Blockchain (BlCh)

Repetition DSy – part 1

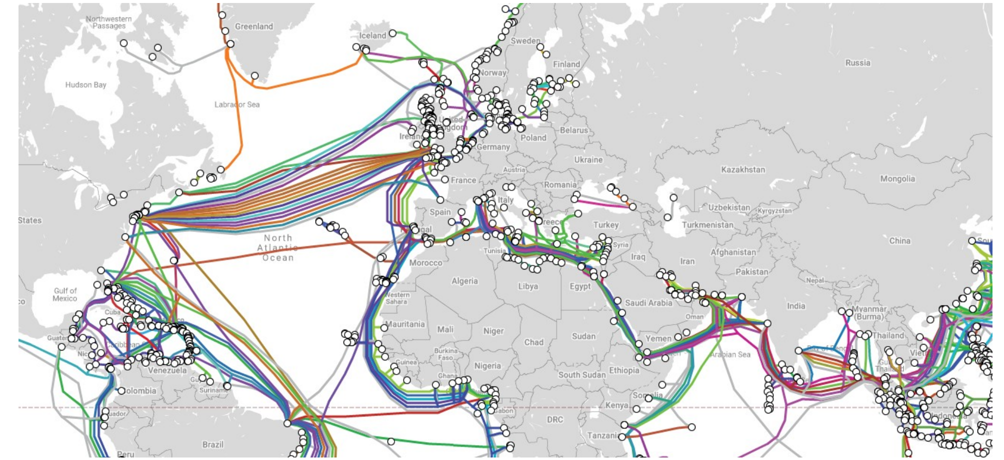
Thomas Bocek

16.09.2024

Lecture 1 + 2

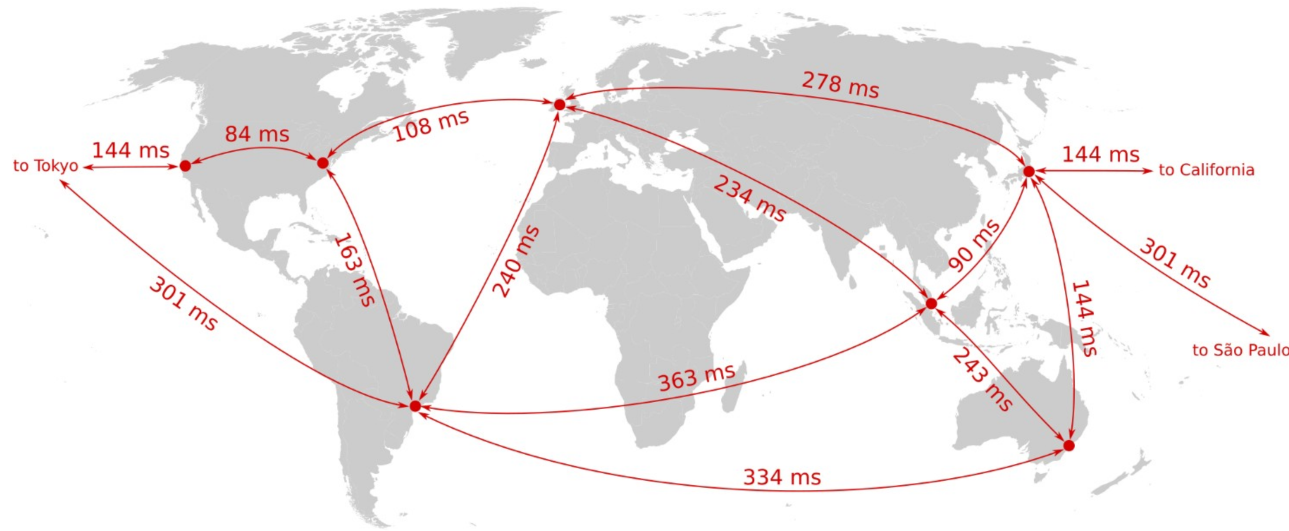
Distributed Systems Motivation

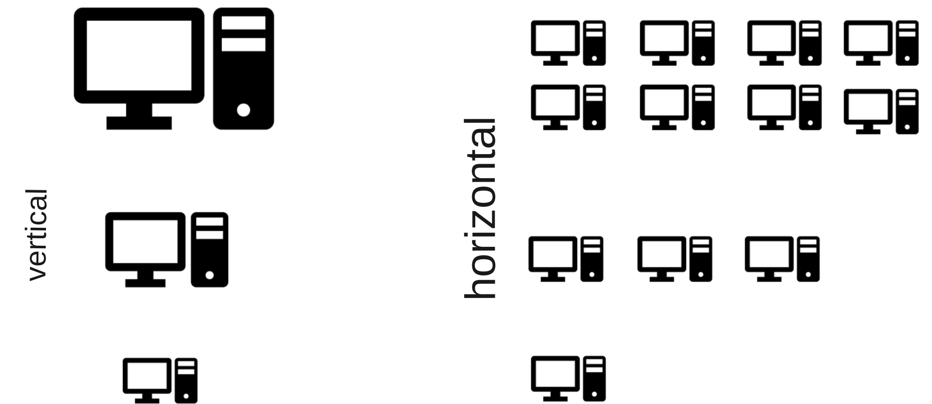
- Why Distributed Systems
  - Scaling
  - Location
  - Fault-tolerance (bitflips, outages)



