CS472 – COMPUTER NETWORKS

Application Layer Slides

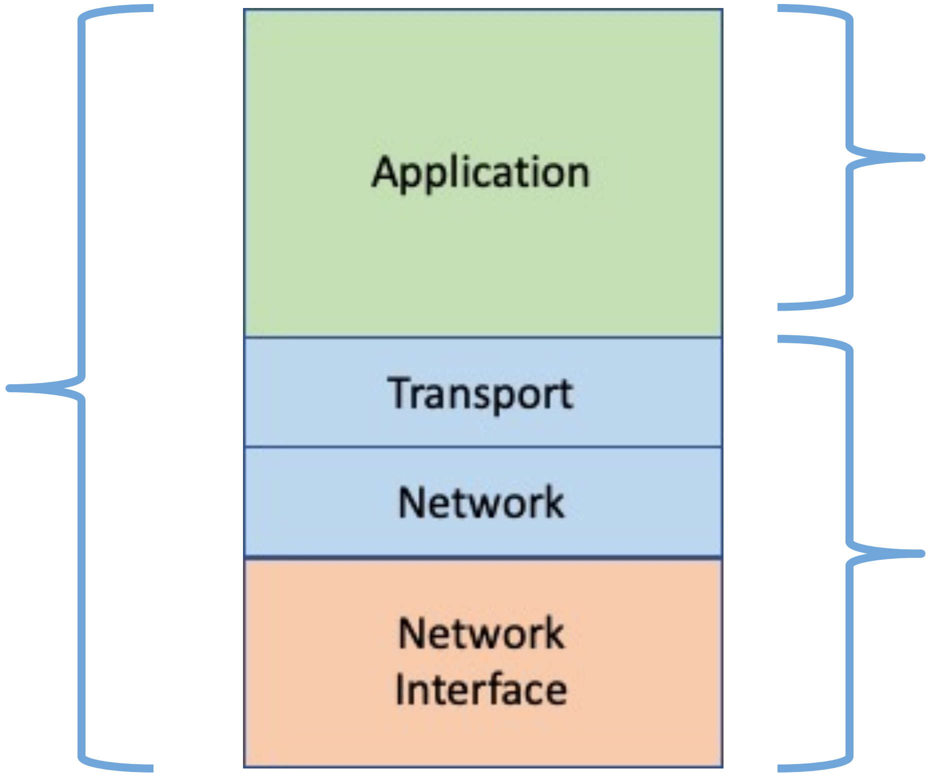
Professor Brian Mitchell

Reading

- Chapter 2.1-2.3; 2.7 & 2.8

Flow of this class review

Focus So Far: A touch on the entire network stack



Now: Diving into the Layer Space (Chapter 2)

Future Lectures will work down the stack – Transport (Chapter 3), Network (Chapter 4/5), and Network Interface (Chapter 6)

Application Layer Topics

- ***Homework #1 focused on basic systems programming mental refreshing***
- Next
  - The application layer – common duties and characteristics
  - HTTP
  - FTP
  - DNS
  - HW2

Review - Protocol

- The heart of the network is the PROTOCOL
- Similar to many things that you already use
    - Telephone conversation
    - Ordering food
    - "What time is it?"
- Definition:

"Defines the messages sent and received and the actions that are performed when messages and network events are received."

They are specified by RFCs

Good place to view RFCs is https://www.rfc-editor.org/

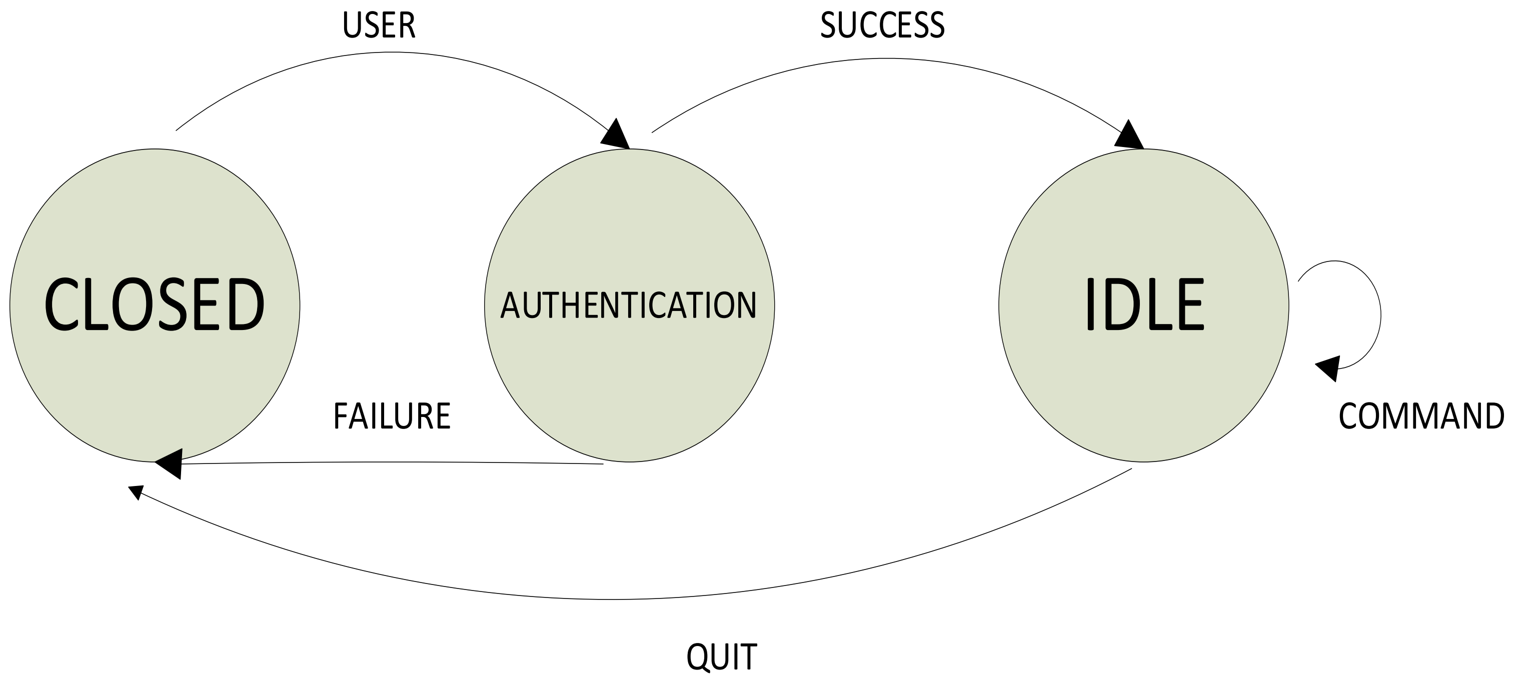
Review - Network Protocols

- Deterministic – only one possible action for each message in each state
  - Deterministic Finite Automata (DFA)
- Time (some concept) – no response in a period of time is also an action
- Well defined messages (what makes them up) & how that you know that they are done (“OVER”)
- State (possibly) – different stages of the conversation
- TRUST

- Examples (“What time is it?”)

- Note that protocols lower in the stack use binary aligned fields in packet headers, up the stack in the application layer they tend to use other attributes such as key:value pairs separated by newline characters (e.g., HTTP)

Simple DFA

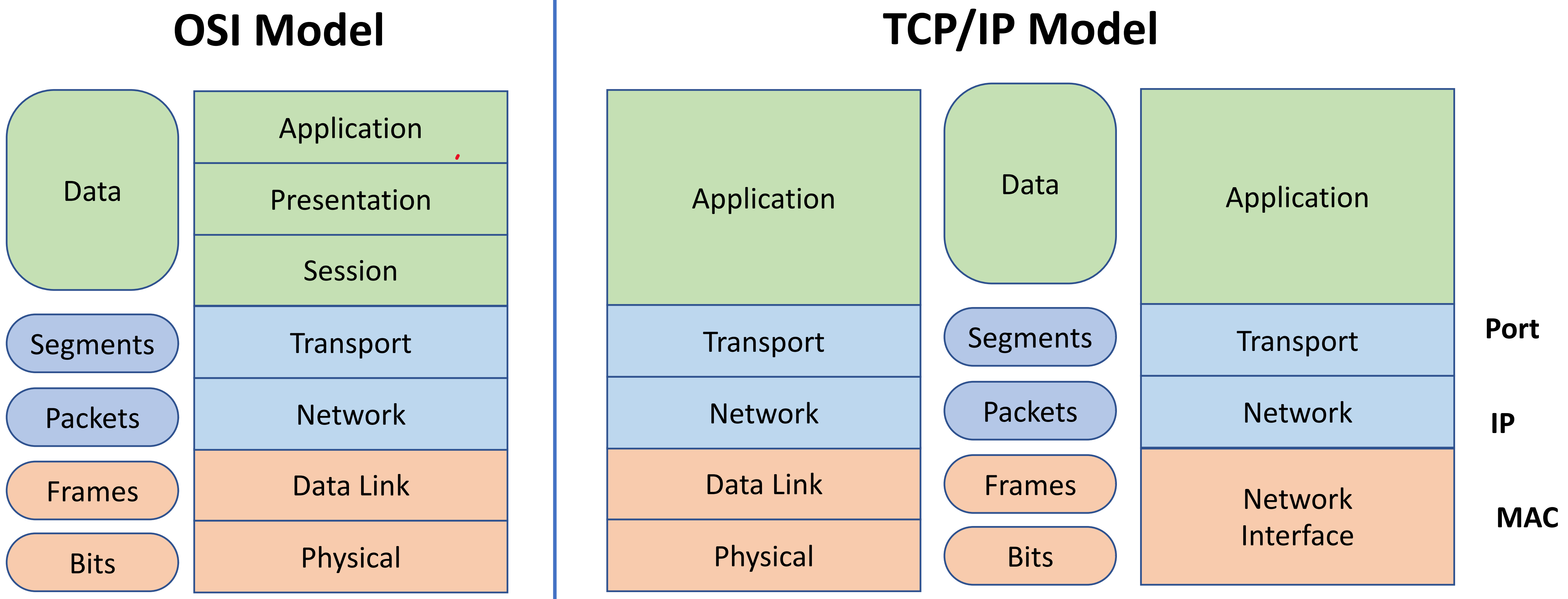


Note that DFAs generally require saving the state of the DFA so that transitions can be properly managed.

Review: TCP/IP Reference Model

- Application Layer (the applications use these protocols to transfer data)
  - HTTP, FTP, SMTP, POP3, etc.
- Transport Layer (this layer connects applications)
  - TCP, UDP and others
- Internetworking Layer (get data from source to destination)
  - IPv4, IPv6
- Link Layer (get data to the next adjacent device – make a “link”)
  - Ethernet, 4G, 5G, Wifi 5, Wifi 6, etc.
- Physical Layer (0->0, 1->1)
  - Could be combined with Link Layer as Host to Network Layer or NIC layer
- Transport Layer Security sublayer added later
- Review Wireshark

Network Reference Models



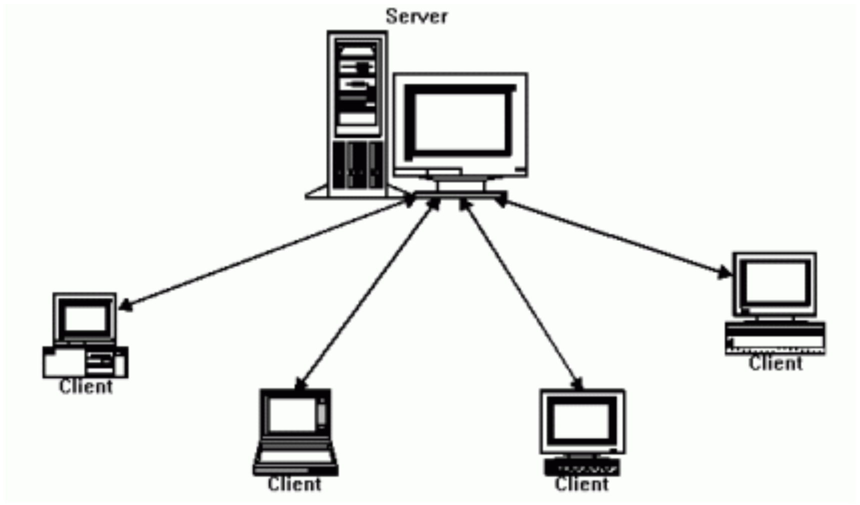
Sometimes its better to think about the Data Link and Physical separately; sometimes its easier to think about combining them into a single Network Interface Level

