

جامعة الإسكندرية
ALEXANDRIA
UNIVERSITY
كلية الحاسبات وعلوم البيانات

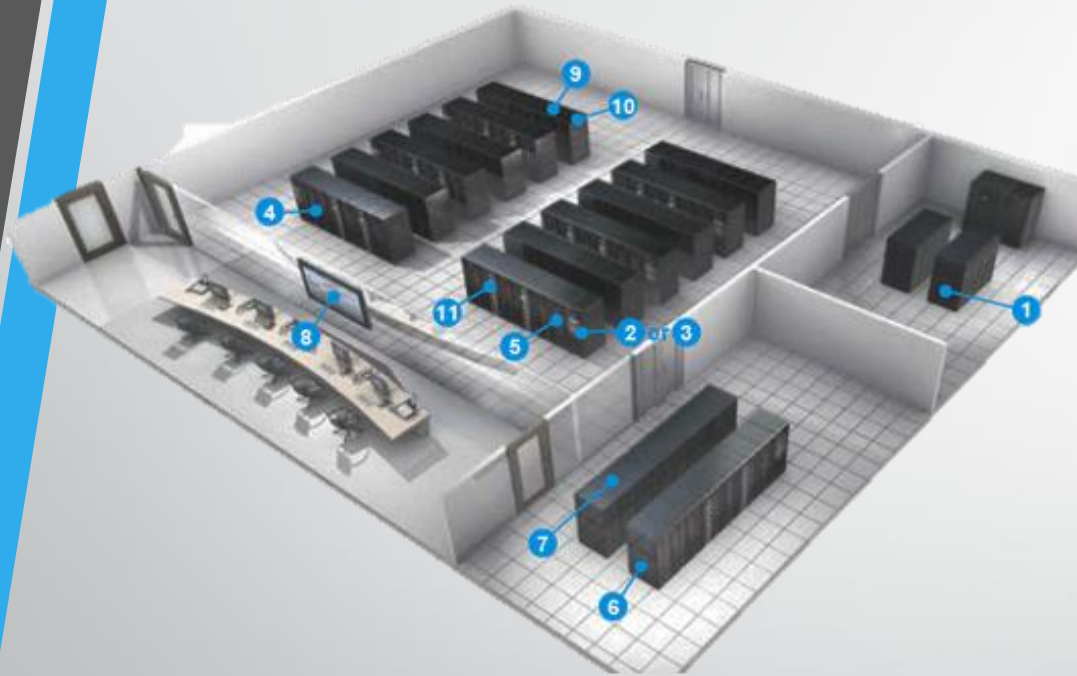


IOT Training

Eng. Mohamed Wael

Traditional Cloud

Capex



Some Problems



If you get hardware devices that can not fulfill your needs?



How to scale up or scale down your computing resources?



Devices may be ideal (not used) for a long time but consuming power and passing working hour.




What about cooling?



What about maintenance?

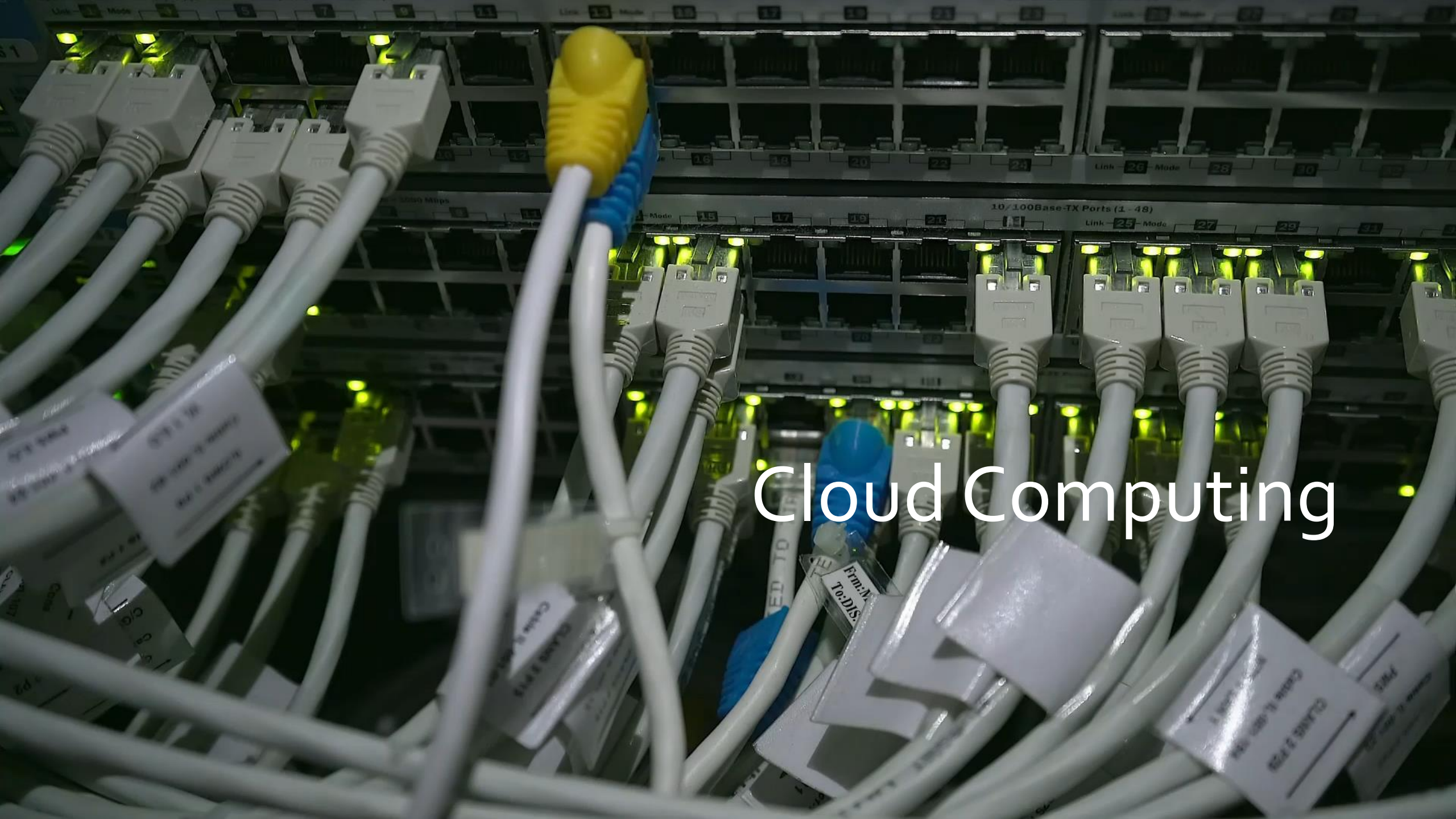


Meaning of Cloud



Cloud refers to something which you don't know exactly the physical location.

Cloud is something that is remote and can be accessed through the Internet.



Cloud Computing

cloud computing

- is the delivery of computing services—including servers, storage, databases, networking, software, analytics, and intelligence—over the internet (“the cloud”) to offer faster innovation, flexible resources, and economies of scale. You typically pay only for cloud services you use, helping you lower your operating costs, run your infrastructure more efficiently, and scale as your business needs change.

Cloud Deployment Model

How is the service delivered to clients?

- **Private clouds**

are built, managed, and owned by a single organization and privately hosted in their own data centers, commonly known as “on-premises” or “on-prem.” They provide greater control, security, and management of data while still enabling internal users to benefit from a shared pool of compute, storage, and network resources.

- **Public clouds**

are owned and operated by third-party [cloud service providers](#), which deliver computing resources like servers and storage over the internet like Microsoft Azure, GCP, AWS and Firebase.

Other types

- Hybrid clouds

combine public and private clouds, bound together by technology that allows data and applications to be shared between them. By allowing data and applications to move between private and public clouds, a hybrid cloud gives your business greater flexibility and more deployment option

- Multicloud

refers to an approach in cloud computing where an organization uses multiple cloud service providers to meet its computing needs and objectives. Instead of relying on a single cloud provider, multicloud strategies involve distributing workloads and applications across multiple cloud platforms, leveraging the strengths and capabilities of different providers.

Benefits of Cloud Computing

- **Cost**

Moving to the cloud helps companies optimize [IT costs](#). This is because cloud computing eliminates the capital expense of buying hardware and software and setting up and running onsite datacenters—the racks of servers, the round-the-clock electricity for power and cooling, and the IT experts for managing the infrastructure. It adds up fast.

- **Global scale**

The benefits of cloud computing services include the ability to scale elastically. In cloud speak, that means delivering the right amount of IT resources—for example, more or less computing power, storage

- **Performance**

The biggest cloud computing services run on a worldwide network of secure datacenters, which are regularly upgraded to the latest generation of fast and efficient computing hardware

- **Reliability**

Cloud computing makes data backup, [disaster recovery](#), and business continuity easier and less expensive because data can be mirrored at multiple redundant sites on the cloud provider's network.

- **Productivity**

Onsite datacenters typically require a lot of “racking and stacking”—hardware setup, software patching, and other time-consuming IT management chores. Cloud computing removes the need for many of these task

Disadvantages



Potential privacy and security risks of putting valuable data on someone else's system in an unknown location



What happens if your supplier suddenly decides to stop supporting a product or system you've come to depend on?



Public Cloud

Private Cloud

Hybrid cloud

No capital expenditures to scale up

Organizations have complete control over resources and security

Applications can be quickly provisioned and deprovisioned

Applications can be quickly provisioned and deprovisioned

Data is not collocated with other organizations' data

Organizations determine where to run their applications

Organizations pay only for what they use

Hardware must be purchased for startup and maintenance

Organizations control security, compliance, or legal requirements

Organizations don't have complete control over resources and security


Organizations are responsible for hardware maintenance and updates



- CapEx is typically a one-time, up-front expenditure to purchase or secure tangible resources. A new building, repaving the parking lot, building a datacenter, or buying a company vehicle are examples of CapEx.
- In contrast, OpEx is spending money on services or products over time. Renting a convention center, leasing a company vehicle, or signing up for cloud services are all examples of OpEx.
- Cloud computing falls under OpEx because cloud computing operates on a consumption-based model. With cloud computing, you don't pay for the physical infrastructure, the electricity, the security, or anything else associated with maintaining a datacenter. Instead, you pay for the IT resources you use. If you don't use any IT resources this month, you don't pay for any IT resources.

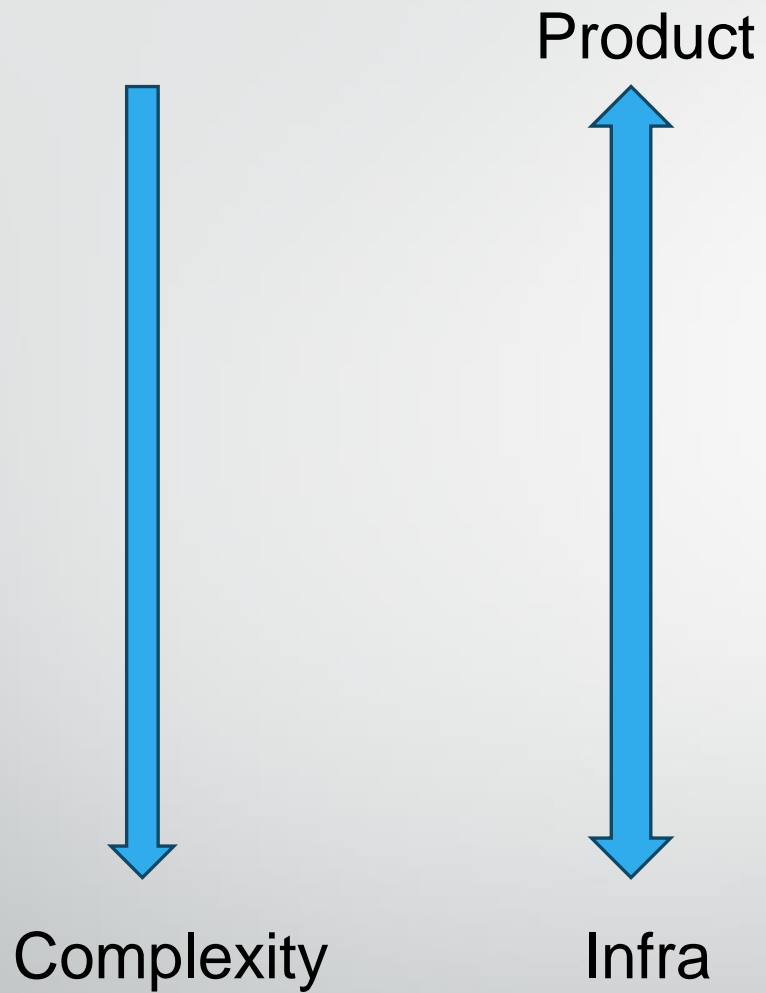
This consumption-based model has many benefits, including:

- No need to purchase and manage costly infrastructure that users might not use to its fullest potential.
- The ability to pay for more resources when they're needed.
- The ability to stop paying for resources that are no longer needed.



What is the service
delivered to clients?