Submarine Cable Map

https://www.inkandswitch.com/local-first.html





OST

Lecture 3

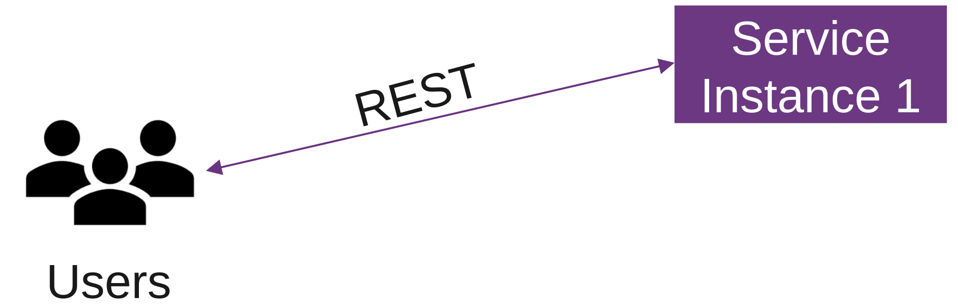
OST

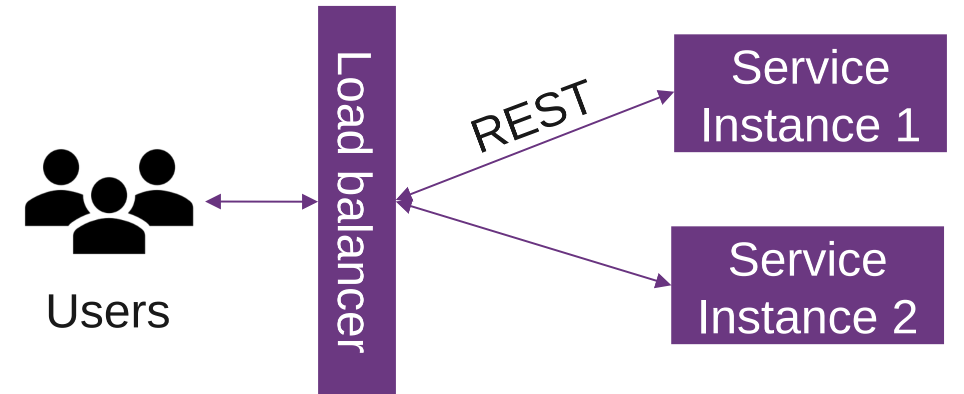
Load Balancing

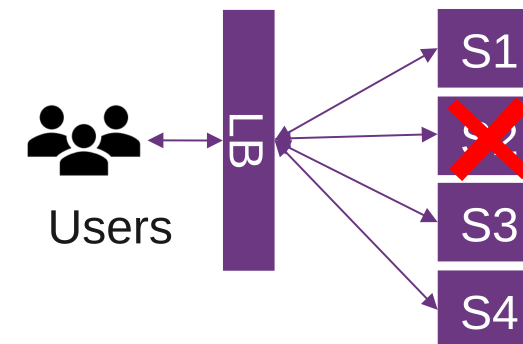
- What is load balancing
  - Distribution of workloads across multiple computing resources
    - Workloads (requests)
    - Computing resources (machines)
  - Distributes client requests or network load efficiently across multiple servers [link]
    - E.g., service get popular, high load on service

→ horizontal scaling

- Why load balancing
  - Ensures high availability and reliability by sending requests only to servers that are online
  - Provides the flexibility to add or subtract servers as demand dictates