Common Themes

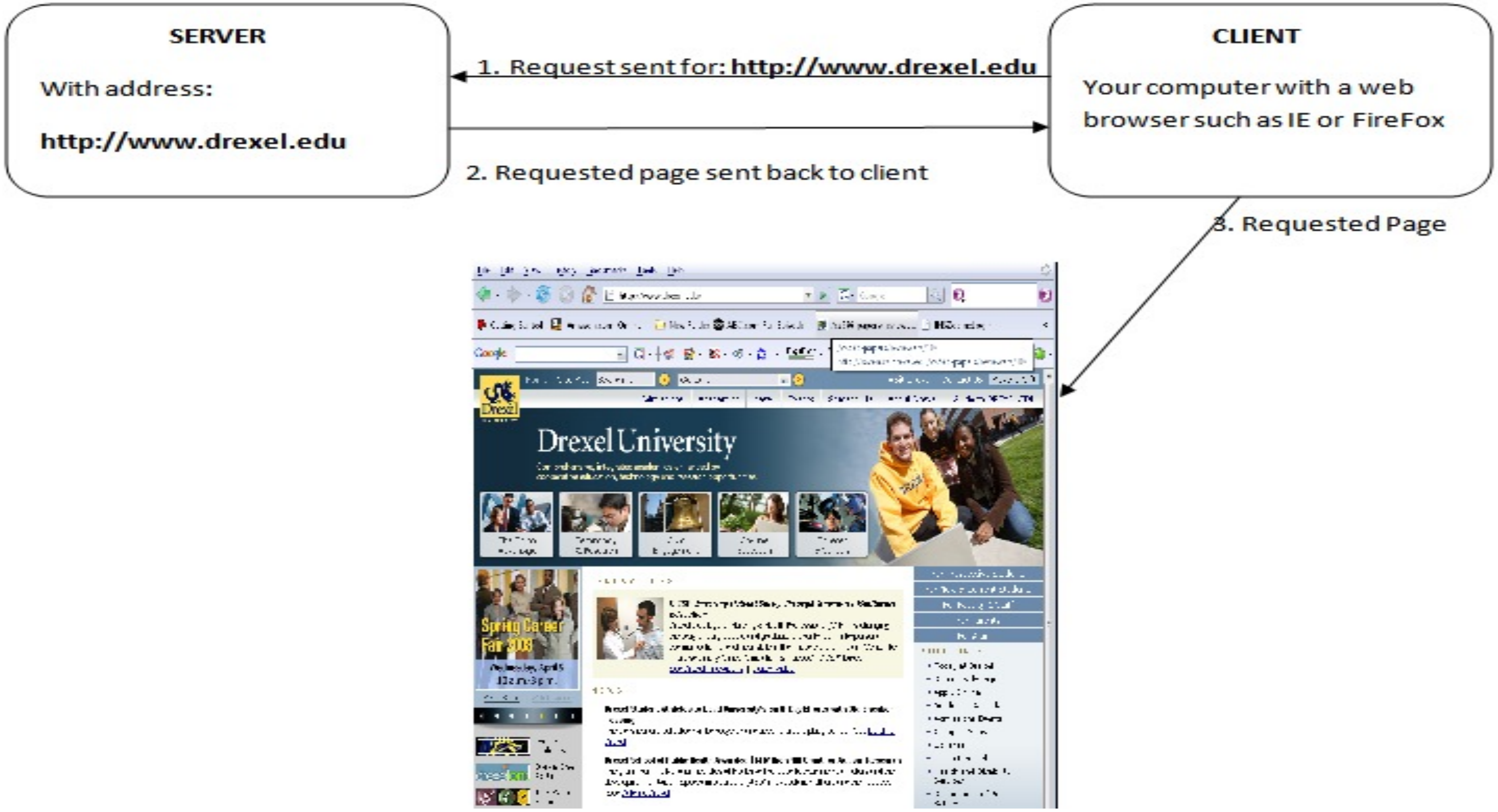
- Common functions which occur at multiple layers due to solving common problems
    - Addressing
    - Connection Control
    - Ordered Delivery
    - Segmentation & Reassembly
    - PDU definition
    - Error detection & Correction
    - Flow Control
    - Multiplexing (more than one lower or upper service)
    - Security
    - Quality of Service (QoS)

Network Application - Architecture

- Client / Server Architecture
  - Web (HTTP)
  - FTP
  - Telnet / SSH
  - E-mail (SMTP)
  - DHCP,
  - DNS
  - Streaming video
  - And many others!!

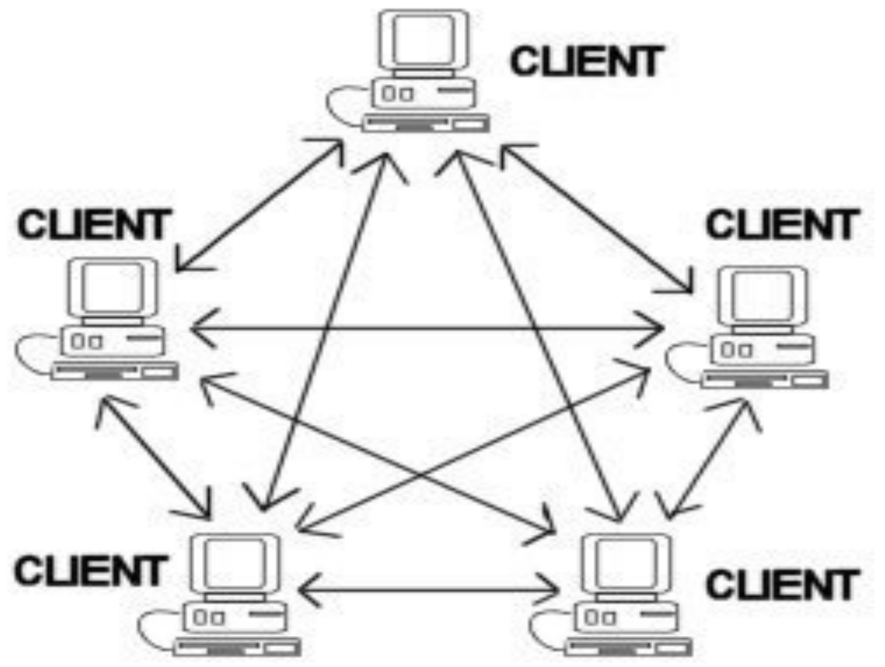


Network Application Architecture


SERVER
With address:
http://www.drexel.edu
1. Request sent for: http://www.drexel.edu
CLIENT
Your computer with a web browser such as IE or FireFox
2. Requested page sent back to client
3. Requested Page
Drexel University
Spring Career Fair 2008

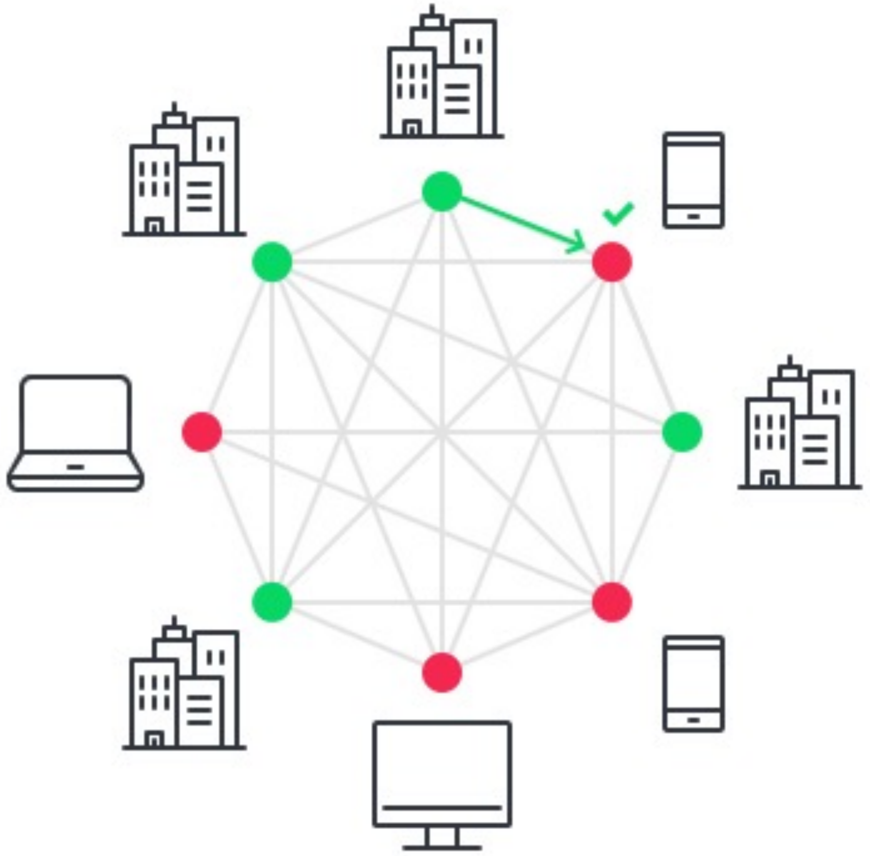
Network Application Architecture

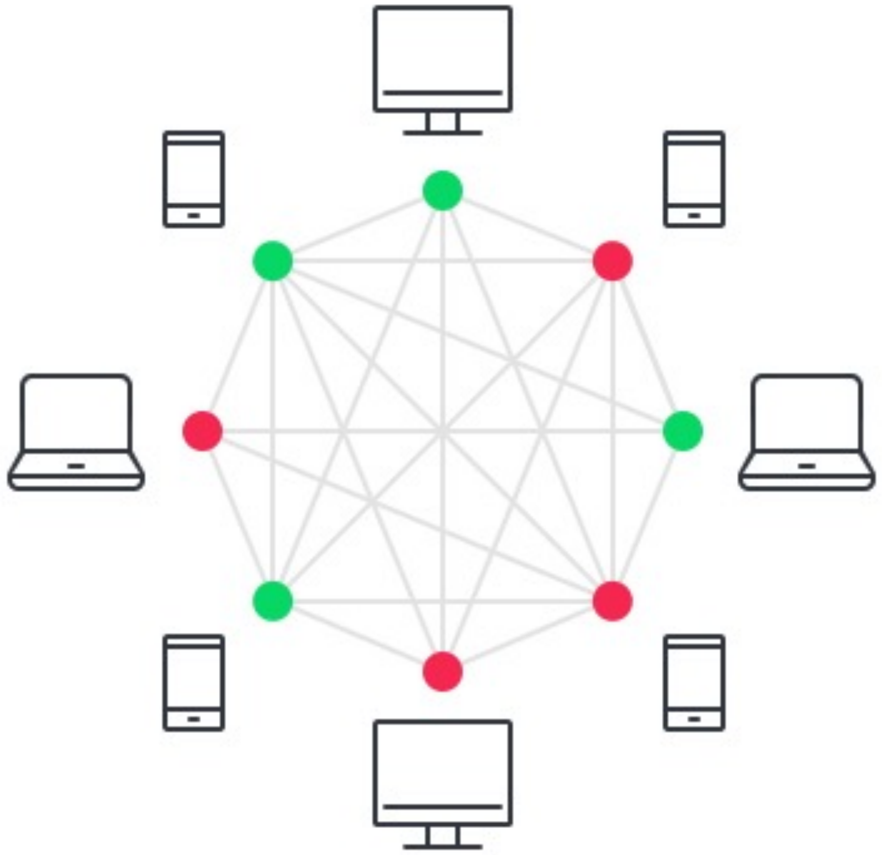
- Peer to Peer (P2P) Architecture – Each node acts as a client and server
  - BitTorrent
  - Others



P2P networks can use rules such as proximity or latency to locate neighbor peers

Blockchain "decentralized" Architecture – A Modern Peer to Peer Architecture





🟥 **Validator Node**

Can both initiate/receive and validate transactions

🟩 **Memeber Node**

Can only initiate/receive transactions

Comparison / Tradeoffs

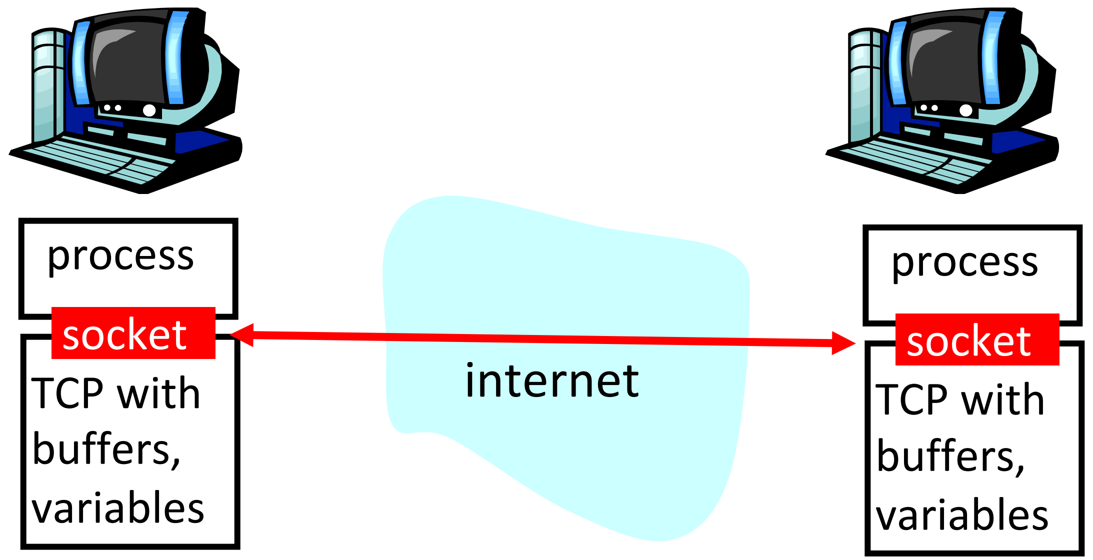
- Identification
- Trust
- Discovery
- Traffic Predictability
- Performance (finding the right pieces)
- Others?

- The start of “IT DEPENDS”!!

- Why would you pick client-server over peer-to-peer?
  - Protocol?
  - Service?
  - Security?
  - Reputation of the server?