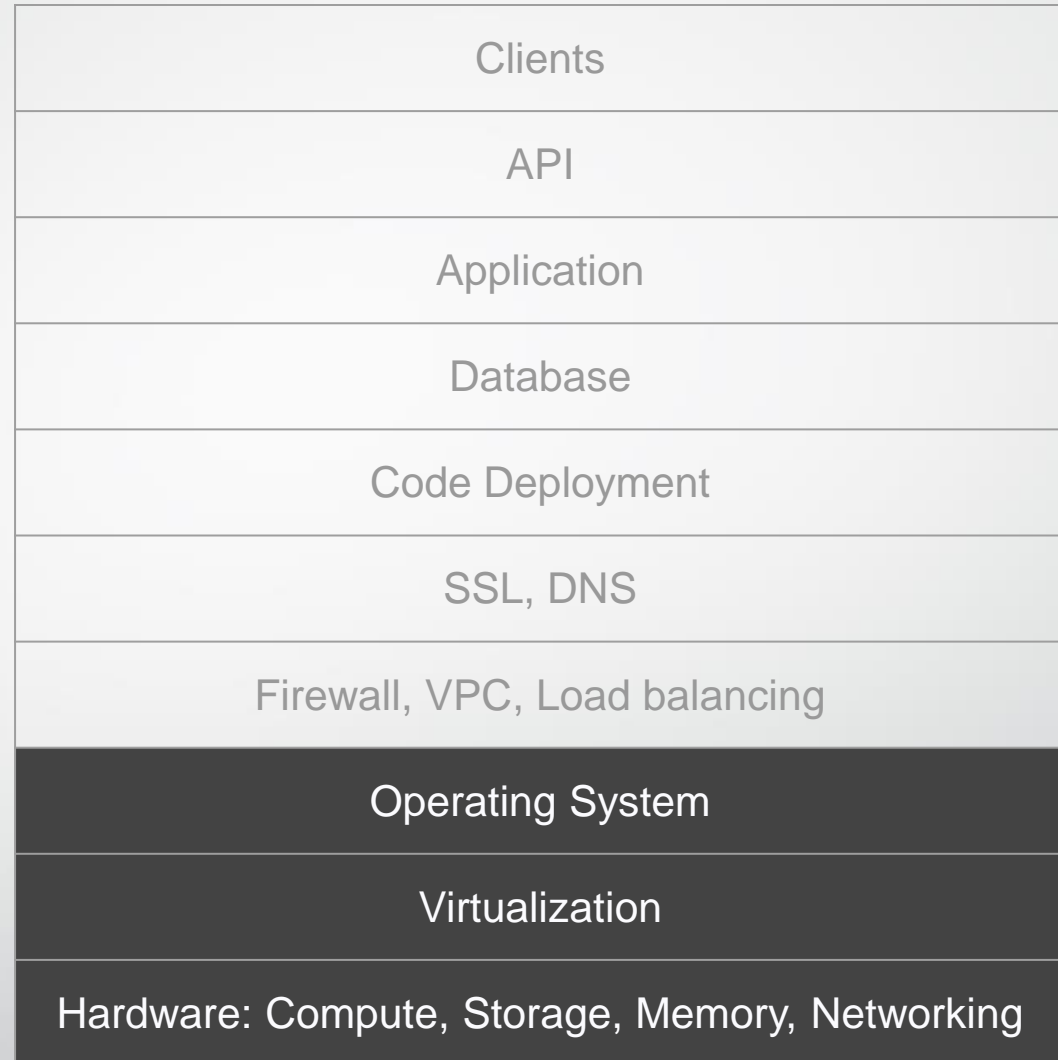
Cloud service “delivery” model



Clients
API
Application
Database
Code Deployment
SSL, DNS
Firewall, VPC, Load balancing
Operating System
Virtualization
Hardware: Compute, Storage, Memory, Networking

IaaS (Abstract common pieces)

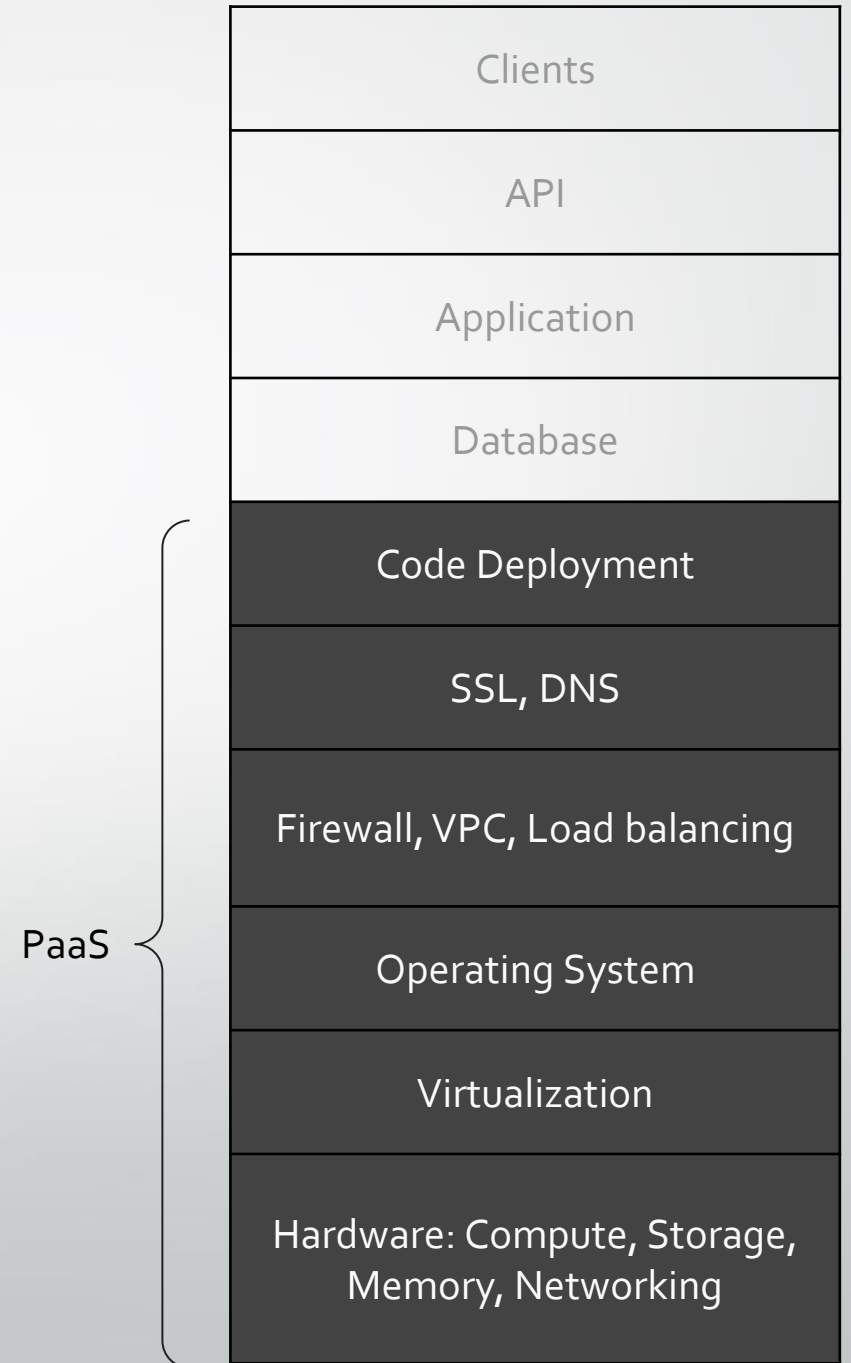
- More Flexibility
- Less Abstraction



PaaS: Abstract more common pieces

- Everyone needs a server framework
- Deploying is a pain
- SSL
- Service model:
 - Provision server automatically
 - Add load balancers
 - Add SSL certificate
 - Pick up code from cloud repository, build, and deploy it
- PaaS for applications

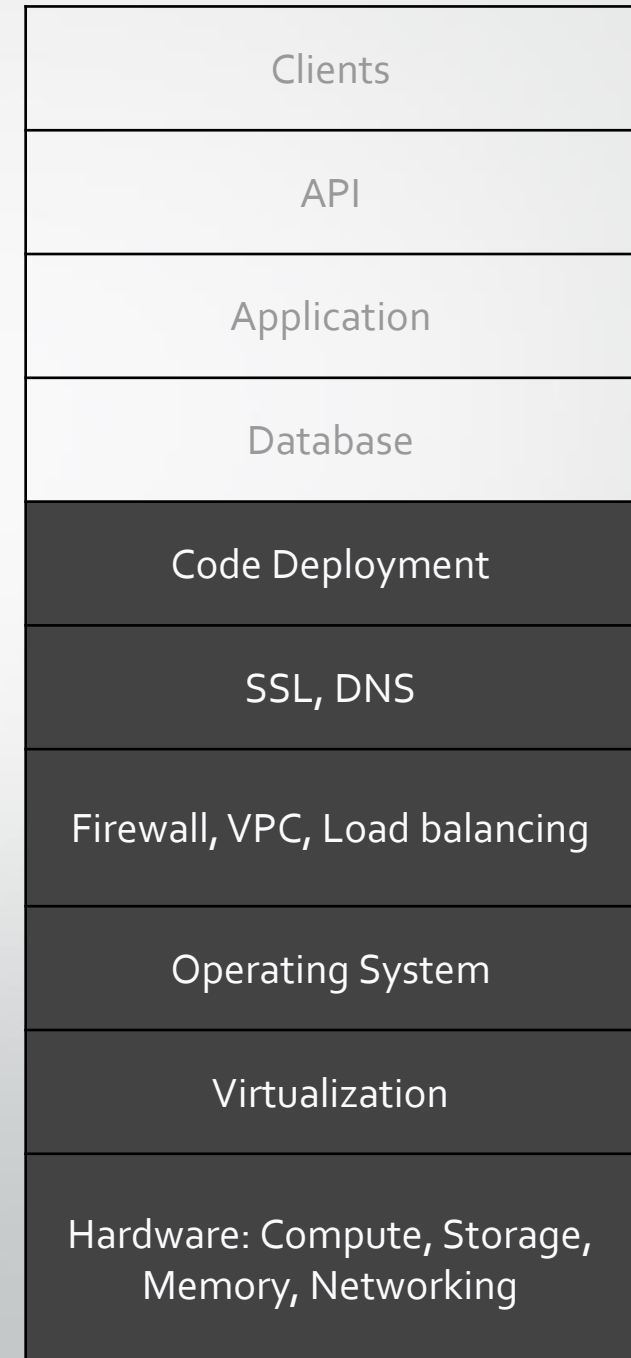
** Many other PaaS solutions, app is just most common*



PaaS

- Auto-deployment
 - Connect to your cloud repository
 - Choose deployment branch and trigger
- Pick language, framework, runtime
 - Server configuration
 - Disk storage as a service, by the GB
- Security
 - Configure firewalls, DNS, IP rotations
 - Add SSL certificate
- Scaling
 - Easy to scale up/down deployment

PaaS





✓ Node.js detected

Source Directory ?

/server

Edit

Type

Web Services

Edit

Environment Variables

Edit

No environment variables defined yet.

Build Command

No build command defined

Edit

Run Command

\$ npm start

Edit

HTTP Port

8080

Edit

HTTP Request Routes

Edit

/

Step 1 of 4

Choose Source

Note: We will request access only to repos or containers you choose.
You can revoke access through the source at any time.



GitHub

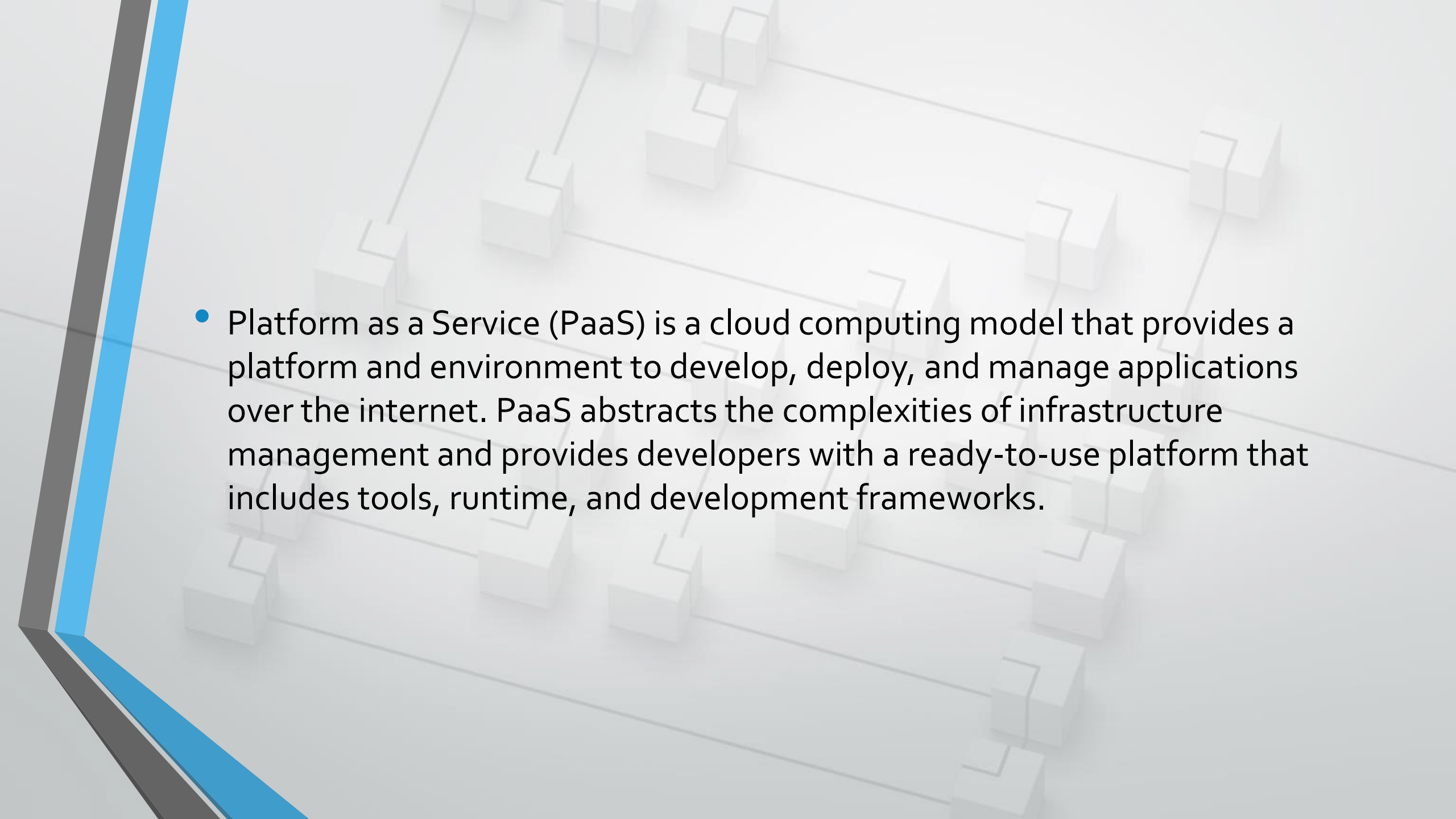


GitLab ↗



Docker Hub

[Create a Registry on DigitalOcean Container Registry](#)

- 
- Platform as a Service (PaaS) is a cloud computing model that provides a platform and environment to develop, deploy, and manage applications over the internet. PaaS abstracts the complexities of infrastructure management and provides developers with a ready-to-use platform that includes tools, runtime, and development frameworks.