OST

CORS

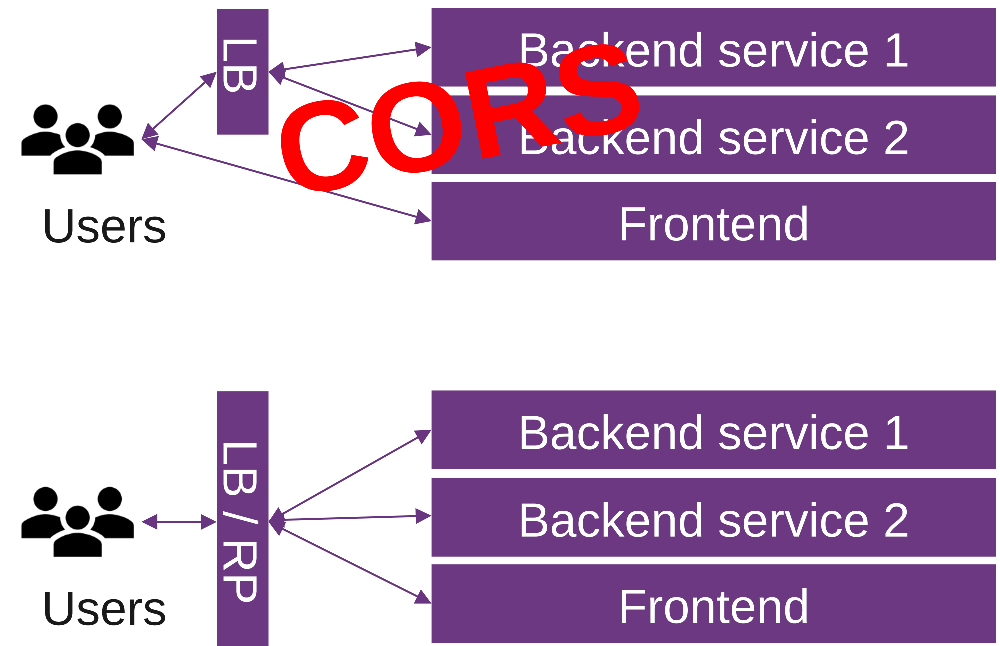
- CORS = Cross-Origin Resource Sharing
  - For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts (among others)
  - Mechanism to instruct browsers that runs a resource from origin A to run resources from origin B
- Solution
  - Use reverse proxy with builtin webserver, e.g., nginx, or user reverse proxy with external webserver.