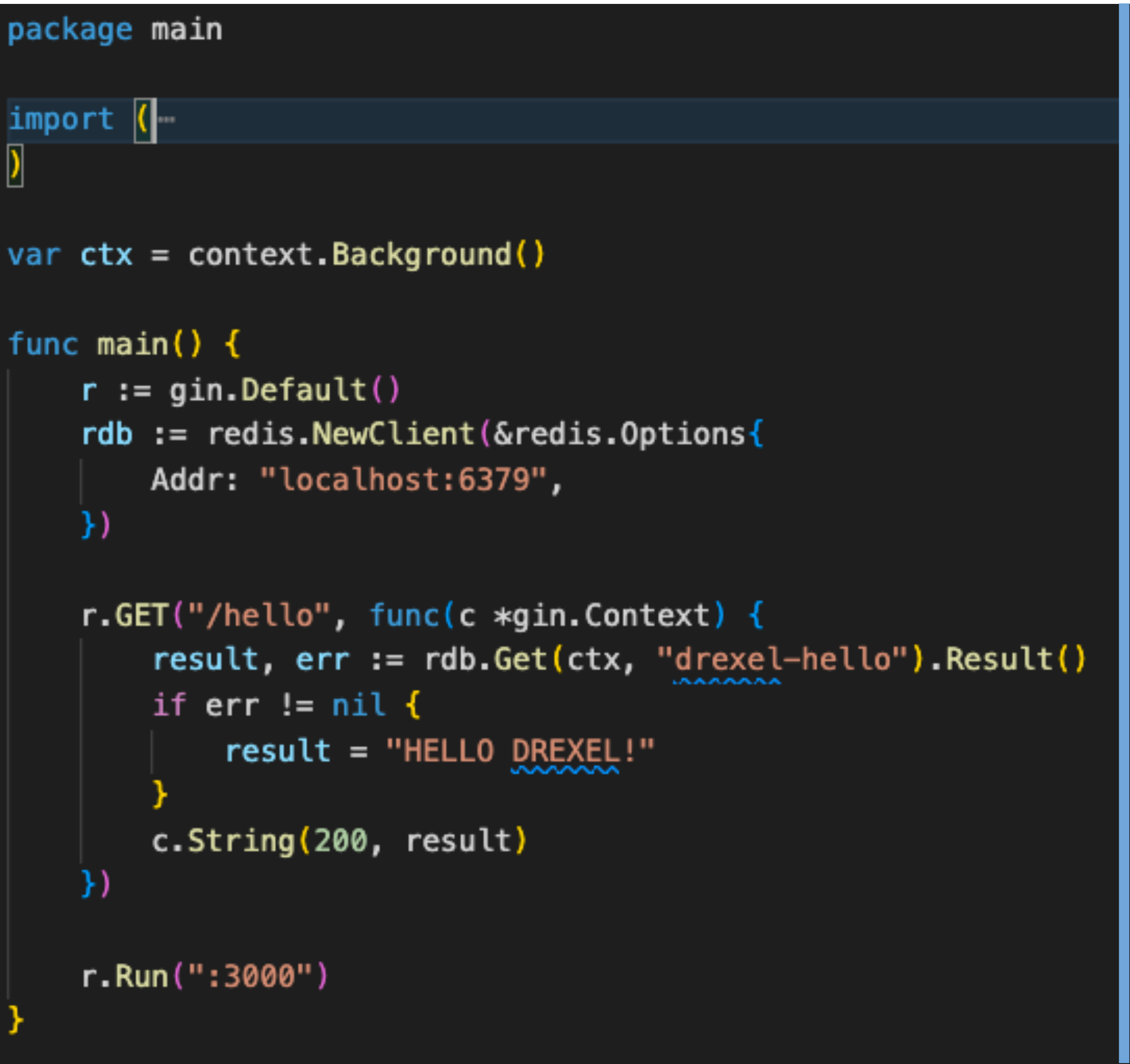
How Processes Communicate

- Interface between Processes and Computer Networks
- Application Programming Interfaces (network “look”)
    - Sockets
    - RPC, GRPC / COM / DCOM



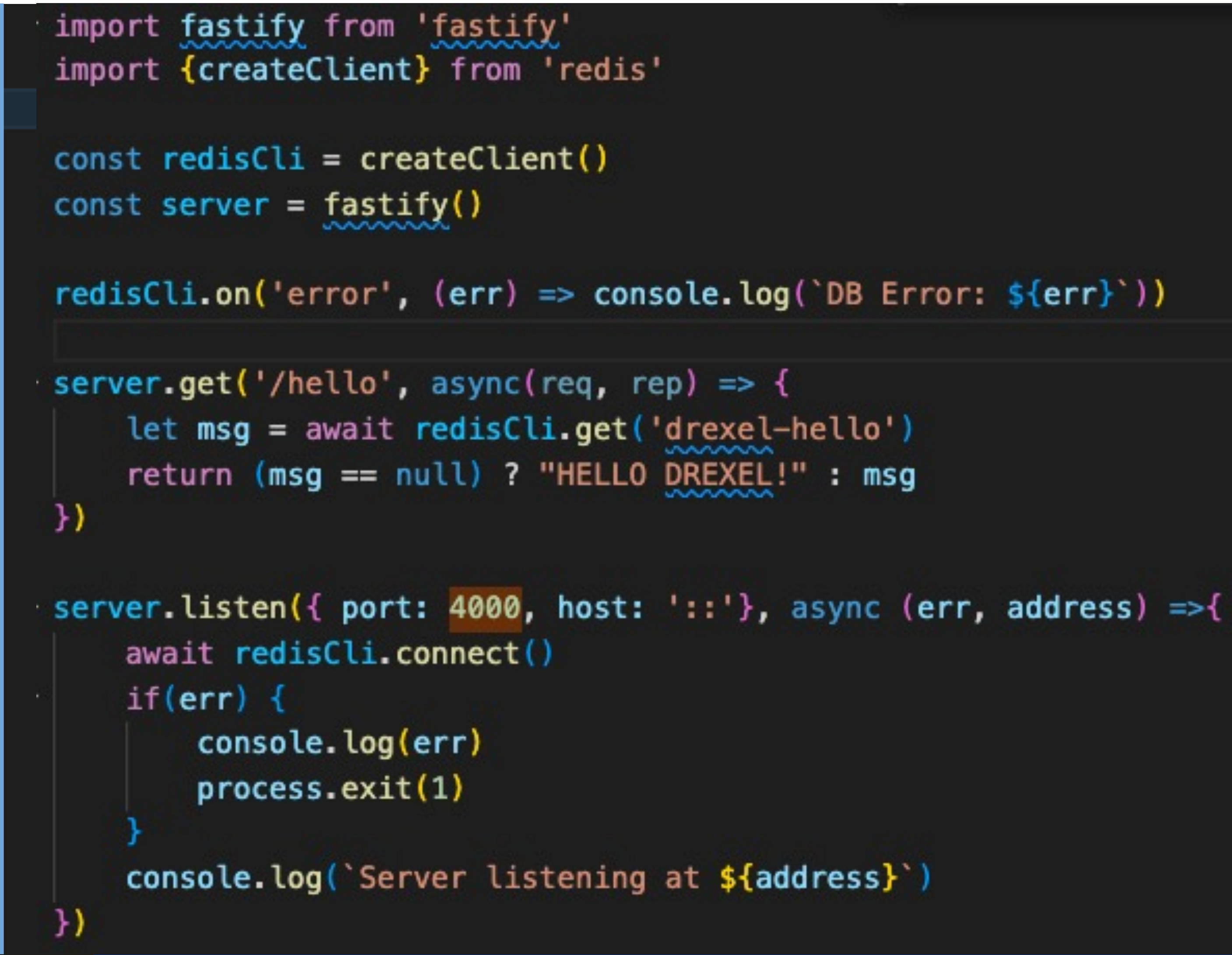
Modern Libraries and Programming Languages have Made This Much Easier

A Highly Scalable Server in Go



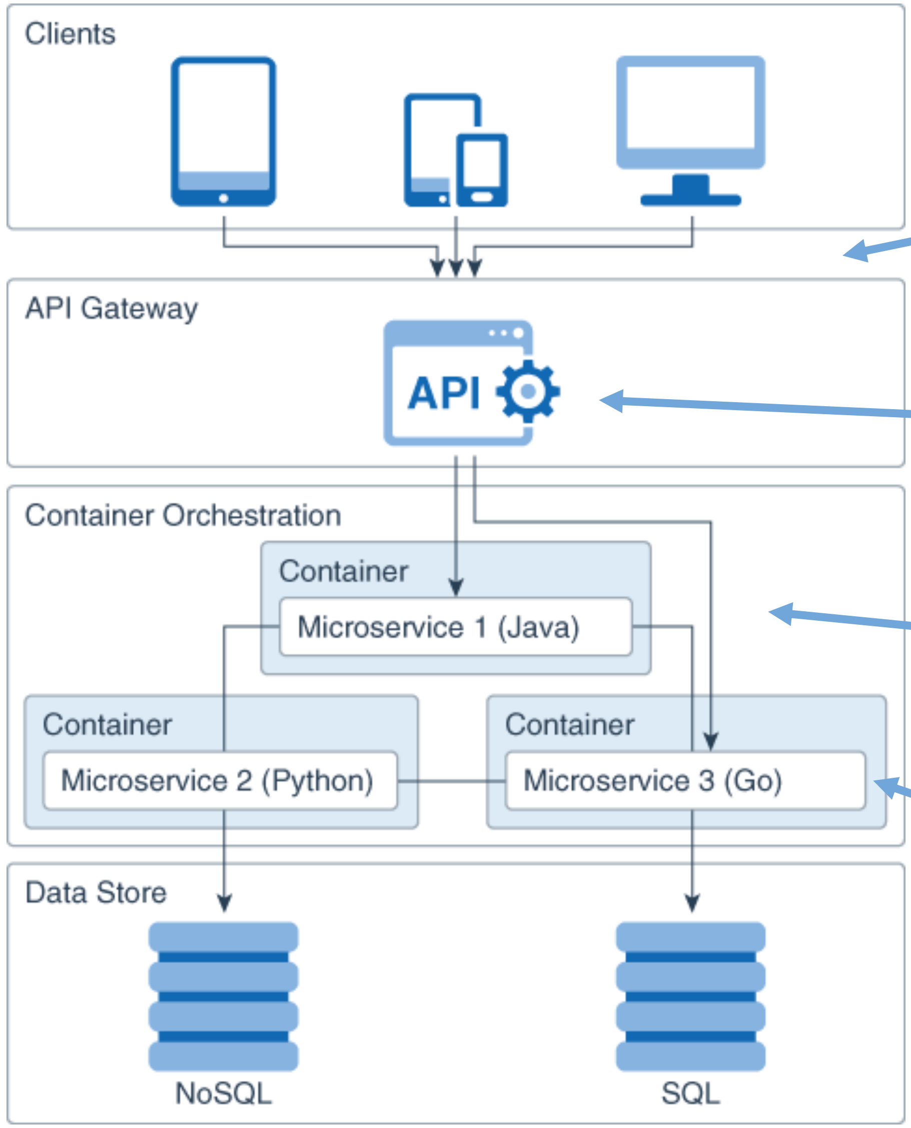
```go
package main

import (⋯
)

var ctx = context.Background()

func main() {
    r := gin.Default()
    rdb := redis.NewClient(&redis.Options{
        Addr: "localhost:6379",
    })

    r.GET("/hello", func(c *gin.Context) {
        result, err := rdb.Get(ctx, "drexel-hello").Result()
        if err != nil {
            result = "HELLO DREXEL!"
        }
        c.String(200, result)
    })

    r.Run(":3000")
}
```

A Highly Scalable Server in Typescript



```typescript
import fastify from 'fastify'
import {createClient} from 'redis'

const redisCli = createClient()
const server = fastify()

redisCli.on('error', (err) => console.log(`DB Error: ${err}`))

server.get('/hello', async(req, rep) => {
    let msg = await redisCli.get('drexel-hello')
    return (msg == null) ? "HELLO DREXEL!" : msg
})

server.listen({ port: 4000, host: '::'}, async (err, address) =>{
    await redisCli.connect()
    if(err) {
        console.log(err)
        process.exit(1)
    }
    console.log(`Server listening at ${address}`)
})
```