Other PaaS Examples

- Database as a service
- Machine learning as a service
- Tracing/logging as a service
- SMS as a service
- ...

Lots of tech companies offer part of their stack as a PaaS

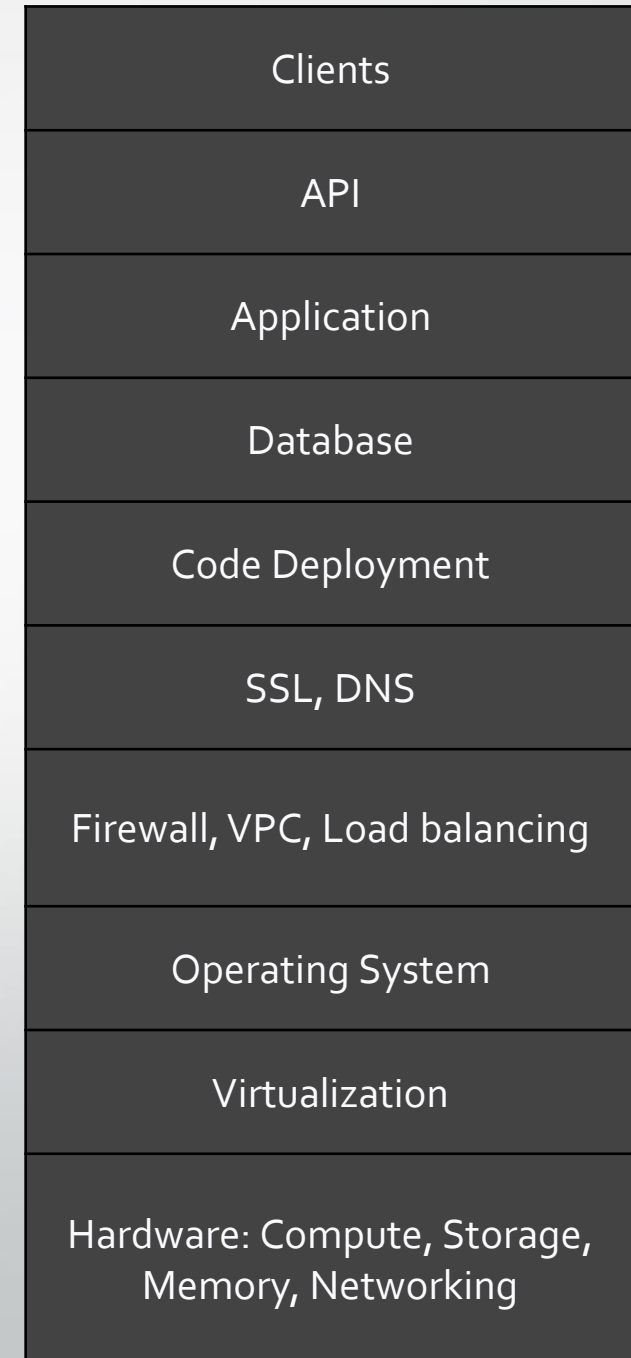
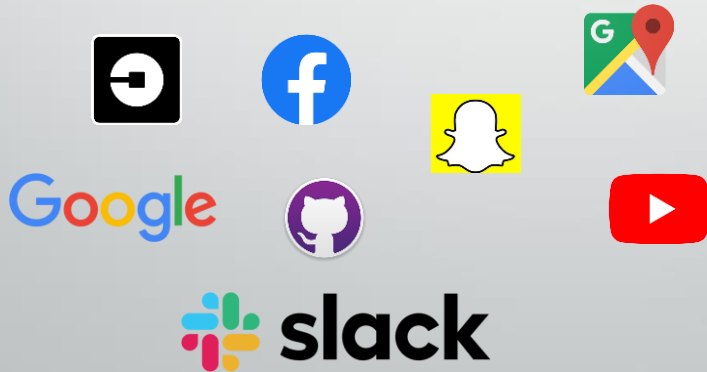
Uber: SaaS

Uber's last mile delivery: PaaS

Developers/other tech companies can use Uber's API to delivery anything

Software as a Service

- Consumer facing applications
- Usually, accessible using web or mobile client
- Expected to be scalable, secure, and available




High availability

- When you're deploying an application, a service, or any IT resources, it's important the resources are available when needed. High availability focuses on ensuring maximum availability, regardless of disruptions or events that may occur.
- When you're architecting your solution, you'll need to account for service availability guarantees. Azure is a highly available cloud environment with uptime guarantees depending on the service. These guarantees are part of the service-level agreements (SLAs).



Scalability

- Another major benefit of cloud computing is the scalability of cloud resources. Scalability refers to the ability to adjust resources to meet demand. If you suddenly experience peak traffic and your systems are overwhelmed, the ability to scale means you can add more resources to better handle the increased demand.
- 




Vertical scaling

- With vertical scaling, if you were developing an app and you needed more processing power, you could vertically scale up to add more CPUs or RAM to the virtual machine. Conversely, if you realized you had over-specified the needs, you could vertically scale down by lowering the CPU or RAM specifications.

Horizontal scaling

- With horizontal scaling, if you suddenly experienced a steep jump in demand, your deployed resources could be scaled out (either automatically or manually). For example, you could add additional virtual machines or containers, scaling out. In the same manner, if there was a significant drop in demand, deployed resources could be scaled in (either automatically or manually), scaling in.



Management in the cloud

Management in the cloud speaks to how you're able to manage your cloud environment and resources. You can manage these:


- Through a web portal.
- Using a command line interface.
- Using APIs.
- Using PowerShell.

Physical infrastructure

- The physical infrastructure for something like Azure starts with datacenters. Conceptually, the datacenters are the same as large corporate datacenters. They're facilities with resources arranged in racks, with dedicated power, cooling, and networking infrastructure.
- As a global cloud provider, Azure has datacenters around the world. However, these individual datacenters aren't directly accessible. Datacenters are grouped into Azure Regions or Azure Availability Zones that are designed to help you achieve resiliency and reliability for your business-critical workloads.

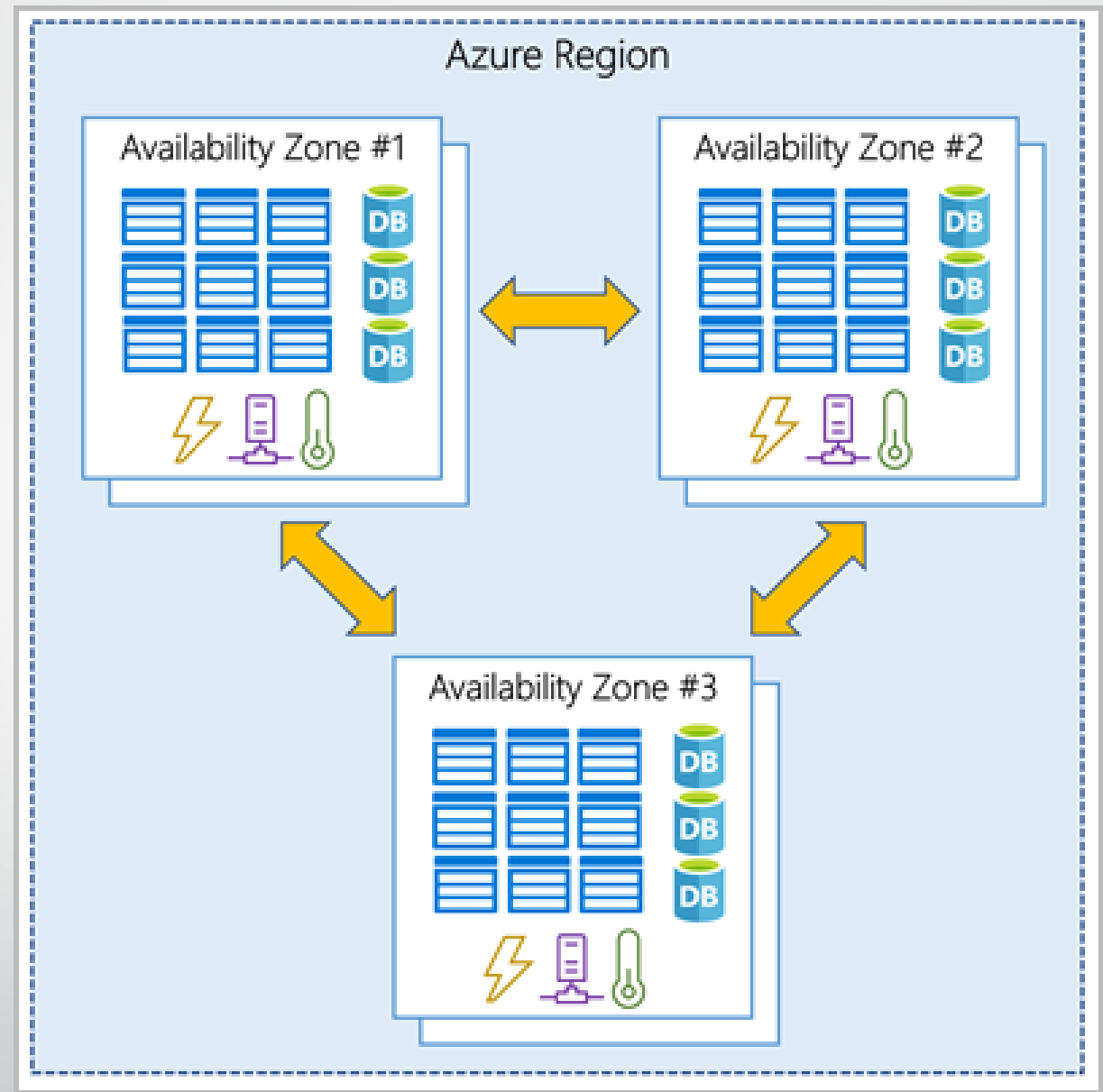
Regions

- A region is a geographical area on the planet that contains at least one, but potentially multiple datacenters that are nearby and networked together with a low-latency network. Azure intelligently assigns and controls the resources within each region to ensure workloads are appropriately balanced.



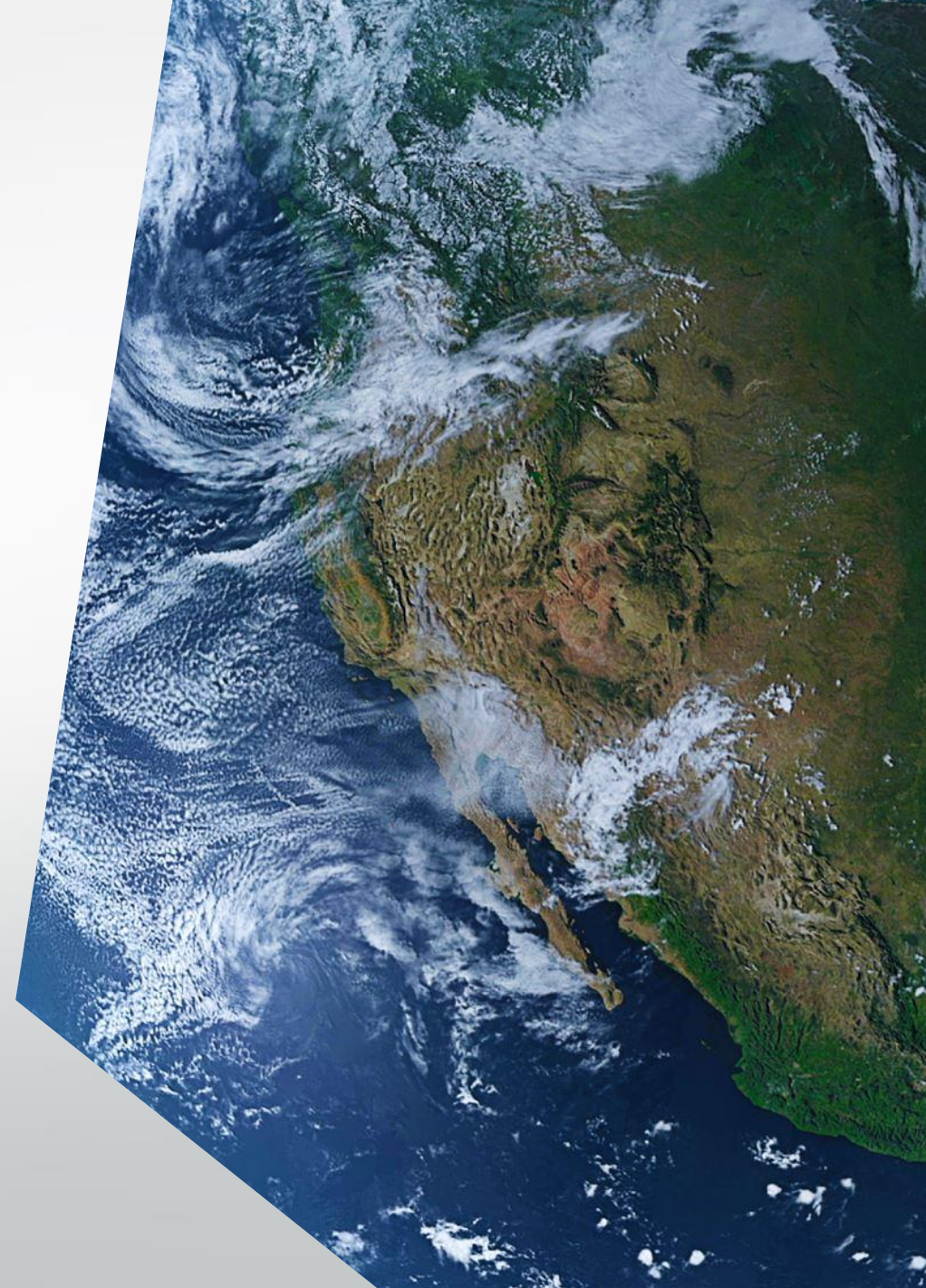
Availability Zones

- Availability zones are physically separate datacenters within an Azure region. Each availability zone is made up of one or more datacenters equipped with independent power, cooling, and networking. An availability zone is set up to be an isolation boundary. If one zone goes down, the other continues working. Availability zones are connected through high-speed, private fiber-optic networks.