→ The client only sees the same origin for the API and the frontend assets

- Access-Control-Allow-Origin: https://foo.example

→ For dev: Access-Control-Allow-Origin: *

- w.Header().Set("Access-Control-Allow-Origin", "*")
- Reverse proxy

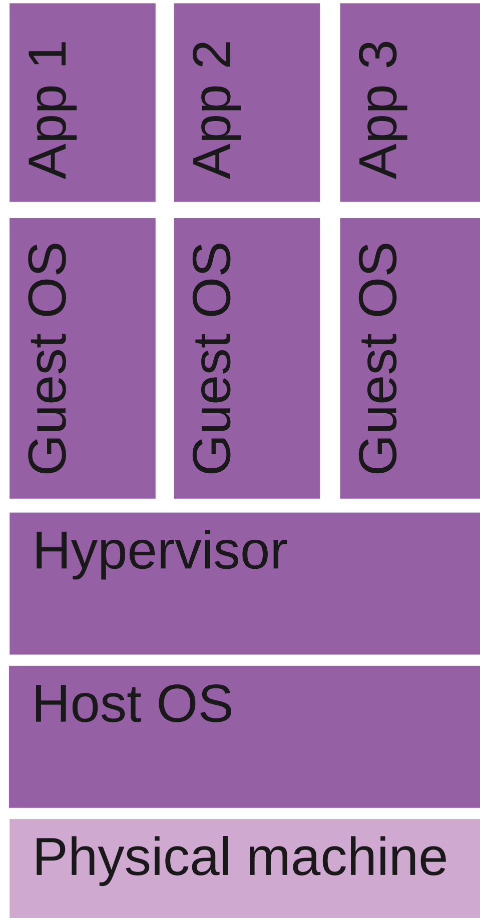


OST

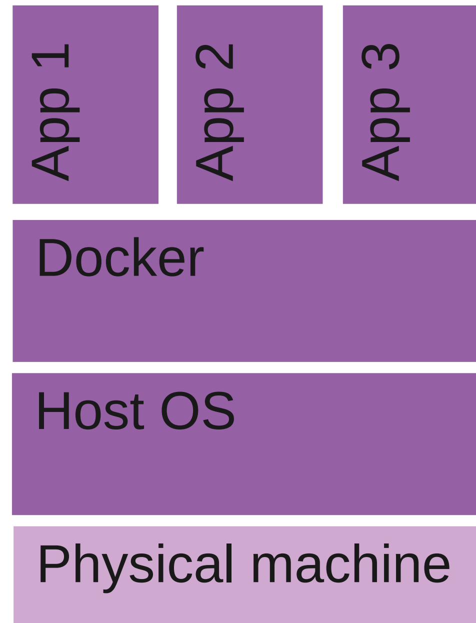
Lecture 4

OST

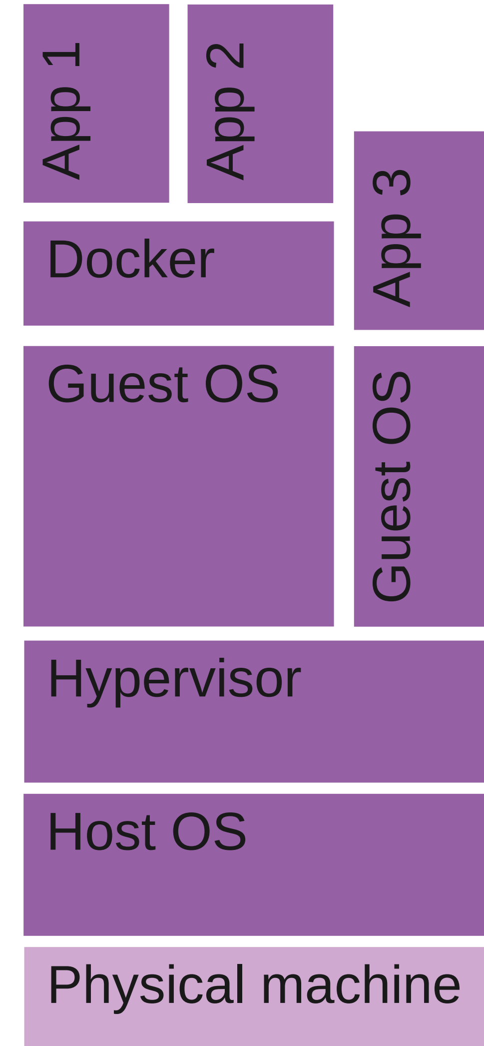
Introduction



- Virtual machines



- Container



- Both

OST

Lecture 5

Distributed Systems Categorization

“Controlled” Distributed Systems

- 1 responsible organization
- Low churn
- Examples:
  - Amazon DynamoDB
  - Client/server
- “Secure environment”
- High availability
- Can be homogeneous / heterogeneous

“Fully” Decentralized Systems

- N responsible organizations
- High churn
- Examples:
  - BitTorrent
  - Blockchain
- “Hostile environment”
- Unpredictable availability
- Is heterogeneous

Distributed Systems Categorization

“Controlled” Distributed Systems

- Mechanisms that work well:
  - Consistent hashing (DynamoDB, Cassandra)
  - Master nodes, central coordinator
- Network is under control or client/server → no NAT issues

“Fully” Decentralized Systems

- Mechanisms that work well:
  - Consistent hashing (DHTs)
  - Flooding/broadcasting - Bitcoin
- NAT and direct connectivity huge problem

OST

Distributed Systems Categorization

“Controlled” Distributed Systems

- Consistency
  - Leader election (Zookeeper, Paxos, Raft)
- Replication principles
  - More replicas: higher availability, higher reliability, higher performance, better scalability, but: requires maintaining consistency in replicas
- Transparency principles apply

“Fully” Decentralized Systems

- Consistency
  - Weak consistency: DHTs
  - Nakamoto consensus (aka proof of work)
  - Proof of stake – Leader election, PBFT protocols - Is Bitcoin eventually consistent?
    - Some argue no, some argue it has even stronger guarantees [link]
- Replication principles apply to fully decentralized systems as well
- Transparency principles apply

OST

Distributed Systems Categorization

- Spring Term – Distributed Systems (DSy)
  - Tightly/loosely coupled
  - Heterogeneous systems
  - Small-scale systems
  - Distributed systems

  (we will also talk about blockchains in this lecture)

- Fall Term – Blockchain (BlCh)
  - Loosely coupled
  - Heterogeneous systems
  - Large-scale systems
  - Decentralized systems

  (we will also talk about distributed systems in this lecture, but DSy is highly recommended)

Lecture 6

Pro/Cons - Opinion

- Monorepo
  - Tight coupling of projects
    - E.g., generating openapi.yml from backend, generate types for frontend → simply copy
  - Everyone sees all code / commits
  - Encourages code sharing within organization
  - Scaling: large repos, specialized tooling
- Polyrepo
  - Loose coupling of projects
    - If you want to generate openapi.yml, you need access from the backend repository to the frontend (e.g., curl+token)
  - Fine grained access control
  - Encourages code sharing across organizations
  - Scaling: many projects, special coordination

- Opinion: Accenture - “From my experience, for a smaller team, starting with mono-repo is always safe and easy to start. Large and distributed teams would benefit more from poly-repo”
- My opinion: for small teams and “independent” project, use polyrepo. (I worked with small teams with mono and polyrepo, I have worked in big projects with polyrepos, but never in a big project with monorepos). If you have a tight coupling between projects (OpenAPI), use monorepos.
- Other opinion (sales pitch): https://monorepo.tools

OST

Lecture 7

Access Token / Refresh Token

- Access Token only short lifetime, e.g., 10min.
  - If public key / secret is known, the content in the token can be trusted, e.g., in the serivce
  - Can have userId, role, etc.
    - No need to query DB for those information, e.g.:

```
type TokenClaims struct {
    MailFrom string `json:"mail_from,omitempty"`
    MailTo   string `json:"mail_to,omitempty"`
    jwt.Claims
}
```

- Refresh Token longer lifetime, e.g., 6 month
  - A refresh token is used to get a new access token
  - IAM / Auth server creates access tokens

- Only access token, with long lifetime
  - If a user credential is revoked – how to inform every service?
- Only refresh token
  - Tightly coupled Service/Auth, every request to Service, Auth needs to be involved for every access
- Access + Refresh token
  - If a user credential is revoked, user has max. 10min more to access service
  - Auth only involved if access token is expired