APIs



API-Based Architectures

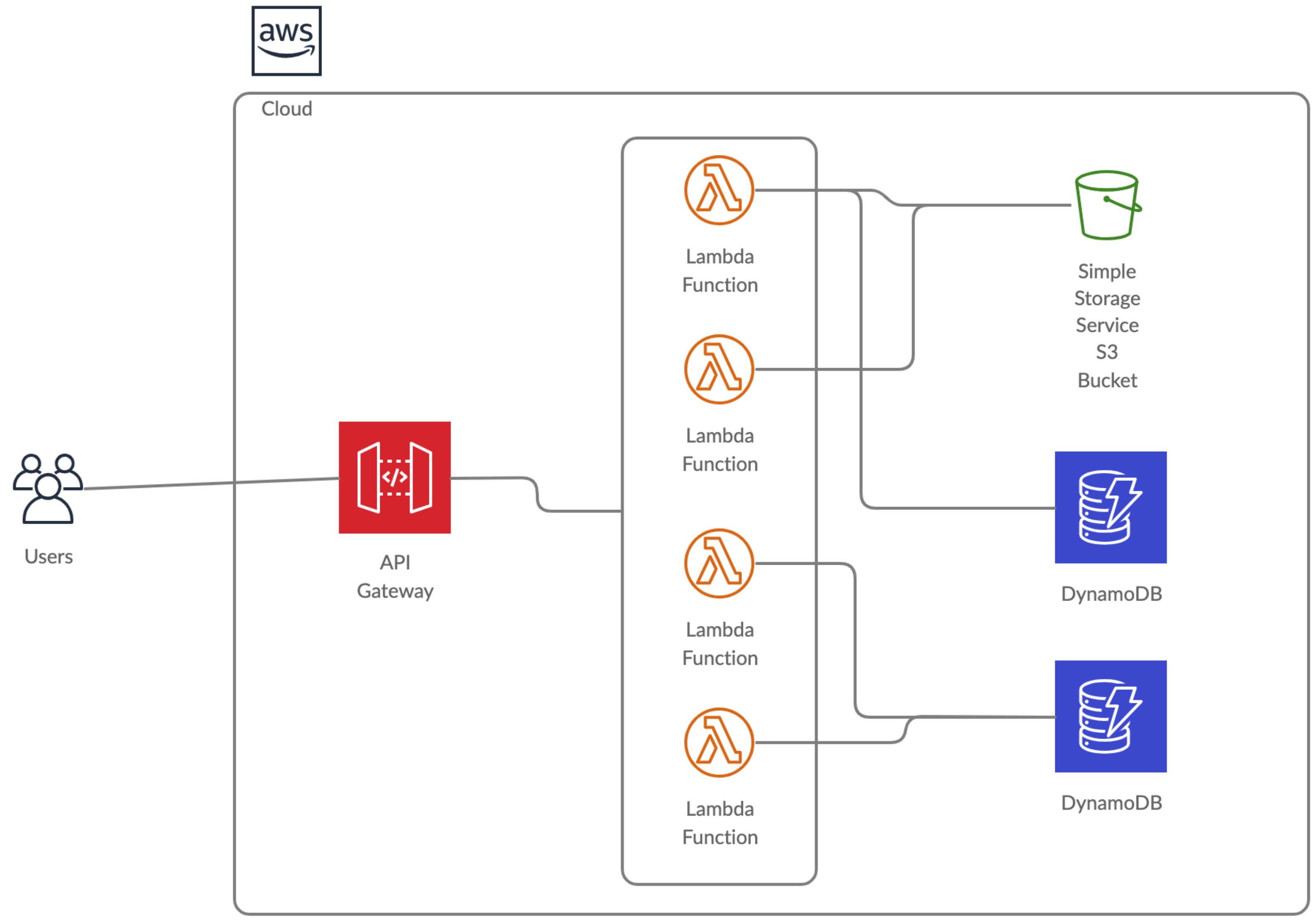
HTTPS Requests (REST)
oAuth Security

Reverse Proxy,
Security Enforcement Point

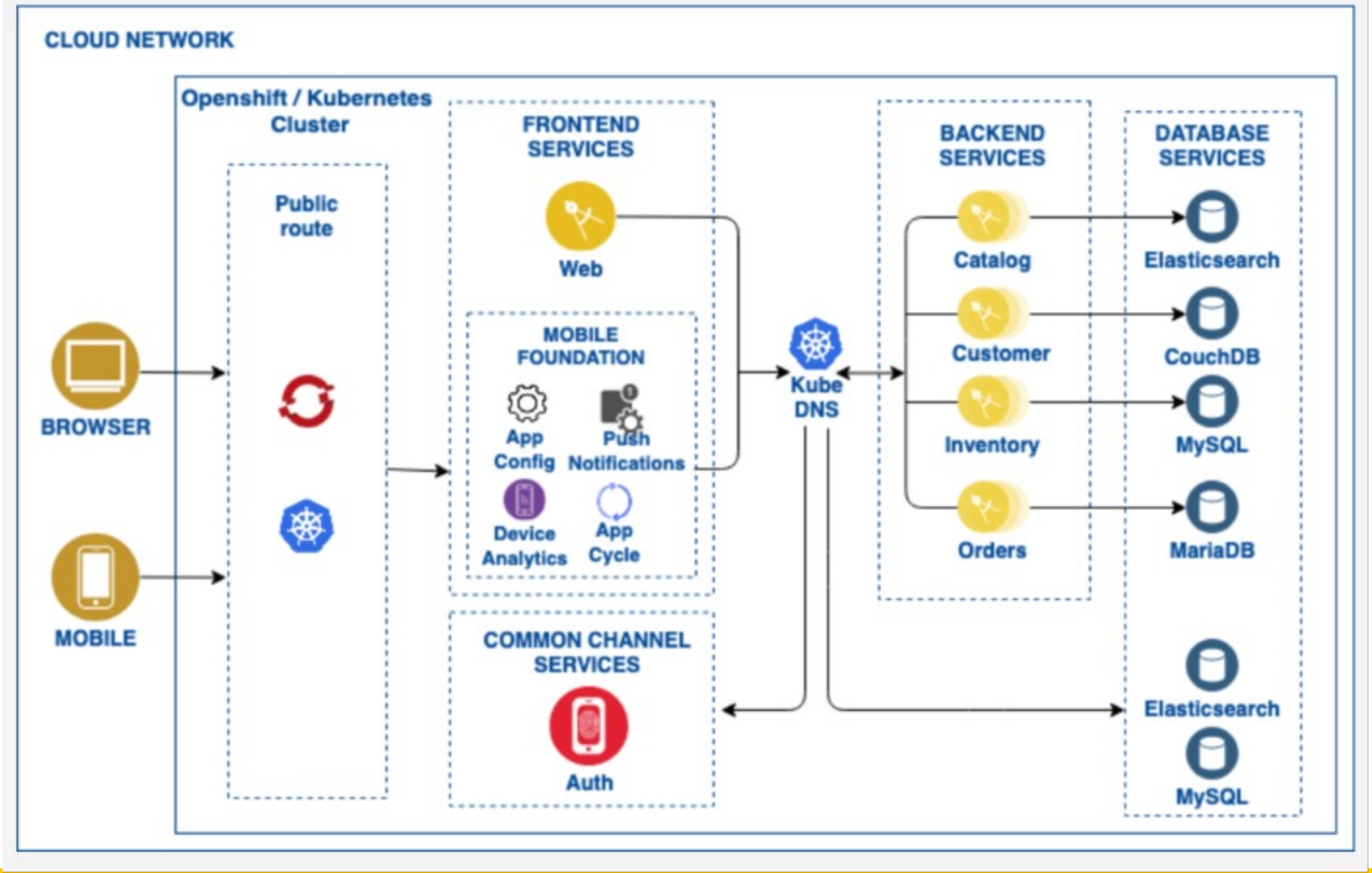
Resource Management,
Supervision

Network-Based Application Code

APIs - Cloud


aws
Cloud
Users
API Gateway
Lambda Function
Lambda Function
Lambda Function
Lambda Function
Simple Storage Service S3 Bucket
DynamoDB
DynamoDB

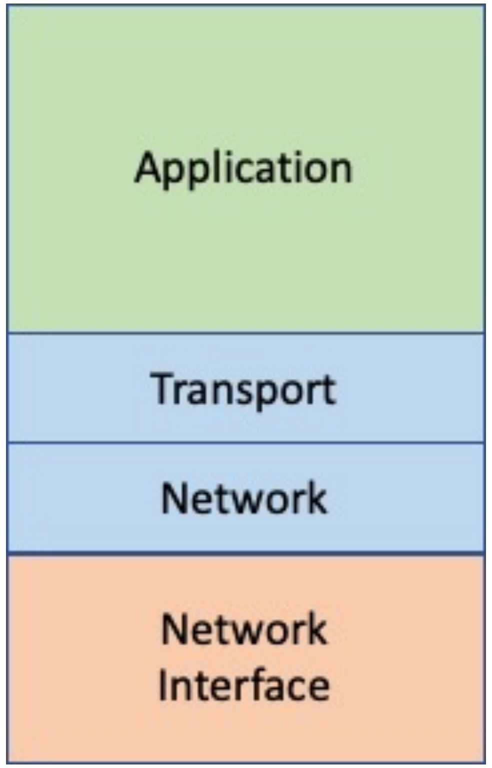
APIs – Kubernetes



What the Transport Layer gives applications

- Data Transfer end-to-end
  - Reliable? (if needed)
- Throughput
- Timing

- Selection of Transport Layer is made on what the application wants the network look like.
- The transport layer also provides a lot of abstractions to the application layer – can you think of a few of these?



Remember the application layer "rides on top" of the transport layer!

Application Layer Choices on Transport Layer – Reliable Stream (TCP), Possible Unreliable Datagram (UDP), Payload – Binary or UTF-8 (Text), Encrypted or Non-Encrypted, etc.

How do we specify who we want to talk to?

- IP addresses (v4, v6)
  - Use DNS to get these, more later
- Port numbers (e.g. 80)
  - Which identify what application we want to talk to
  - Can specify service, but port number does not represent service absolutely.

- Internet Endpoint (socket)
  - (IP address, port number, TCP/UDP)
- Internet Connection (socketpair)
  - (source IP, destination IP, source port, destination port, TCP/UDP)
- Must be unique (describe servers)

IMPORTANT: The abstraction the Transport layer provides to the application layer is a tuple:

(Source IP, Destination IP, Source Port, Destination Port, and Protocol)

Port usage

- Port is represented by a 16-bit integer
  - Too few going forward?
- Some ports have been reserved to support common services (/etc/services). For example:
  - FTP (TCP port 21 or port 990 with TLS)
  - SSH (TCP port 22)
  - SMTP (TCP port 25)
  - DNS (UDP port 53)
  - HTTP (TCP port 80 or port 443 with TLS)
- Must usually be superuser/administrator to use < 1024.
- Clients use port 0 and OS gives them an unused port number.
- IANA keeps track on who has registered the port – but does not guarantee that service is at that port.

Application Layer Protocols

- Define:
  - Type of message exchanged
  - Syntax of various message types
  - Semantics of various fields
  - Rules for determining when and how a process sends requests and responses

- All protocols do this, but transport protocols are usually one to one (unicast)
  - IPv4 primarily uses unicast and broadcast
  - IPv6 addressing this – eliminates broadcast, and expands on multi-cast
  - There will be others!

Addressing Recall

- Lets assume we have a network mask of 255.255.255.0
- Lets next assume our IP address is 192.168.22.44

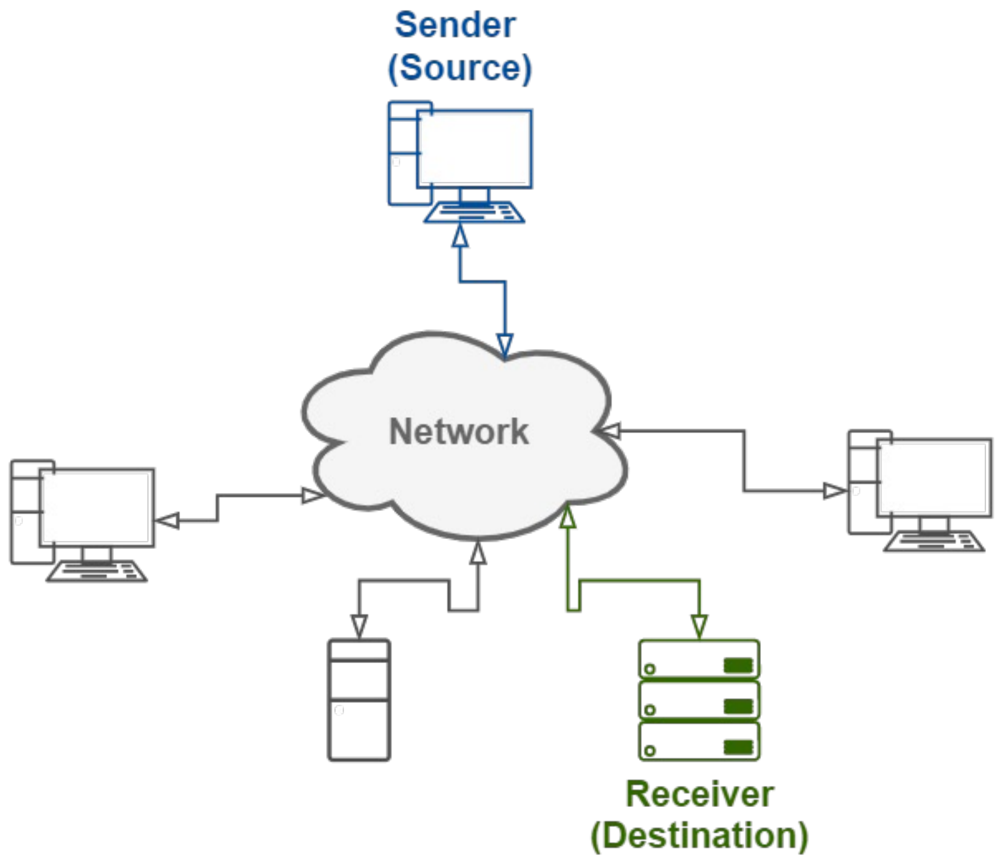
- What network are we on?
  - Bitwise AND of your IP address and the mask
  - In this case its 192.168.22.0