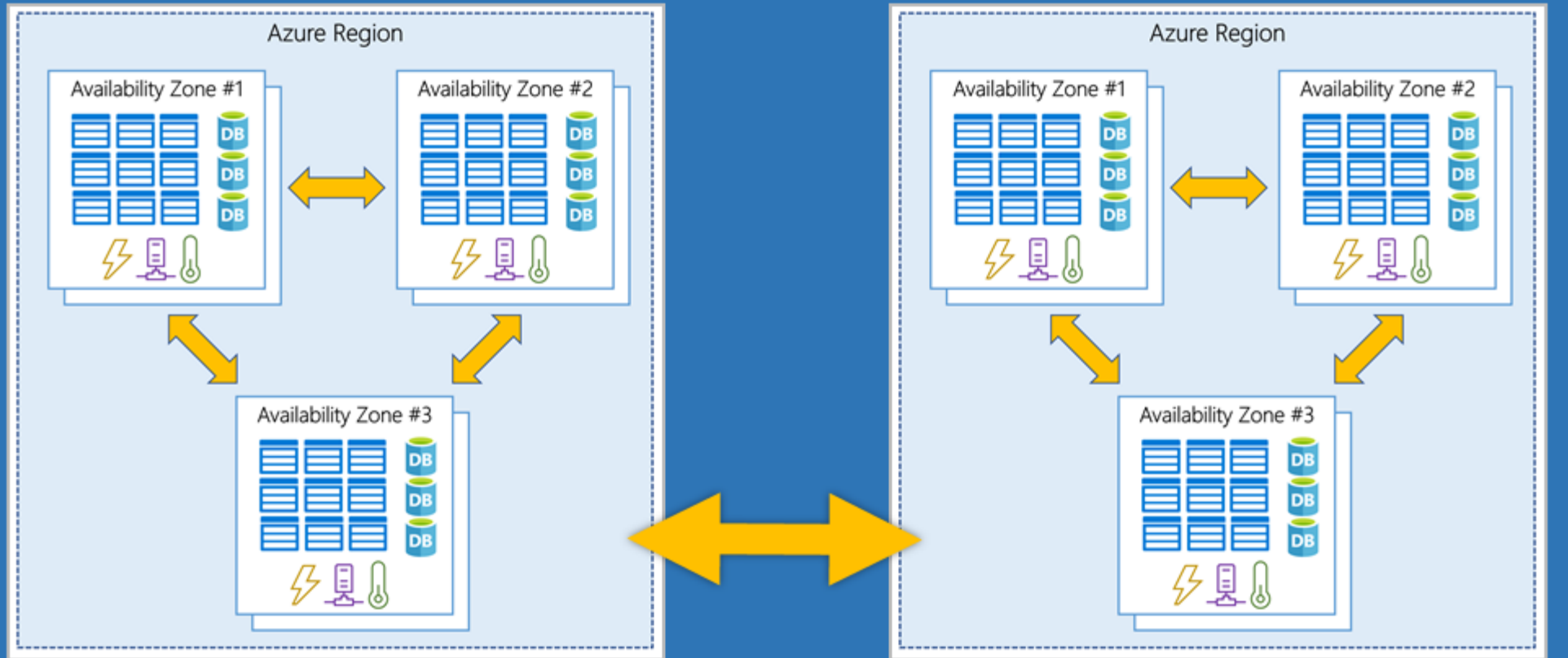
Region pairs

- Most Azure regions are paired with another region within the same geography (such as US, Europe, or Asia) at least 300 miles away. This approach allows for the replication of resources across a geography that helps reduce the likelihood of interruptions because of events such as natural disasters, civil unrest, power outages, or physical network outages that affect an entire region. For example, if a region in a pair was affected by a natural disaster, services would automatically fail over to the other region in its region pair.



Geography

Region Pair



An abstract digital graphic on the left side of the slide. It features several blue, three-dimensional cubes or rectangular blocks arranged in a staggered, isometric pattern. The surfaces of these blocks are covered with a dense, glowing blue binary code (0s and 1s). Bright blue light beams emanate from some of the block's openings and corners. A single red light dot is visible at the bottom left. The background is a gradient from dark blue to light grey.

resources

- A resource is the basic building block of something like Azure. Anything you create, provision, deploy, etc. is a resource. Virtual Machines (VMs), virtual networks, databases, cognitive services, etc. are all considered resources within Azure.

Serverless computing

- Serverless computing is a cloud computing model where developers can build and run applications without needing to manage the underlying server infrastructure

Key features of serverless computing include:

- Automatic Scaling: Serverless platforms can automatically scale your application based on the incoming traffic or events. You don't have to worry about provisioning additional servers during traffic spikes.
- Pay-as-You-Go Billing: You are billed based on the actual usage of your functions or code snippets, rather than paying for a fixed amount of server resources.
- No Server Management: Developers are relieved from the burden of server management, including operating system updates, patches, and server provisioning.
- High Availability

A 3D perspective illustration of a network topology. It features several white, ring-shaped nodes of varying sizes connected by thin white lines. The nodes are arranged in a non-linear fashion, with some acting as hubs. The entire scene is set against a light gray background with a subtle gradient. In the bottom-left corner, there are decorative diagonal stripes in blue and dark gray. The text "Network Applications" is overlaid on the right side of the image in a large, black, sans-serif font.

Network Applications

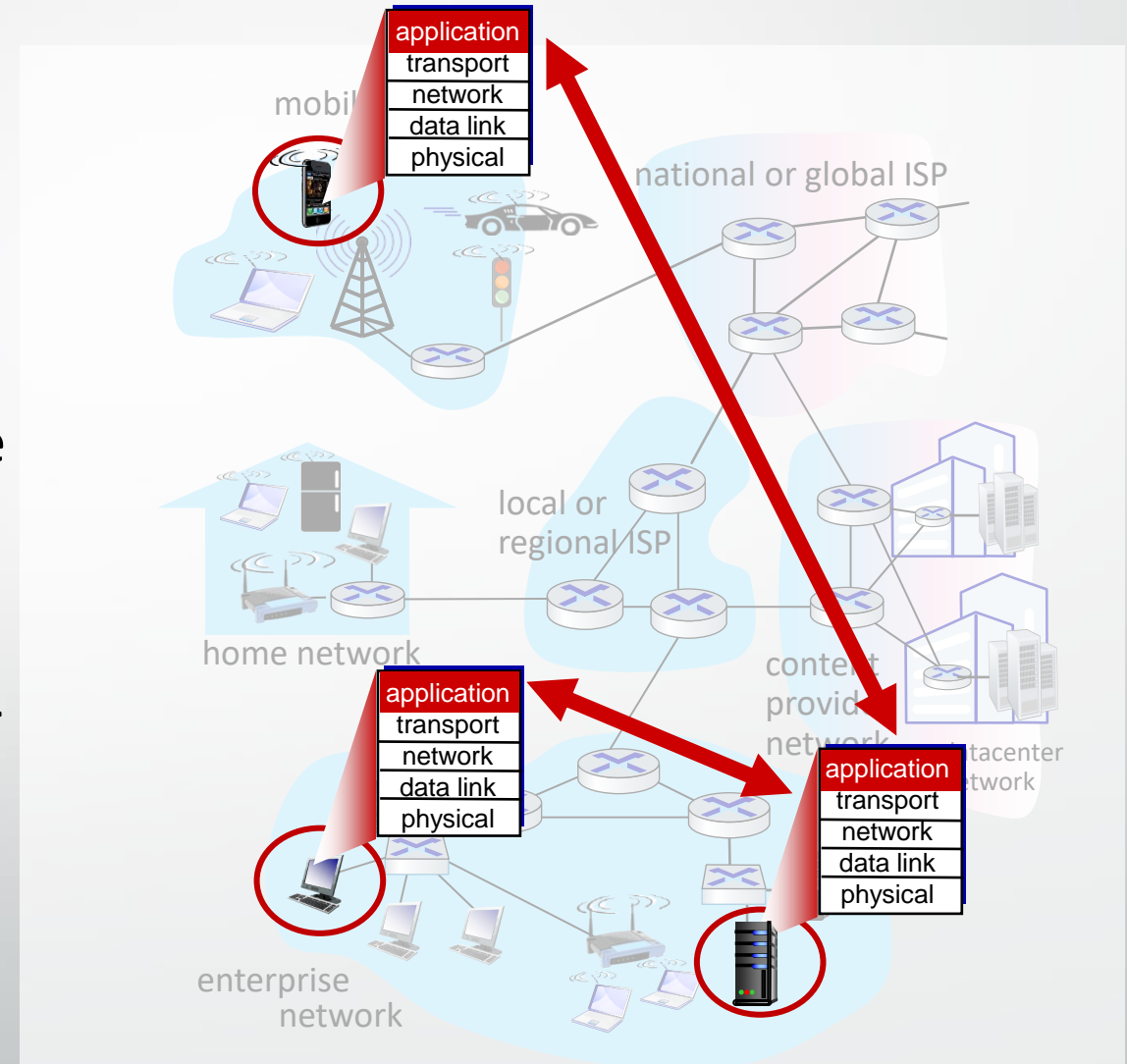
Creating a network app

write programs that:

- run on (different) end systems
- communicate over network
- e.g., web server software communicates with browser software

no need to write software for network-core devices

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation





Client-Server Architecture

A revolution in networking

Client- Server Architecture



Machines are expensive,
applications are
complex



Seperate the application
into two components



Expensive workload can
be done on the server



Clients call servers to
perform expensive tasks



Remote procedure call
(RPC) was born

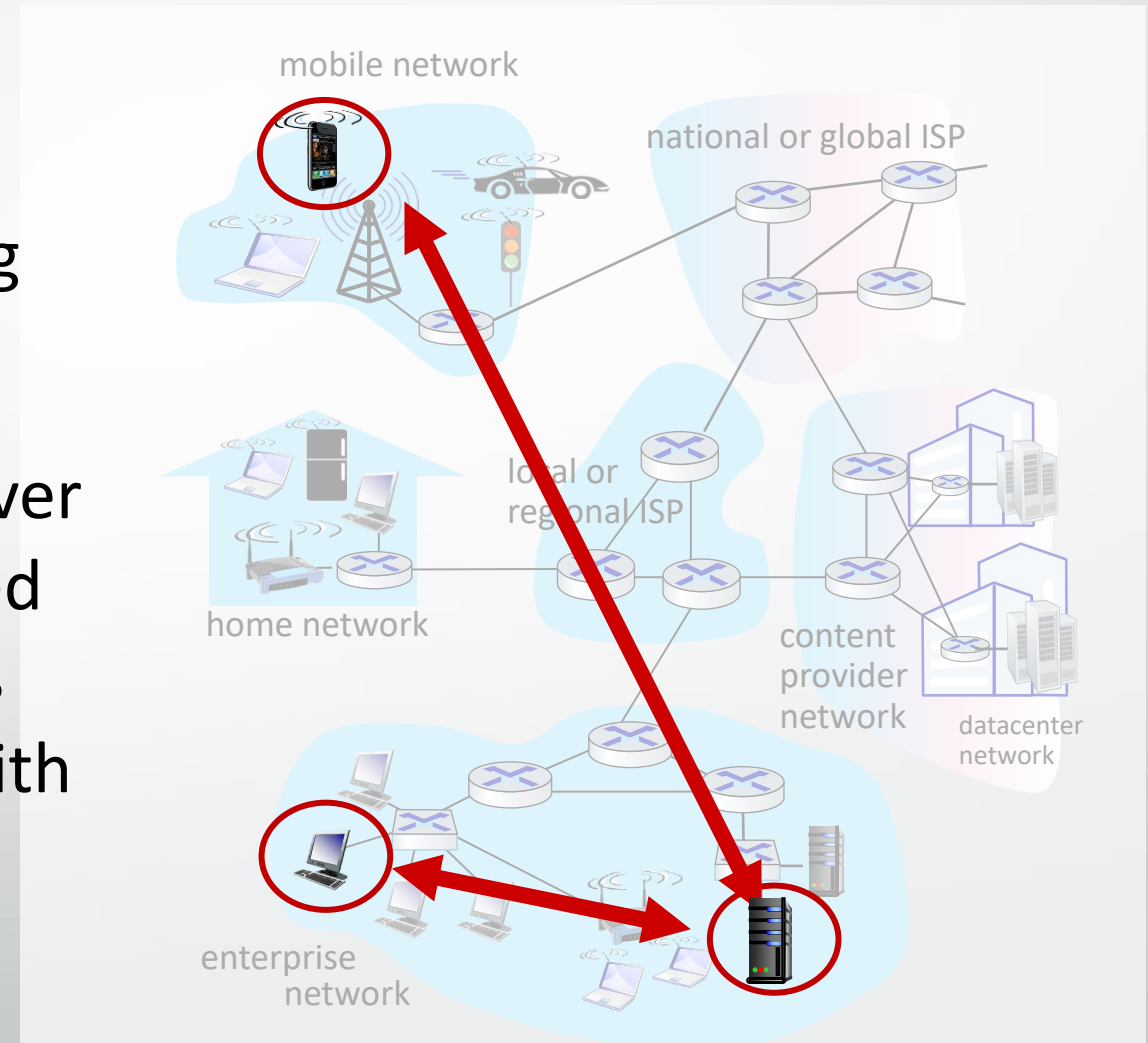
Client-server paradigm

server:

- always-on host
- permanent IP address
- often in data centers, for scaling

clients:

- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other
- examples: HTTP, IMAP, FTP