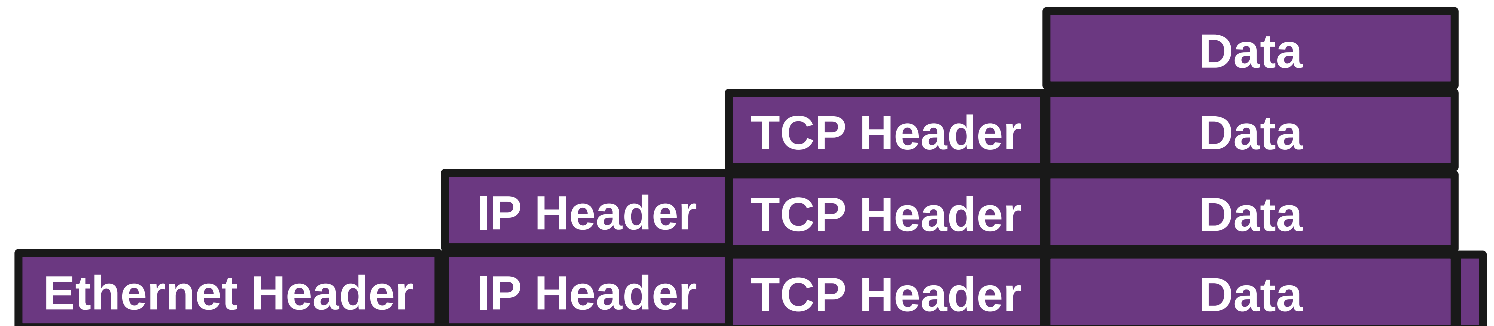
OST

Lecture 8

Networking: Layers

- Networking: Each vendor had its own proprietary solution - not compatible with another solution
  - IPX/SPX – 1983, AppleTalk 1985, DECnet 1975, XNS 1977
- Nowadays most vendors build compatible networks hardware/software from different vendors
  - Cisco, Dell, HP, Huawei, Juniper, Lenovo, Linksys, Netgear, MicroTik, Siemens, Ubiquiti, etc.
- Goal of layers: interoperability
  - 1984: ISO 7498 - The Basic Reference Model for Open Systems Interconnection

OSI model	"Internet model"
Application	Application
Presentation	
Session	
Transport	Transport
Network	Internet
Data link	Link
Pysical	



OST

TCP/IP from an Application Developer View

- Server in golang (repo)
  - git clone https://github.com/tbocek/DSy
  - Download GoLand, or others
  - go run server.go → server
- Listening on TCP port 8081
  - Return string in uppercase
- Node.js version
  - Download WebStorm, or other
- Client:
  - nc localhost 8081

