2.jpg

80



close



....

# HTTP 1.1

- Persisted TCP Connection
- Low Latency & Low CPU Usage
- Streaming with Chunked transfer
- Pipelining (disabled by default)
- Proxying & Multi-homed websites

# HTTP/2

- SPDY
- Compression
- Multiplexing
- Server Push
- Secure by default
- Protocol Negotiation during TLS (NPN/ALPN)

# HTTP over QUIC (HTTP/3)

- Replaces TCP with QUIC (UDP with Congestion control)
- All HTTP/2 features
- Without HOL

# Summary

- HTTP Anatomy
- HTTP/1.0 over TCP
- HTTP/1.1 over TCP
- HTTP/2 over TCP
- HTTP over QUIC (HTTP/3)



# WebSockets

Bidirectional communications on the web

# Agenda

- HTTP
- WebSockets
- WebSockets Handshake
- WebSockets use cases
- WebSockets Example
- WebSockets Pros and Cons

# HTTP 1.0



open

close

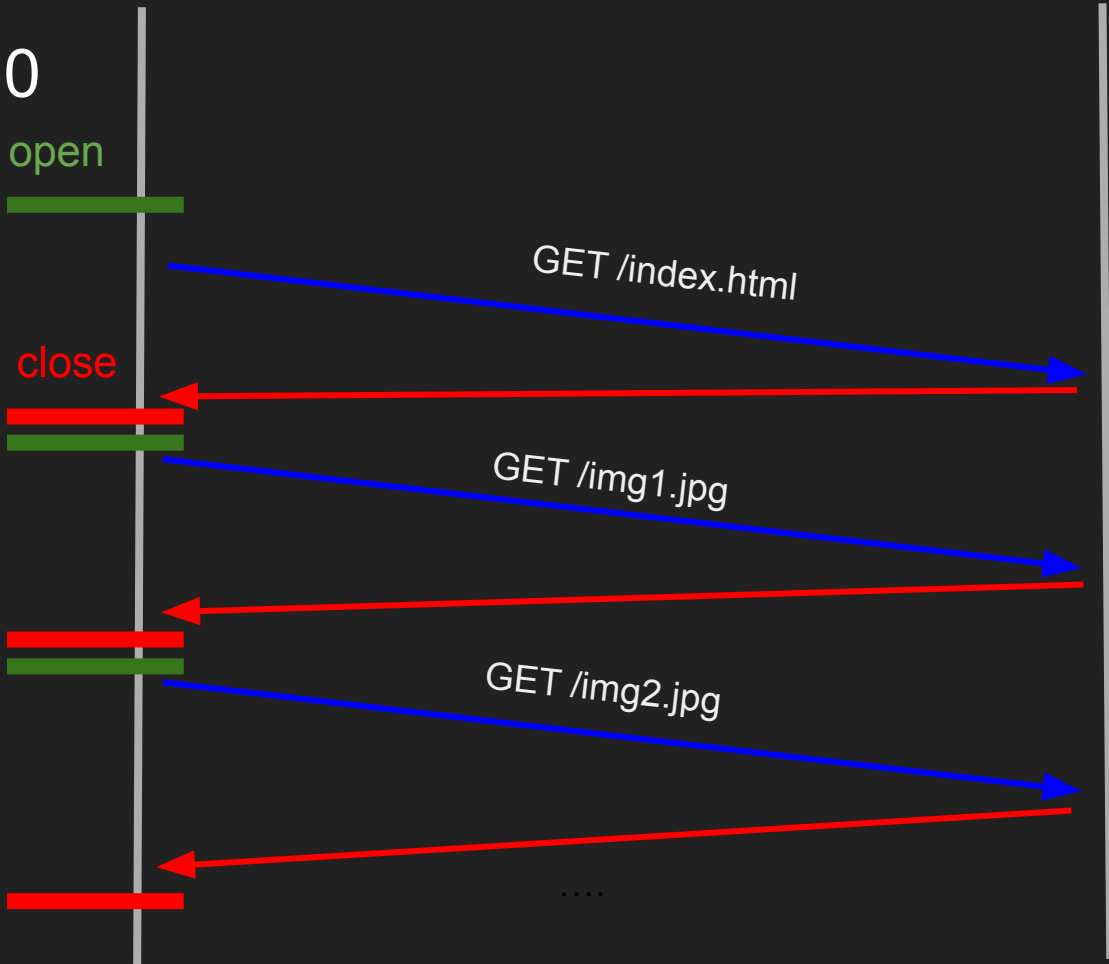
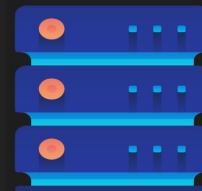
GET /index.html

GET /img1.jpg

GET /img2.jpg

....

80

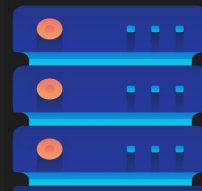


# HTTP 1.1

open



80



GET /index.html

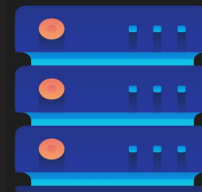
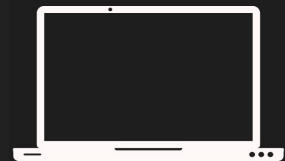
GET /img1.jpg

GET /img2.jpg

close

....

# WebSockets



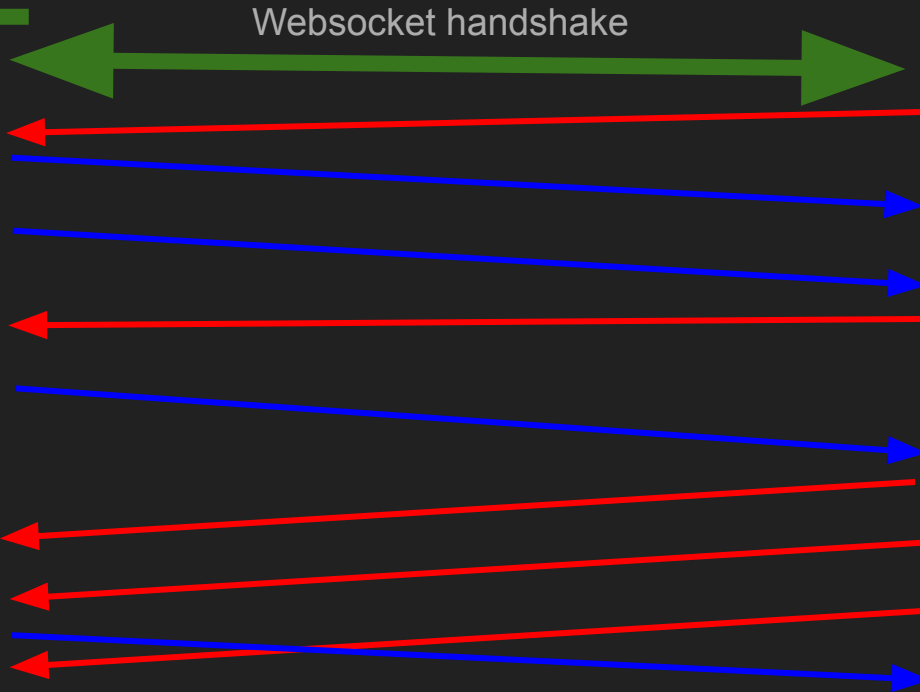
open

Websocket handshake

80

close

....



# WebSockets Handshake ws:// or wss://



# WebSocket Handshake

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

Client

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmerc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

Server

# WebSockets use cases

- Chatting
- Live Feed
- Multiplayer gaming
- Showing client progress/logging



# WebSockets Pros and Cons

## Pros

- Full-duplex (no polling)
- HTTP compatible
- Firewall friendly (standard)

## Cons

- Proxying is tricky
- L7 LB challenging (timeouts)
- Stateful, difficult to horizontally scale

# Do you have to use WebSockets?

- No
- Rule of thumb - do you absolutely need bidirectional communication?
- Long polling
- Server Sent Events

# Summary

- HTTP
- WebSockets
- WebSockets Handshake
- WebSockets use cases
- WebSockets Example
- WebSockets Pros and Cons

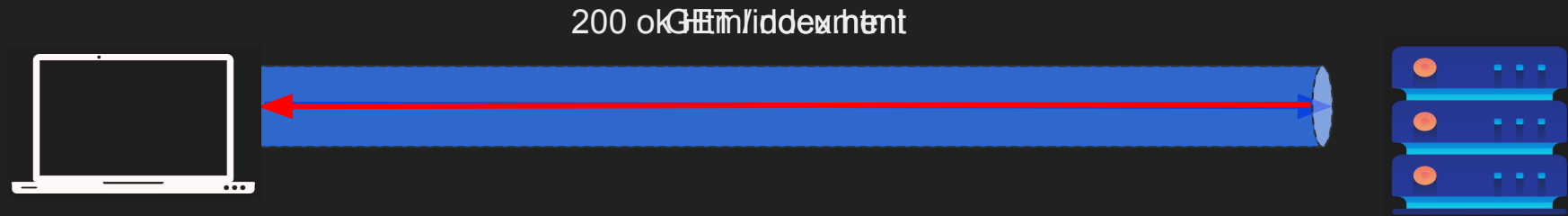
# HTTP/2

Improving HTTP/1.1

# Agenda

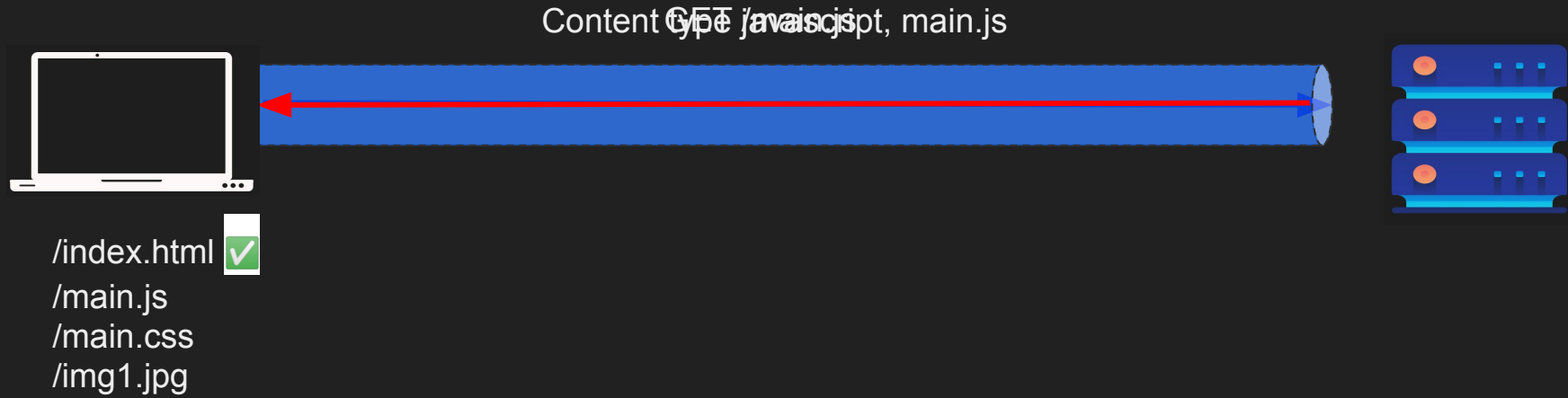
- How HTTP 1.1 Works?
- How HTTP/2 Works?
- HTTP/2 Pros & Cons
- H1 vs H2 performance test!

# HTTP 1.1

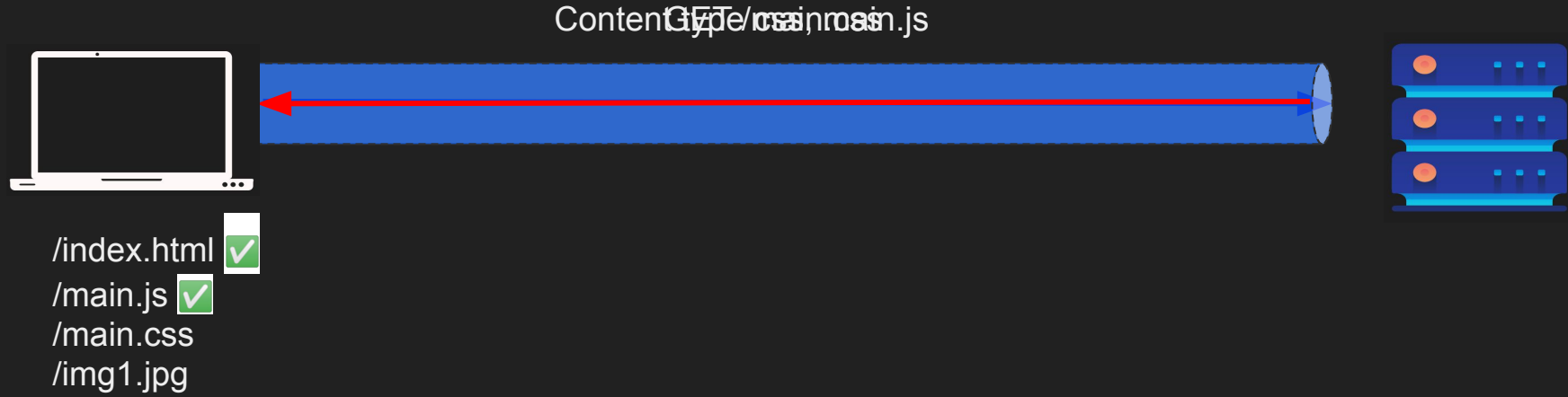


....

# HTTP 1.1



# HTTP 1.1



....

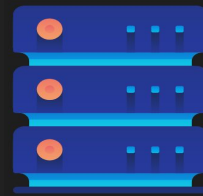
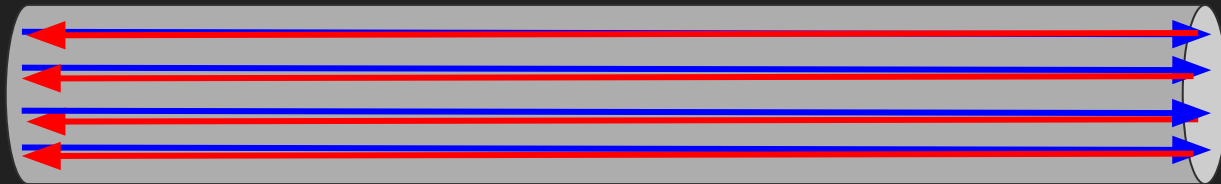
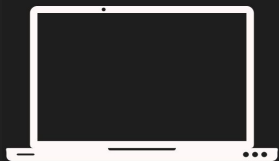


# HTTP 1.1 (Browsers use 6 connections)



# HTTP/2

GET /main.js (stream1)  
GET /main.css (stream3)  
GET /img1.jpg (stream5)  
GET /img2.jpg (stream7)



# HTTP/2 with Push



# HTTP/2 Pros

- Multiplexing over Single Connection (save resources)
- Compression (Headers & Data)
- Server Push
- Secure by default
- Protocol Negotiation during TLS (ALPN)

## HTTP/2 Cons

- TCP head of line blocking
- Server Push never picked up
- High CPU usage

# Summary

- How HTTP 1.1 Works?
- How HTTP/2 Works?
- HTTP/2 Pros & Cons
- H1 vs H2 performance test

# HTTP/3

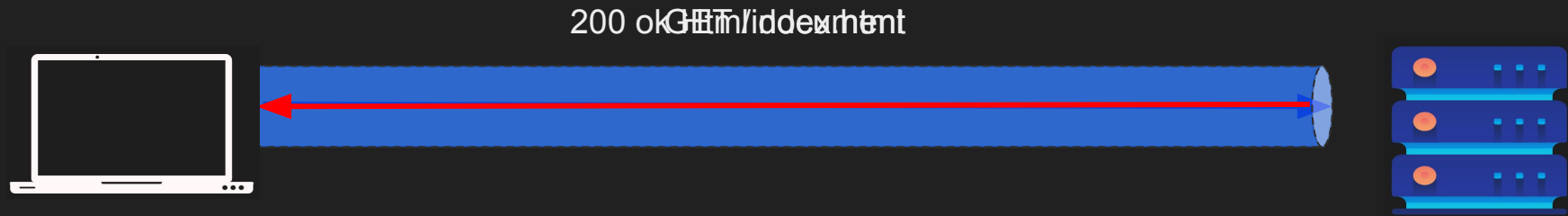
HTTP over QUIC  
Multiplexed streams

# Agenda

- How HTTP 1.1 Works?
- How HTTP/2 Works?
- HTTP/2 Cons
- How HTTP/3 & QUIC saves the day
- HTTP/3 & QUIC Pros & Cons

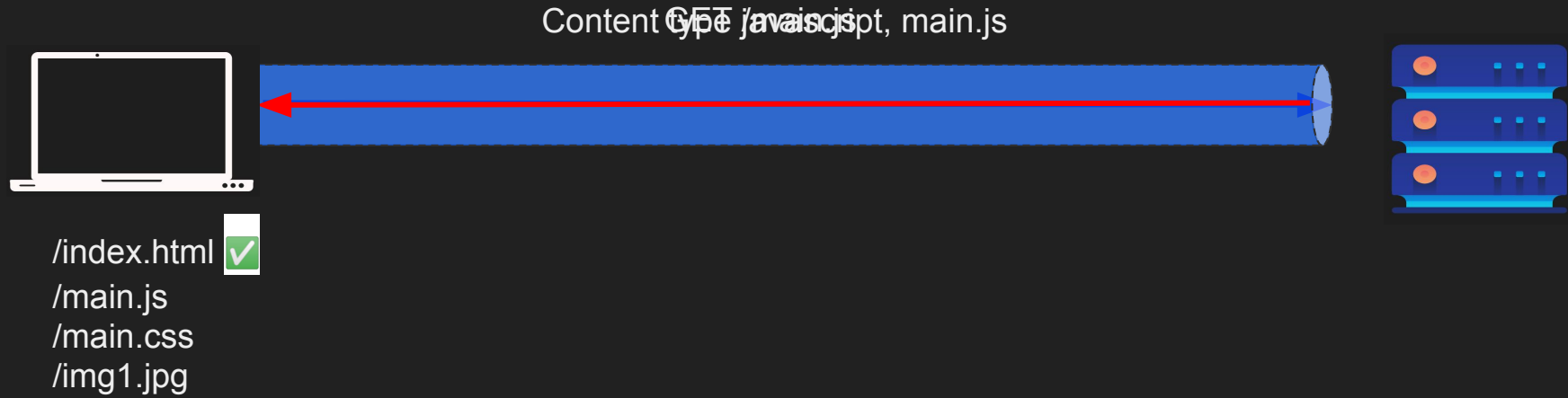


# HTTP 1.1

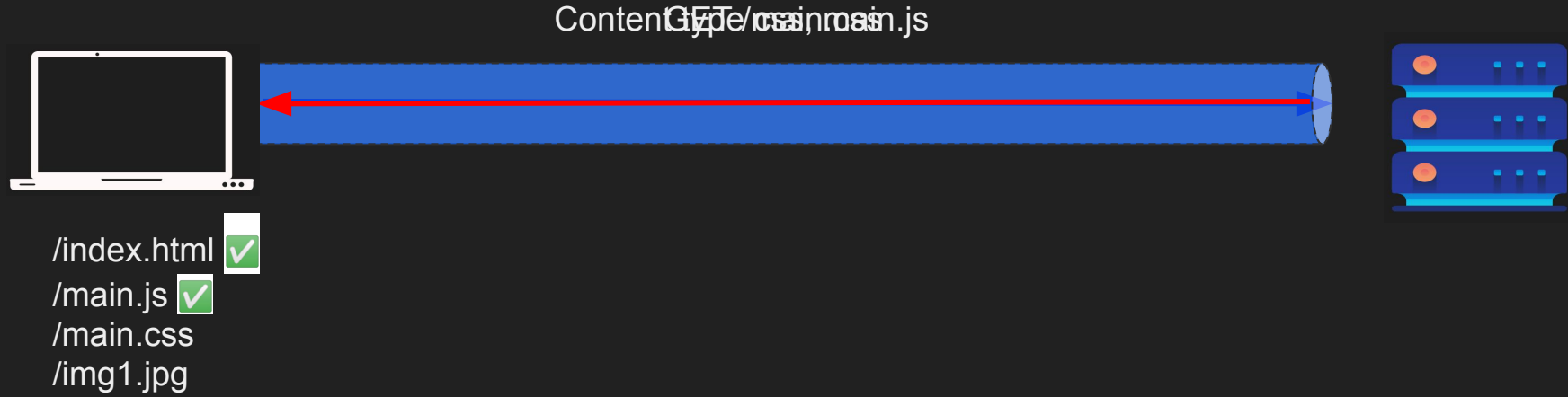


....