Processes communicating

process: program running within a host

- within same host, two processes communicate using **inter-process communication** (defined by OS)
- processes in different hosts communicate by exchanging **messages**

clients, servers

client process: process that initiates communication

server process: process that waits to be contacted

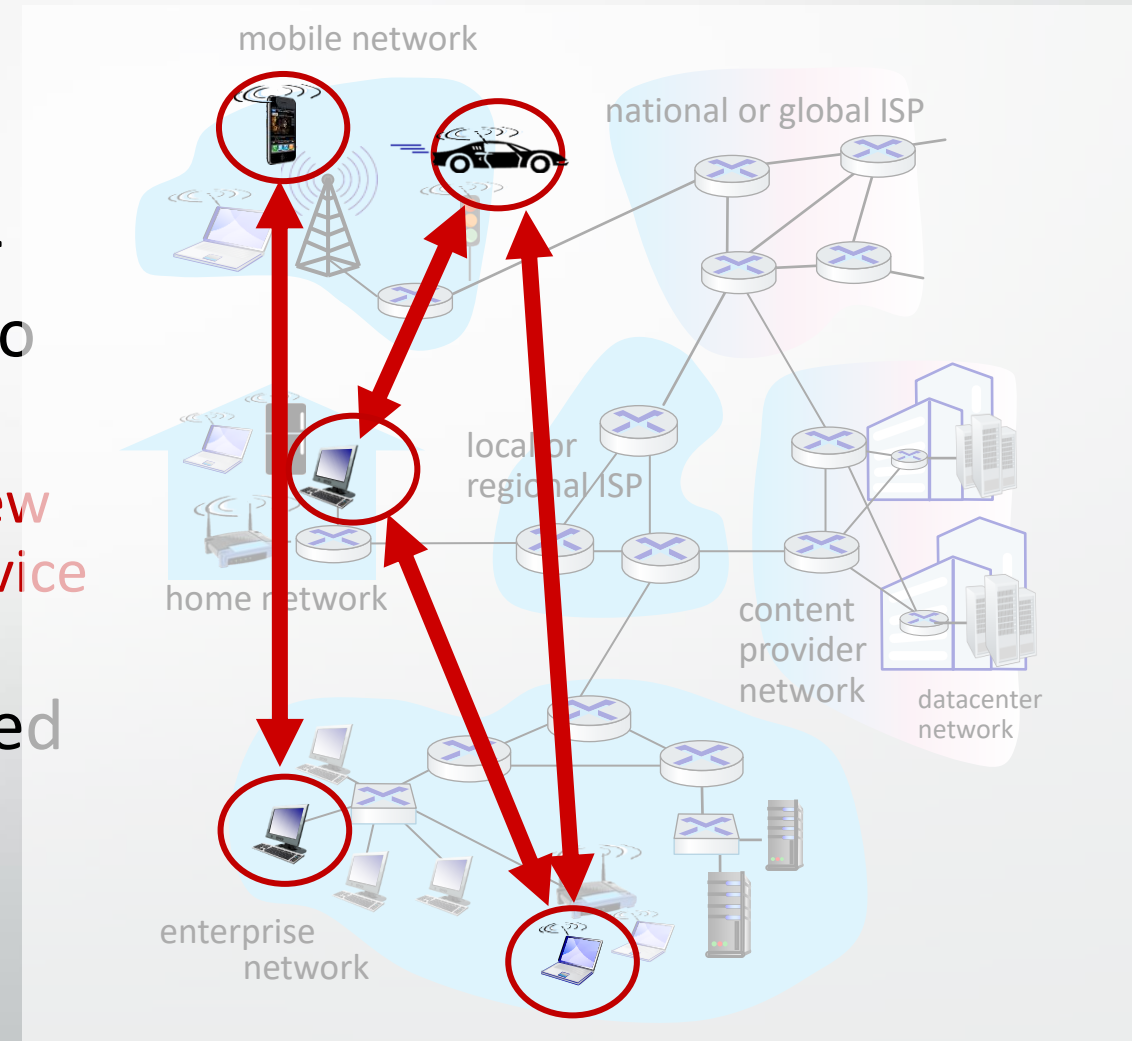
- note: applications with P2P architectures have client processes & server processes

Client-Server Architecture Benefits

- Servers have beefy hardware
- Clients have commodity hardware
- Clients can still perform lightweight tasks
- Clients no longer require dependencies
- However, we need a communication model

Peer-peer architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
 - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
 - complex management
- example: P2P file sharing



Protocol “layers” and reference models

Networks are complex,
with many “pieces”:

- hosts
- routers
- links of various media
- applications
- protocols
- hardware, software

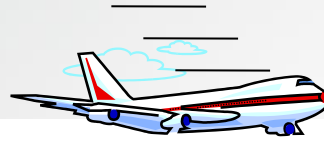
Question: is there any
hope of *organizing*
structure of network?

- and/or our *discussion*
of networks?

Why do we need a communication model?

- Agnostic applications
 - Without a standard model, your application must have knowledge of the underlying network medium
 - Imagine if you have to author different version of your apps so that it works on wifi vs 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

Example: organization of air travel



end-to-end transfer of person plus baggage

ticket (purchase)

baggage (check)

gates (load)

runway takeoff

airplane routing

ticket (complain)

baggage (claim)

gates (unload)

runway landing

airplane routing

airplane routing

How would you *define/discuss* the *system* of airline travel?

- a series of steps, involving many services

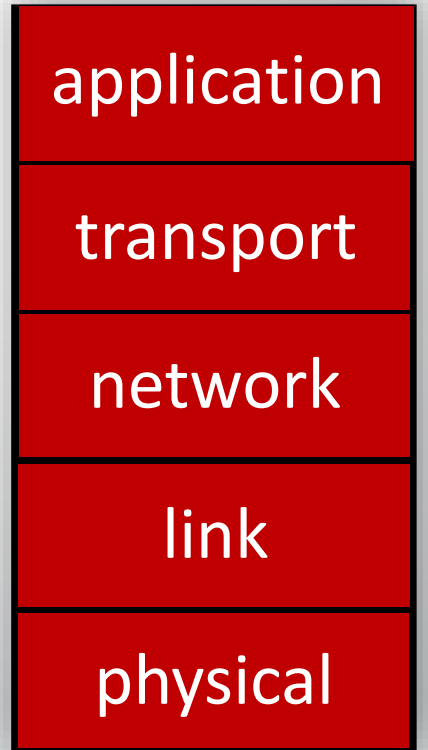
Why layering?

Approach to designing/discussing complex systems:

- explicit structure allows identification, relationship of system's pieces
 - layered *reference model* for discussion
- modularization eases maintenance, updating of system
 - change in layer's service *implementation*: transparent to rest of system
 - e.g., change in gate procedure doesn't affect rest of system

Layered Internet protocol stack

- *application*: supporting network applications
 - HTTP, IMAP, SMTP, DNS
- *transport*: process-process data transfer
 - TCP, UDP
- *network*: routing of datagrams from source to destination
 - IP, routing protocols
- *link*: data transfer between neighboring network elements
 - Ethernet, 802.11 (WiFi), PPP
- *physical*: bits “on the wire”



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

An application-layer protocol defines:

- **types of messages exchanged**,
 - e.g., request, response
- **message syntax**:
 - what fields in messages & how fields are delineated
- **message semantics**
 - meaning of information in fields
- **rules** for when and how processes send & respond to messages

open protocols:

- defined in RFCs, everyone has access to protocol definition
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype, Zoom

Protocol properties

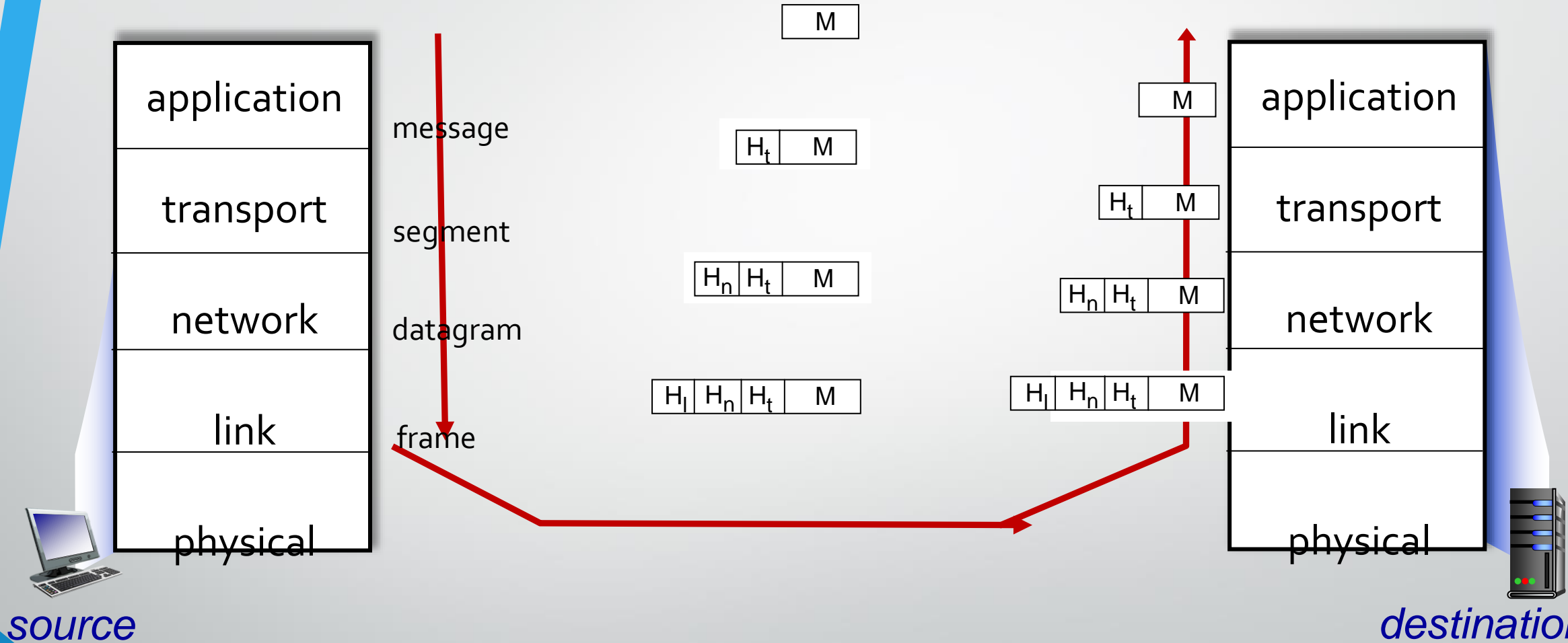
- Data format
 - Text based (plain text, JSON, XML)
 - Binary (protobuf, RESP, h2, h3)
- Transfer mode
 - Message based (UDP, HTTP)
 - Stream (TCP, WebRTC)
- Addressing system
 - DNS name, IP, MAC
- Directionality
 - Bidirectional (TCP)
 - Unidirectional (HTTP)
 - Full/Half duplex



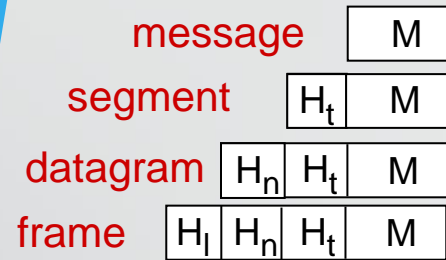
Protocol properties

- 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

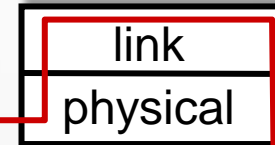
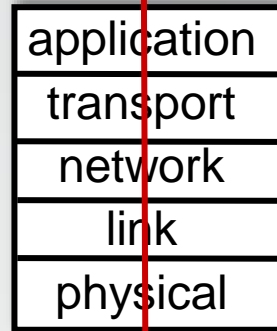
Services, Layering and Encapsulation



Encapsulation: an end-end view

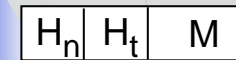
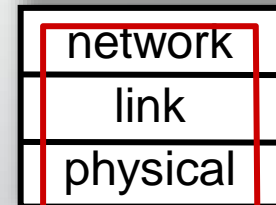
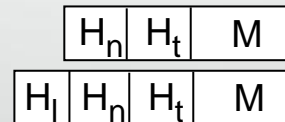
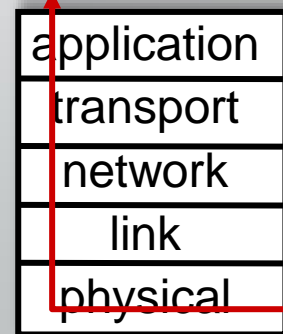
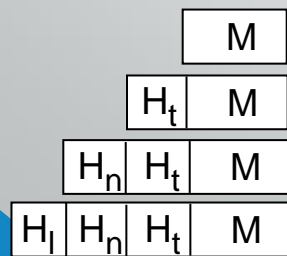


source



switch

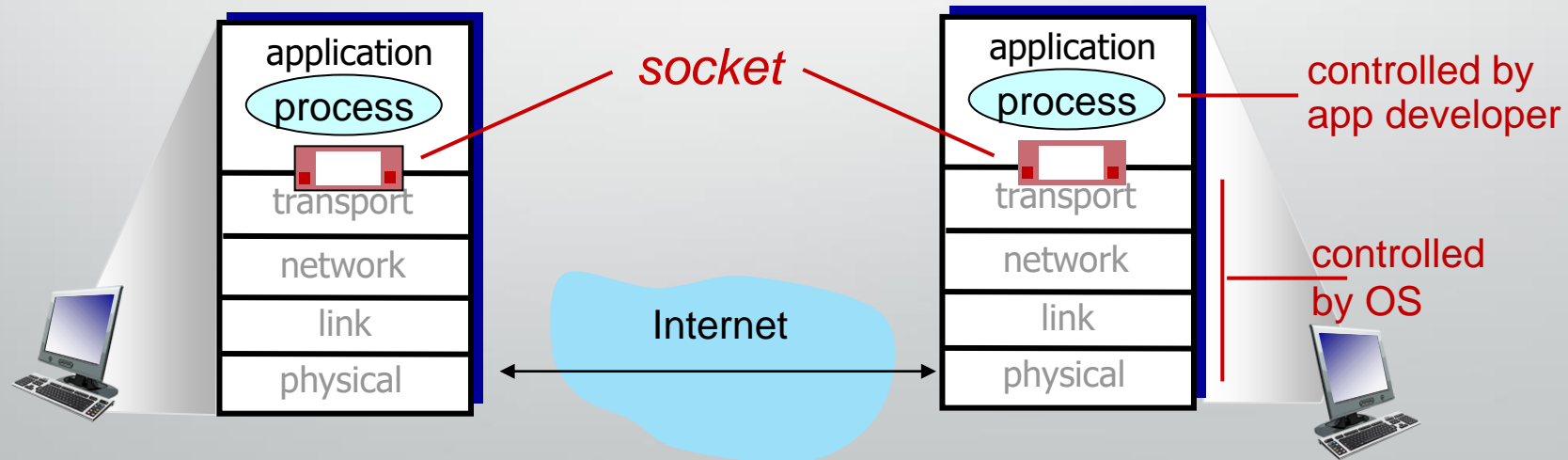
destination



router

Sockets

- process sends/receives messages to/from its **socket**
- socket analogous to door
 - sending process shoves message out door
 - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
 - two sockets involved: one on each side



Ports

- Each service must have a unique port number to be accessible on a given host. When a client or a peer needs to connect to a particular service or peer, it needs to specify not only the IP address but the port that the service process is listening on. Sockets are an API (application programming interface) for connecting to network services. A socket is bound to a port and allows a program to send and receive data with another program



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

Doesn't work everywhere

- Notification service
- Chatting application
- Very Long requests
- What if client disconnects?