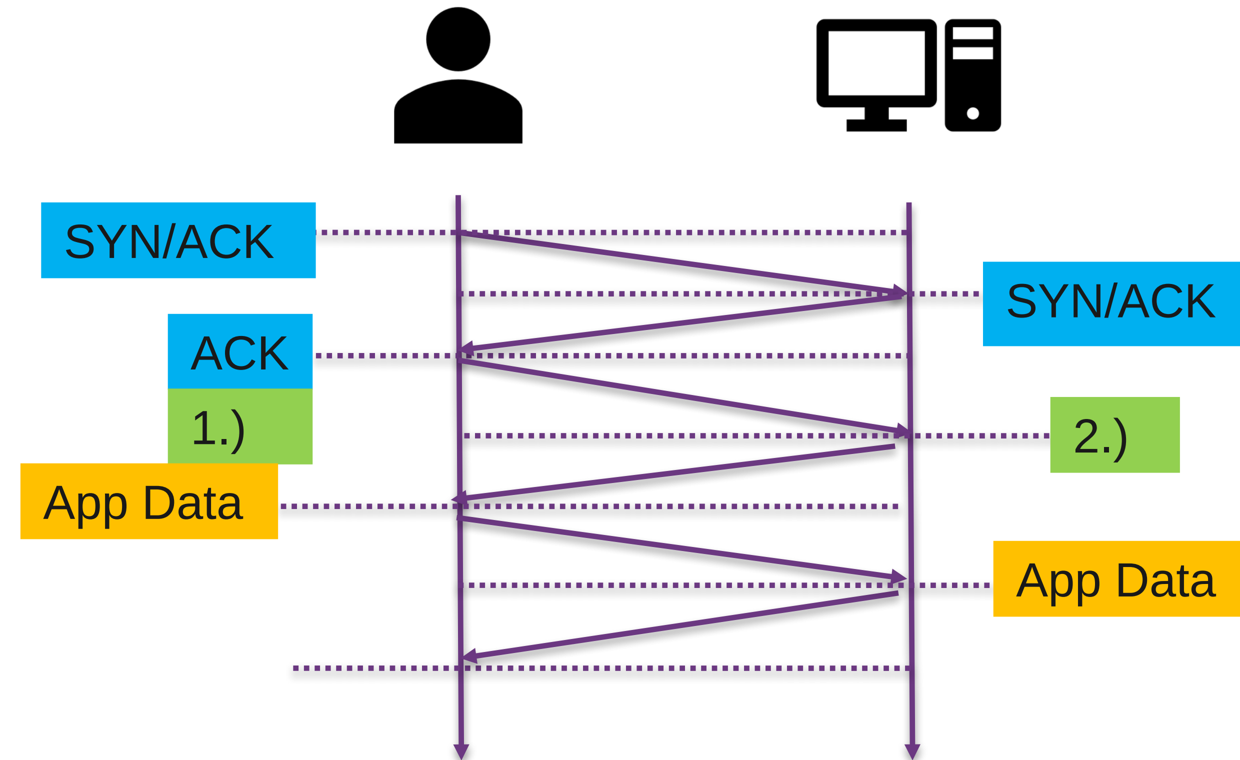
```
const net = require('net');
const server = new net.Server();
server.listen(8081, function() {
  console.log('Launching server...');
});

server.on('connection', function(socket) {
  socket.on('data', function(chunk) {
    console.log(`Data received from client: ${chunk.toString()}`);

socket.write(chunk.toString().toUpperCase() + "\n");
  });
});
```

```
package main
import ("bufio"
     "fmt"
     "net"
     "strings")
func main() {
     fmt.Println("Launching server...")
     ln, _ := net.Listen("tcp", ":8081") // listen
on all interfaces
     for {
          conn, _ := ln.Accept() // accept connection
on port
          message, _ :=
bufio.NewReader(conn).ReadString('\n') //read line
          fmt.Print("Message Received:",
string(message))
          newMessage := strings.ToUpper(message)
//change to upper
          conn.Write([]byte(newMessage + "\n"))
//send upper string back
     }
}
```

```
PING sydney.edu.au (129.78.5.8) 56(84) bytes of data.
64 bytes from scilearn.sydney.edu.au (129.78.5.8): icmp_seq=1 ttl=233 time=307 ms
64 bytes from scilearn.sydney.edu.au (129.78.5.8): icmp_seq=2 ttl=233 time=305 ms
64 bytes from scilearn.sydney.edu.au (129.78.5.8): icmp_seq=3 ttl=233 time=305 ms
```

Layer 4 – TCP + TLS

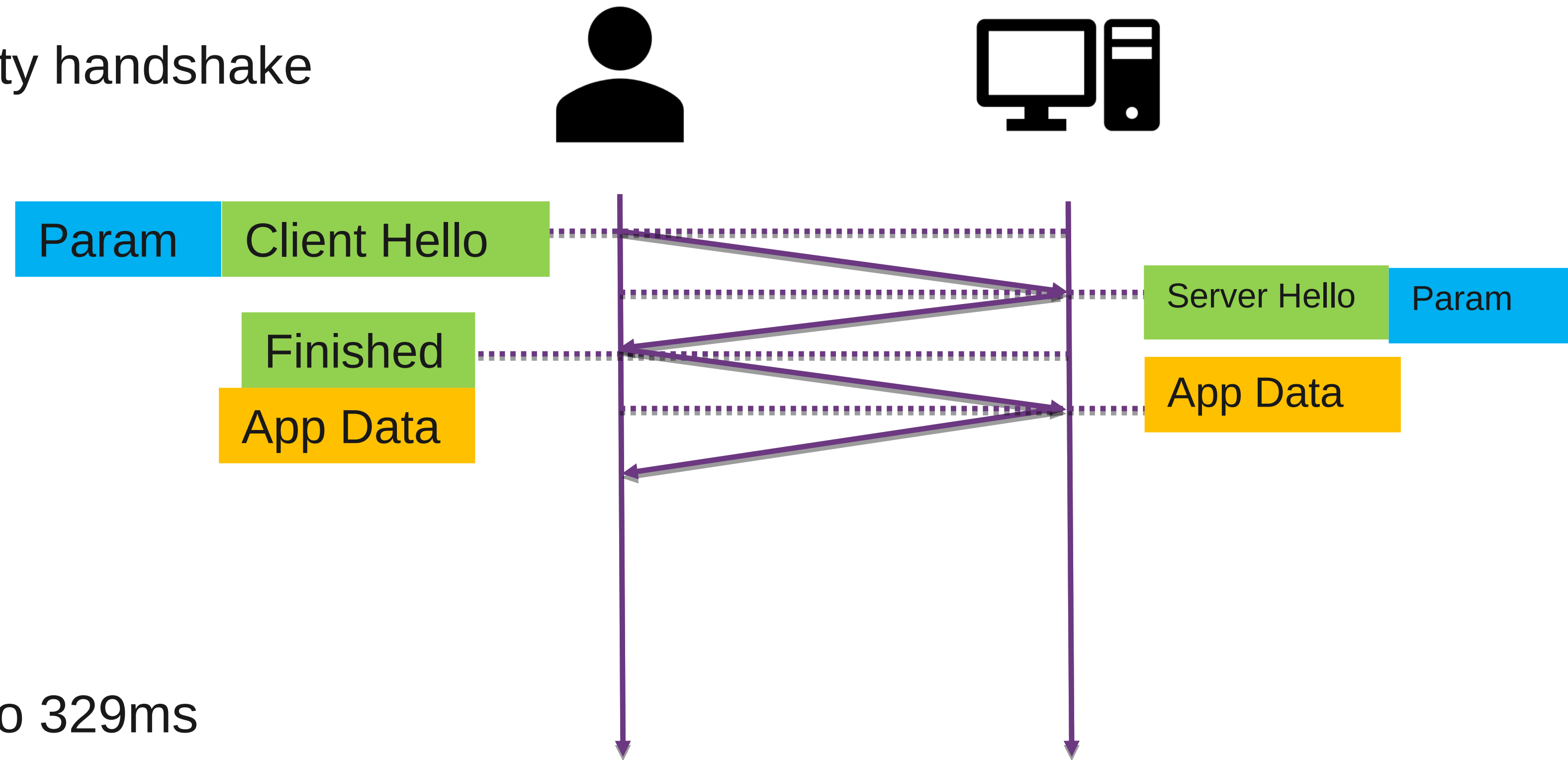
- Ping to Australia: 329ms
  - One way ~ 165ms
- TCP + TLS handshake:
  - 3RTT = 987ms! No data sent yet
- TLS 1.3, finished Aug 2018
  - 1 RTT instead of 2
    - 1.) Client Hello, Key Share
    - 2.) Server Hello, key Share, Verify Certificate, Finished
  - 0 RTT possible, for previous connections, loosing perfect forward secrecy
  - 90% of browsers used already support it