- Often times the gateway will end with "1" so 192.168.22.1

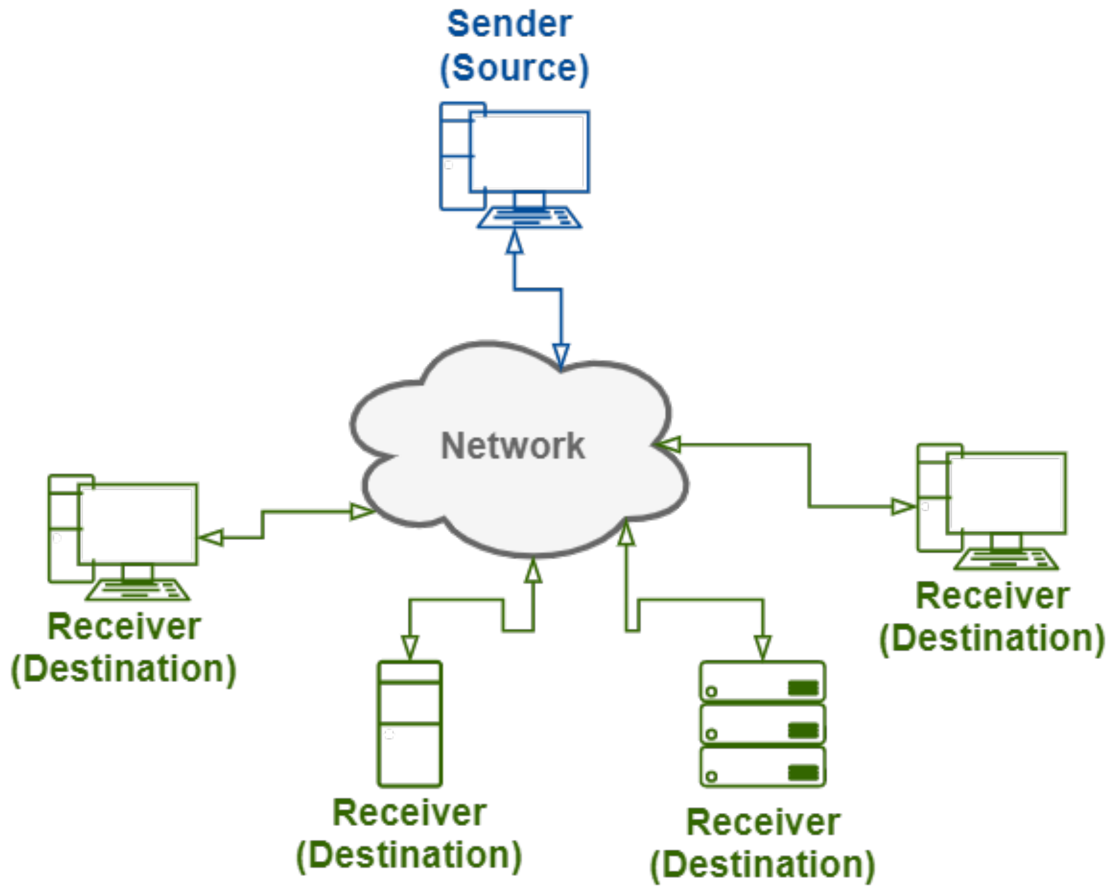
- Broadcast address is you take your network address first:
  - 192.168.22.0, then you replace all of the zero bits with "1"
  - My network is 192.168.22.[00000000]
  - 192.168.22.[11111111] = 192.168.22.255

Different Ways to Exchange Messages

Unicast

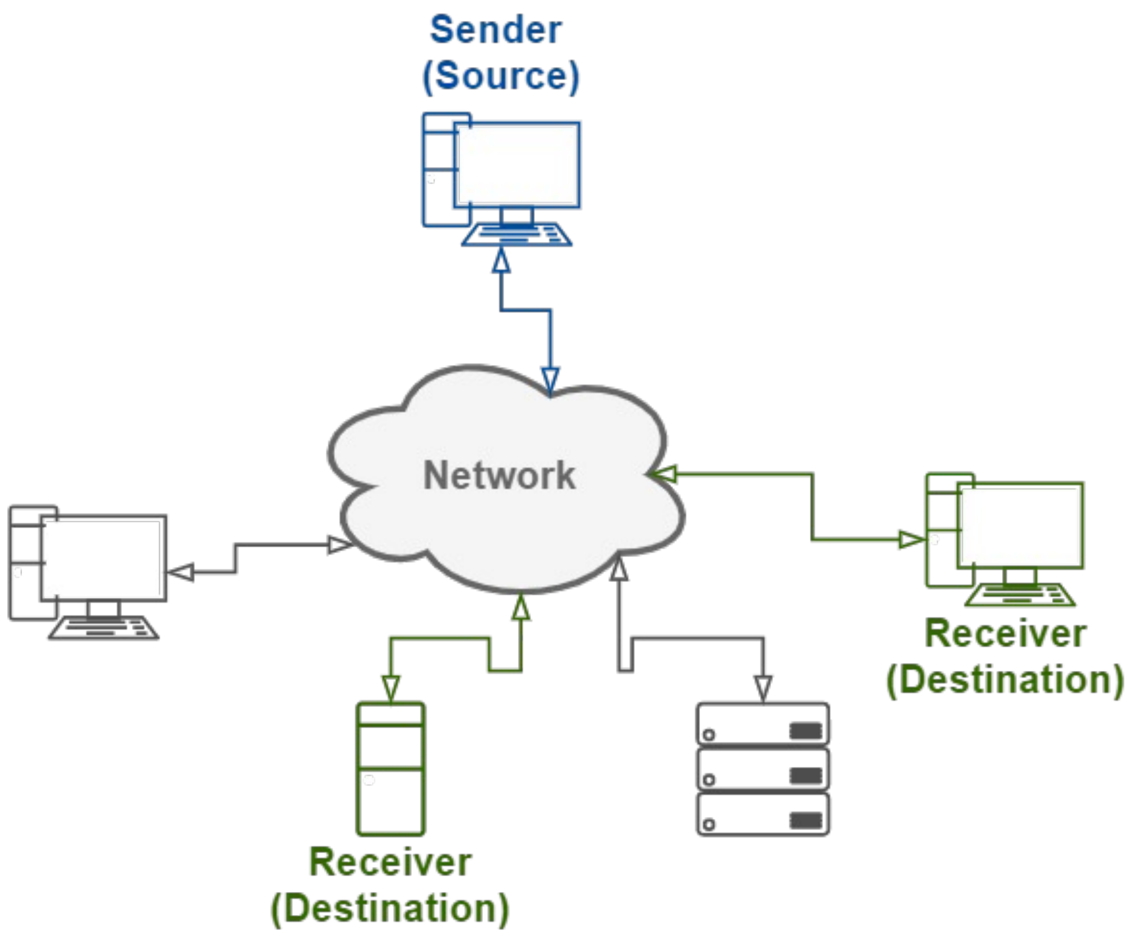


Broadcast

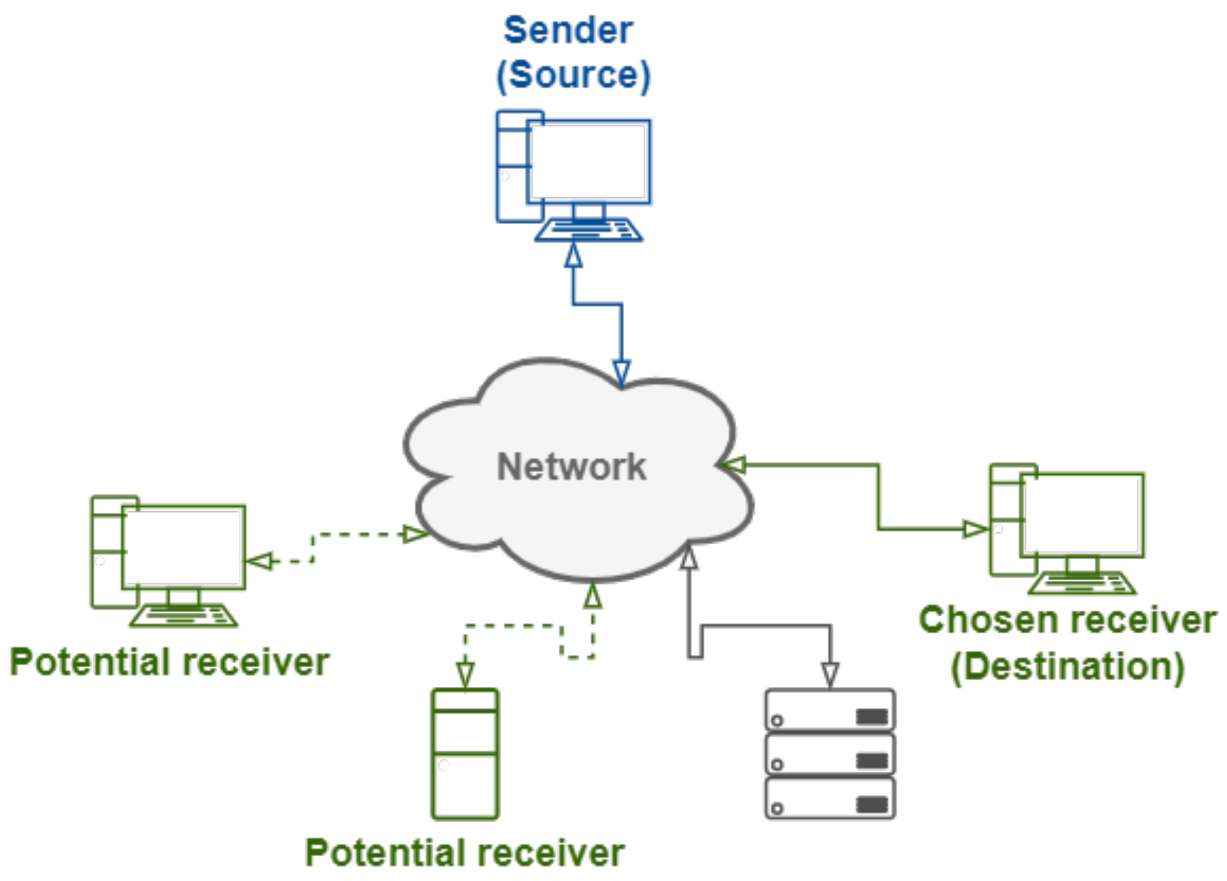


Different Ways to Exchange Messages

Multicast



Anycast



HTTP

- RFC 1945 (1.0), 2616 (1.1), 7540 (2.0), and 9114(3.0)
  - Client / Server Architecture
- Communication using HTTP messages
- HTTP defines how Web clients requests OBJECTS (used to be only web pages) and how servers transfers objects back.
- HTML is the data (to start)
  - But can be anything (XML, PHP, ASPX, HTML5)
  - Are there issues with the data format not being specified?
- Objects addressable by URLs
  - Uniform Resource Locator
  - Hides physical location (local or remote)

HTTP

- Widely known for driving the web, but its used to power many other things
- HTTP is:
  - A messaging format – how and when connections are stood up and torn down to exchange data
  - An on the wire format – Text for < HTTP/1.1 and Binary for HTTP/2.0 and later
  - A set of Predefined HTTP Verbs - **GET, PUT, POST, DELETE** - are the most common, but also include **HEAD, PATCH, OPTIONS** and **TRACE**
  - Negotiation and Control Driven by a text driven set of key-value attributes exchanged between the client and server – also called HTTP Headers

Some of the standard HTTP Headers
https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

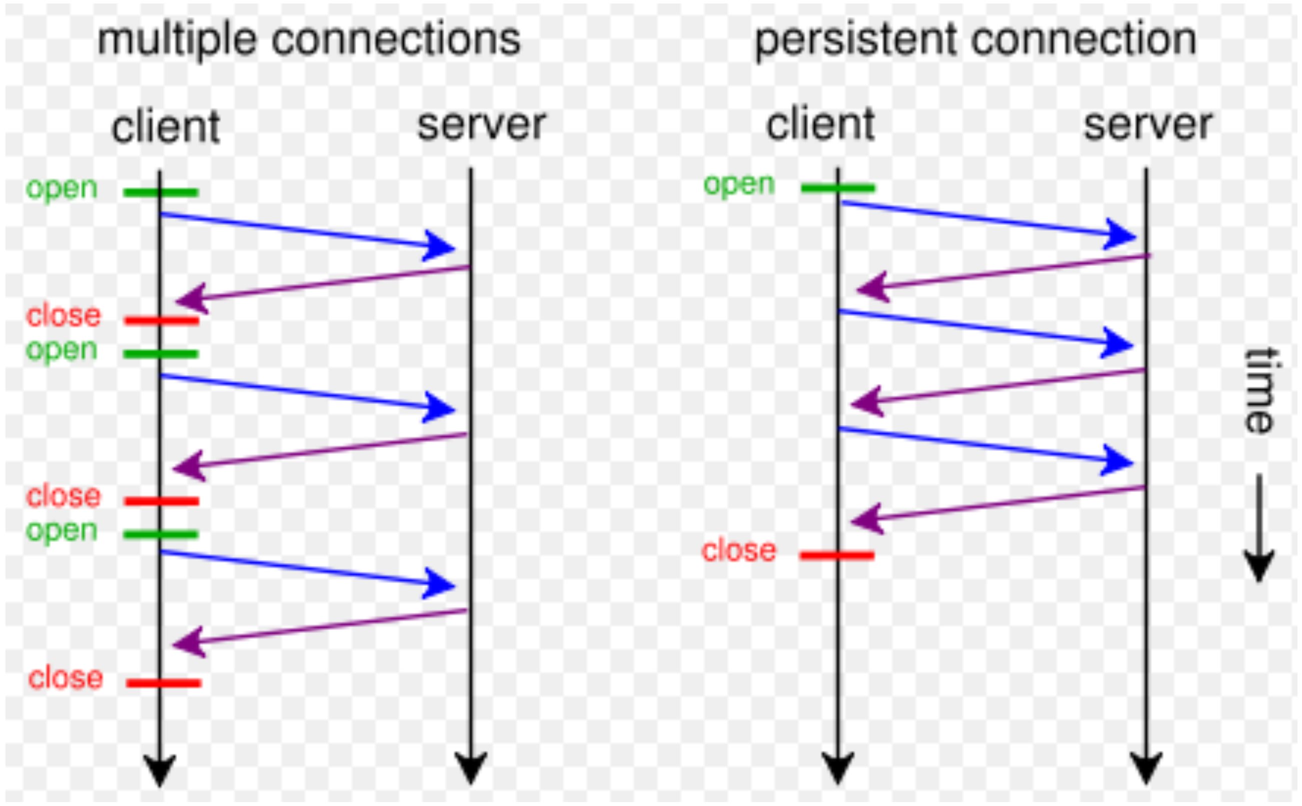
URL

http://www.cs.drexel.edu/~bmitchell/cs472/class2.pptx

- Step 1: Take hostname and translate via DNS to IP address
  - www.cs.drexel.edu -> 192.25.203.108 (locally cache that)
- Step 2: Take method (http) and translate to a port number and protocol:
  - http = TCP over port 80 by default
  - https = TCP over port 443 w/ TLS by default
  - Http or https with a :xxx behind the host name: HTTP over port xxx, or HTTP w/ TLS over port xxx respectively – for example: http://www.cs.drexel.edu:8080/~bmitchell/cs472/class2.pptx
- Step 3: Connect to IP address and port number and GET the rest
  - GET ~bmitchell/cs472/class2.pptx
- Step 4: Read the response