# HTTP 1.1



# HTTP 1.1

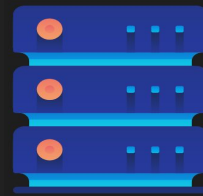
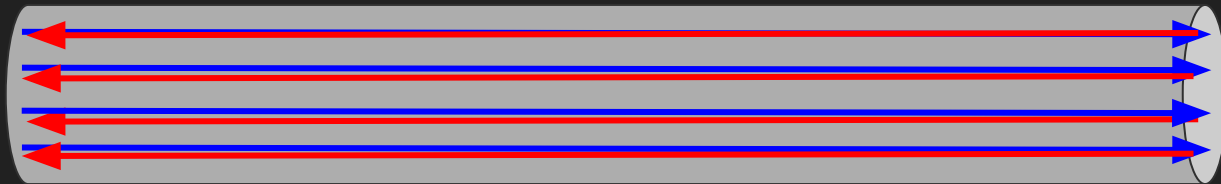
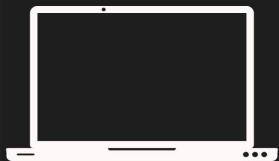


# HTTP 1.1 (Browsers use 6 connections)



# HTTP/2

GET /main.js (stream1)  
GET /main.css (stream3)  
GET /img1.jpg (stream5)  
GET /img2.jpg (stream7)

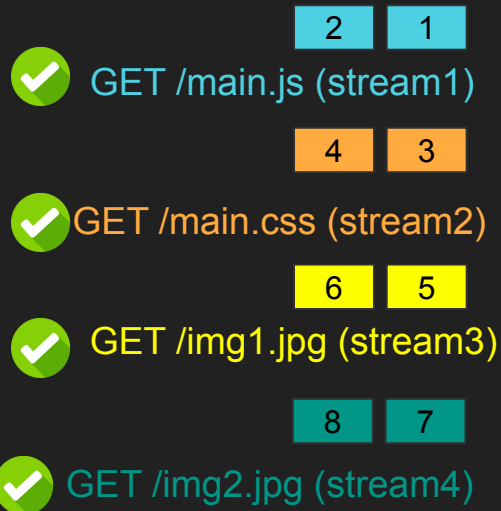
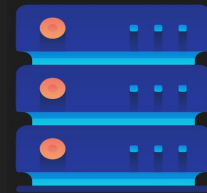
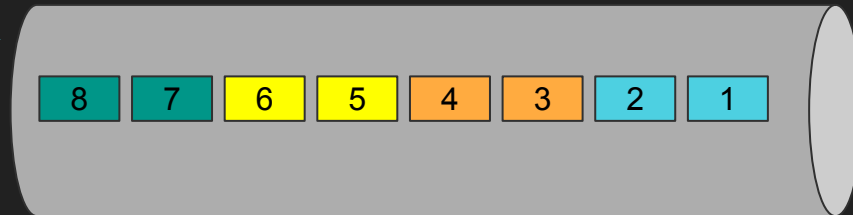


# TCP head of line blocking

- TCP segments must be delivered in order
- But streams don't have to
- Blocking requests

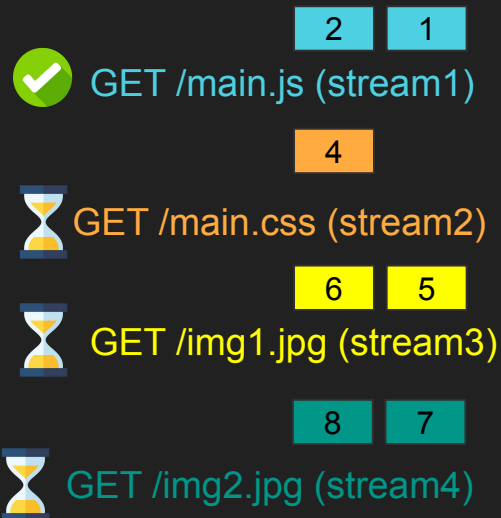
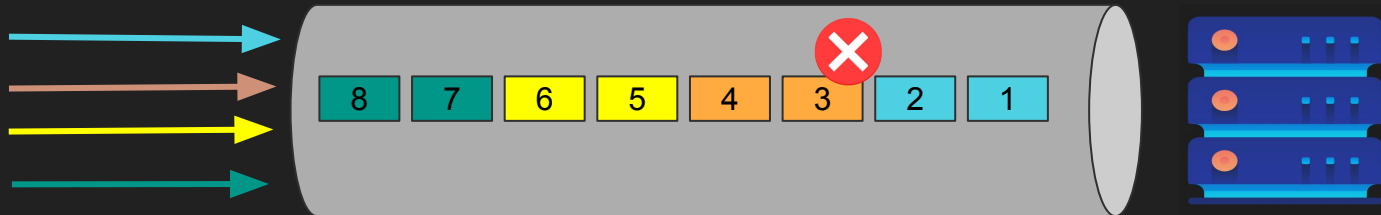
# HTTP/2 TCP HOL

GET /main.js (stream1)  
GET /main.css (stream2)  
GET /img1.jpg (stream3)  
GET /img2.jpg (stream4)



# HTTP/2 TCP HOL

GET /main.js (stream1)  
GET /main.css (stream2)  
GET /img1.jpg (stream3)  
GET /img2.jpg (stream4)



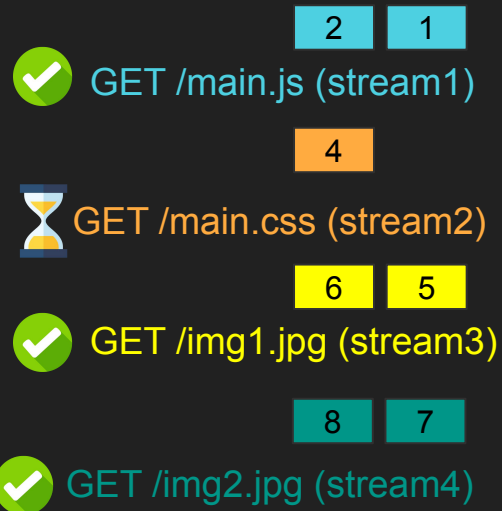
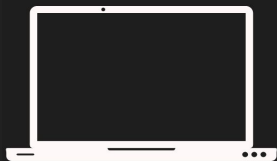
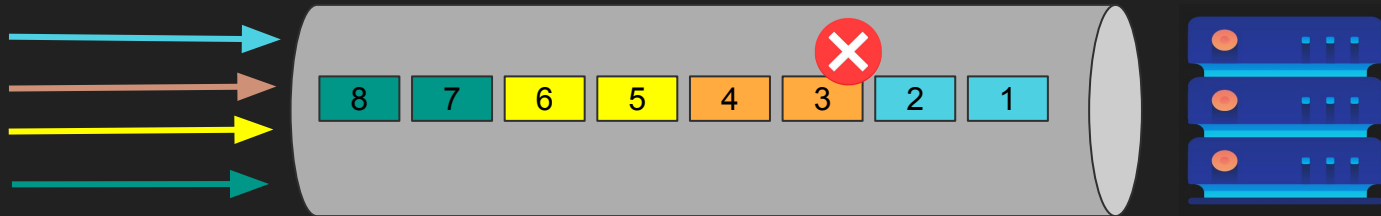


# HTTP/3 & QUIC

- HTTP/3 uses QUIC
- Like HTTP/2, QUIC has streams
- But QUIC use UDP instead
- Application decides the boundary

# HTTP/3 Streams

GET /main.js (stream1)  
GET /main.css (stream2)  
GET /img1.jpg (stream3)  
GET /img2.jpg (stream4)



# HTTP/3 & QUIC Pros

- QUIC has many other benefits
- Merges Connection setup + TLS in one handshake
- Has congestion control at stream level
- Connection migration (connectionID)
- Why not HTTP/2 over QUIC?
  - Header compression algorithm

## HTTP/3 & QUIC Cons

- Takes a lot of CPU (parsing logic)
- UDP could be blocked
- IP Fragmentations is the enemy

# Summary

- How HTTP 1.1 Works?
- How HTTP/2 Works?
- HTTP/2 Cons
- How HTTP/3 & QUIC saves the day
- HTTP/3 & QUIC Pros & Cons

# gRPC

Taking HTTP/2 to the next level

# Agenda

- Motivation
  - Client/Server Communication
  - Problem with Client Libraries
  - Why gRPC was invented?
- gRPC
  - Unary gRPC
  - Server Streaming
  - Client Streaming
  - Bidirectional

- Coding!
- gRPC Pros & Cons
- I'm tired of new protocols
- Summary

# Client Server Communication

- SOAP, REST, GraphQL
- SSE, WebSockets
- Raw TCP



# The Problem with Client Libraries

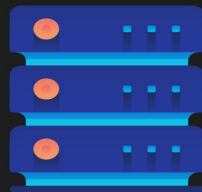
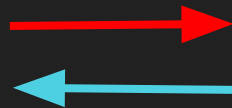
- Any communication protocol needs client library for the language of choice
  - SOAP Library
  - HTTP Client Library
- Hard to maintain and patch client libraries
  - HTTP/1.1 HTTP/2, new features, security etc.

# Why gRPC was invented?

- Client Library: One library for popular languages
- Protocol: HTTP/2 (hidden implementation)
- Message Format: Protocol buffers as format

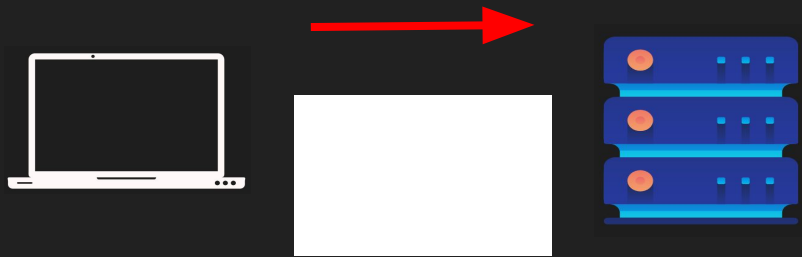
# gRPC modes

- Unary RPC
- Server streaming RPC
- Client streaming RPC
- Bidirectional streaming RPC



# gRPC modes

- Unary RPC
- Server streaming RPC
- Client streaming RPC
- Bidirectional streaming RPC



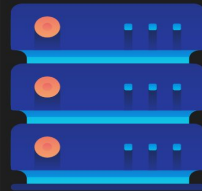
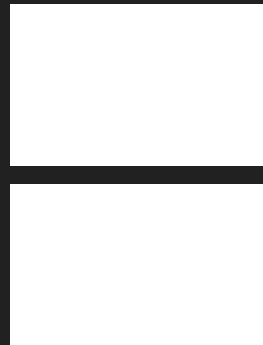
# gRPC modes

- Unary RPC
- Server streaming RPC
- Client streaming RPC
- Bidirectional streaming RPC



# gRPC modes

- Unary RPC
- Server streaming RPC
- Client streaming RPC
- Bidirectional streaming RPC



# Coding time!

- Todo application (server, client) with gRPC
- `createTodo ( )`
- `readTodos ( ) //synchronous`
- `readTodos ( ) //server stream`

# gRPC Pros & Cons

## Pros

- Fast & Compact
- One Client Library
- Progress Feedback (upload)
- Cancel Request (H2)
- H2/Protobuf

## Cons

- Schema
- Thick Client
- Proxies
- Error handling
- No native browser support
- Timeouts (pub/sub)



## Can I write my own protocol too?

- Yes, you can, Spotify did (Hermes) but guess what
- Only you will be using it so...
- Spotify moved to gRPC not because of limitation of Hermes but because they are isolated.

# Summary

- Motivation
  - Client/Server Communication
  - Problem with Client Libraries
  - Why gRPC was invented?
- gRPC
  - Unary gRPC
  - Server Streaming
  - Client Streaming
  - Bidirectional

- Coding!
- gRPC Pros & Cons
- I'm tired of new protocols
- Summary

# WebRTC

Realtime communication on the web

# Agenda

- WebRTC Overview
- WebRTC Demystified
  - NAT, STUN, TURN, ICE, SDP, Signaling the SDP
- Demo
- WebRTC Pros & Cons
- More WebRTC content beyond this content

# WebRTC Overview

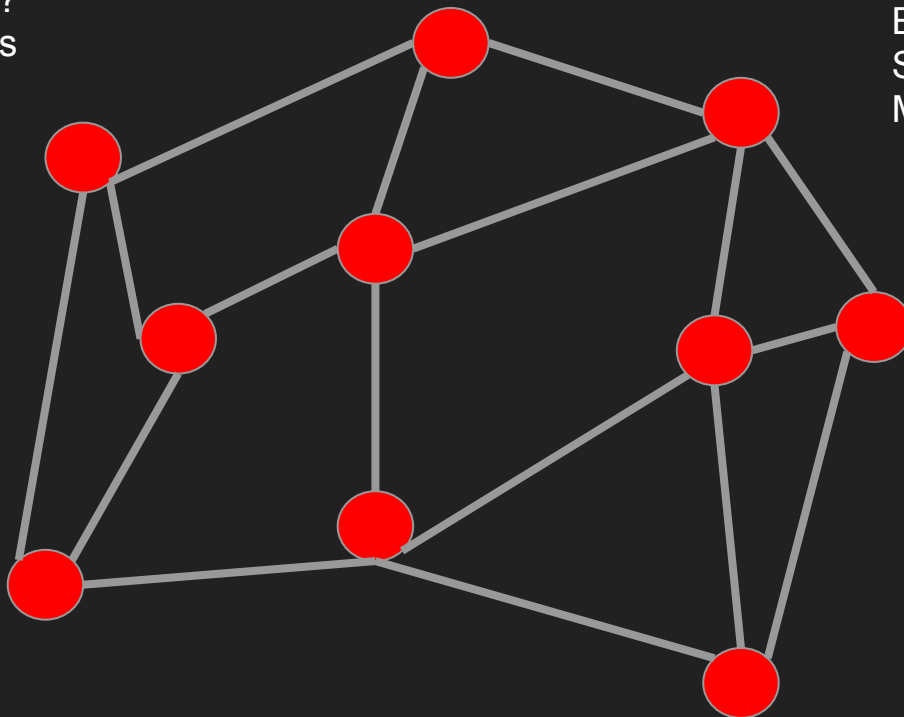
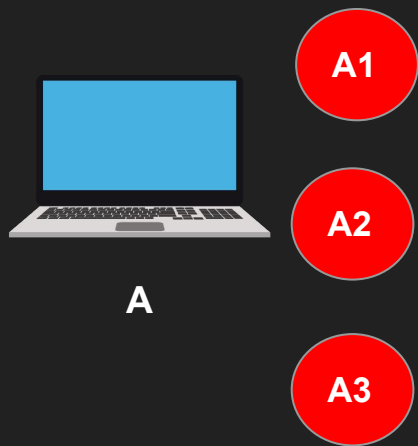
- Stands for Web Real-Time Communication
- Find a peer to peer path to exchange video and audio in an efficient and low latency manner
- Standardized API
- Enables rich communications browsers, mobile, IOT devices

# WebRTC Overview

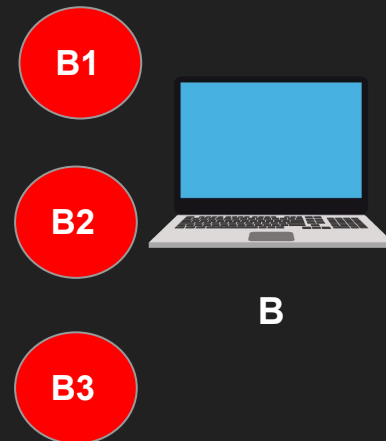
- A wants to connect to B
- A finds out all possible ways the public can connect to it
- B finds out all possible ways the public can connect to it
- A and B signal this session information via other means
  - WhatsApp, QR, Tweet, WebSockets, HTTP Fetch..
- A connects to B via the most optimal path
- A & B also exchanges their supported media and security

# WebRTC Overview

How can people reach me?  
(A1, A2, A3) are candidates  
Security Parameters  
Media options