OST

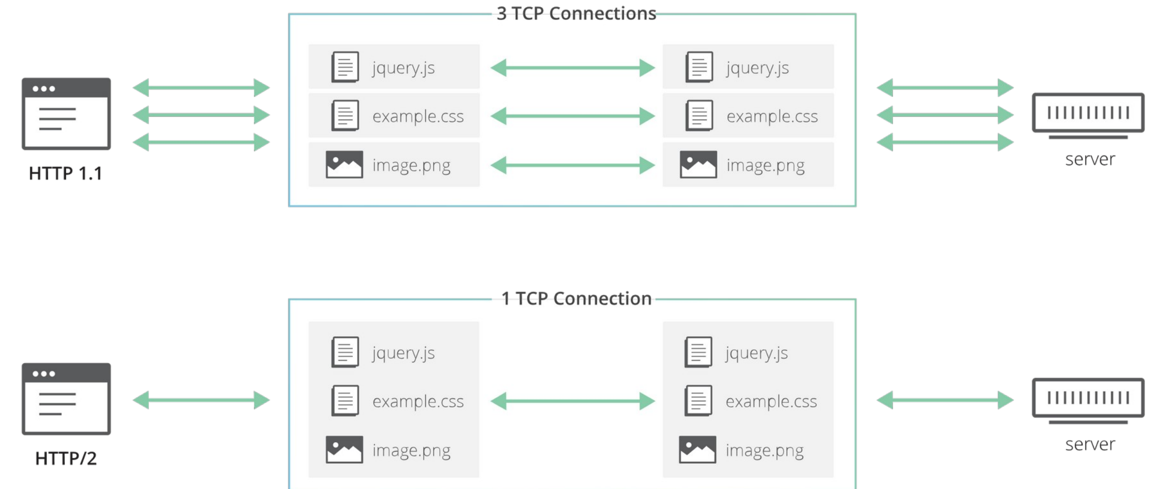
QUIC / HTTP/3

- QUIC: 1RTT connection + security handshake
  - For known connections: 0RTT
  - Built in security
  - “Google's 'QUIC' TCP alternative slow to excite anyone outside Google” [link] (9%, 25%, 75%)
    - Facebook
    - Cloudflare, state of HTTP
- Example Australia: from 987ms to 329ms