HTTP

- Stateless (important)
- Versions
  - 1.0 – Non-Persistent
    - w/ optional encryption
  - 1.1 – Persistent
    - w/ optional encryption
  - 2.0 – like 1.1
    - w/ layering
    - w/ multiplexing
    - w/ binary transport
    - w/ mandatory encryption
  - 3.0 – like 1.1
    - Over UDP
    - Connections handled higher up
    - w/ mandatory encryption

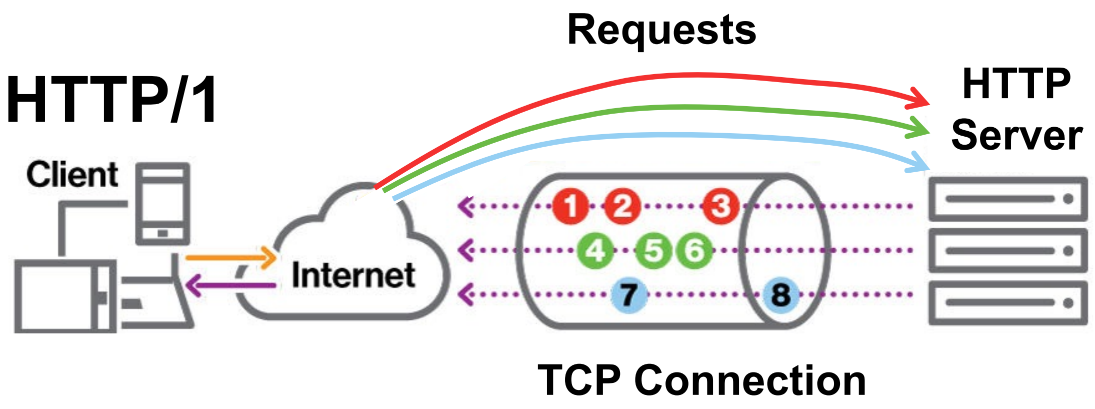


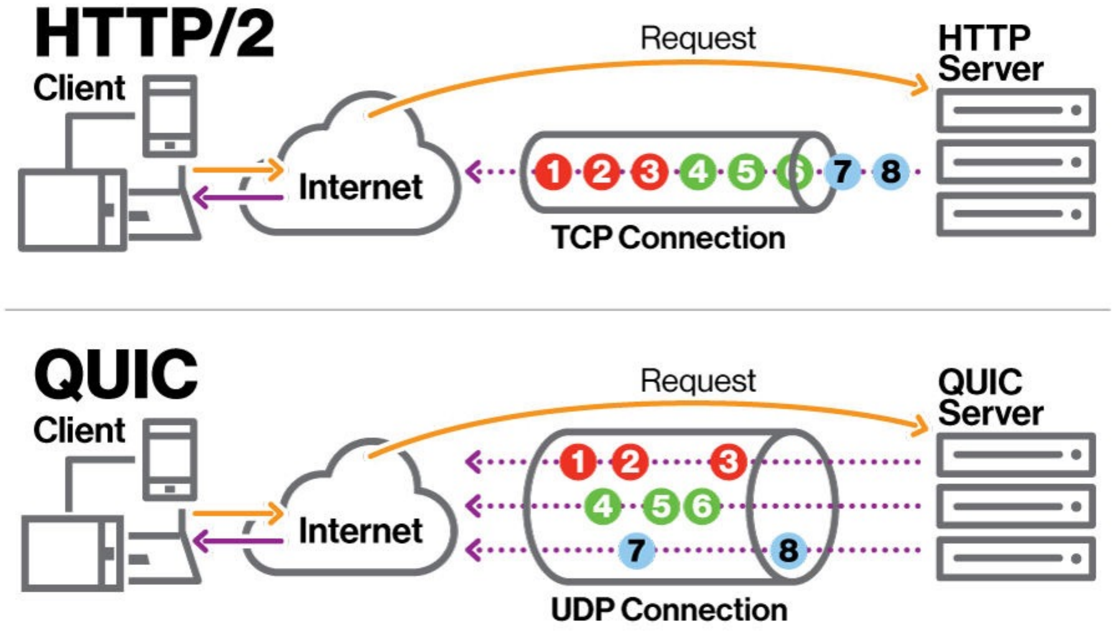
Why is being stateless important? When state is needed how is it managed?

HTTP variations V1.1, 2, and 3 (emerging)

HTTP/1 Text Based	HTTPS/1 Text Based	HTTP/2 Binary	HTTP/3 Binary
	TLS	TLS	QUIC
TCP	TCP	TCP	UDP
IP	IP	IP	IP
Ethernet WiFi MoCA	Ethernet WiFi MoCA	Ethernet WiFi MoCA	Ethernet WiFi MoCA
1997 Note HTTP/1 is really version 1.1 – 1.0 deprecated	**1997**	**2015**	**Not officially Released – Reference Implementations Exist**

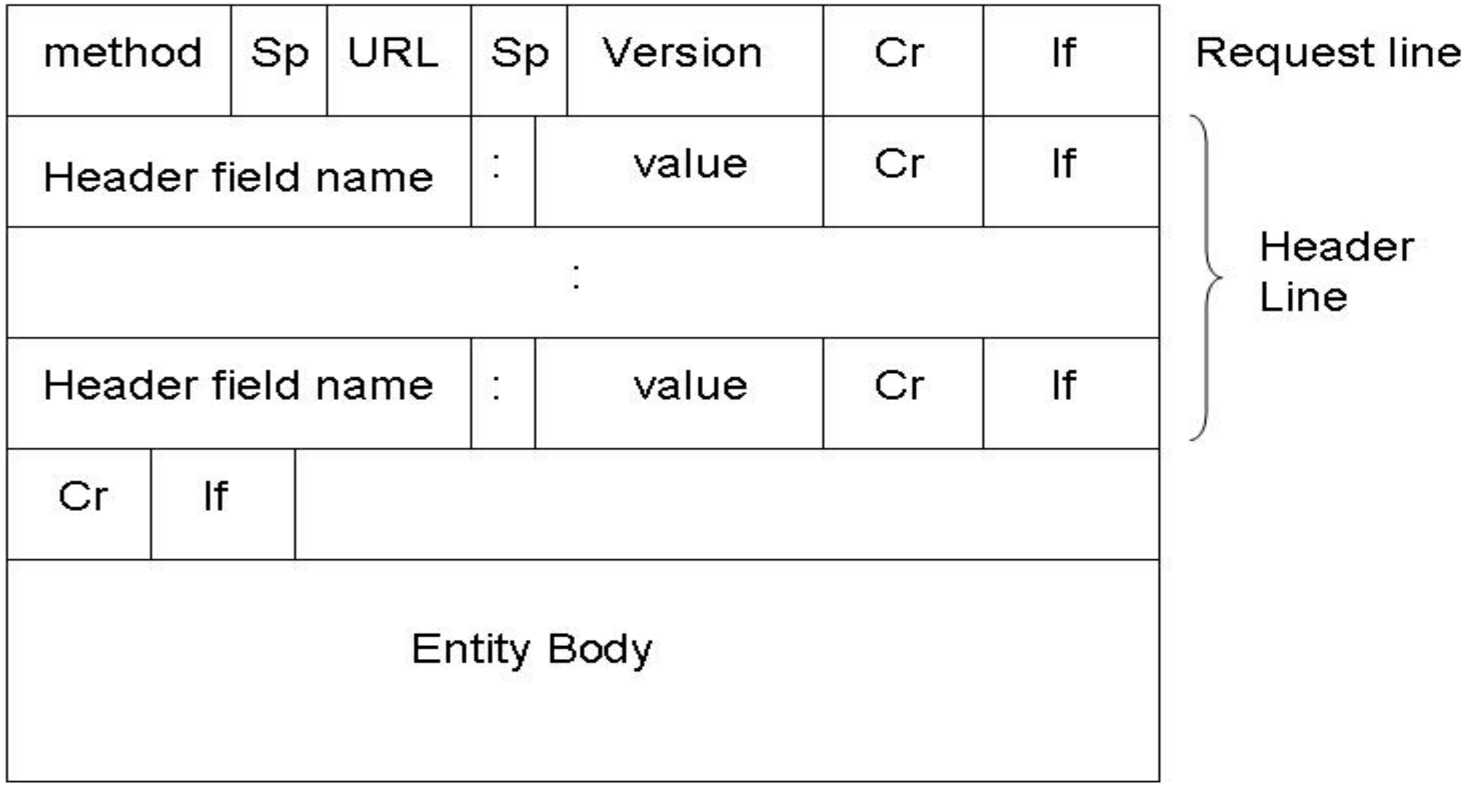
HTTP Message Exchange Format





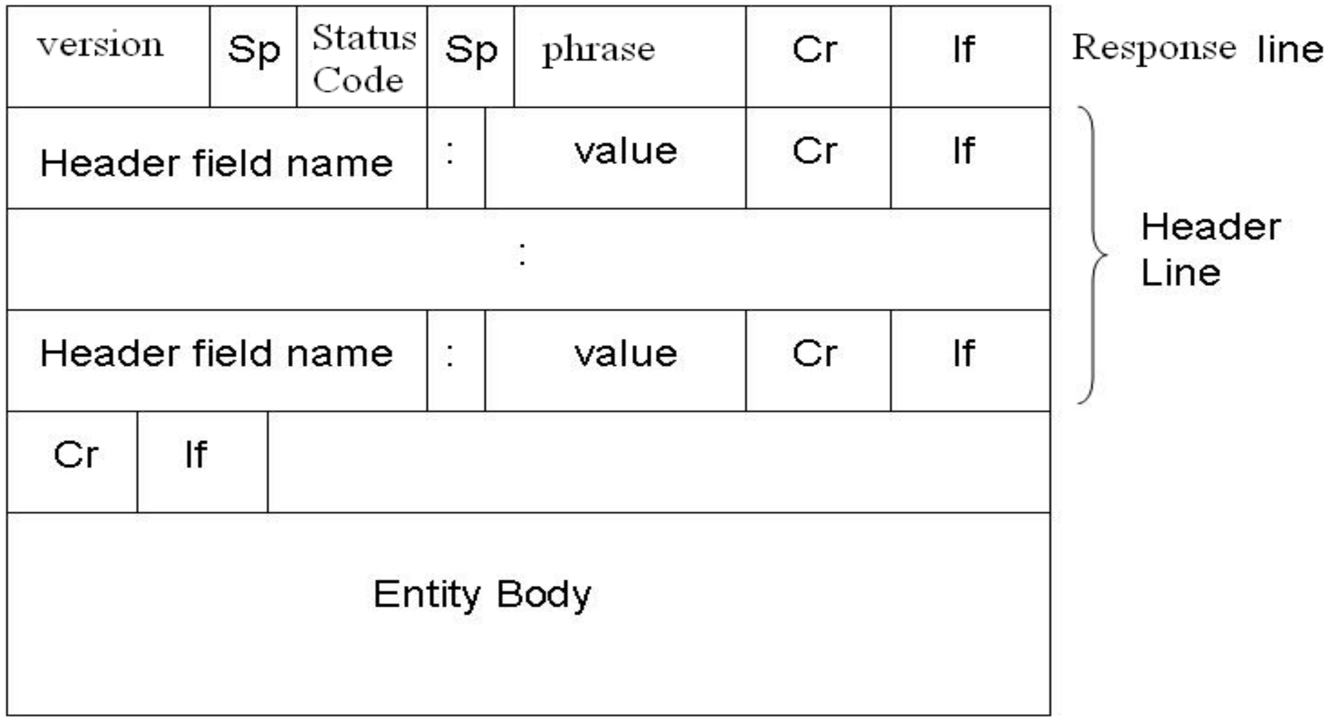
HTTP 2 & 3 Allow Requests and Responses to flow Asynchronously; HTTP 1.x is request/reply. Think about a web page and all of the resources

HTTP Request Message



Notice as compared to ARP, the transport also contains a header but its not binary aligned, it uses key value fields terminated by \r\n. Body starts after blank \r\n line

HTTP Response Message



Lets try something:

curl –v https://www.cs.drexel.edu

HTTP Protocol Versions Can be Negotiated Client Requested Example

One cool feature of HTTP is that via headers and status codes, HTTP protocol versions can be negotiated between clients and servers. HTTP status codes are well defined in the standard. See https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

```
-> GET foo.com HTTP/1.1


<- Status-Code: 200/OK
```

Simple exchange, server supports protocol requested by client

```
-> GET foo.com HTTP/1.1
   Upgrade: http/2
   Connection: Upgrade

<- Status-Code: 101/Switching-Protocols
   Upgrade: http/2
```