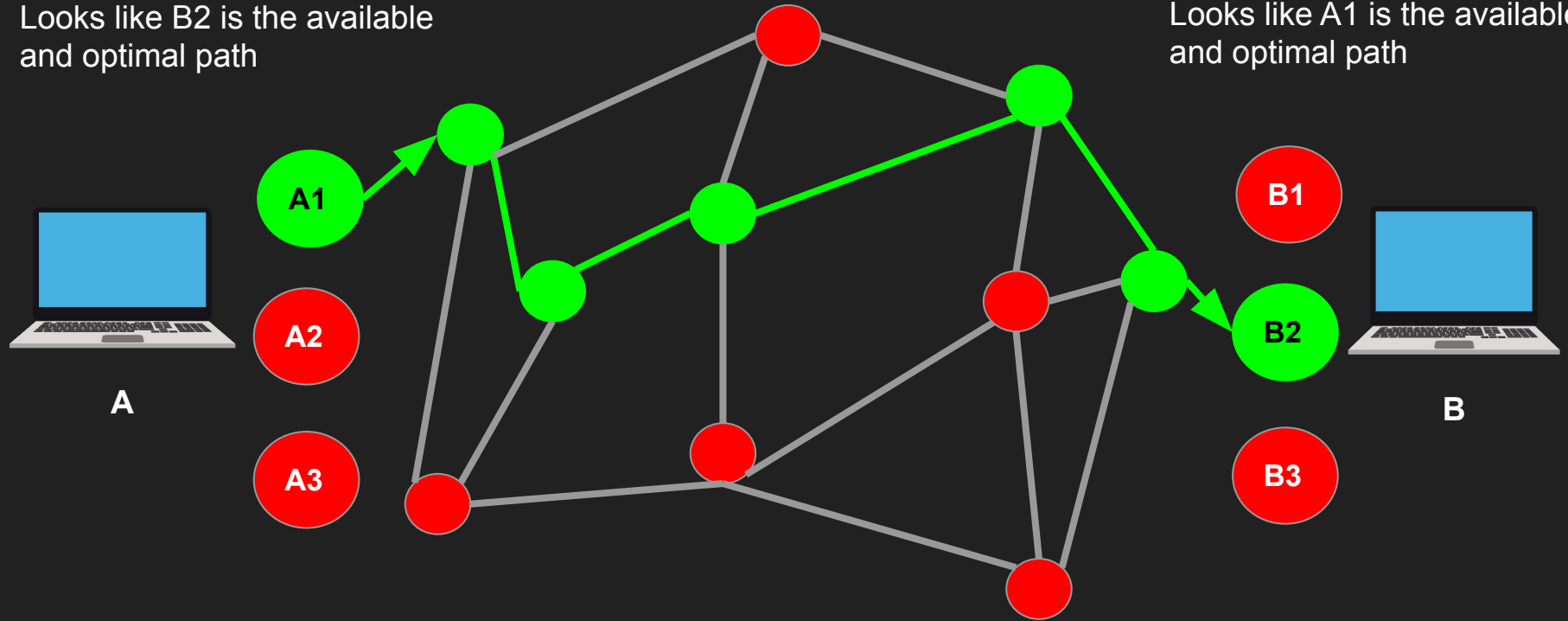
How can people reach me?  
B1, B2, B3 are candidates  
Security Parameters  
Media options



# WebRTC Overview

Looks like B2 is the available  
and optimal path

Looks like A1 is the available  
and optimal path





# WebRTC Demystified

- NAT
- STUN, TURN
- ICE
- SDP
- Signaling the SDP

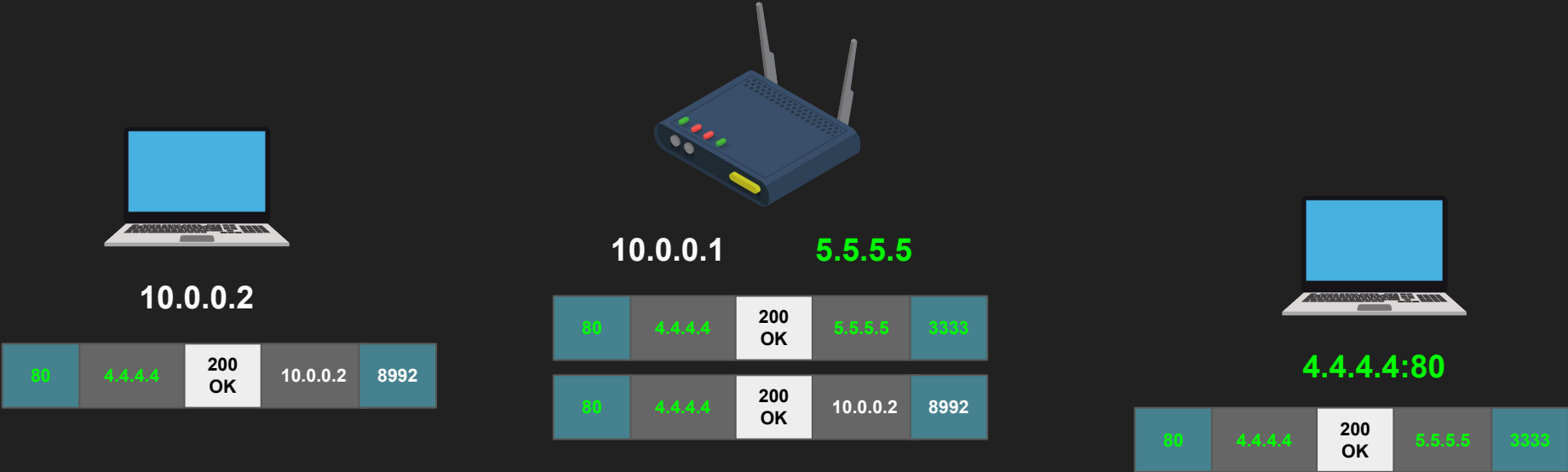
# Network Address Translation



Internal IP	Internal Port	External IP	Ext. Port	Dest IP	Dest Port
10.0.0.2	8992	5.5.5.5	3333	4.4.4.4	80



# Network Address Translation



Internal IP	Internal Port	Ext IP	Ext Port	Dest IP	Dest Port
10.0.0.2	8992	5.5.5.5	3333	4.4.4.4	80



# NAT Translations Method

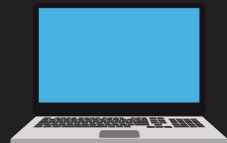
- One to One NAT (Full-cone NAT)
- Address restricted NAT
- Port restricted NAT
- Symmetric NAT

# One to One NAT (Full cone NAT)

- Packets to external IP:port on the router always maps to internal IP:port without exceptions

Internal IP	Internal Port	Ext IP	Ext Port	Dest IP	Dest Port
10.0.0.2	8992	5.5.5.5	3333	4.4.4.4	80

# One to One NAT



10.0.0.2



10.0.0.1

5.5.5.5



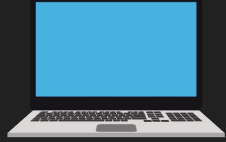
Internal IP	Internal Port	Ext IP	Ext Port	Dest IP	Dest Port
10.0.0.2	8992	5.5.5.5	3333	4.4.4.4	80
10.0.0.2	9999	5.5.5.5	4444	3.3.3.3	80

# Address Restricted NAT

- Packets to external IP:port on the router always maps to internal IP:port as long as source address from packet matches the table (regardless of port)
- Allow if we communicated with this host before

Internal IP	Internal Port	Ext IP	Ext Port	Dest IP	Dest Port
10.0.0.2	8992	5.5.5.5	3333	4.4.4.4	80

# Address Restricted NAT



10.0.0.2



10.0.0.1

5.5.5.5



80	4.4.4.4	200 OK	5.5.5.5	3333
80	3.3.3.3	200 OK	5.5.5.5	3333
8080	3.3.3.3	200 OK	5.5.5.5	3333
80	9.9.1.2	200 OK	5.5.5.5	3333

Internal IP	Internal Port	Ext IP	Ext Port	Dest IP	Dest Port
10.0.0.2	8992	5.5.5.5	3333	4.4.4.4	80
10.0.0.2	9999	5.5.5.5	4444	3.3.3.3	80

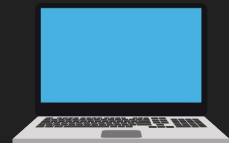


# Port Restricted NAT

- Packets to external IP:port on the router always maps to internal IP:port as long as source address and port from packet matches the table
- Allow if we communicated with this host:port before

Internal IP	Internal Port	Ext IP	Ext Port	Dest IP	Dest Port
10.0.0.2	8992	5.5.5.5	3333	4.4.4.4	80

# Port Restricted NAT



10.0.0.2



10.0.0.1

5.5.5.5



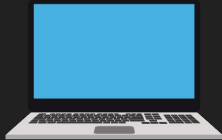
Internal IP	Internal Port	Ext IP	Ext Port	Dest IP	Dest Port
10.0.0.2	8992	5.5.5.5	3333	4.4.4.4	80
10.0.0.2	9999	5.5.5.5	4444	3.3.3.3	80
10.0.0.2	8888	5.5.5.5	2222	3.3.3.3	8080

# Symmetric NAT

- Packets to external IP:port on the router always maps to internal IP:port as long as source address and port from packet matches the table
- Only Allow if the full pair match

Internal IP	Internal Port	Ext IP	Ext Port	Dest IP	Dest Port
10.0.0.2	8992	5.5.5.5	3333	4.4.4.4	80

# Symmetric NAT



10.0.0.2



10.0.0.1

5.5.5.5



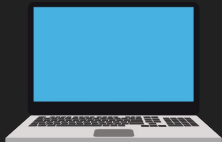
80	4.4.4.4	200 OK	5.5.5.5	3333
22	3.3.3.3	200 OK	5.5.5.5	3333
8080	3.3.3.3	200 OK	5.5.5.5	3333
23	9.9.1.2	200 OK	5.5.5.5	3333

Internal IP	Internal Port	Ext IP	Ext Port	Dest IP	Dest Port
10.0.0.2	8992	5.5.5.5	3333	4.4.4.4	80
10.0.0.2	9999	5.5.5.5	4444	3.3.3.3	80
10.0.0.2	8888	5.5.5.5	2222	3.3.3.3	8080

# STUN

- Session Traversal Utilities for NAT
- Tell me my public ip address/port through NAT
- Works for Full-cone, Port/Address restricted NAT
- Doesn't work for symmetric NAT
- STUN server port 3478, 5349 for TLS
- Cheap to maintain

# STUN Request



10.0.0.2

8992	10.0.0.2	STN	9.9.9.9	3478
------	----------	-----	---------	------

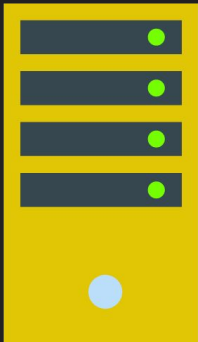


10.0.0.1

5.5.5.5

8992	10.0.0.2	STN	9.9.9.9	3478
------	----------	-----	---------	------

3333	5.5.5.5	STN	9.9.9.9	3478
------	---------	-----	---------	------

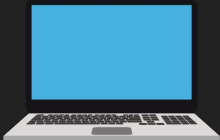


9.9.9.9:3478  
(STUN Sever)

3333	5.5.5.5	STN	9.9.9.9	3478
------	---------	-----	---------	------

Internal IP	Internal Port	Ext IP	Ext Port	Dest IP	Dest Port
10.0.0.2	8992	5.5.5.5	3333	9.9.9.9	3478

# STUN Response



10.0.0.2

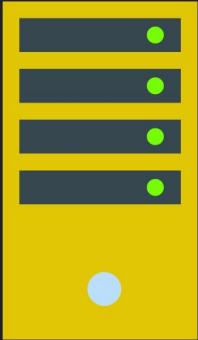
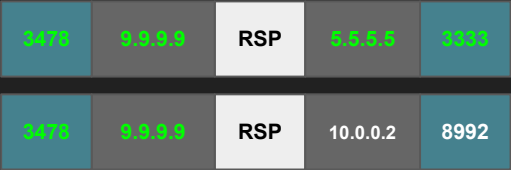


You are  
5.5.5.5:3333



10.0.0.1

5.5.5.5



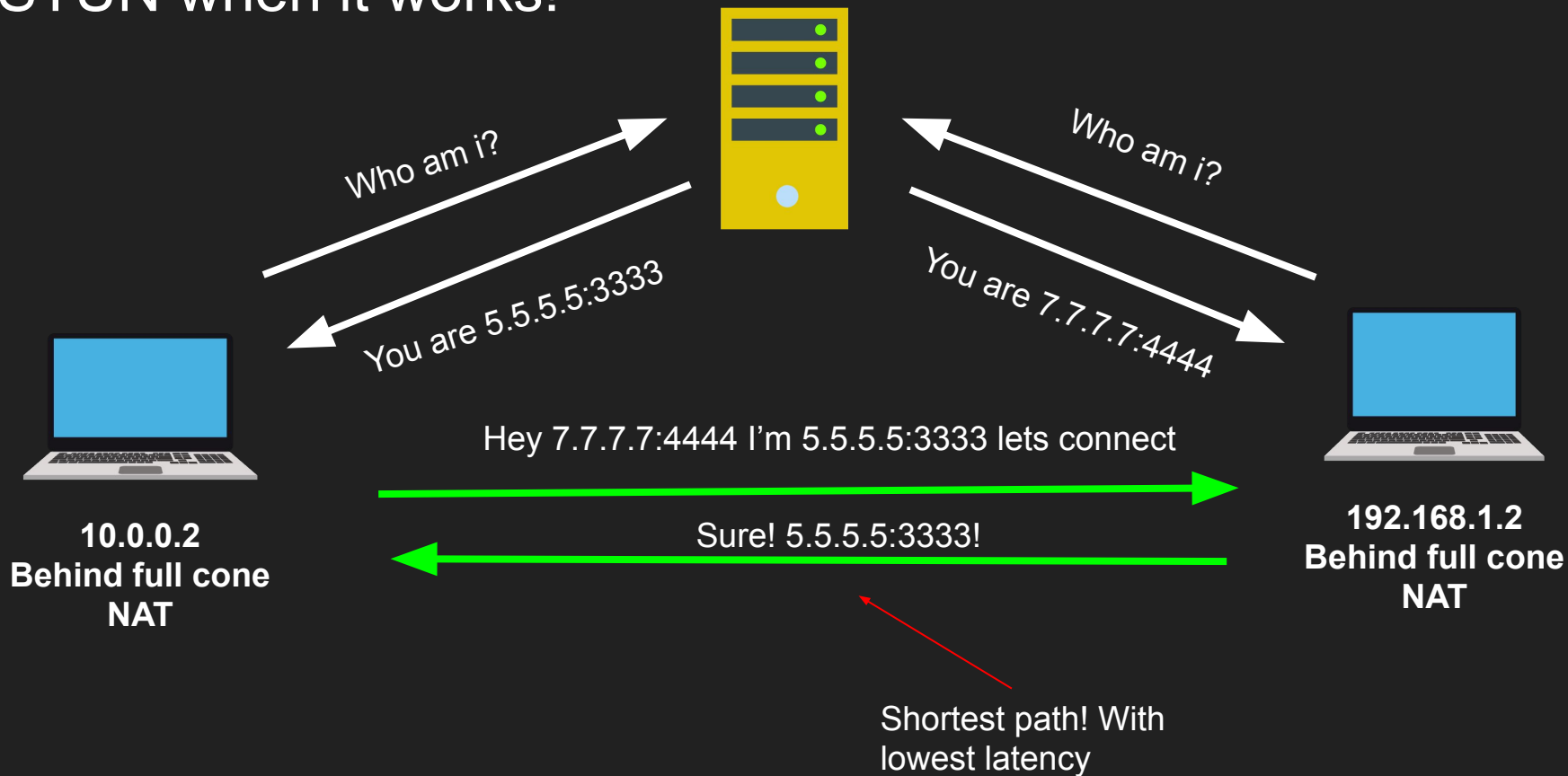
9.9.9.9:3478  
(STUN Sever)

You are  
5.5.5.5:3333



Internal IP	Internal Port	Ext IP	Ext Port	Dest IP	Dest Port
10.0.0.2	8992	5.5.5.5	3333	9.9.9.9	3478

# STUN when it works!





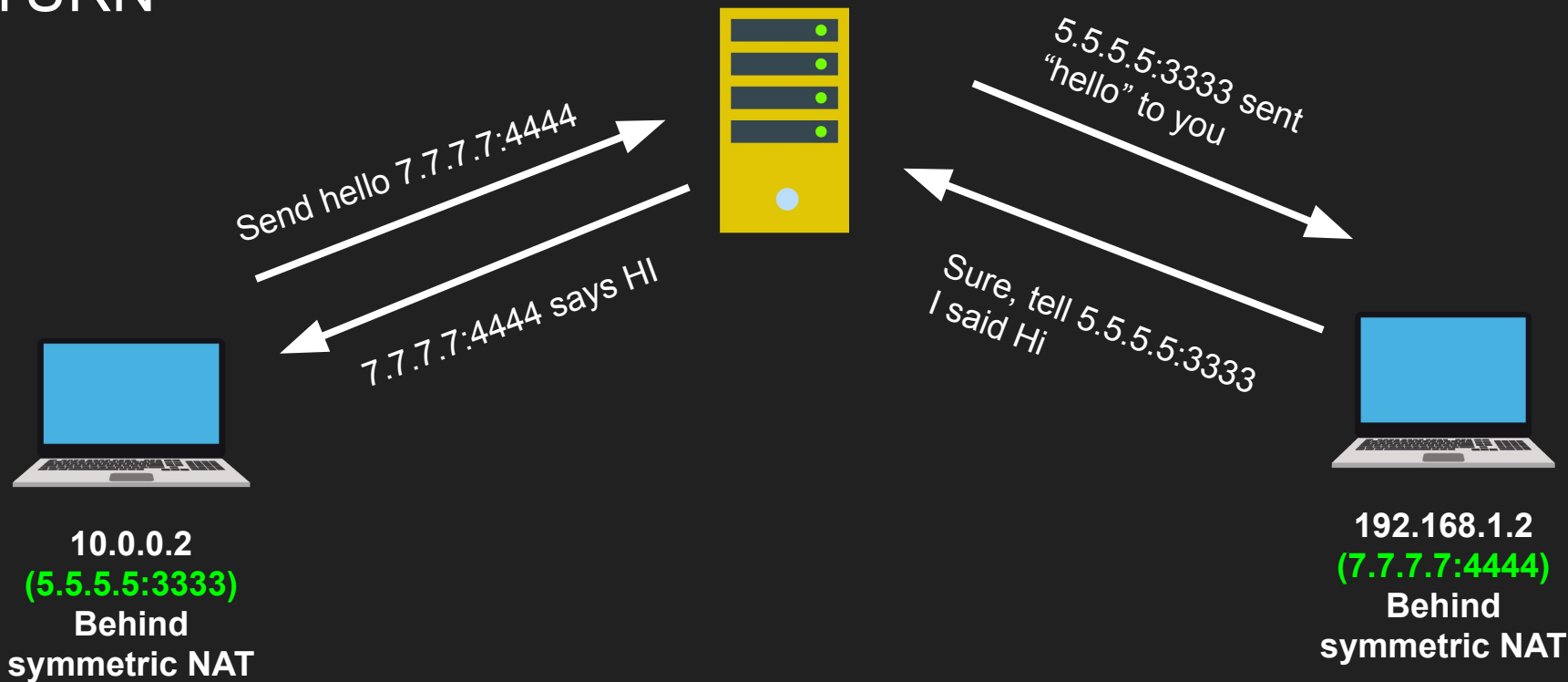
# STUN when it doesn't!