Request/Response



t0

t2

Request
processing
time

t30

t32

Request

Response





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
 - A reading is just received(sent from device)
- Push model is good for certain cases

What is Push?

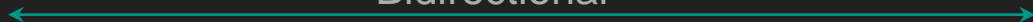
- Client connects to a server
- Server sends data to the client
- Client doesn't have to request anything
- Protocol must be bidirectional
- Used in Websockets



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



Short Polling

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

Where request/response isn't ideal

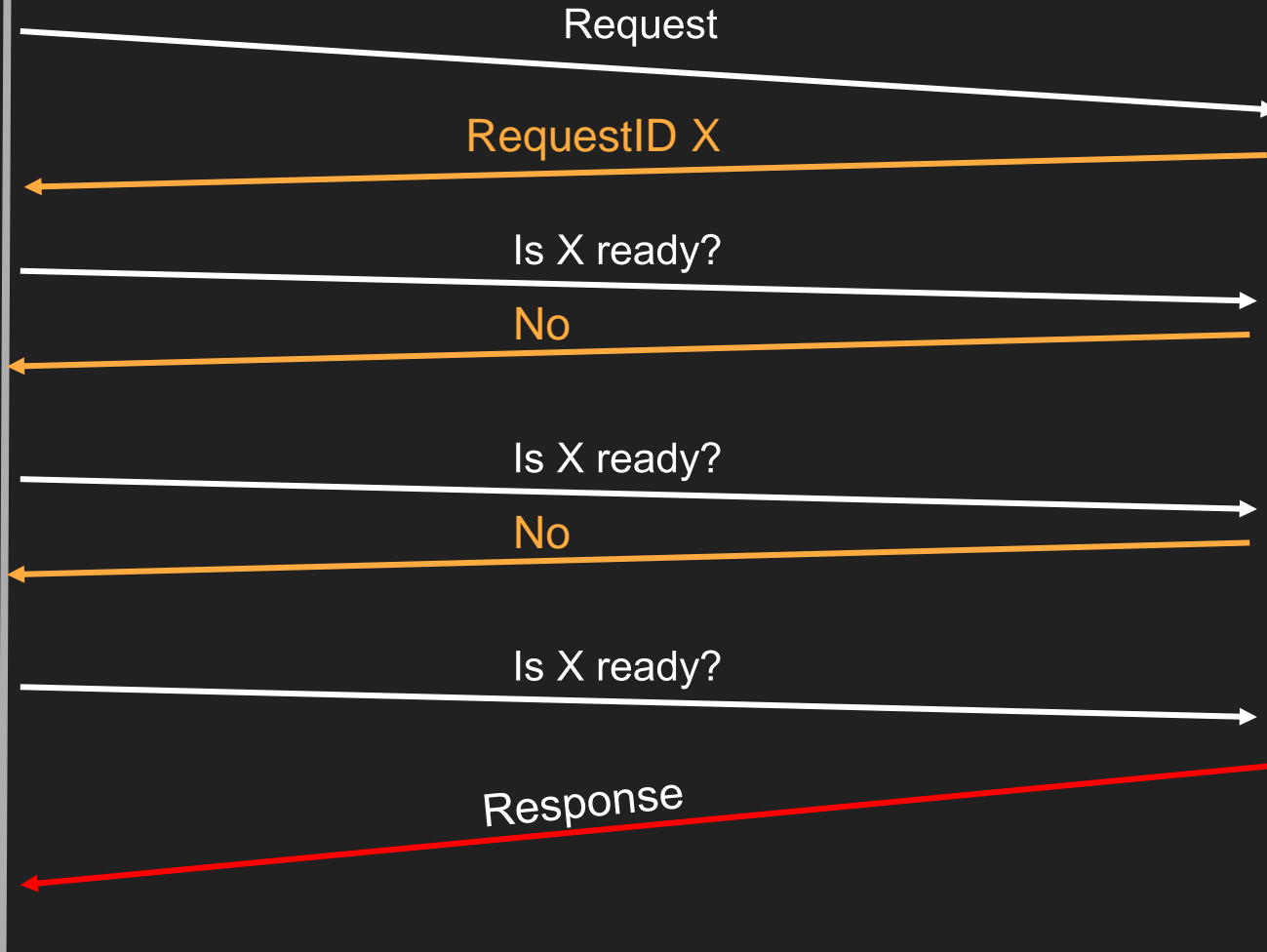
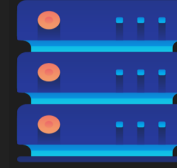
- A request takes long time to process
 - Upload a youtube video
- Polling is a good communication style

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

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



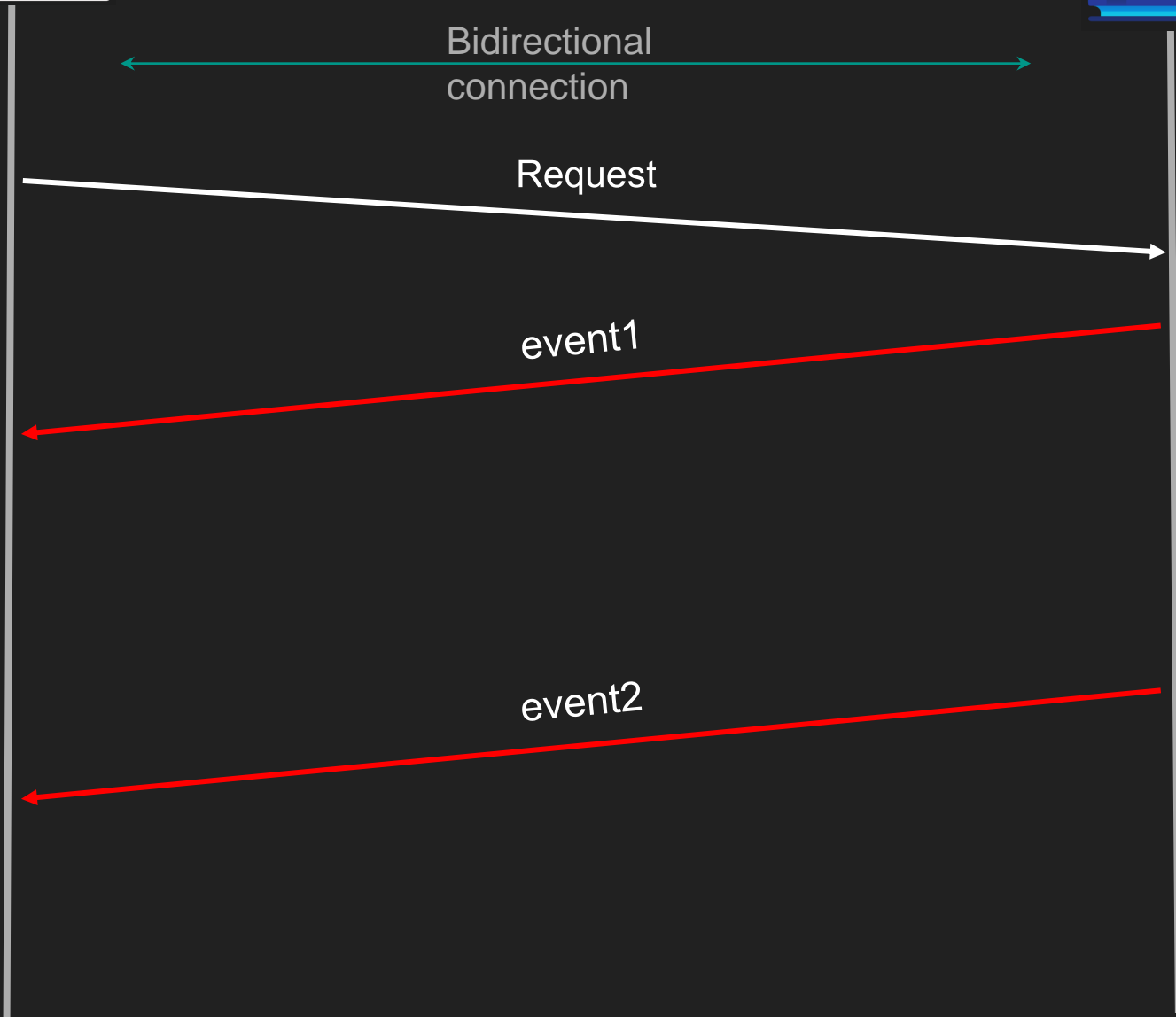
Bidirectional
connection

Request

*Backend gets
message*

event1

event2



Server Sent Events Pros and Cons

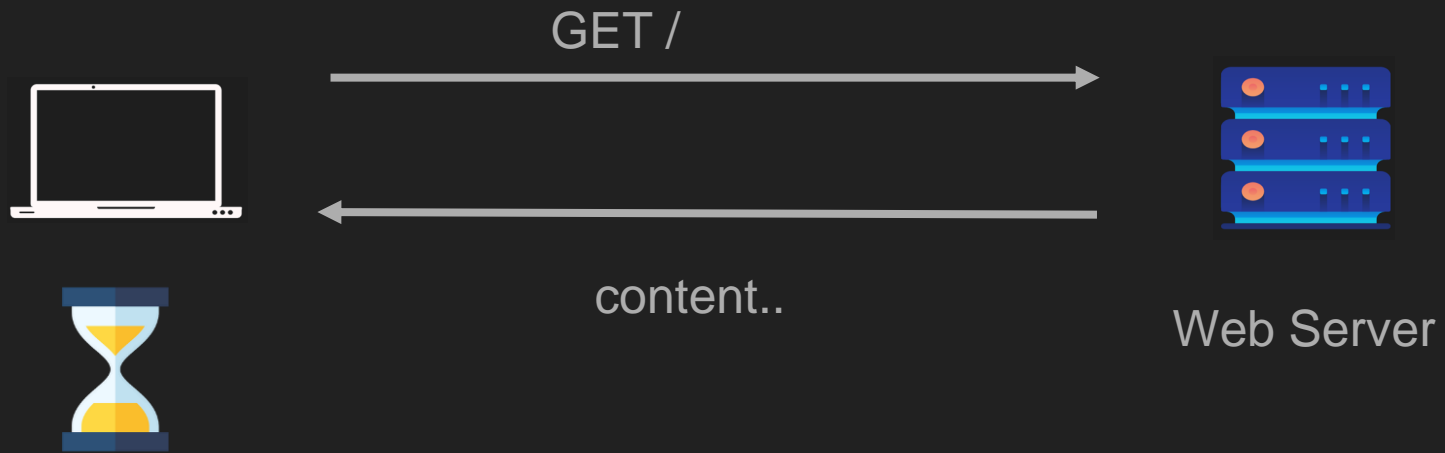
- Pros
 - Real time
 - Compatible with Request/response
- Cons
 - Clients must be online
 - Clients might not be able to handle
 - Polling is preferred for light clients
 - HTTP/1.1 problem (6 connections)



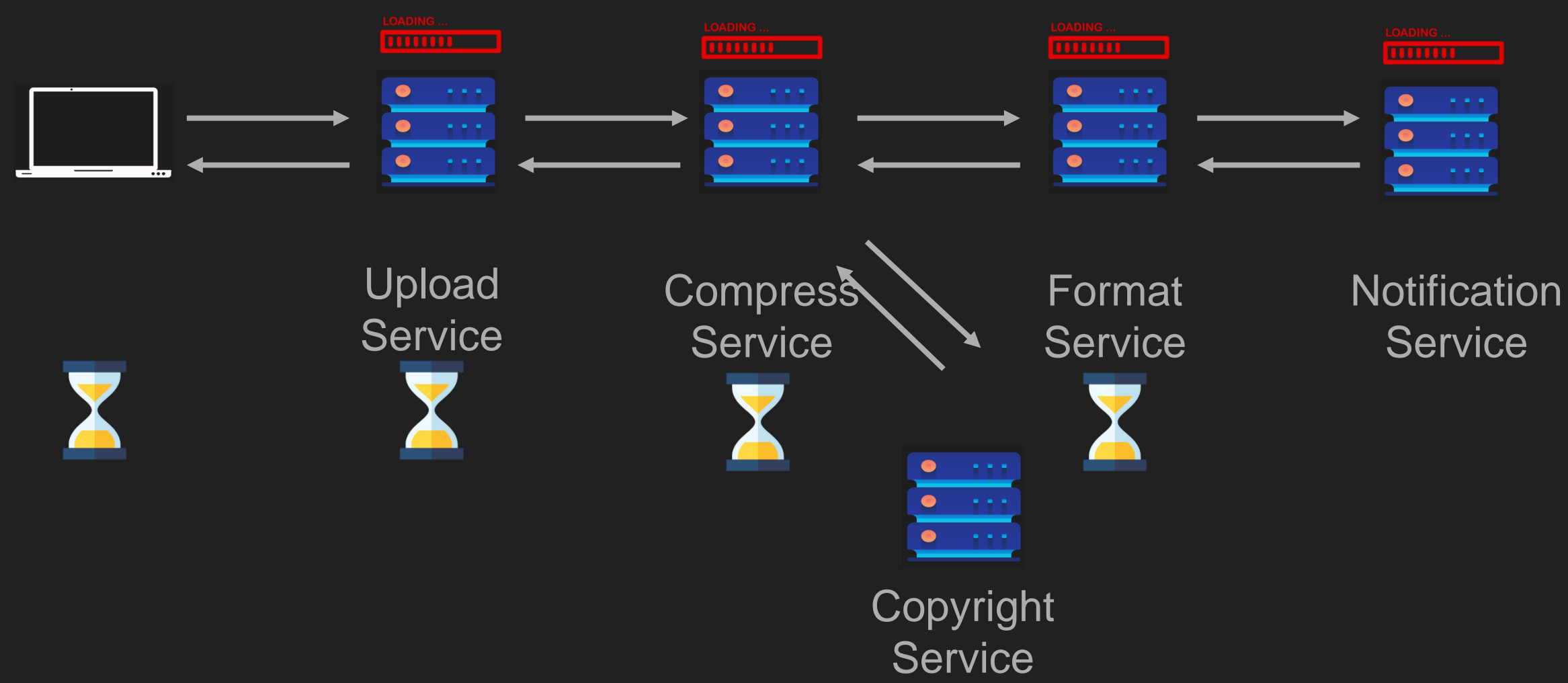
Publish Subscribe

One publisher many readers

Request Response



Where it breaks



Request/Response pros and cons

Pros

- Elegant and Simple
- Scalable

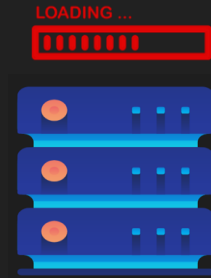
Cons

- 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

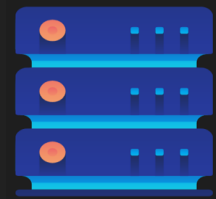
480p video

1080p video

4k video

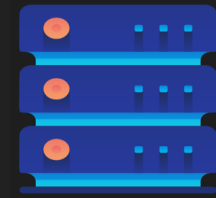
Message Queue
topics/channels

LOADING ...