Client requests server to upgrade to http/2, server indicates that upgrade is OK

```
-> GET foo.com HTTP/1.1
   Upgrade: http/3, http/2
   Connection: Upgrade

<- Status-Code: 101/Switching-Protocols
   Upgrade: http/2
```

Client requests server to upgrade and gives server a preferred list of protocols, server picks the one it wants and returns it

HTTP Protocol Versions Can be Negotiated Server Enforced Example

One cool feature of HTTP is that via headers and status codes, HTTP protocol versions can be negotiated between clients and servers. HTTP status codes are well defined in the standard. See https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

```
-> GET foo.com HTTP/1.1


<- Status-Code: 426/Upgrade-Required
   Upgrade: http/3, http/2

-> Get foo.com HTTP/1.1
   Upgrade: http/2
   Connection: Upgrade

<- Status-Code: 101/Switching-Protocols
   Upgrade: http/2
```

Client wants to use 1.1, but server responds that it supports http/3 and http/2 with http3 being preferred

Client then initiates protocol transfer to what it wants, http/2 in this case

```
-> GET foo.com HTTP/1.1


<- Status-Code: 426/Upgrade-Required
   Upgrade: http/2

-> Get foo.com HTTP/1.1
   Upgrade: http/3
   Connection: Upgrade

<- Status-Code: 505/HTTP-Version-Not-Supported
   Upgrade: http/2, http/1.1
```

The client is trying to upgrade to http/3 but the server does not support it, so it returns a 505 with a list of protocols it supports in preferred order

HTTP basic operation

- GET all objects, but that could have problems
- How does the protocol adapt and evolve to different situations?
- How do we handle:
  - Getting the same object over and over?
  - Telling them who we are? (remember we're stateless)
  - Going to multiple sites?
  - Handling different types of objects?
  - (Don't worry about security – we'll talk about it later)

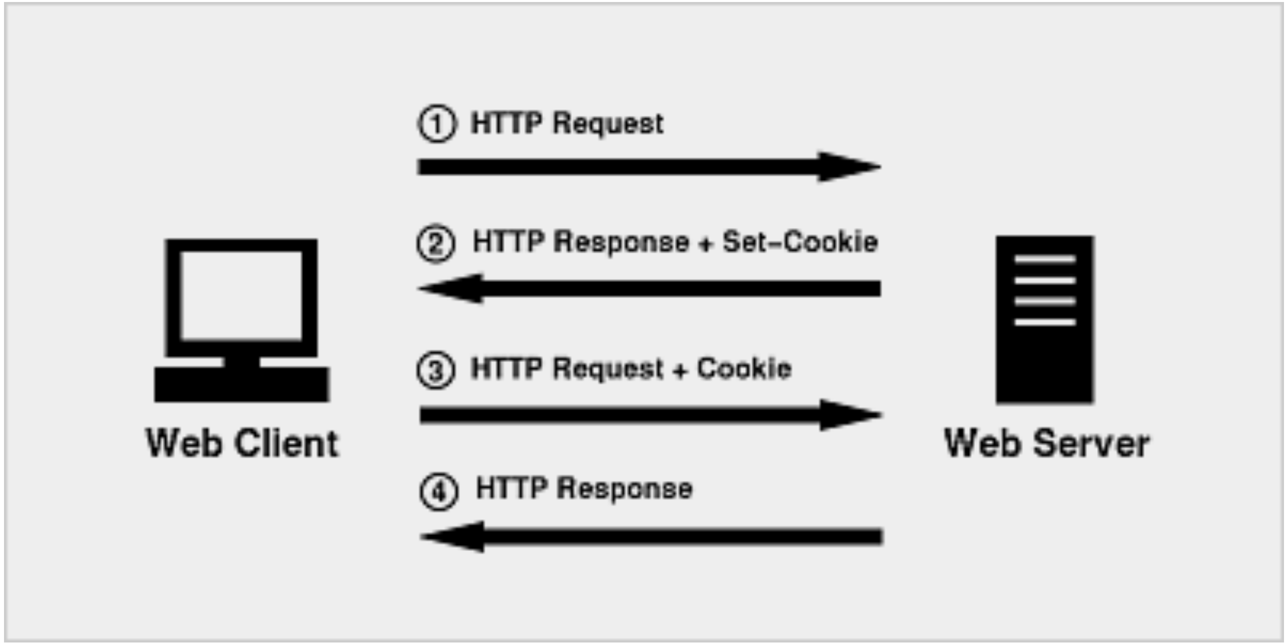
- HTTP will show how a protocol was designed and evolved over time to be better and more efficient.

Optimizations in responses – the Conditional GET

```
GET /sample.html HTTP/1.1
Host:  www.example.com

HTTP/1.1 200 OK
Date:  Tues, 27 Dec 2020 05:25:11 GMT
Last-Modified:  Thurs, 10 Apr 2008 13:24:52 GMT

GET /sample.html HTTP/1.1
Host:  www.example.com
If-Modified-Since:  Thurs, 10 Apr 2008 13:24:52 GMT

HTTP/1.1 304 Not Modified
Date: Tues, 27 Dec 2020 05:25:58 GMT
```

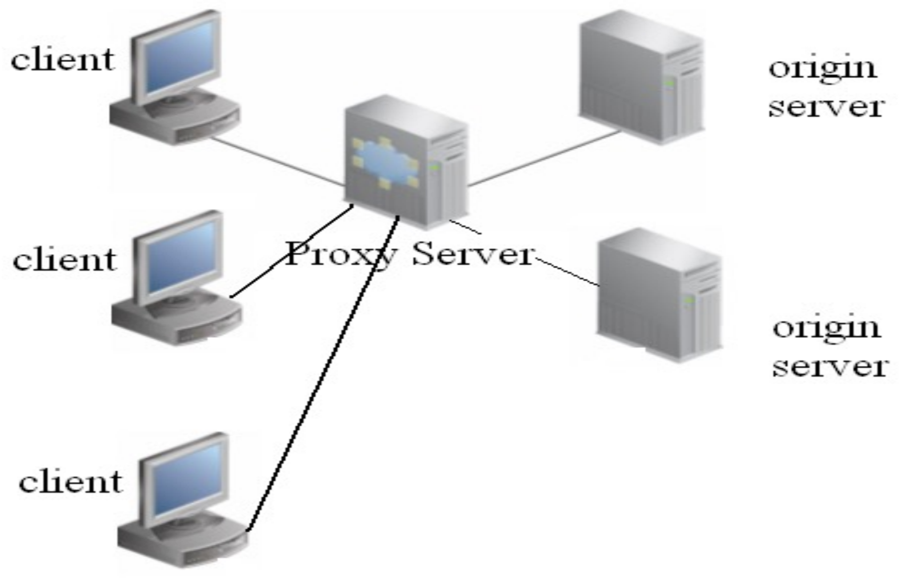
HTTP authentication

- HTTP is a stateless protocol, so how to we “login”?
- We have to have the client return an object which is a identifier on subsequent requests



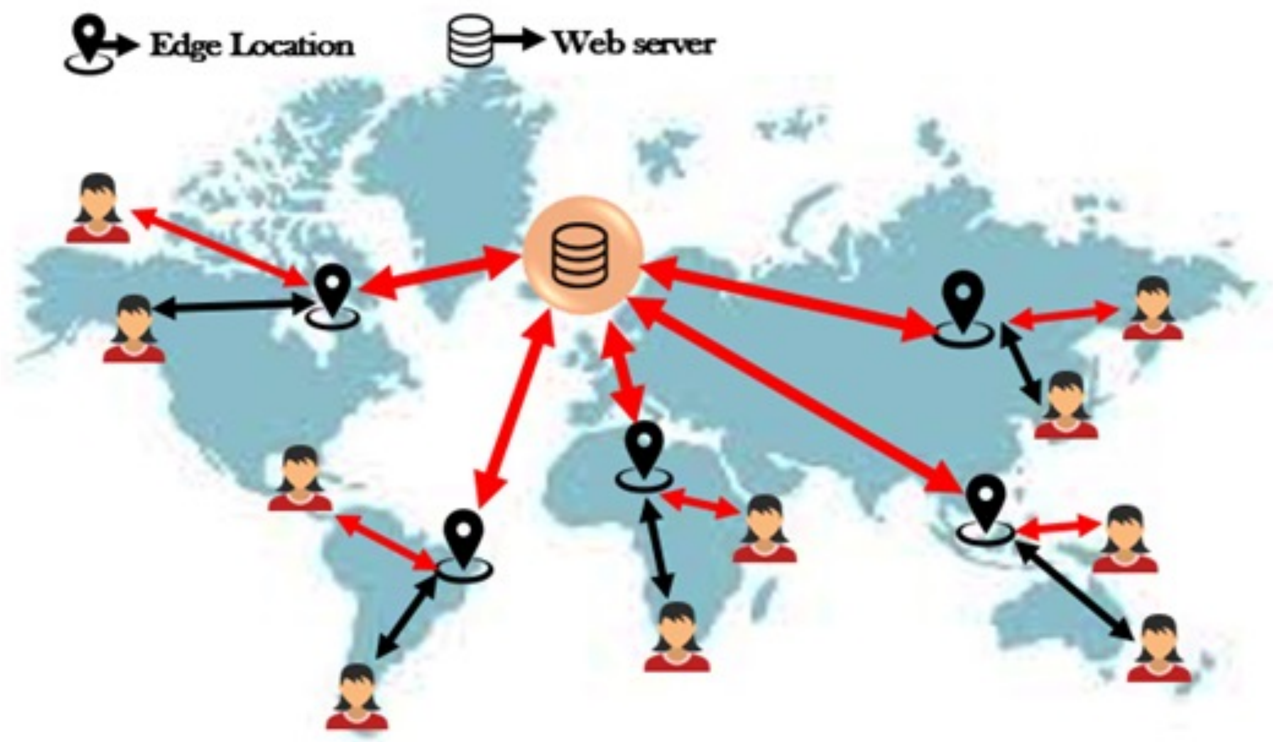
Optimizations in implementations – Web Caching

- To increase performance – content is distributed across a number of servers, each with a complete copy.
- Local DNSes are pointed to the closest copy.



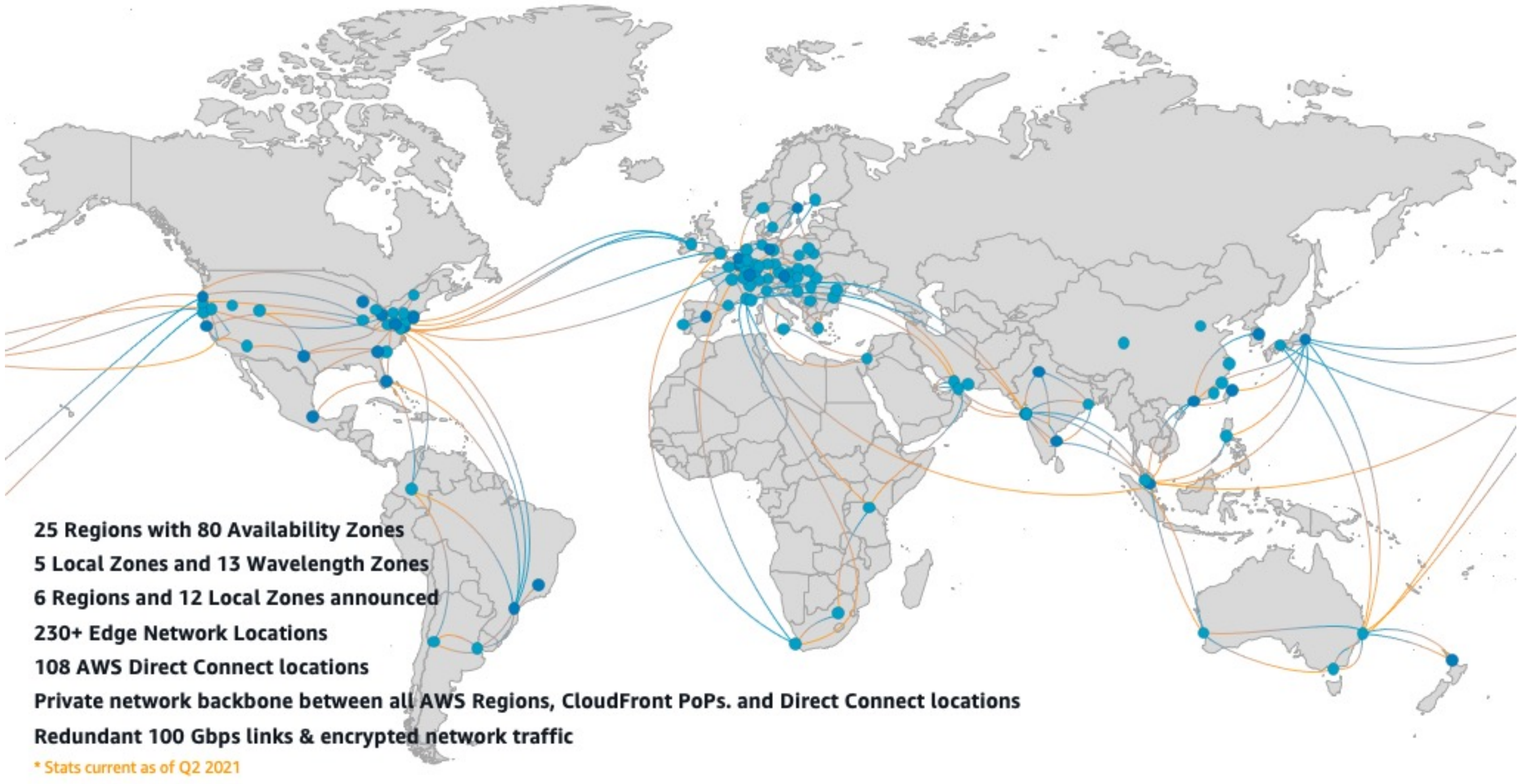
Optimizations in implementations – Content Delivery Networks (CDNs)

- To increase performance – content is distributed and delivered from a location closest to the person requesting it



Optimizations in implementations – Use the public internet for the last mile only, internal routing over an optimized private network

- To increase performance – jump on the AWS, Azure, or GCP network at the "first mile" and then jump off at the last mile to reach the destination.