# TURN

- Traversal Using Relays around NAT
- In case of Symmetric NAT we use TURN
- It's just a server that relays packets
- TURN default server port 3478, 5349 for TLS
- Expensive to maintain and run

# TURN



# ICE

- Interactive Connectivity Establishment
- ICE collects all available candidates (local IP addresses, reflexive addresses – STUN ones and relayed addresses – TURN ones)
- Called ice candidates
- All the collected addresses are then sent to the remote peer via SDP

# SDP

- Session Description Protocol
- A format that describes ice candidates, networking options, media options, security options and other stuff
- Not really a protocol its a format
- Most important concept in WebRTC
- The goal is to take the SDP generated by a user and send it “somehow” to the other party

# SDP Example

v=0

o=- 9148204791819634656 3 IN IP4 127.0.0.1

s=-

t=0 0

a=group:BUNDLE audio video data

a=msid-semantic: WMS kyaiqbOs7S2h3EoSHabQ3JIBqZ67cFqZmWFN

m=audio 50853 RTP/SAVPF 111 103 104 0 8 107 106 105 13 126

c=IN IP4 192.168.1.64

a=rtcp:50853 IN IP4 192.168.1.64

a=candidate:3460887983 1 udp 2113937151 192.168.1.64 50853 typ host generation 0

a=candidate:3460887983 2 udp 2113937151 192.168.1.64 50853 typ host generation 0

...

# Signaling

- SDP Signaling
- Send the SDP that we just generated somehow to the other party we wish to communicate with
- Signaling can be done via a tweet, QR code, Whatsapp, WebSockets, HTTP request DOESN'T MATTER! Just get that large string to the other party

# WebRTC Demystified

1. A wants to connect to B
2. A creates an “offer”, it finds all ICE candidates, security options, audio/video options and generates SDP, the offer is basically the SDP
3. A signals the offer somehow to B (whatsapp)
4. B creates the “answer” after setting A’s offer
5. B signals the “answer” to A
6. Connection is created
- 7.



# WebRTC Demo

- We will connect two browsers (Browser A & Browser B)
- A will create an offer (sdp) and set it as local description
- B will get the offer and set it as remote description
- B creates an answer sets it as its local description and signal the answer (sdp) to A
- A sets the answer as its remote description
- Connection established, exchange data channel

# WebRTC Pros & Cons

- Pros
  - P2p is great ! low latency for high bandwidth content
  - Standardized API I don't have to build my own
- Cons
  - Maintaining STUN & TURN servers
  - Peer 2 Peer falls apart in case of multiple participants (discord case)

# More WebRTC stuff!

So more to discuss beyond this content

# Media API

- `getUserMedia` to access microphone, video camera
- `RTCPConnection.addTrack(stream)`
- <https://www.html5rocks.com/en/tutorials/webrtc/basics/>

## onIceCandidate and addIceCandidate

- To maintain the connection as new candidates come and go
- onIceCandidate tells user there is a new candidate after the SDP has already been created
- The candidate is signaled and sent to the other party
- The other party uses addIceCandidate to add it to its SDP

# Set custom TURN and STUN Servers

```
const iceConfiguration = {  
  iceServers: [{  
    urls: 'turn:turnserver.company.com:3478',  
    username: 'optional-username',  
    credentials: 'auth-token'},  
    { urls: "stun:stun.services.mozilla.com",  
      username: "test@mozilla.com",  
      credential: "webrtcdemo"}  
  ]  
}  
  
const pc = new RTCPeerConnection(configuration);
```

# Create your own STUN & TURN server

- COTURN open source project
- <https://github.com/coturn/coturn>

# Public STUN servers

- `stun1.1.google.com:19302`
- `stun2.1.google.com:19302`
- `stun3.1.google.com:19302`
- `stun4.1.google.com:19302`
- `stun.stunprotocol.org:3478`



# Many ways to HTTPS

And how each affect latency

# HTTPS Communication Basics

- Establish Connection
- Establish Encryption
- Send Data
- Close Connection (when absolutely done)

# HTTPs communications

- HTTPS over TCP with TLS 1.2
- HTTPS over TCP with TLS 1.3
- HTTPS over QUIC
- HTTPS over TCP fast Open
- HTTPS over TCP with TLS 1.3 0RTT
- HTTPS over QUIC with 0RTT

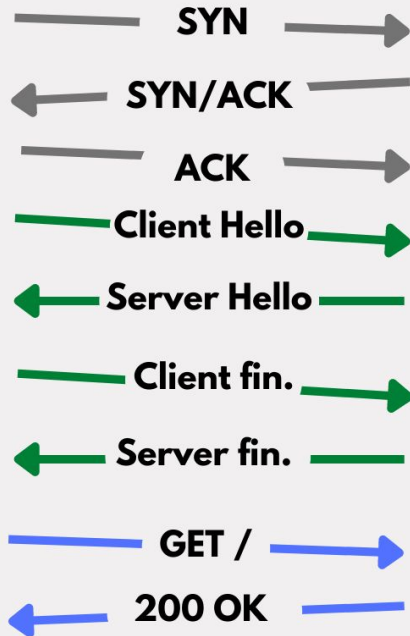
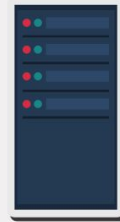
# HTTPS over TCP With TLS 1.2

# HTTPS over TCP with TLS 1.2

Client

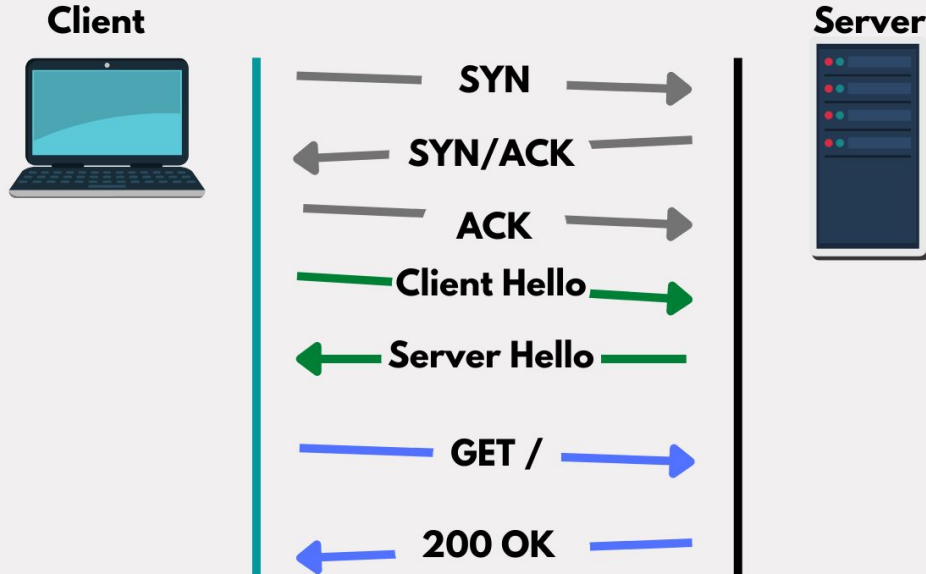


Server



# HTTPS over TCP With TLS 1.3

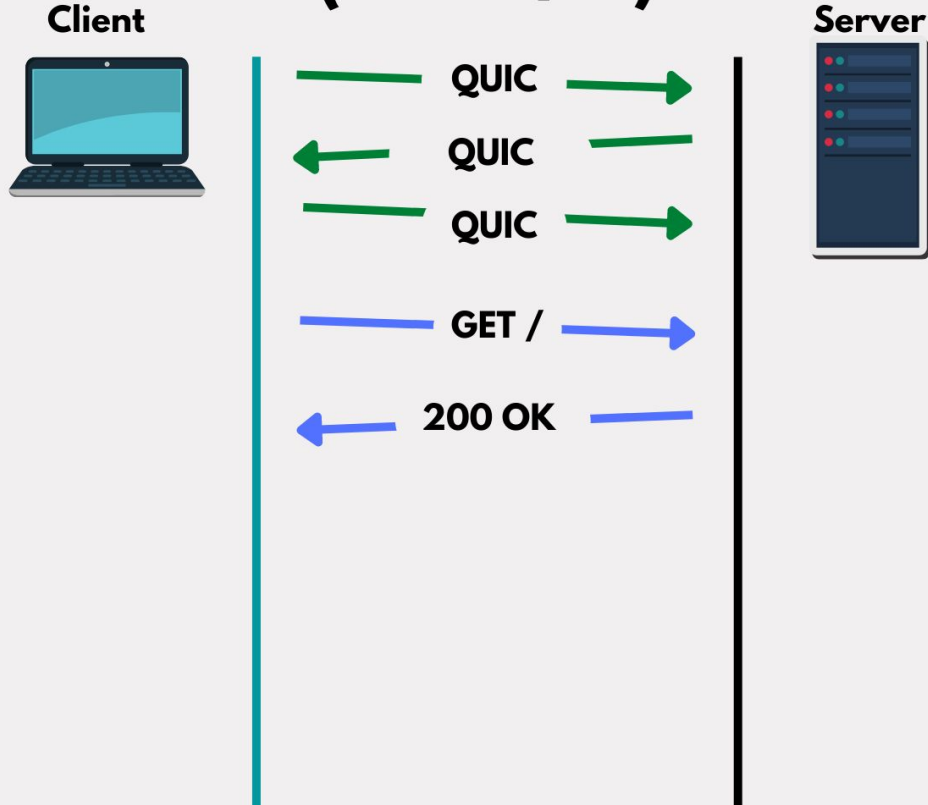
# HTTPS over TCP with TLS 1.3



# HTTPS over QUIC (HTTP/3)

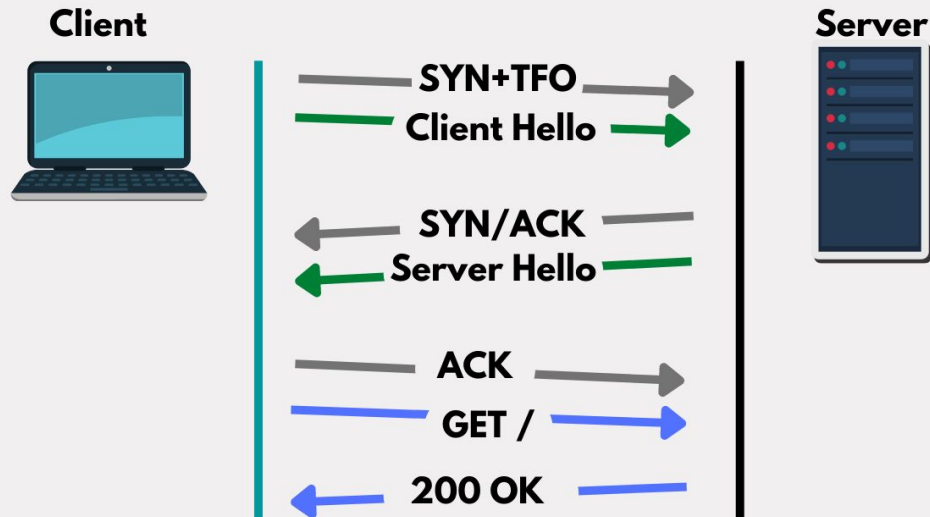


# HTTPS over QUIC (HTTP/3)



# HTTPS over TFO with TLS 1.3

# HTTPS over TCP Fast Open with TLS 1.3 (theory)



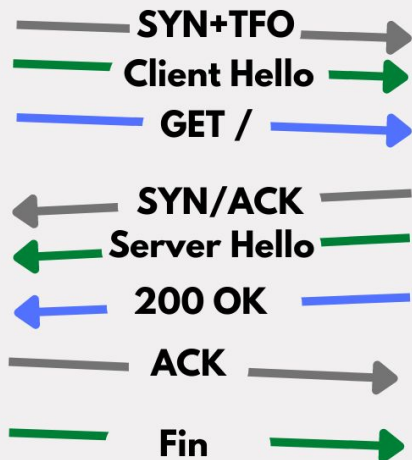
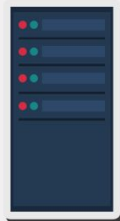
# HTTPS over TFO with TLS 1.3 0RTT

# HTTPS over TCP Fast Open with TLS 1.3 ORTT (theory)

Client

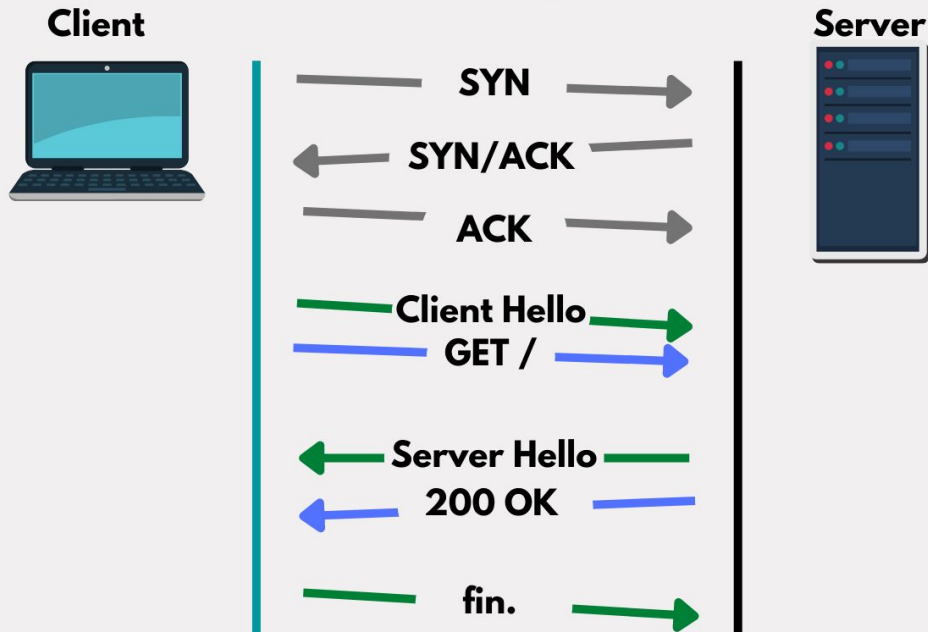


Server



# HTTPS over TCP with TLS1.3 0RTT

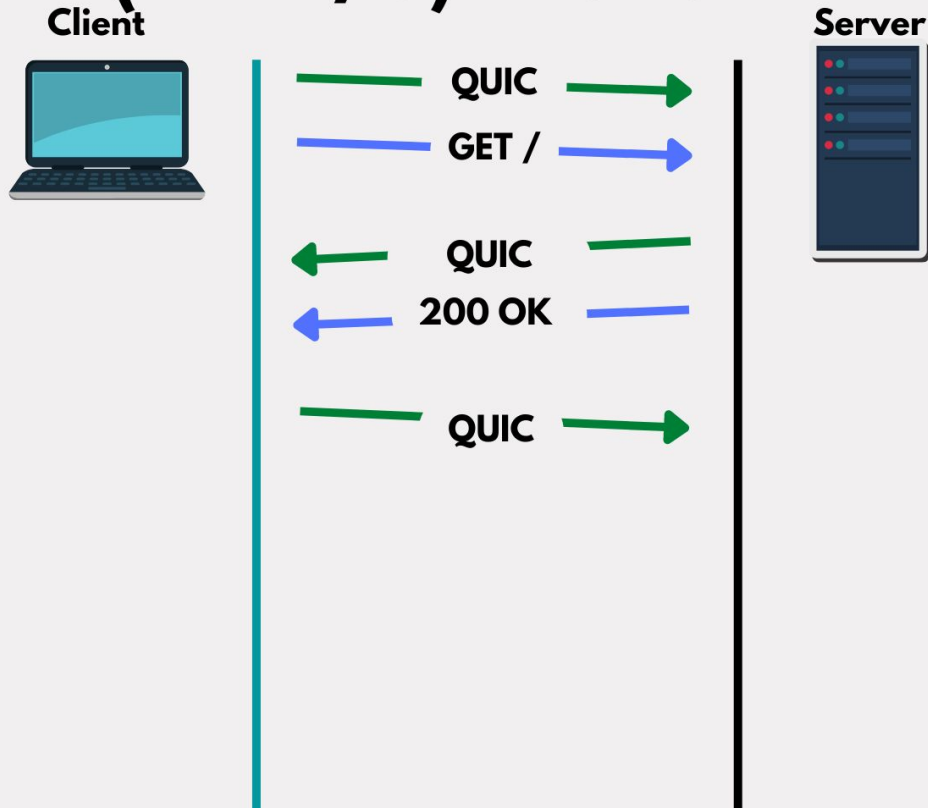
# HTTPS over TCP with TLS 1.3 ORTT



# HTTPS over QUIC 0RTT



# HTTPS over QUIC (HTTP/3) with ORTT



# Backend Execution Patterns

How Backends accept, dispatch and execute requests

# Process vs Thread

What is the difference?