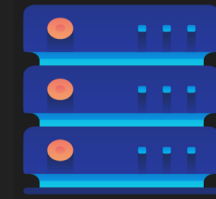


Compress
Service

LOADING ...



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 generals problem)
- Complexity
- Network saturation

The background is a light gray with a subtle grid of thin lines. Scattered across the grid are numerous squares of varying sizes and shades of gray. Some squares are solid, while others are outlined. In the bottom-left corner, there is a large, stylized graphic element consisting of several overlapping, slanted rectangular shapes in dark gray and bright blue.

HTTP Protocol



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

Method

PATH

Protocol

Headers

Body

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		

Protocol

Code

Code Text

Headers

Body

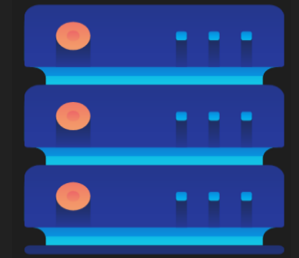
HTTP

open



GET /

80



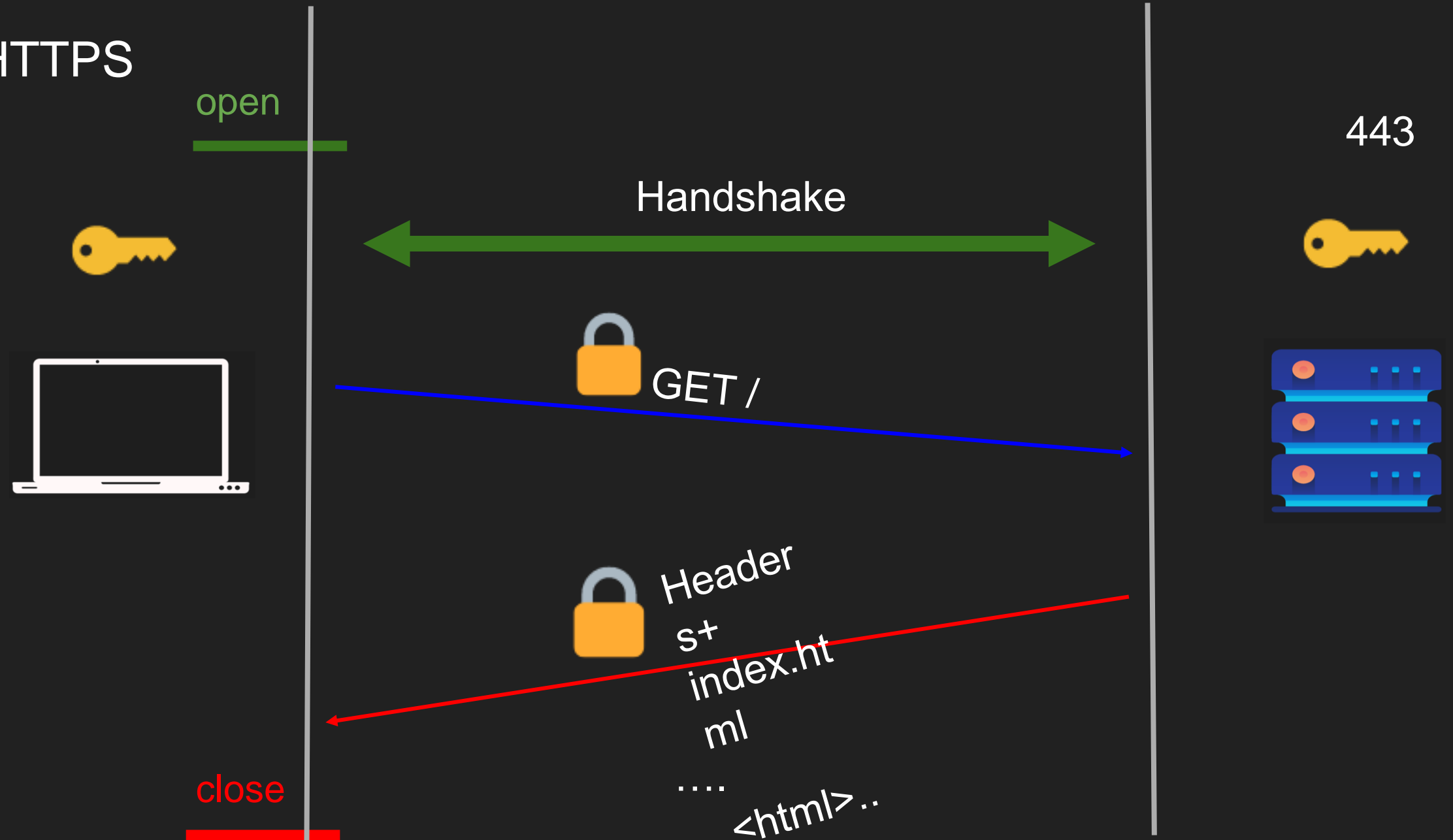
Headers+
index.html

<html>...

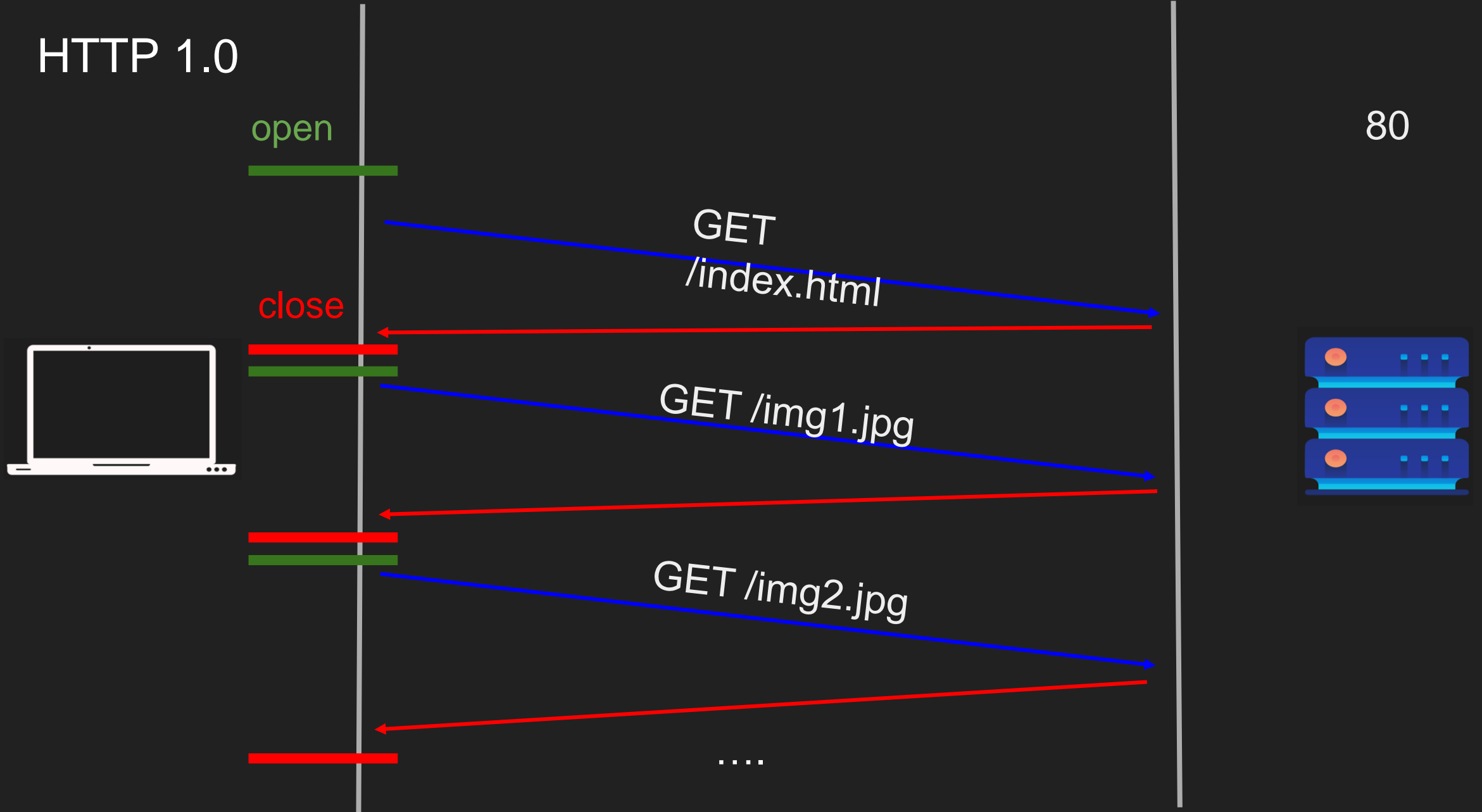
close

....

HTTPS



HTTP 1.0



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



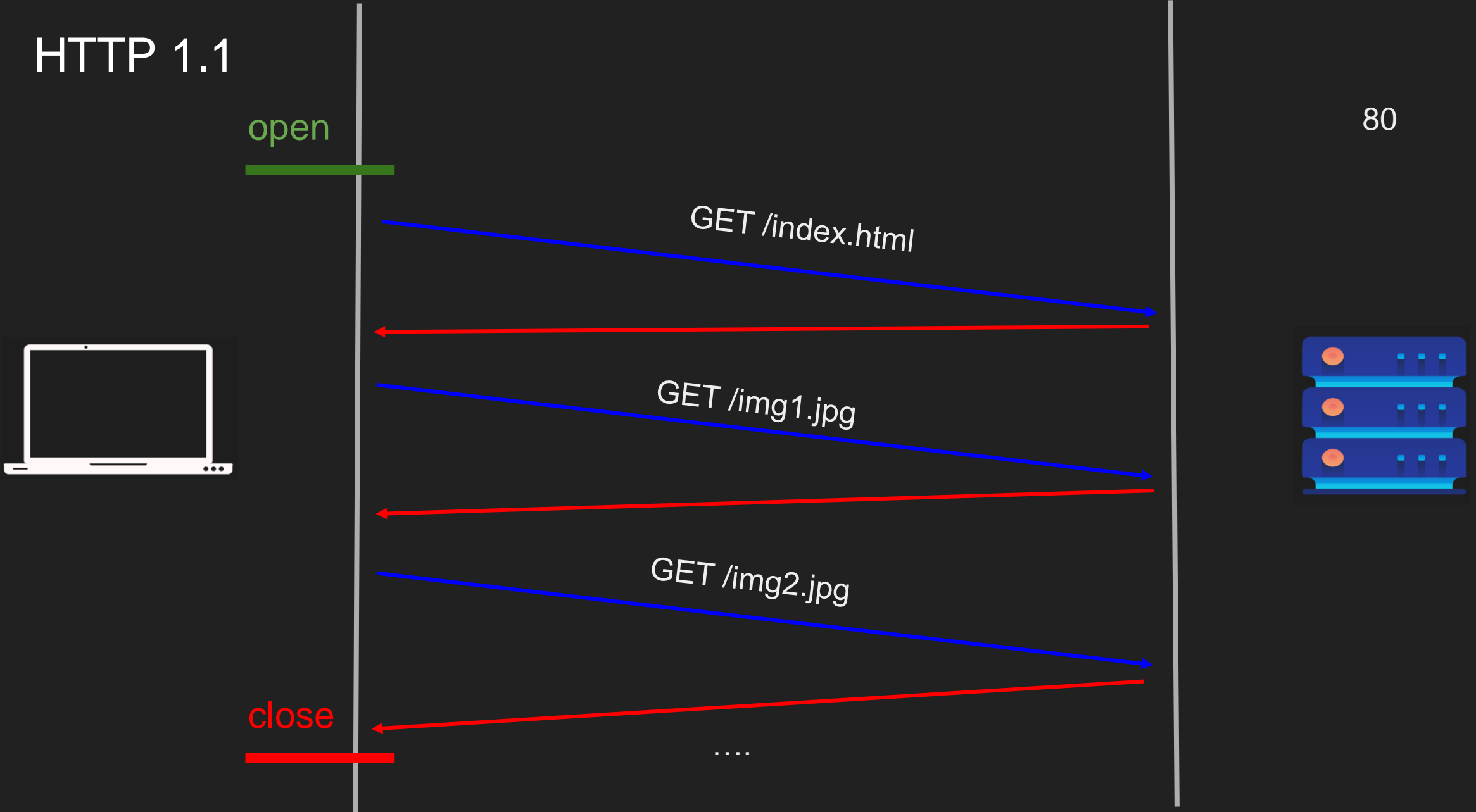
GET /index.html

GET /img1.jpg

GET /img2.jpg

....