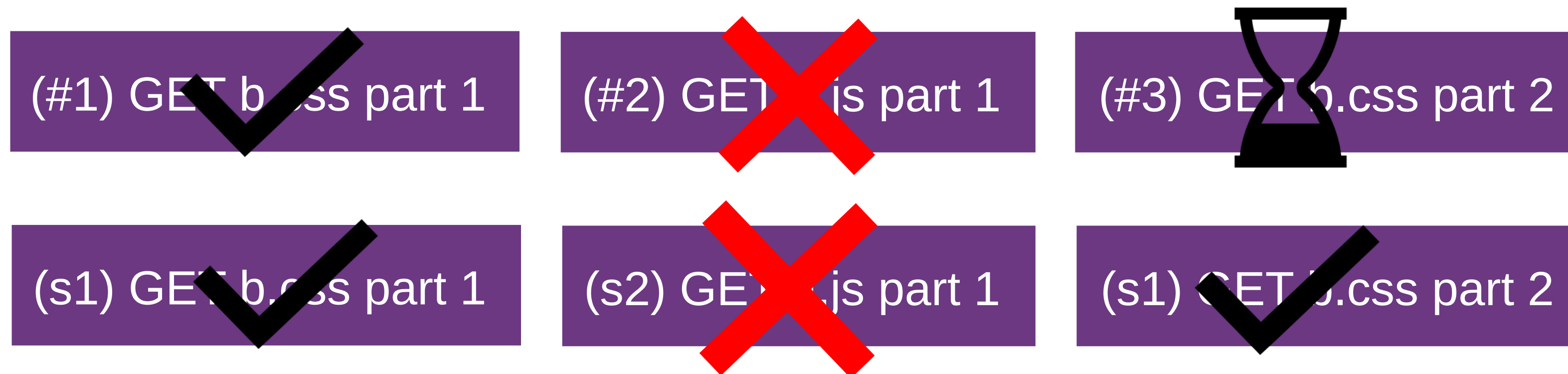


QUIC / HTTP3

- Multiplexing in HTTP/2
  - [HTTP/1 → HTTP/2]
- HTTP/2: Head-of-line blocking
  - One packet loss, TCP needs to be ordered
  - QUIC can multiplex requests: one stream does not affect others
- HTTP/3 is great, but…
  - NAT → SYN, ACK, FIN, conntrack knows when connection ends, not with QUIC, timeouts, new entries, many entries
  - HTTP header compression, referencing previous headers
  - Many TCP [optimizations]



source: https://blog.cloudflare.com/the-road-to-quic/



OST

Lecture 9

OST

Examples

- Static site generation: dsl.i.ost.ch
  - Componets: nginx
  - Java daemon who reacts on file changes in a director. If markdown file changes → create HTML, copy it to nginx directory
- Server side rendering (e.g., handlebarsjs)
  - Simple example: ssr.go (no template)
  - Components: go-based server
- SPA
  - Components: node server, go server
- Hydration
  - Best of both worlds, but adds complexity, needs JavaScript in the backend
  - Overview: source

Server ← - - - → Browser

	Server Rendering	"Static SSR"	SSR with (Re)hydration	CSR with Prerendering	Full CSR
Overview:	An application where input is navigation requests and the output is HTML in response to them.	Built as a Single Page App, but all pages prerendered to static HTML as a build step, and the JS is **removed**.	Built as a Single Page App. The server prerenders pages, but the full app is also booted on the client.	A Single Page App, where the initial shell/skeleton is prerendered to static HTML at build time.	A Single Page App. All logic, rendering and booting is done on the client. HTML is essentially just script & style tags.
Authoring:	Entirely server-side	Built as if client-side	Built as client-side	Client-side	Client-side
Rendering:	Dynamic HTML	Static HTML	Dynamic HTML **and** JS/DOM	Partial static HTML, then JS/DOM	Entirely JS/DOM
Server role:	Controls all aspects.	Delivers static HTML	Renders pages	Delivers static HTML	Delivers static HTML
Pros:	👍 TTI = FCP 👍 Fully streaming	👍 Fast TTFB 👍 TTI = FCP 👍 Fully streaming	👍 Flexible	👍 Flexible 👍 Fast TTFB	👍 Flexible 👍 Fast TTFB
Cons:	👎 Slow TTFB 👎 Inflexible	👎 Inflexible 👎 Leads to hydration	👎 Slow TTFB 👎 TTI >>> FCP 👎 Usually buffered	👎 TTI > FCP 👎 Limited streaming	👎 TTI >>> FCP 👎 No streaming
Scales via:	Infra size / cost	build/deploy size	Infra size & JS size	JS size	JS size
Examples:	Gmail HTML, Hacker News	Docusaurus, Netflix*	Next.js, Razzle, etc	Gatsby, Vuepress, etc	Most apps

Lecture 10

OST

Deployment Strategies

- Many strategies and variations [link, link, link]
- Rolling Deployment
  - New version is gradually deployed to replace the old version - without taking the entire system down at once
    - + Minimal downtime, low risk
    - - Complexity, longer deployment times
- Blue-Green Deployment
  - 2 environments, current prod (blue), current prod with new release (green). Test, then switch
    - + Instant rollback, 0 downtime
    - - 2 prod environments, keep data in sync
- Canary Releases
  - Canary in a coal mine - new version to a small group of users or servers first, if all goes well, more users
    - + Risk reduction, user feedback
    - - Complexity, inconsistencies
- Feature Toggle
  - Fine grained canary, set feature for specific users
    - + More risk reduction, specific user feedback
    - - Increase complexity of codebase, config management
- Big Bang
  - Deploy everything at once
    - + Simple
    - - High risk, limited rollback