# What is a Process?

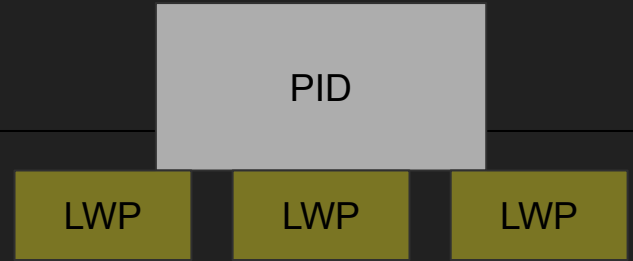
- A set of instructions
- Has an isolated memory
- Has a PID
- Scheduled in the CPU



PID

# What is a Thread?

- Light weight Process (LWP)
- A set of instructions
- Shares memory with parent process
- Has a ID
- Scheduled in the CPU



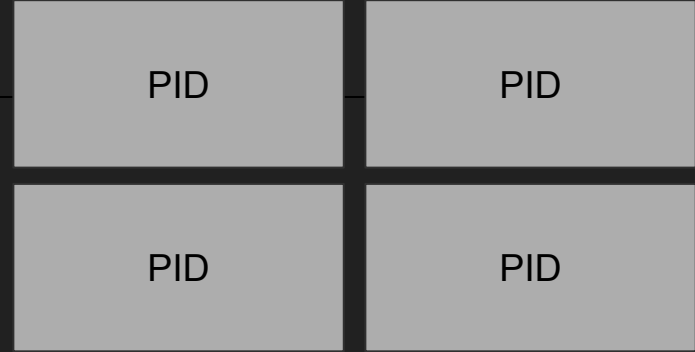
# Single Threaded Process

PID

- One Process with a single thread
- Simple
- Examples NodeJS

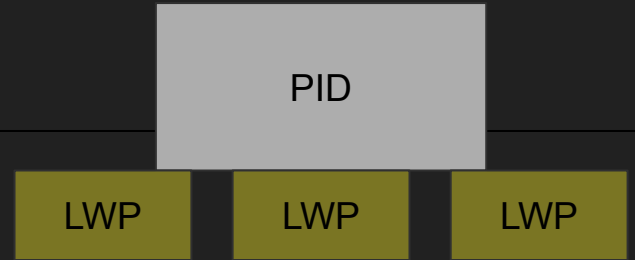
# Multi-Processes

- App has multiple processes
- Each has its own Memory
- Examples NGINX/Postgres
- Take advantage of multi-cores
- More memory but isolated
- Redis backup routine (COW)



# Multi-Threaded

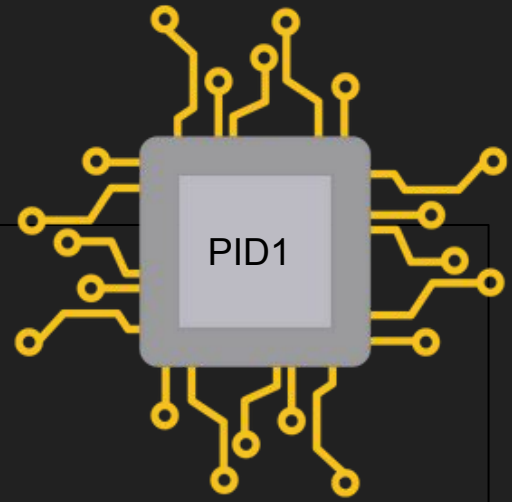
- One Process, multiple threads
- Shared Memory (compete)
- Take advantage of multi-cores
- Require less memory
- Race conditions
- Locks and Latches (SQL Server)
- Examples Apache, Envoy





# How many is too many?

- Too many processes/threads
- CPU context switch
- Multiple Cores help
- Rule of thumb -> # Cores = # Processes



# How Connections are Established

SYN/Accept Queues

# Connection Establishment

- TCP Three way handshake
- SYN/SYN-ACK/ACK
- But what happens on the backend?



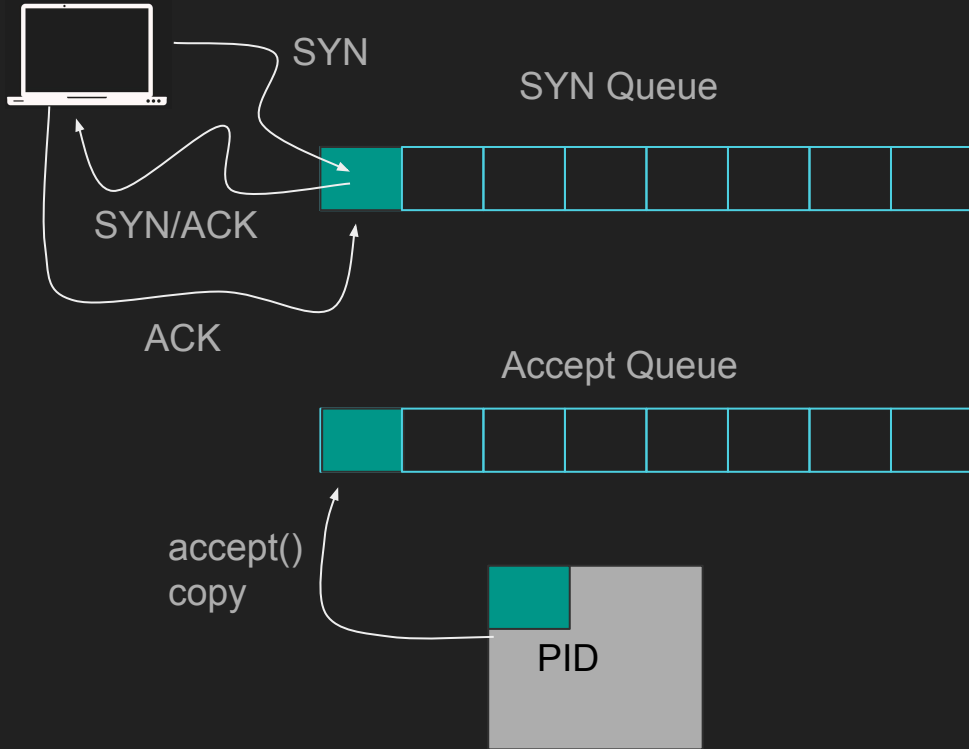
# Connection Establishment

- Server Listens on an address:port
- Client connects
- Kernel does the handshake creating a connection
- Backend process “Accepts” the connection

# Connection Establishment

- Kernel creates a socket & two queues SYN and Accept
- Client sends a SYN
- Kernel adds to SYN queue, replies with SYN/ACK
- Client replies with ACK
- Kernel finish the connection
- Kernel removes SYN from SYN queue
- Kernel adds full connection to Accept queue
- Backend accepts a connection, removed from accept queue
- A file descriptor is created for the connection

# Connection Establishment



# Problems with accepting connections

- Backend doesn't accept fast enough
- Clients who don't ACK
- Small backlog

# Reading and Sending Data

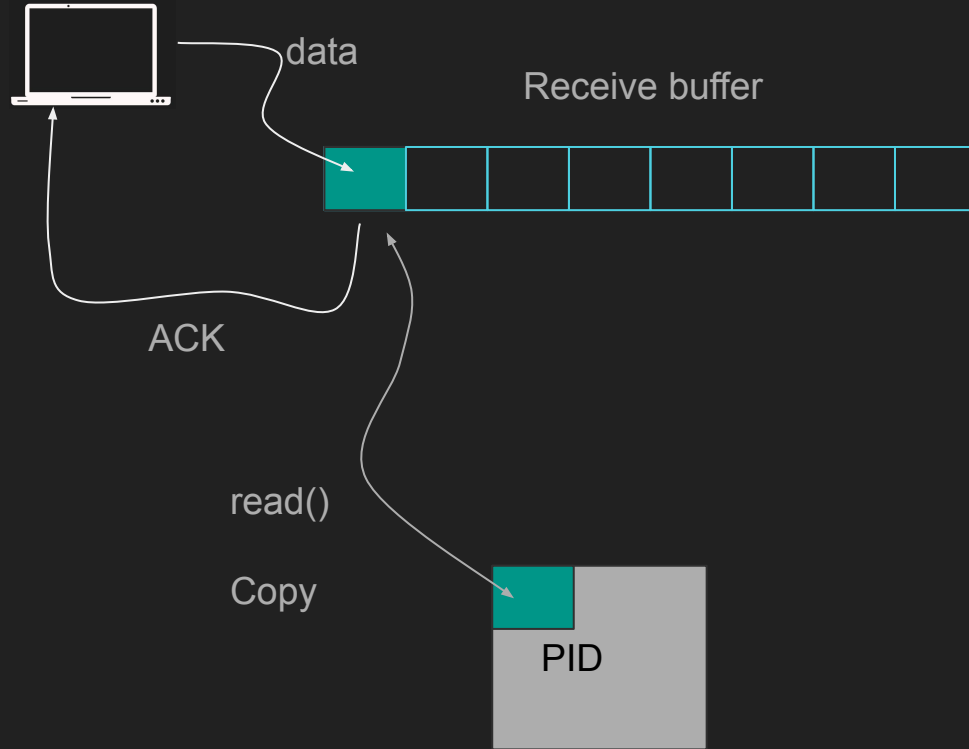
Receive vs Send buffers



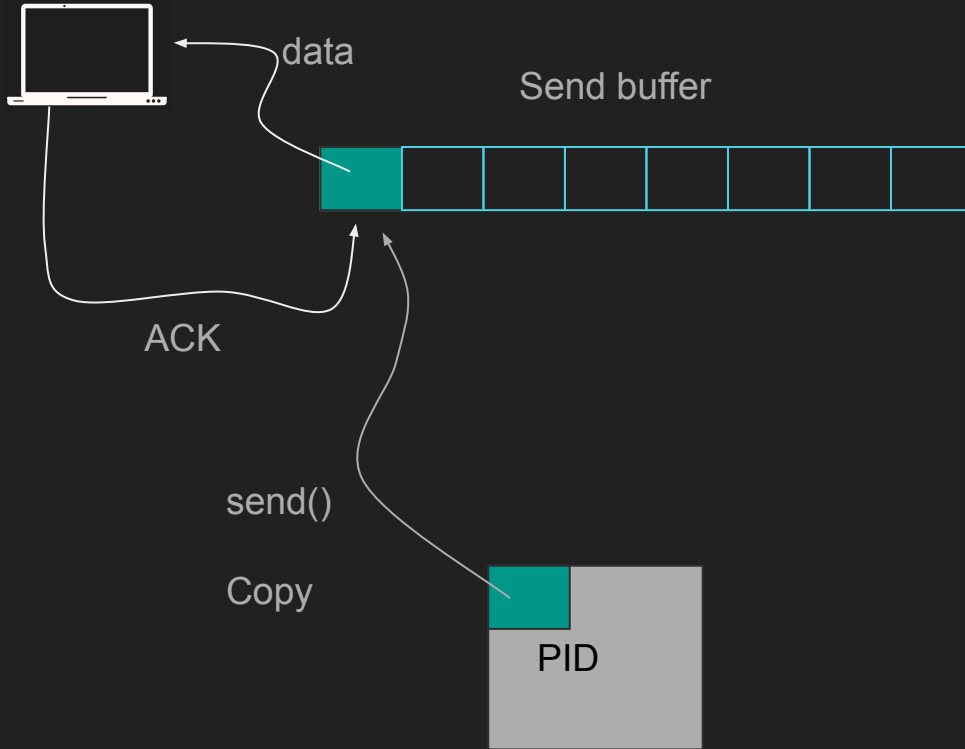
## Send and receive buffers

- Client sends data on a connection
- Kernel puts data in receive queue
- Kernel ACKs (may delay) and update window
- App calls read to copy data

# Receive buffers



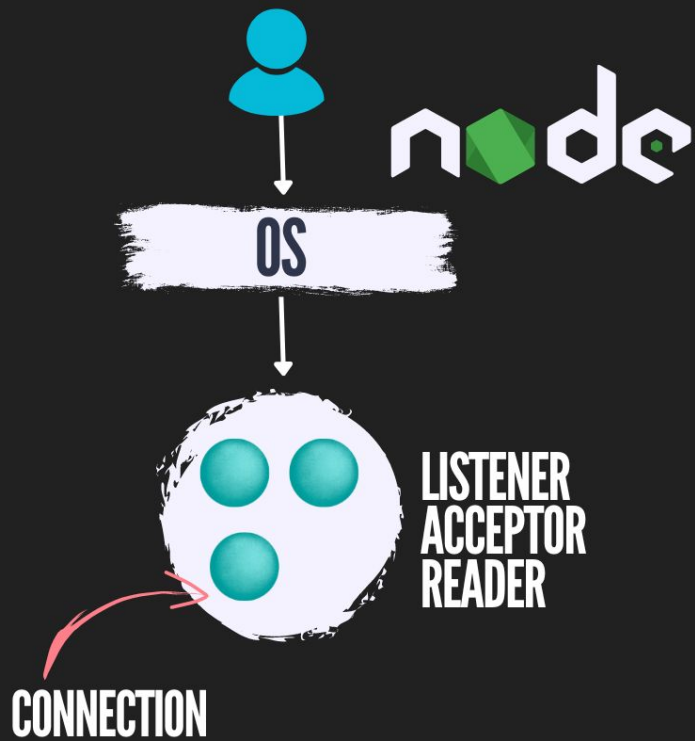
# Send buffers



## Problems with reading and sending

- Backend doesn't read fast enough
- Receive queue is full
- Client slows down

# Single Listener/Single Worker Thread



# SINGLE THREAD

# Single Listener/Multiple Worker threads



**MULTIPLE THREADS**  
W/ SINGLE ACCEPTOR

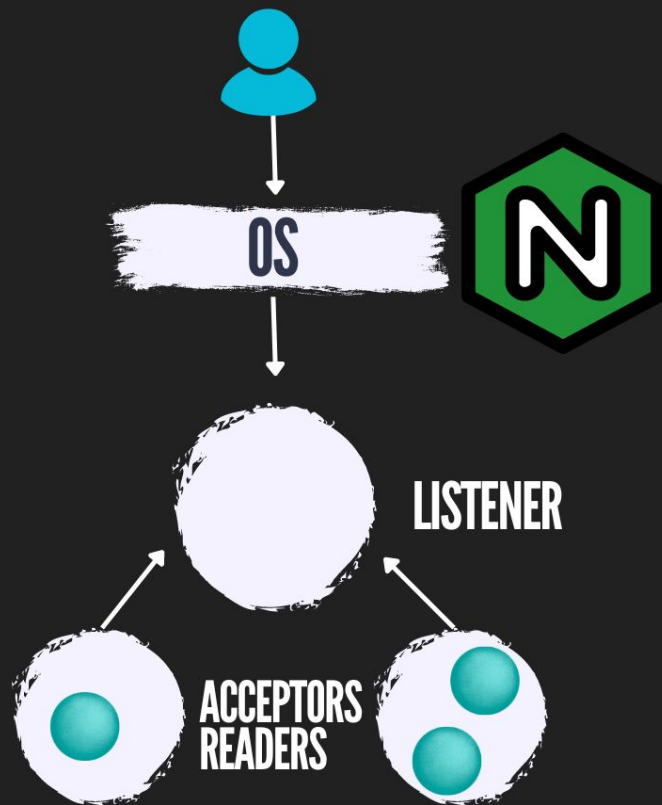


# Single Listener/Multiple Worker threads with load balancing



**MULTIPLE THREADS**  
W/MESSAGE LOAD BALANCING

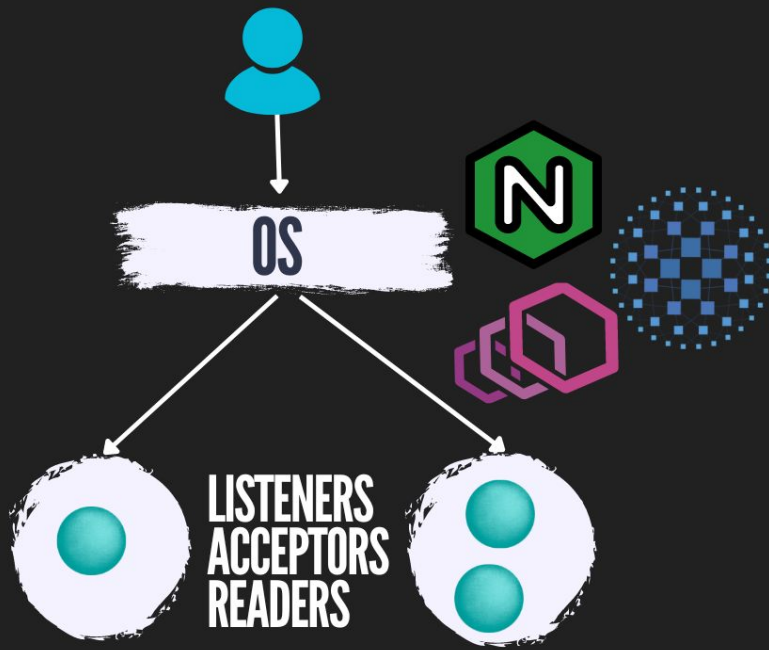
# Multiple Threads single Socket



**MULTIPLE THREADS**  
W/ MULTIPLE ACCEPTORS

# Multiple Listeners on the same port

SO\_REUSEPORT



**MULTIPLE THREADS**  
W/ SOCKET SHARDING (SO\_REUSEPORT)

# Idempotency

Resending the Request without affecting backend

# What is idempotency?

- API /postcomment
- Takes comment and appends it to table
- What if the user/proxy retries it?
- You will have duplicate comments
- Very bad for financial systems



# What is idempotency?

- Idempotent request can be retried without affecting backend
- Easy implementation send a requestId
- If requestId has been processed return
- Also known as idempotency token

# In HTTP

- GET is idempotent
- POST isn't, but we can make it
- Browsers and proxies treat GET as idempotent
- Make sure your GETs are

# Proxying & Load Balancing

Core of backend engineering

# Proxy vs Reverse Proxy

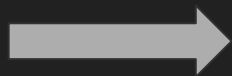
A fundamental component of backend networking

# Layer 4 vs Layer 7 Load balancers

A fundamental component of backend networking

# Agenda

- Layer 4 vs Layer 7
- Load Balancer
- Layer 4 Load Balancer (pros and cons)
- Layer 7 Load Balancer (pros and cons)