close



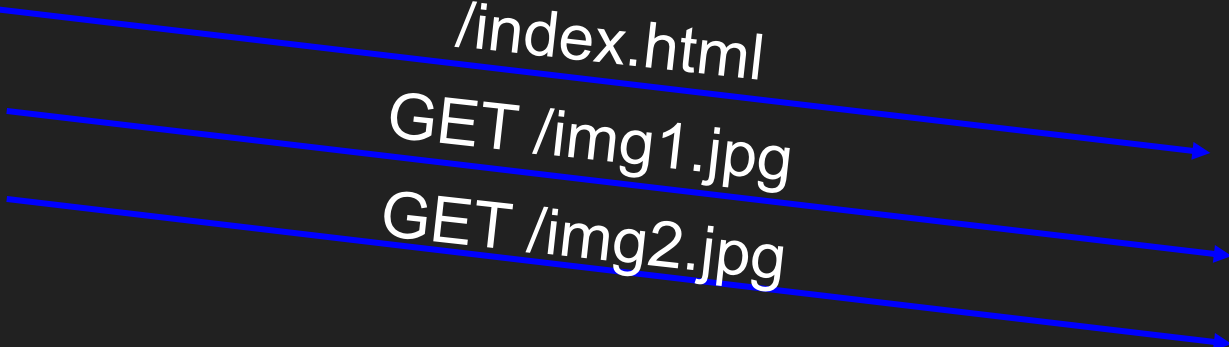
HTTP 1.1 Pipelining



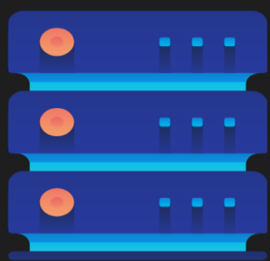
open



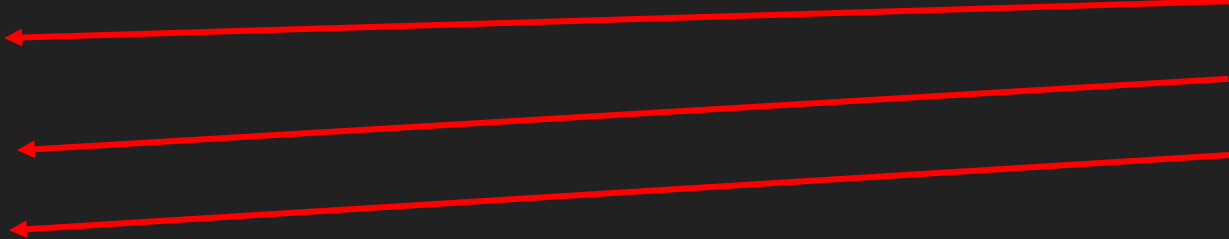
GET /index.html
GET /img1.jpg
GET /img2.jpg



80



close



....

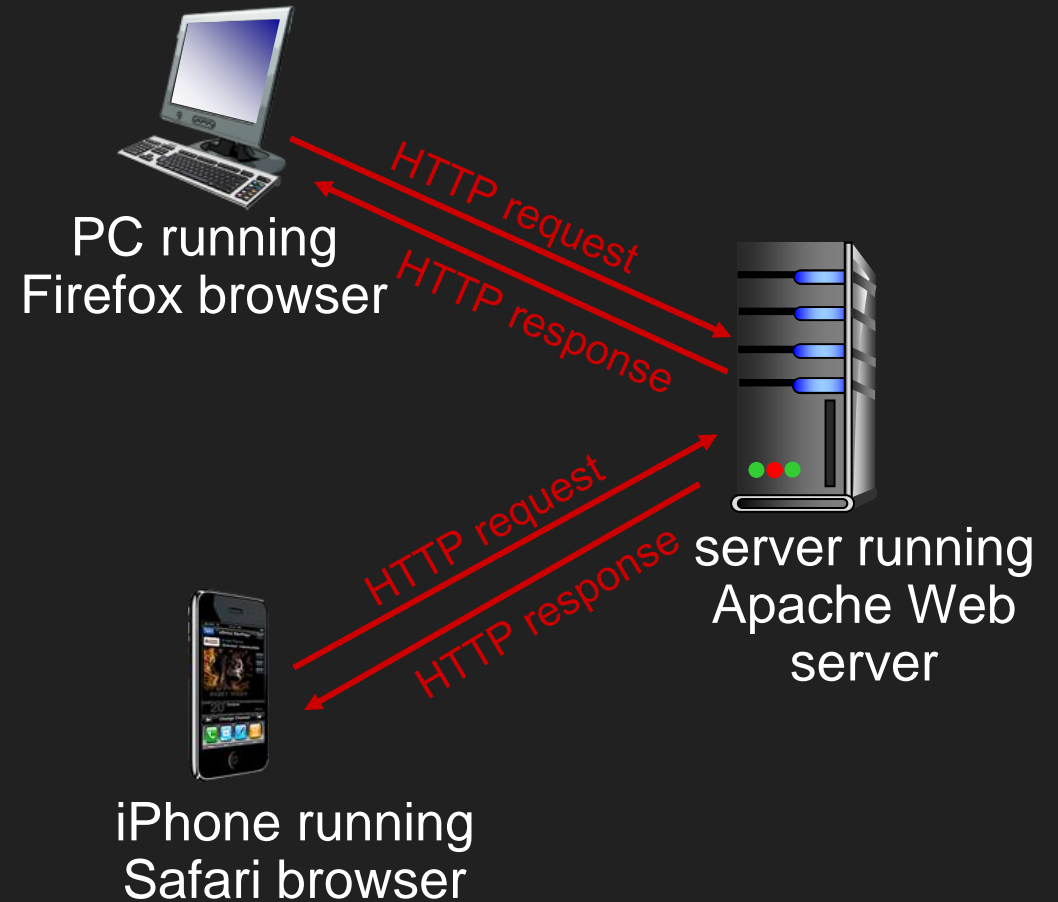
HTTP 1.1

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

HTTP overview

HTTP: hypertext transfer protocol

- Web's application-layer protocol
- client/server model:
 - *client*: browser that requests, receives, (using HTTP protocol) and “displays” Web objects
 - *server*: Web server sends (using HTTP protocol) objects in response to requests



HTTP overview (continued)

HTTP uses TCP:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is “stateless”

- server maintains *no* information about past client requests

aside

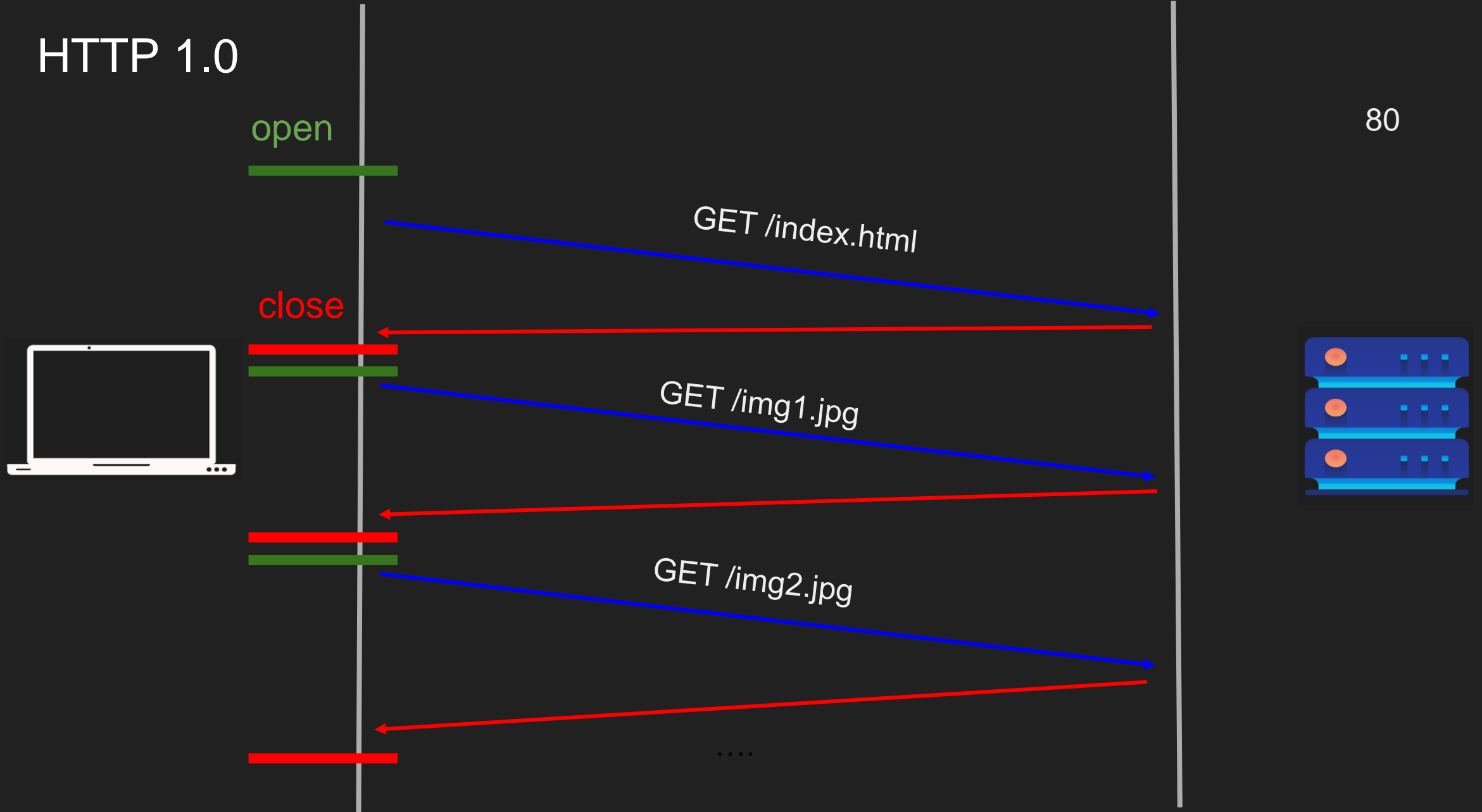
protocols that maintain “state” are complex!

- past history (state) must be maintained
- if server/client crashes, their views of “state” may be inconsistent, must be reconciled

WebSockets

Bidirectional communications on the web

HTTP 1.0



HTTP 1.1

open



80



GET
/index.html

GET /img1.jpg

GET /img2.jpg

close

....

WebSockets



open

80

Websocket
handshake

close

...

WebSockets Handshake ws:// or wss://



Identification

- Identification is the process of providing a unique label or username to a user or entity so that the system can recognize and differentiate them from others. It's the first step in granting access to a system. In other words, identification establishes the "who" aspect of access control by assigning a label or identifier to a user, but it doesn't verify the user's identity or grant access on its own.

Authentication

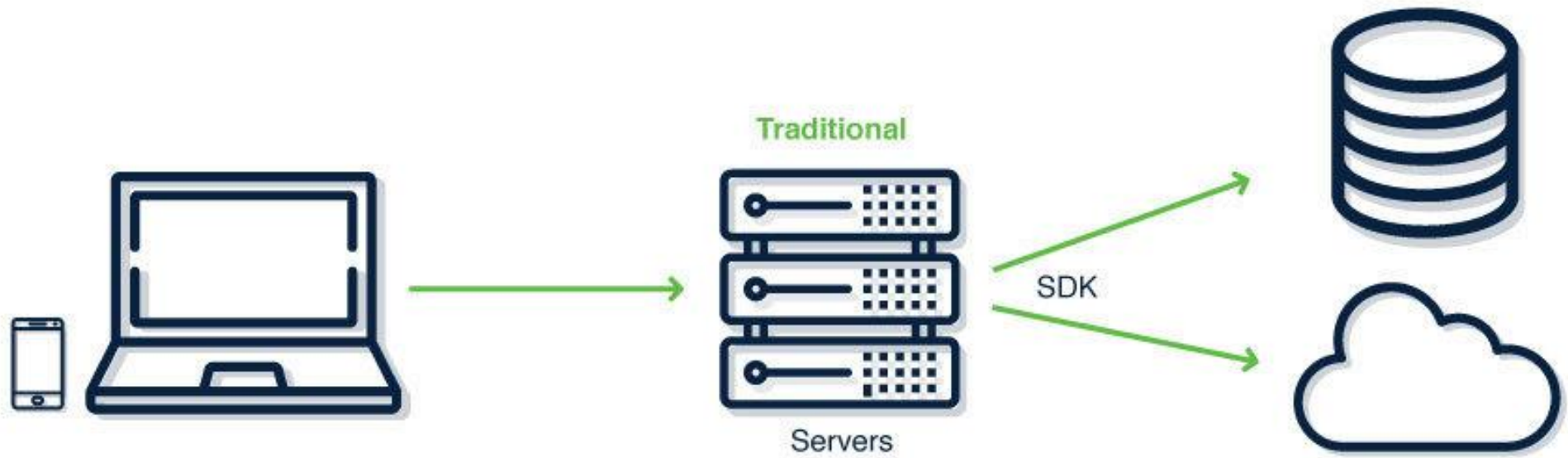
- Authentication is the process of verifying the claimed identity of a user or entity. It ensures that the user is who they claim to be before granting access. Authentication mechanisms involve the use of credentials (such as passwords, PINs, biometric data, smart cards, etc.) to validate the identity of the user. Once a user's identity is verified, they are considered authenticated and may proceed to access the system or resources.

Authorization

- Authorization is the process of determining what actions or resources a user or entity is permitted to access after they have been authenticated. It involves defining and enforcing the level of access and privileges that a user has within a system. Authorization is based on rules, policies, and permissions assigned to specific identities or groups. It answers the question, "What is the user allowed to do once they are authenticated?"



firebase



Interface ESp32 with Firebase



Assignment one

Write a report about the following topics:

- OSI Model vs TCP/IP
- HTTP and its different versions
- Pub/sub model
- MQTT Protocol
- ID token vs Access Token
- Firebase Realtime Database

Assignment two

- Write a client-side code that gets readings from IR Sensor, then authenticate (use keypad to write password) and send readings to Firebase RTDB. After that, you should get readings from Firebase in ESP32 and map it to a value that can be sent to the servo motor and display it in LCD and send it to the servo motor.
- Bonus in Assignment 2: you can use cloud functions to do mapping instead of doing it in an ESp32 device and send a timestamp for every reading.

Assignment three

- Write code that does two of the following operation read, update and delete from the Firebase RTDB.
- Bonus in Assignment 3: you can do all three operations.

Assignment four (Bonus)

- Authenticate using ID token instead of Email&Password or Custom Token

Assignment five (Bonus)

- Send Multiple sensor readings from ESP to Firebase RTDB and Then get them in your ESP again, send a query filter that order them by specific value you choose and select all values that start with the specific value you choose also.