Layer 7 Application

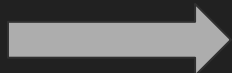
Application

Layer 6 Presentation

Presentation

Layer 5 Session

Session



Layer 4 Transport

Transport

Layer 3 Network

Network

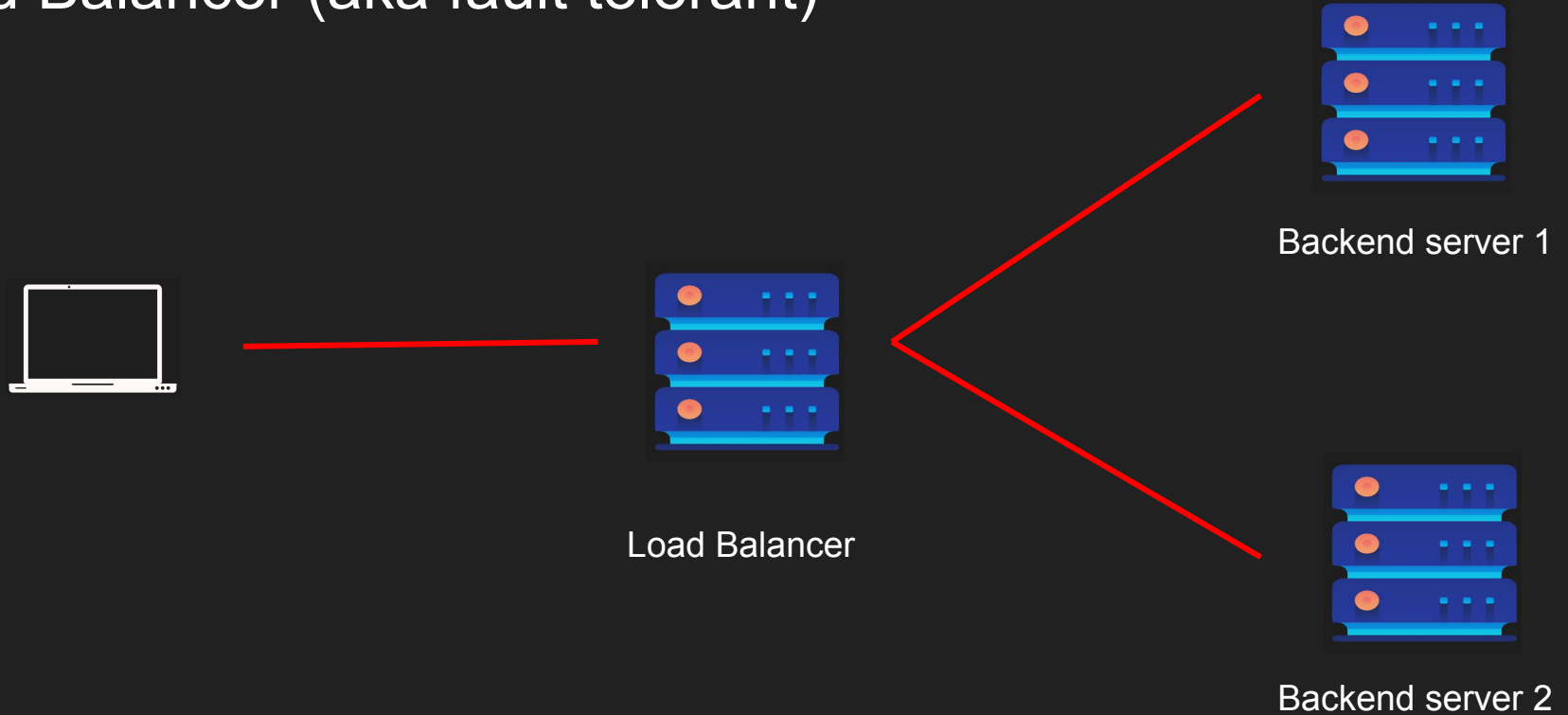
Layer 2 Data Link

Data Link

Layer 1 Physical

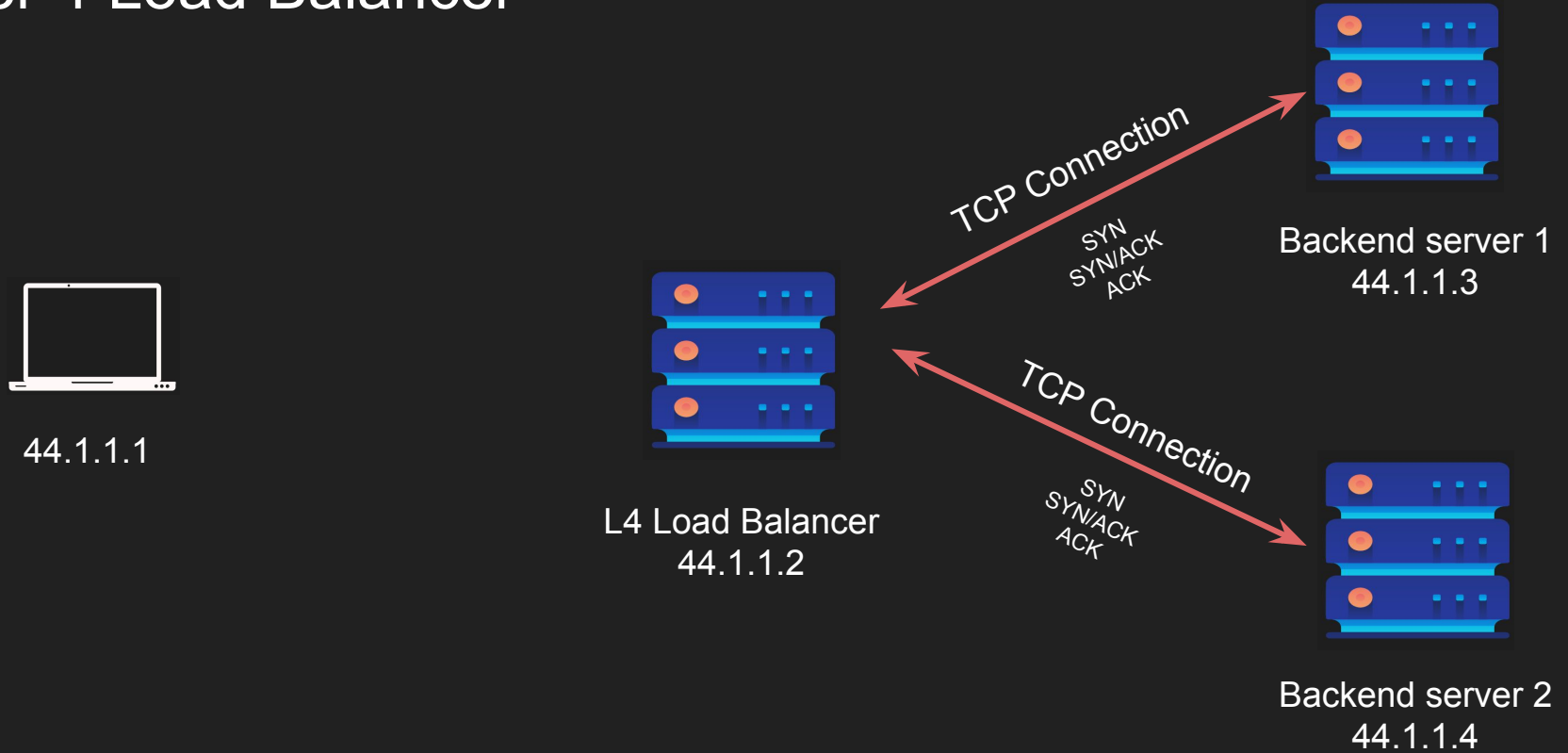
Physical

# Load Balancer (aka fault tolerant)

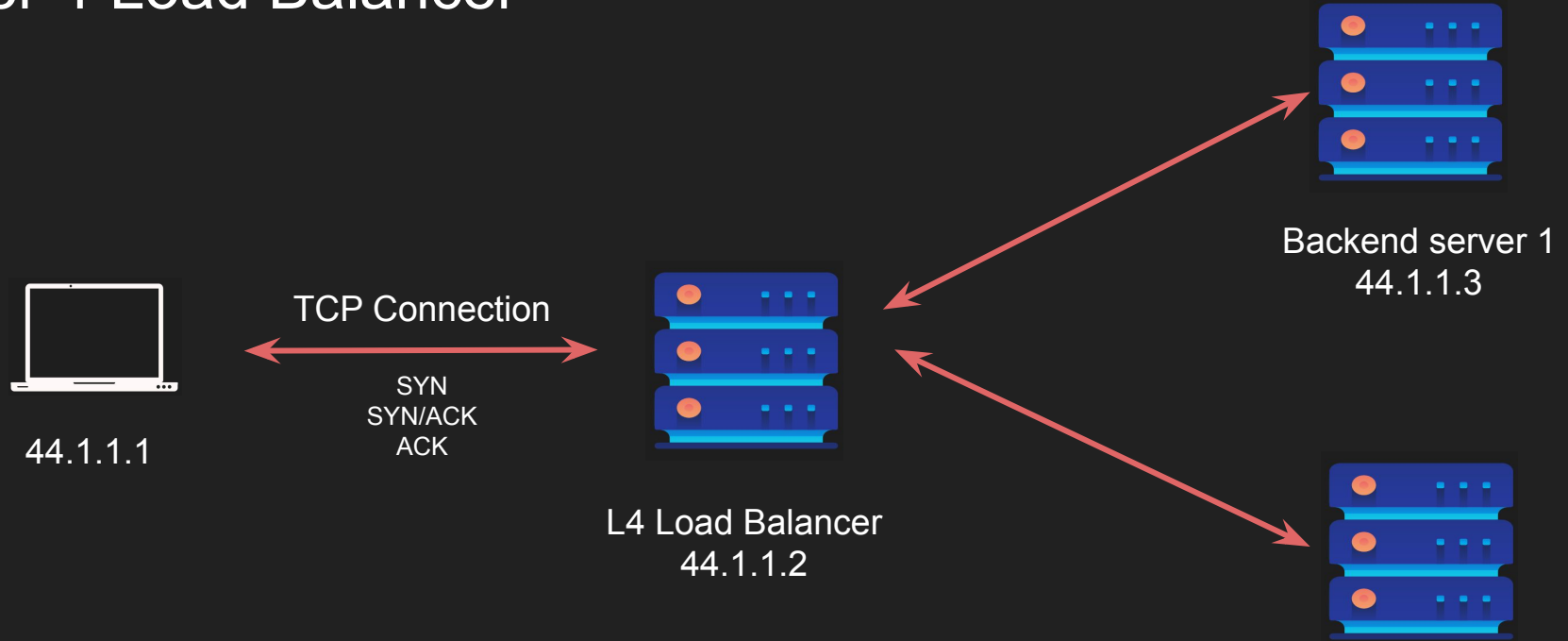




# Layer 4 Load Balancer

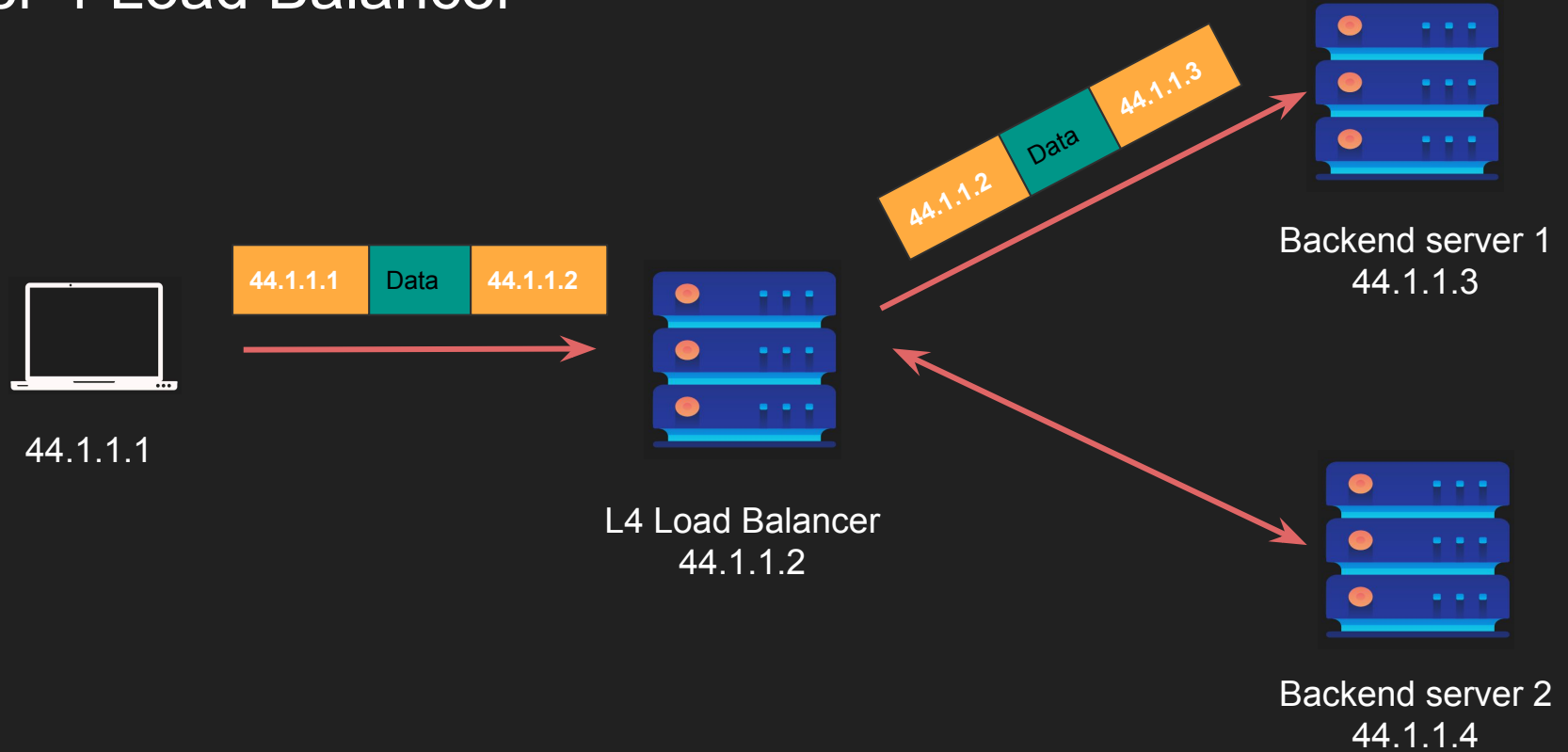


# Layer 4 Load Balancer

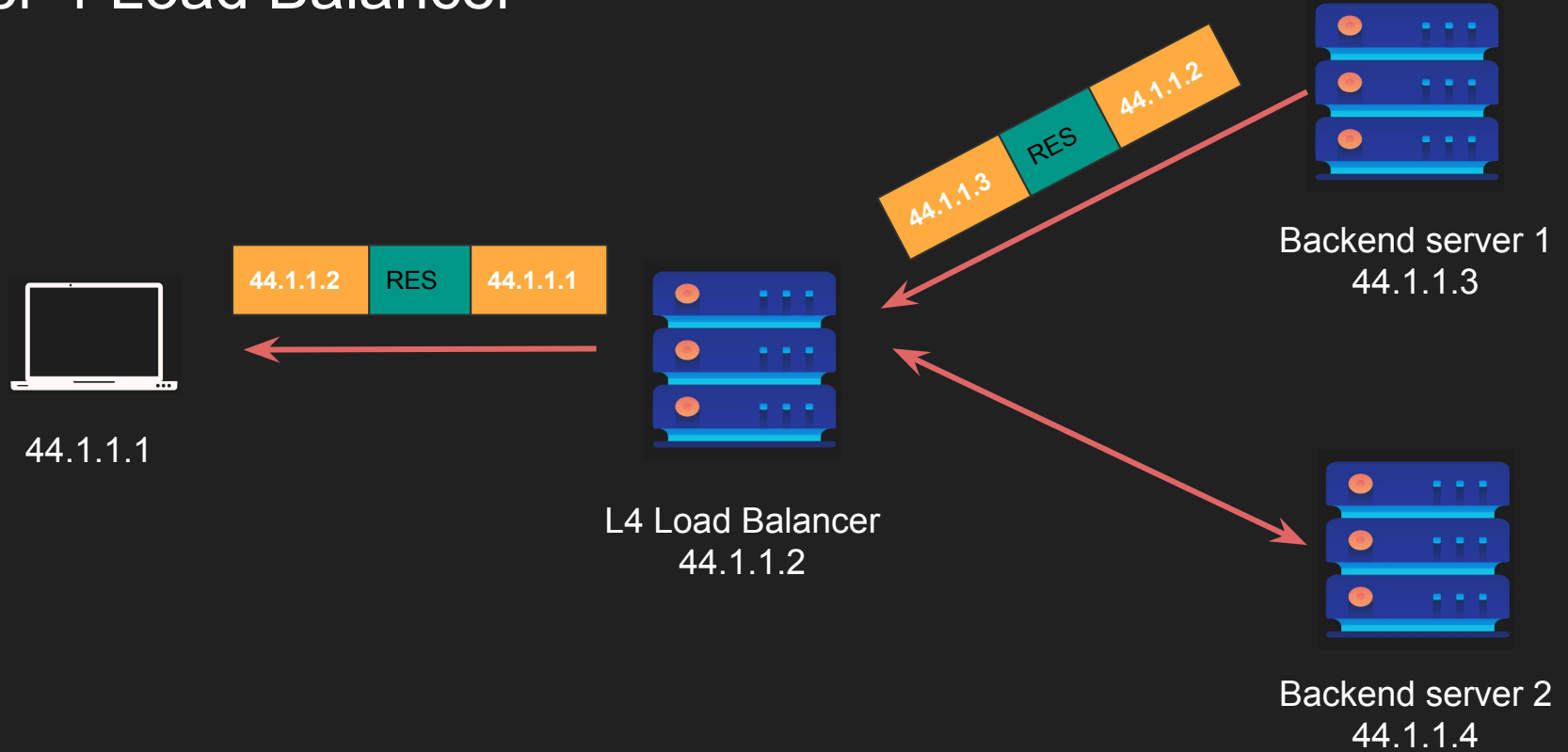


When a client connects to the L4 load balancer, the LB chooses one server and all segments for that connections go to that server

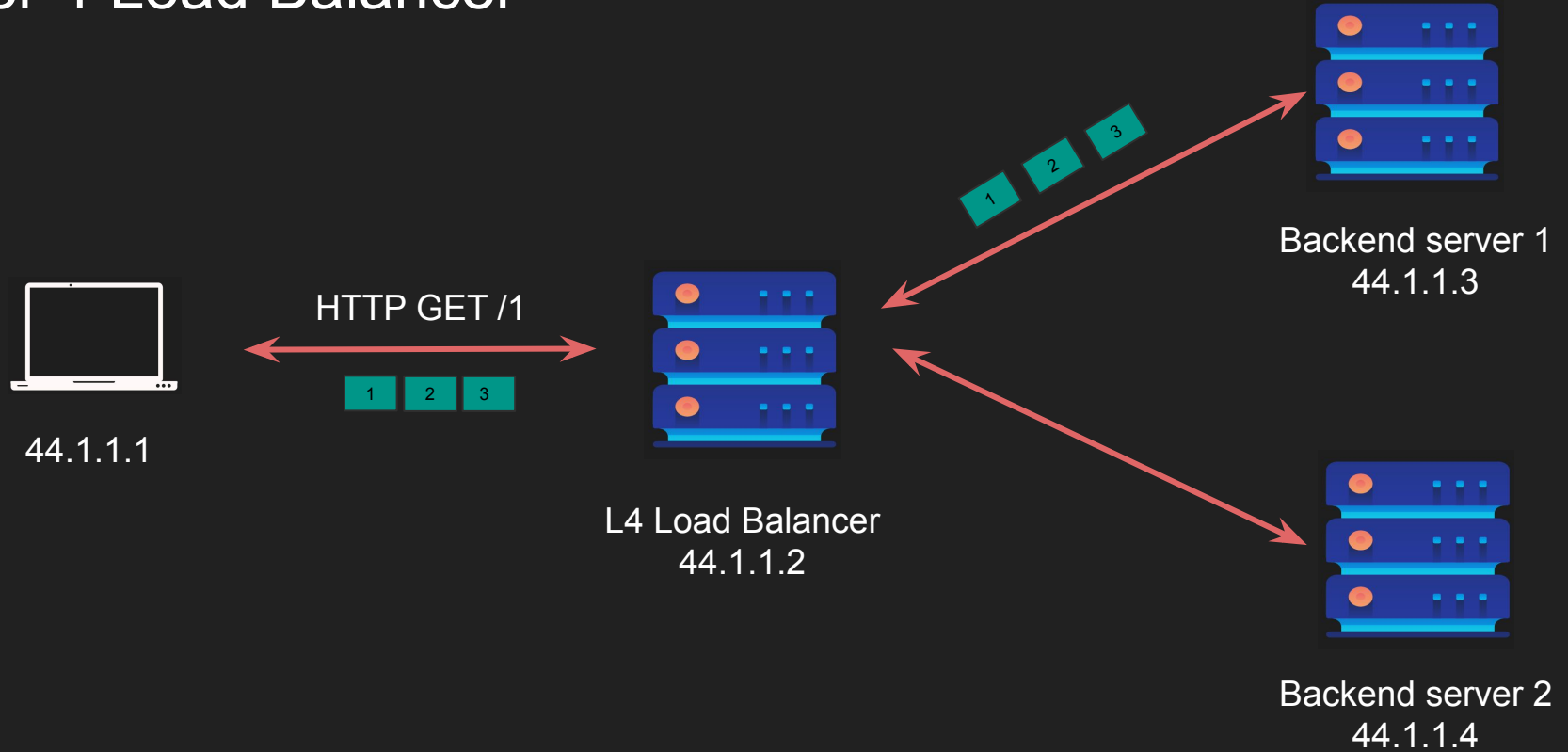
# Layer 4 Load Balancer



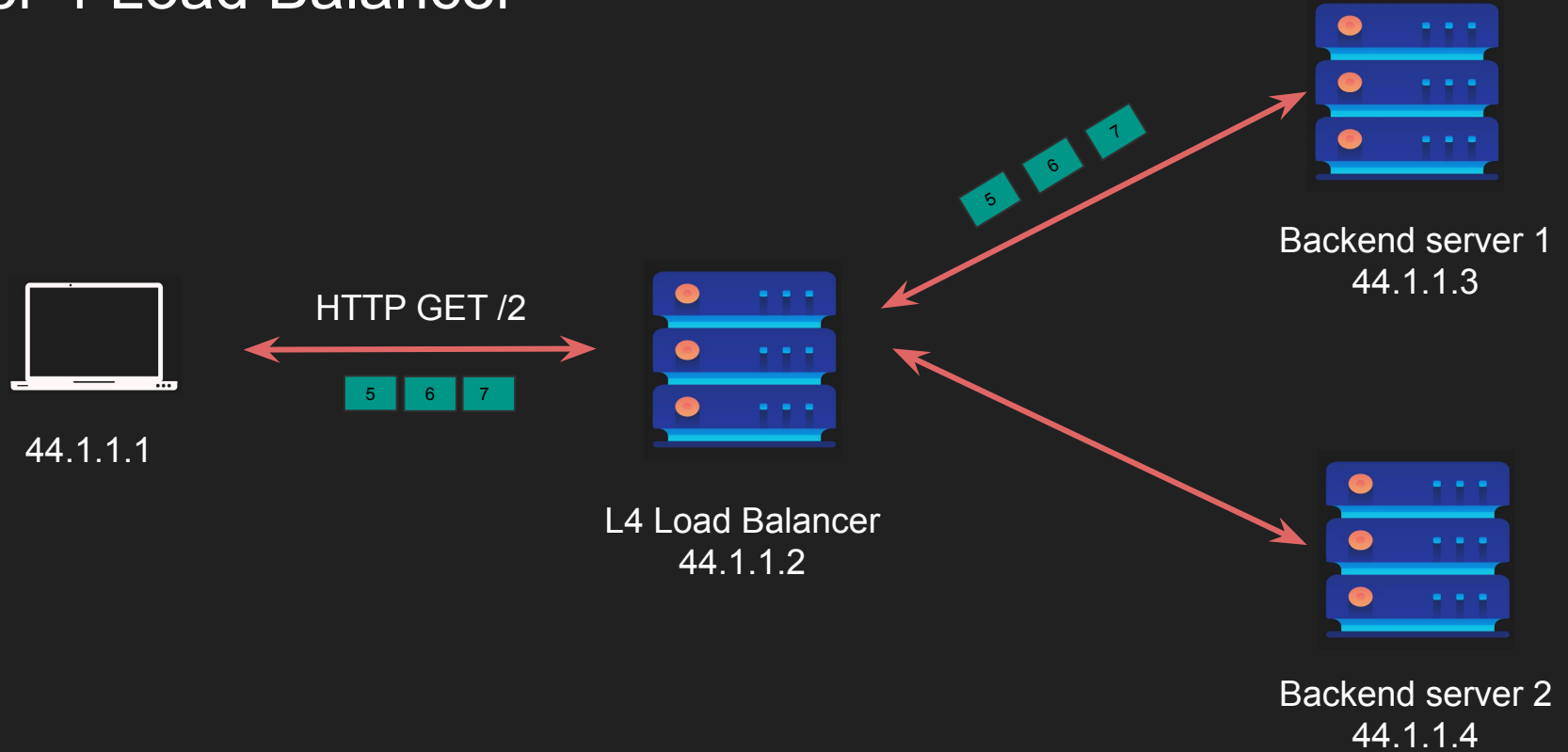
# Layer 4 Load Balancer



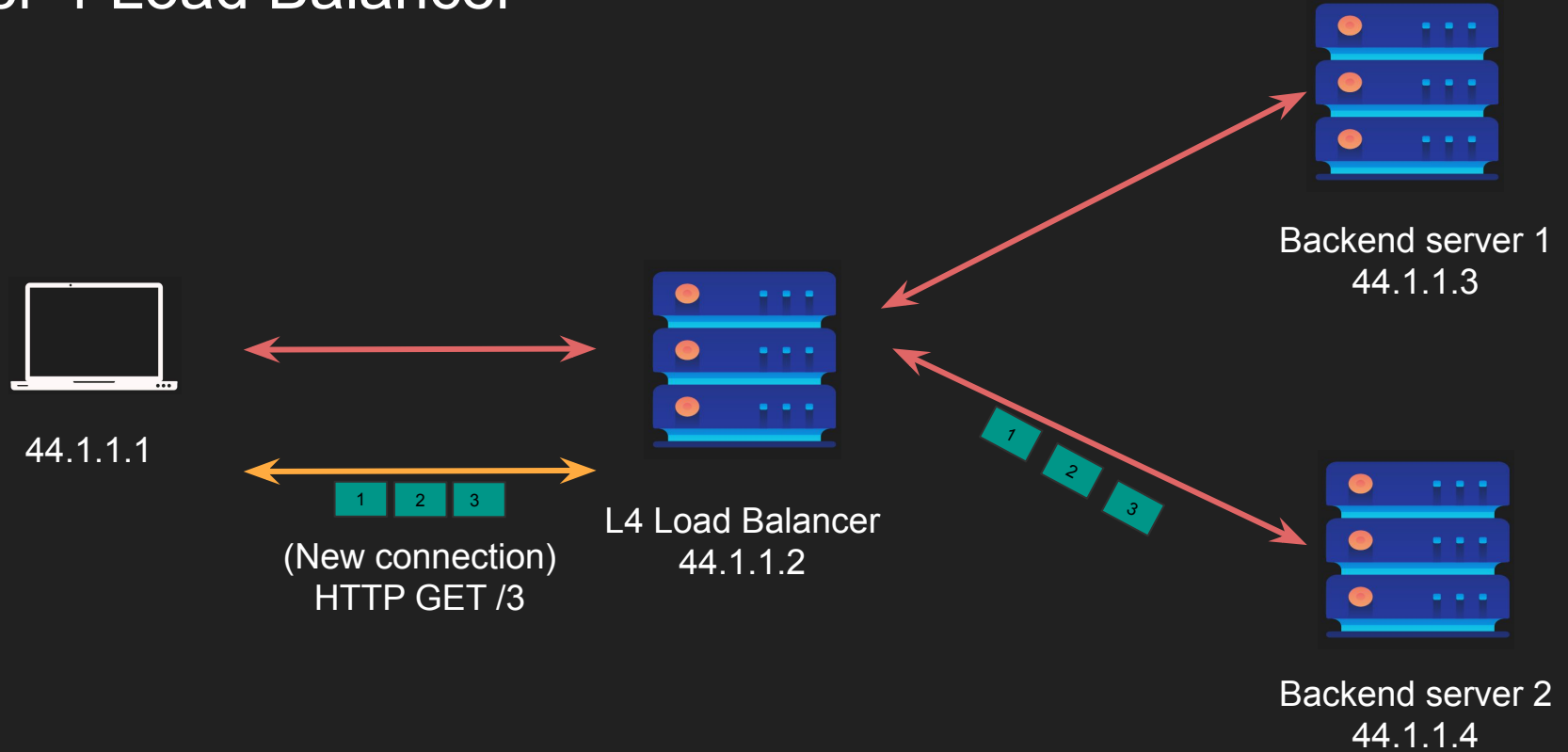
# Layer 4 Load Balancer



# Layer 4 Load Balancer



# Layer 4 Load Balancer



# Layer 4 Load Balancer (Pros and Cons)

## Pros

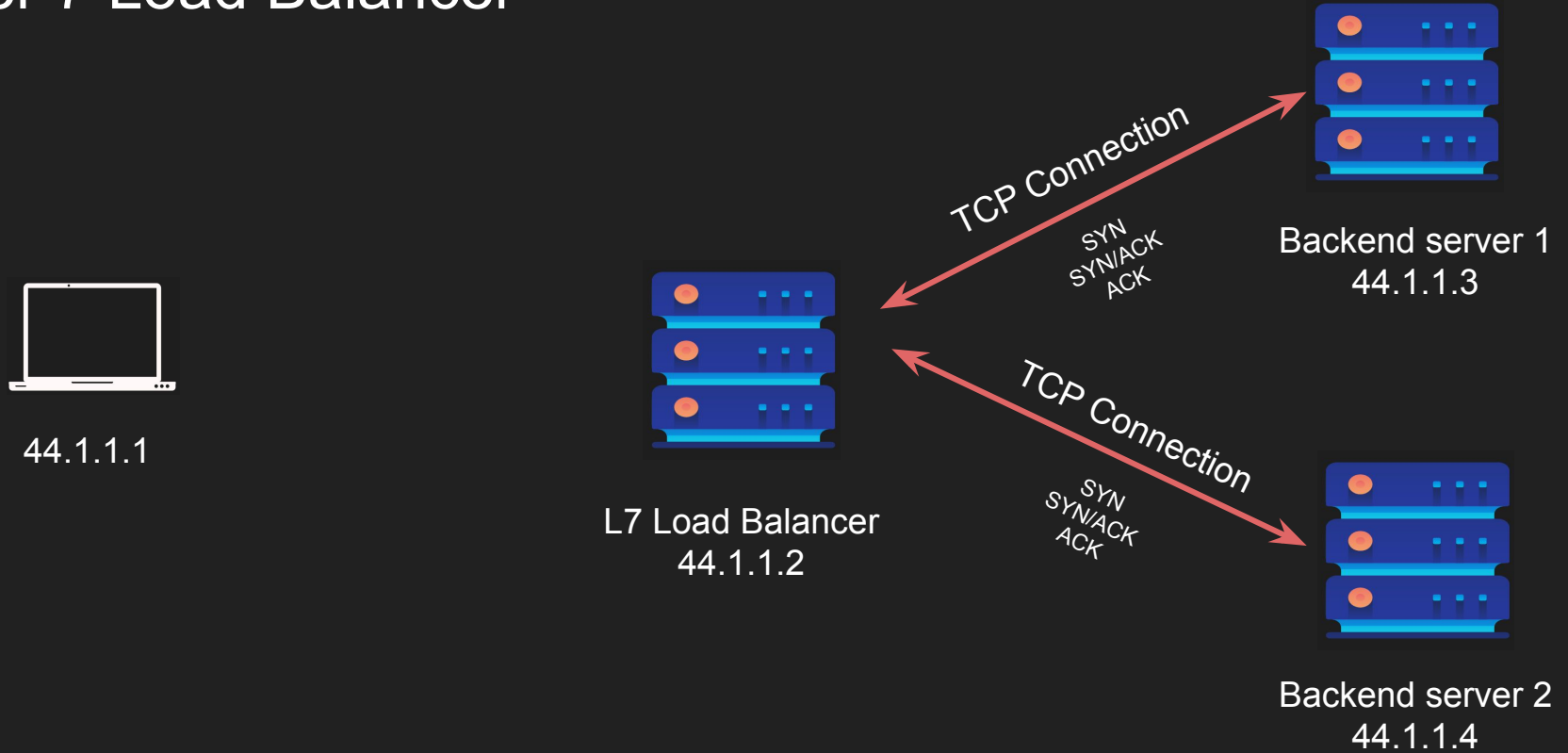
- Simpler load balancing
- Efficient (no data lookup)
- More secure
- Works with any protocol
- One TCP connection (NAT)

## Cons

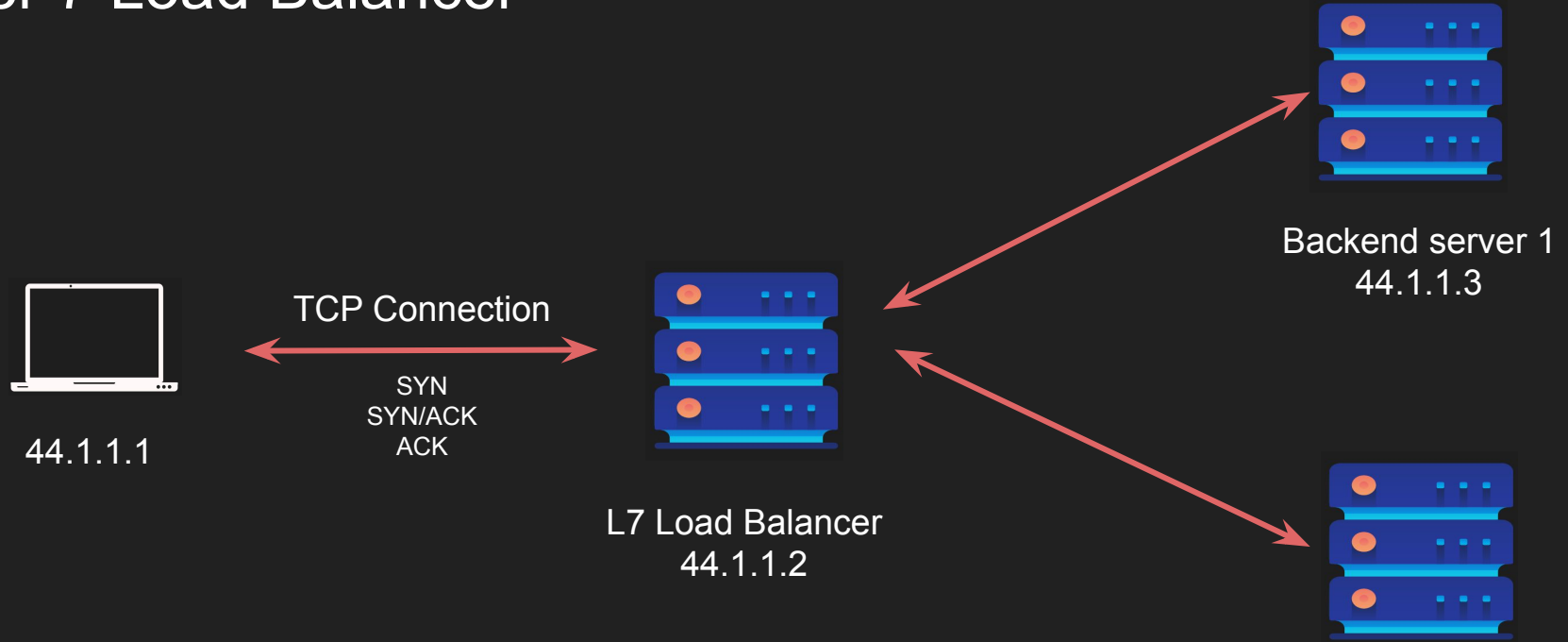
- No smart load balancing
- NA microservices
- Sticky per connection
- No caching
- Protocol unaware (can be dangerous) bypass rules



# Layer 7 Load Balancer

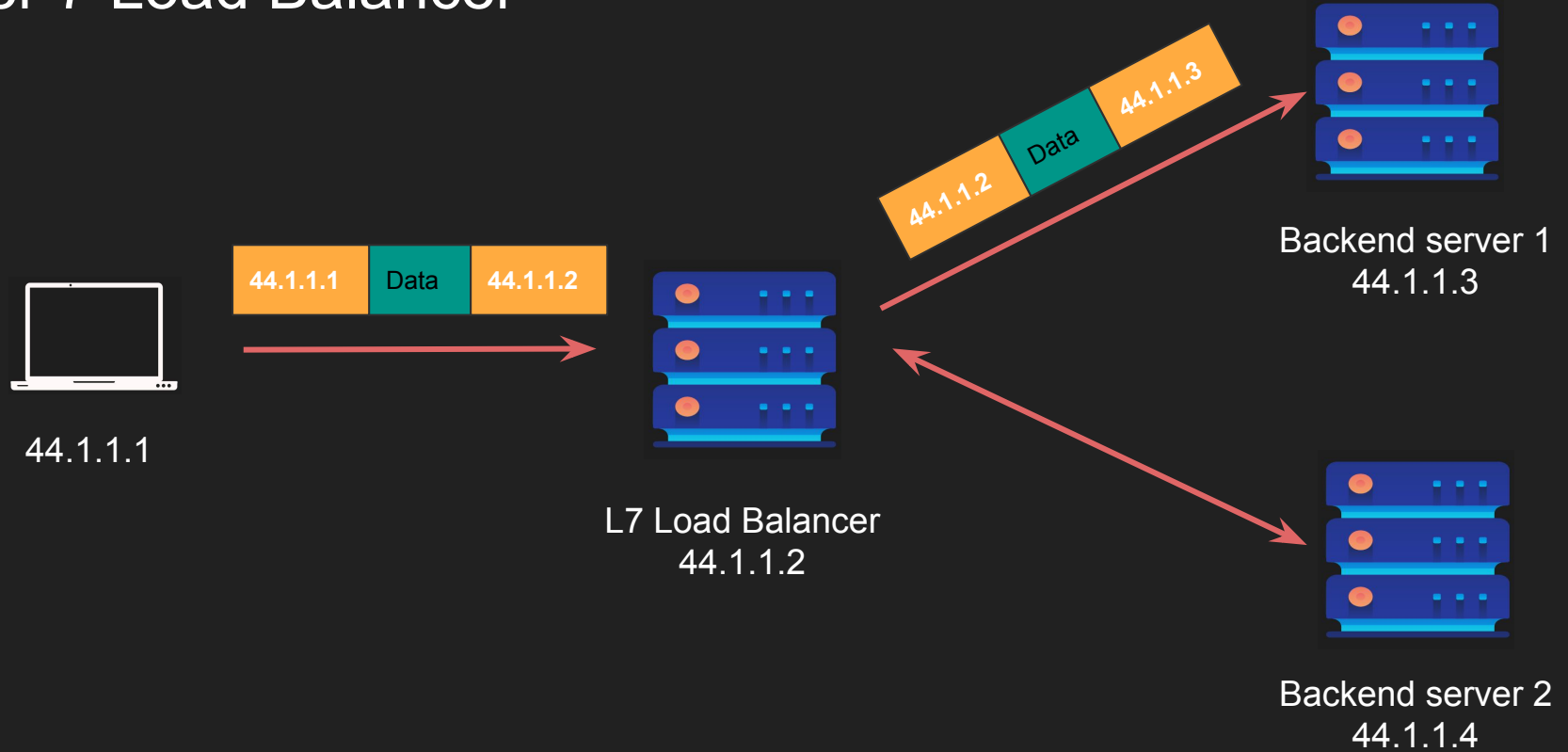


# Layer 7 Load Balancer

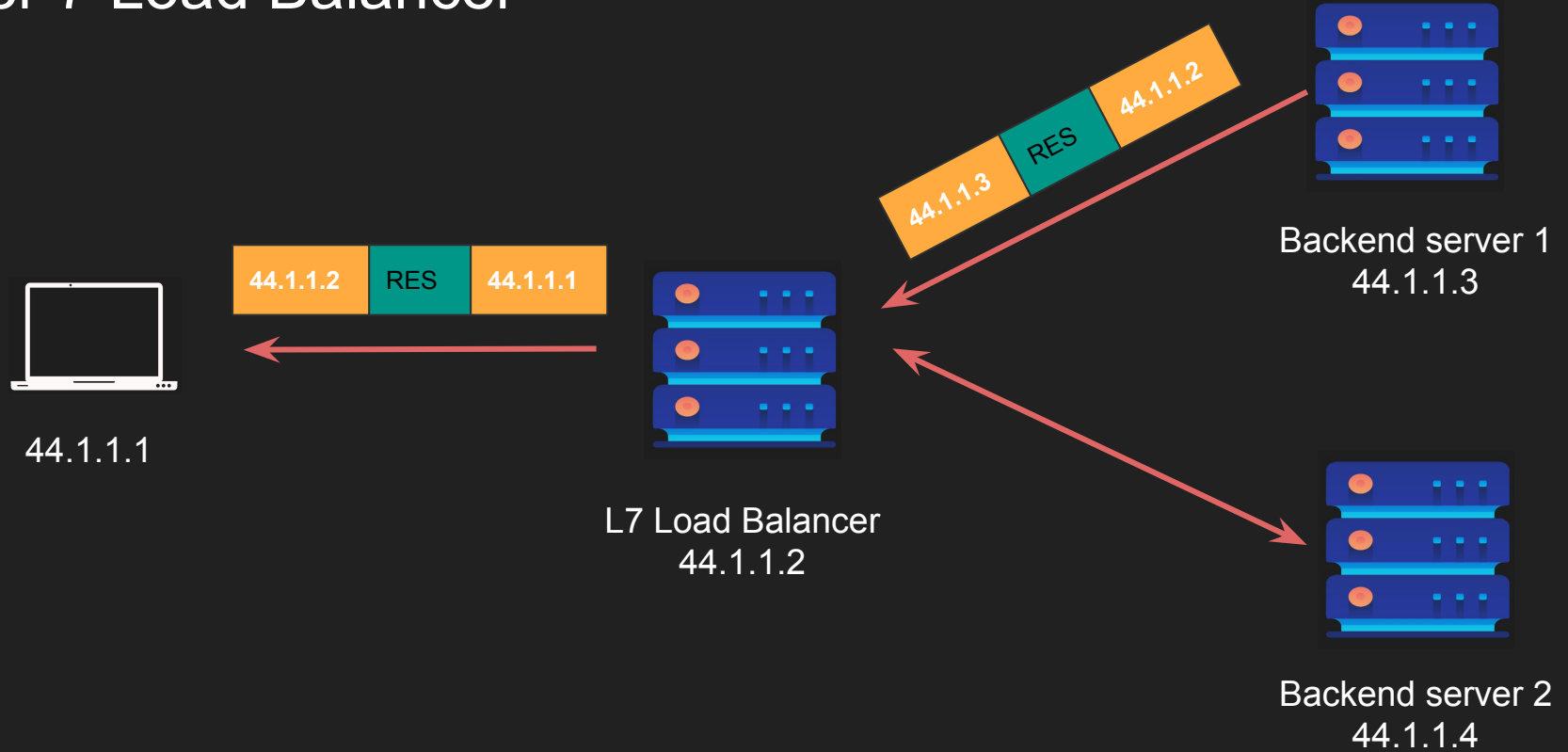


When a client connects to the L7 load balancer, it becomes protocol specific. Any logical “request” will be forwarded to a new backend server. This could be one or more segments

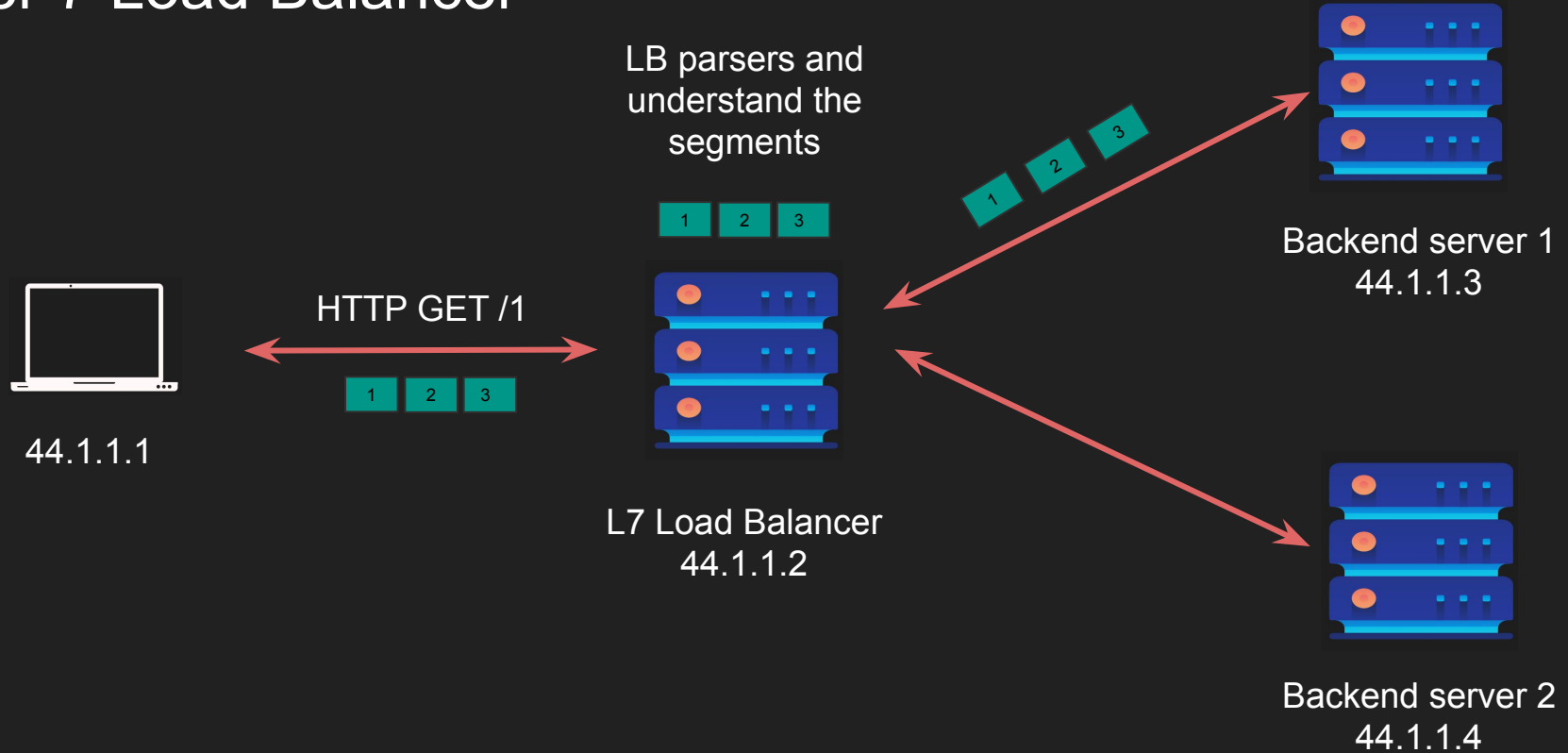
# Layer 7 Load Balancer



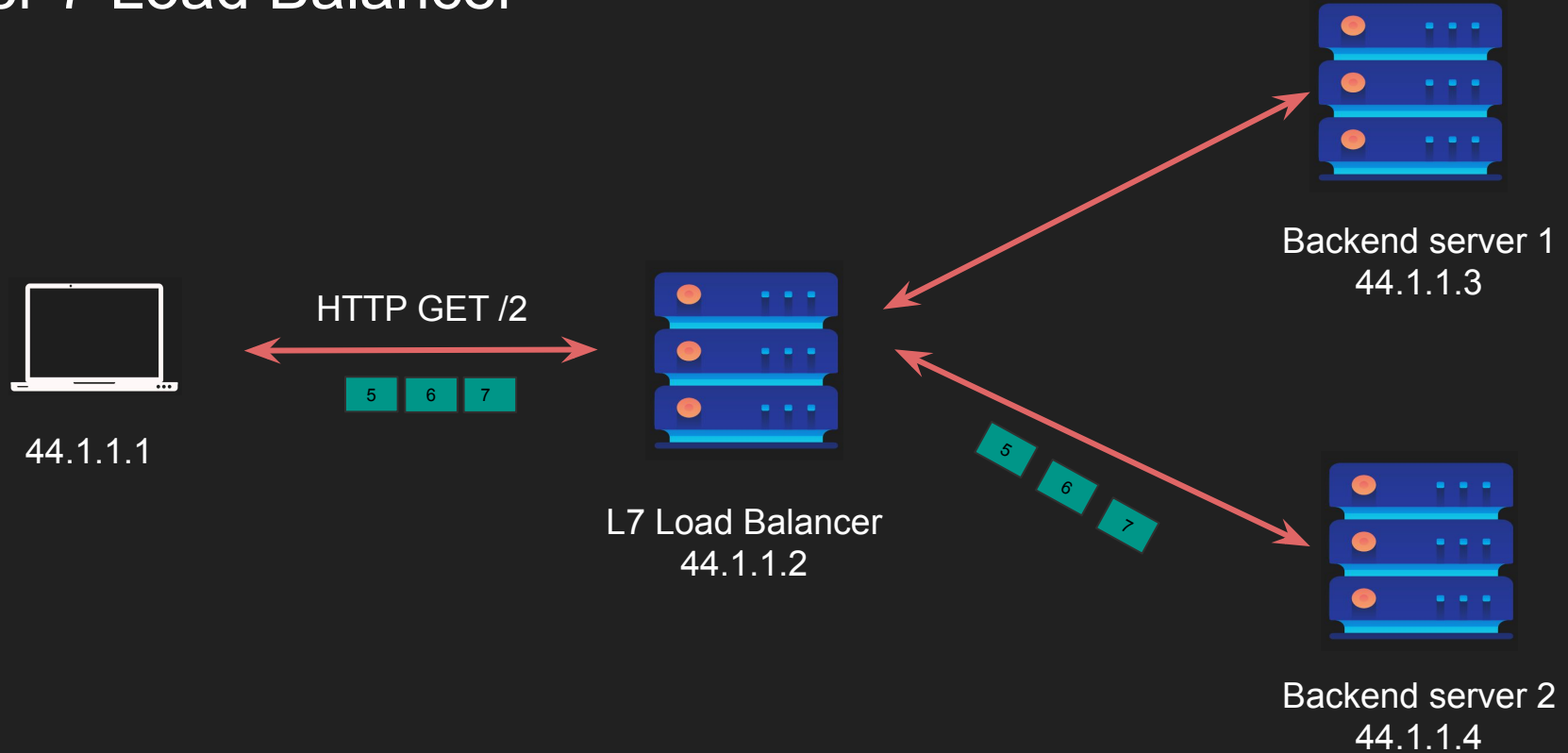
# Layer 7 Load Balancer



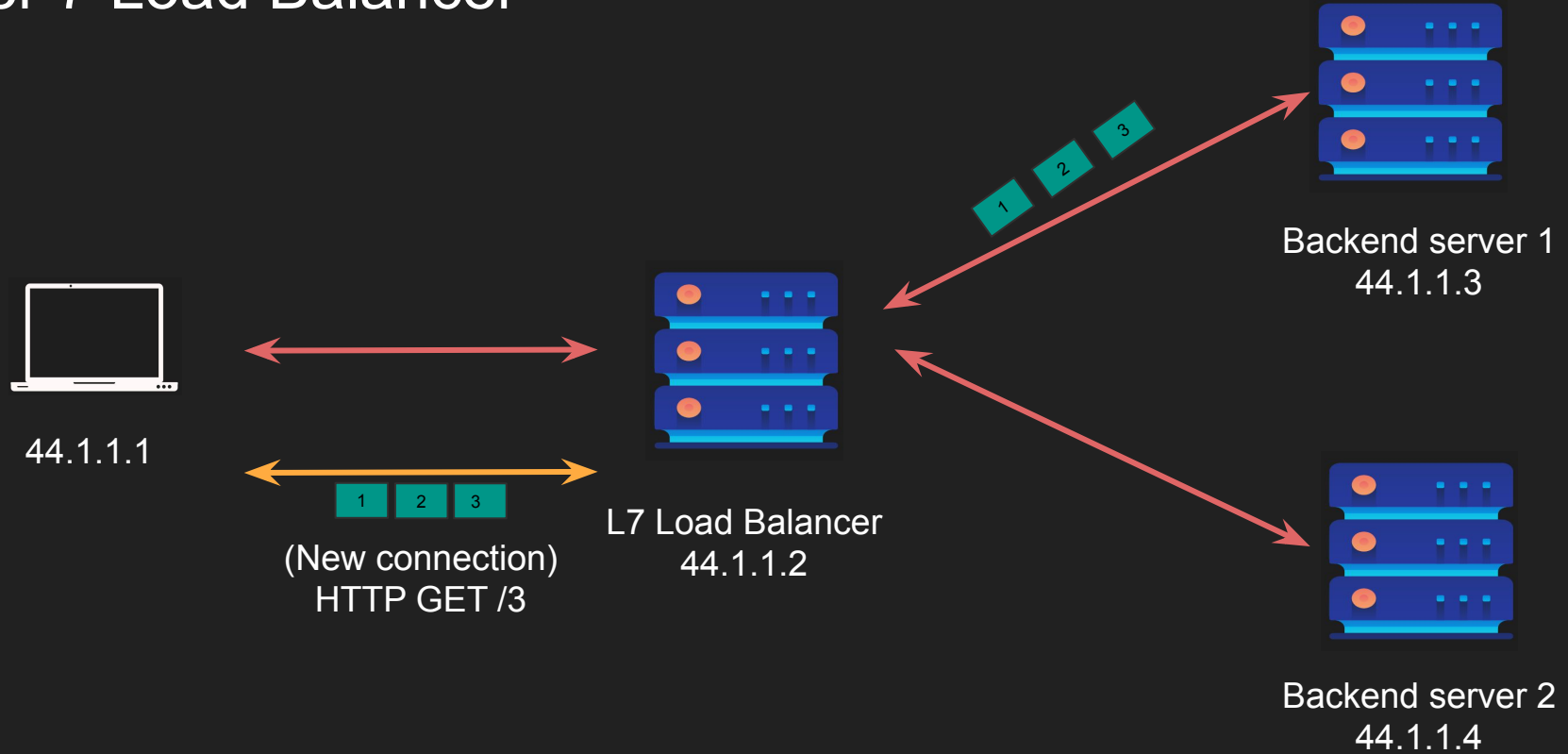
# Layer 7 Load Balancer



# Layer 7 Load Balancer



# Layer 7 Load Balancer



# Layer 7 Load Balancer (Pros and Cons)

## Pros

- Smart load balancing
- Caching
- Great for microservices
- API Gateway logic
- Authentication

## Cons

- Expensive (looks at data)
- Decrypts (terminates TLS)
- Two TCP Connections
- Must share TLS certificate
- Needs to buffer
- Needs to understand protocol



# Summary

- Layer 4 vs Layer 7
- Load Balancer
- Layer 4 Load Balancer (pros and cons)
- Layer 7 Load Balancer (pros and cons)