25 Regions with 80 Availability Zones

5 Local Zones and 13 Wavelength Zones

6 Regions and 12 Local Zones announced

230+ Edge Network Locations

108 AWS Direct Connect locations

Private network backbone between all AWS Regions, CloudFront PoPs. and Direct Connect locations

Redundant 100 Gbps links & encrypted network traffic

* Stats current as of Q2 2021

Other topics in HTTP

- Extensibility
  - How can I add new things? Headers? Responses? Messages?
- Compatibility
  - How can a client and server agree on the same options?
  - What about if a server doesn't support what the client sends?
  - What about if a client doesn't support what the server responds with?

Data security

- Data can be secured by SSL/TLS using implicit method (also called the “web” model)
  - Port is inherently secure or insecure
  - Port 80 = insecure, port 443 = secure (https://)

- What about the data?
- How can I check to see if the data is right?

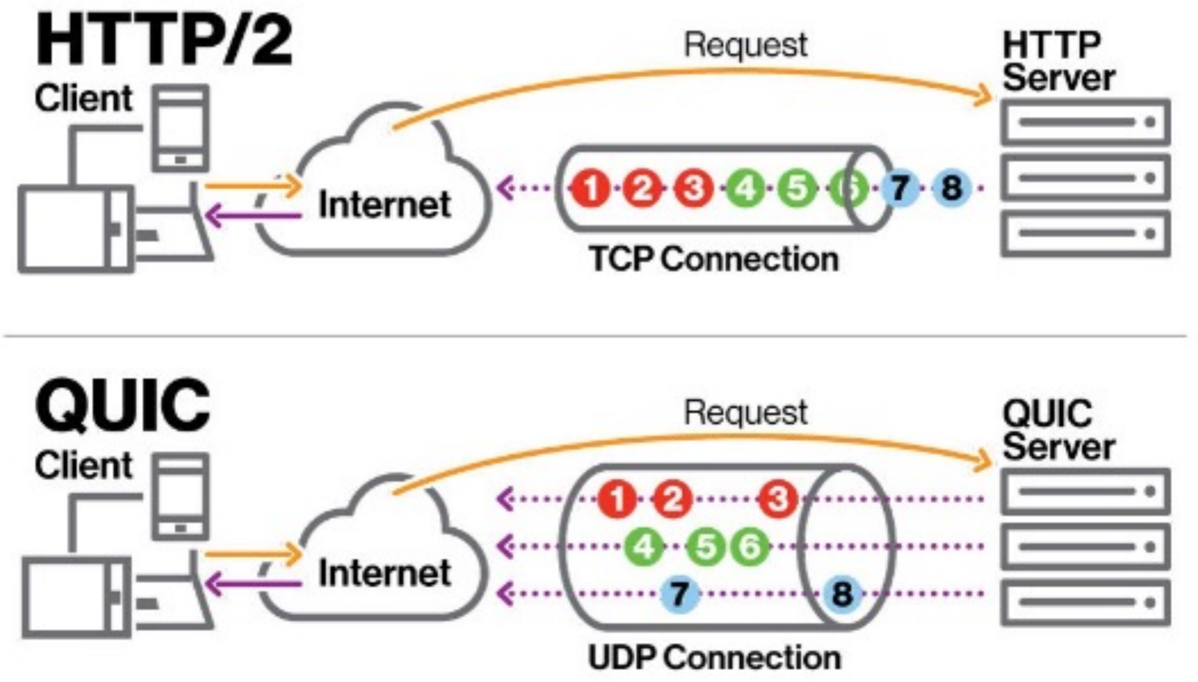
- No login/authentication, data is basically public (it’s all about sharing, right?)

- Performance?
- Does it depend on the number of hosts?

Support for optional data encryption is only in HTTP/1, HTTP/1.1, and has been dropped in HTTP2 and HTTP3

HTTP 2.0 (RFC 7540) & HTTP3.0 (RFC 9114)

- Basically the same methods, status codes, URIs and most header fields from HTTP 1.1.
- It's goal was to make HTTP faster and not have one stream of data – allows HTTP to have "stream-ids" and download items in parallel.
- So, how do we know if a browser or server supports it?



HTTP Protocols can be negotiated - Client Requested

One cool feature of HTTP is that via headers and status codes, HTTP protocol versions can be negotiated between clients and servers. HTTP status codes are well defined in the standard. See https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

```
-> GET foo.com HTTP/1.1

<- Status-Code: 200/OK
```

Simple exchange, server supports protocol requested by client

```
-> GET foo.com HTTP/1.1
   Upgrade: http/2
   Connection: Upgrade

<- Status-Code: 101/Switching-Protocols
   Upgrade: http/2
```

Client requests server to upgrade to http/2, server indicates that upgrade is OK

```
-> GET foo.com HTTP/1.1
   Upgrade: http/3, http/2
   Connection: Upgrade

<- Status-Code: 101/Switching-Protocols
   Upgrade: http/2
```

Client requests server to upgrade and gives server a preferred list of protocols, server picks the one it wants and returns it

HTTP Protocols can be negotiated – Server Mandated

One cool feature of HTTP is that via headers and status codes, HTTP protocol versions can be negotiated between clients and servers. HTTP status codes are well defined in the standard. See https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

```
-> GET foo.com HTTP/1.1


<- Status-Code: 426/Upgrade-Required
   Upgrade: http/3, http/2

-> Get foo.com HTTP/1.1
   Upgrade: http/2
   Connection: Upgrade

<- Status-Code: 101/Switching-Protocols
   Upgrade: http/2
```

Client wants to use 1.1, but server responds that it supports http/3 and http/2 with http3 being preferred

Client then initiates protocol transfer to what it wants, http/2 in this case

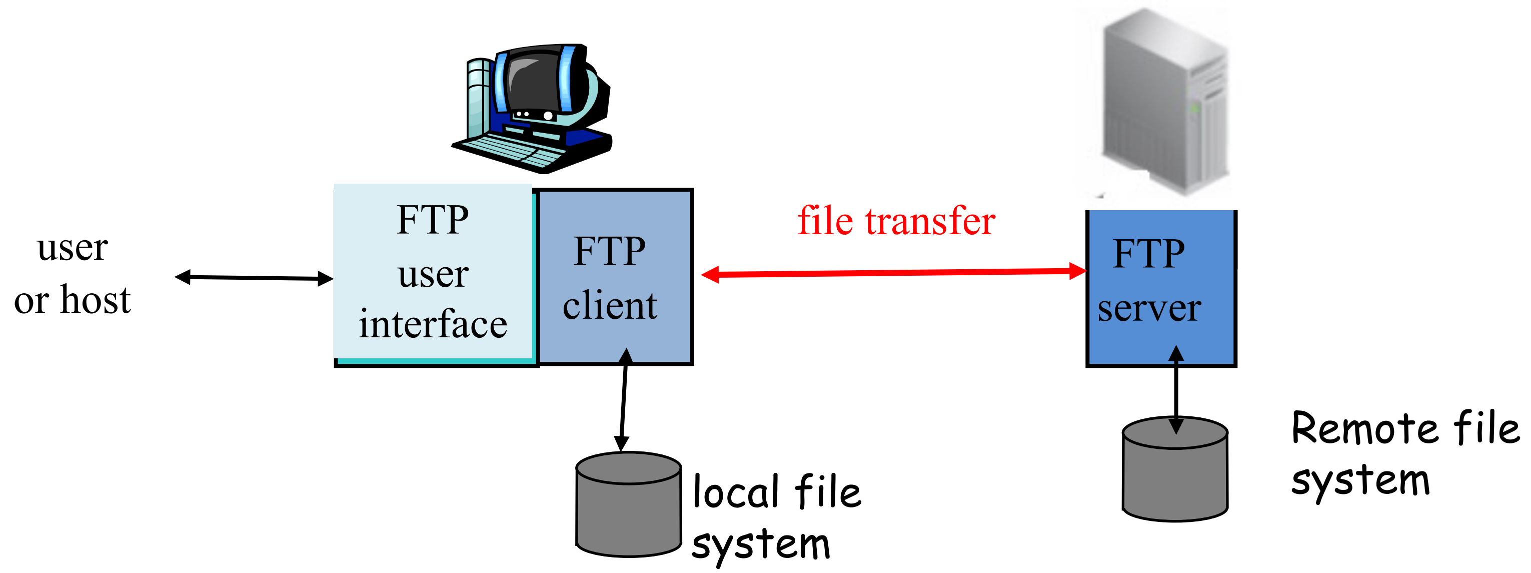
```
-> GET foo.com HTTP/1.1


<- Status-Code: 426/Upgrade-Required
   Upgrade: http/2

-> Get foo.com HTTP/1.1
   Upgrade: http/3
   Connection: Upgrade

<- Status-Code: 505/HTTP-Version-Not-Supported
   Upgrade: http/2, http/1.1
```

The client is trying to upgrade to http/3 but the server does not support it, so it returns a 505 with a list of protocols it supports in preferred order

15

FTP

- RFC 959
- File transfer between client and server
- FTP server listens at TCP port 21 (for commands), another port is used for the data transfer (typically port 20)



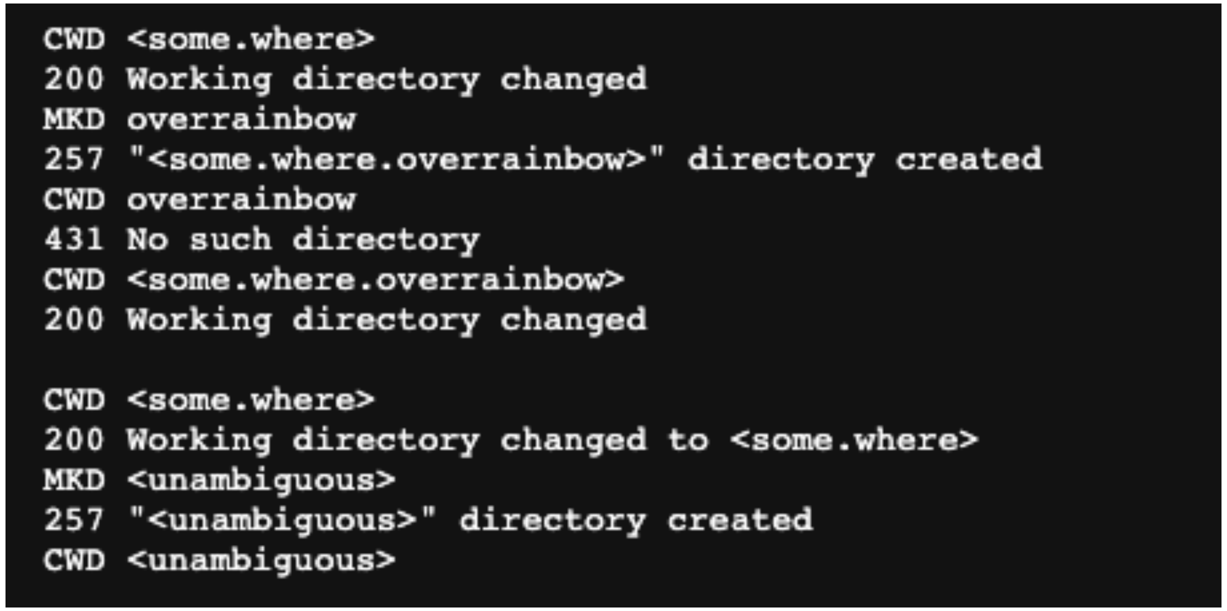
How FTP works

- Client connects to TCP port 21
  - Logs in with usercode and password (usually)
  - But other ways are available
- Can change directory
  - Both locally and remotely
- Can push or pull files (GET/PUT)
- Opens up a separate port for the data transfer (usually TCP port 20), but can be anything
  - Server usually calls the client back, but can also be configured for client to call server
- Borrows from the telnet (RFC 854) protocol to exchange information via commands

- Goal here = different protocols have different "conversations", no matter what layer.

FTP commands

- USER
- PASS
- CWD
- PWD
- PASV
- PORT
- RETR
- STOR
- LIST



```
CWD <some.where>
200 Working directory changed
MKD overrainbow
257 "<some.where.overrainbow>" directory created
CWD overrainbow
431 No such directory
CWD <some.where.overrainbow>
200 Working directory changed

CWD <some.where>
200 Working directory changed to <some.where>
MKD <unambiguous>
257 "<unambiguous>" directory created
CWD <unambiguous>
```

- Each has a list of 3 number Reponses which tell if the command was successful or not.

FTP security & modes

- IMPLICIT SSL
  - Two sets of ports – one unsecure (20/21), one secure (989/990)
- EXPLICIT SSL
  - One set of ports (20/21) – commands to change between insecure and secure (AUTH TLS) and back (CCC)
  - Can secure control and data channels independently (PROT)
- Which is more secure?

- Login as part of protocol (usually usercode/password) – other methods available
- Anonymous login can be configured (default off) with password = identification.

File transfer protocols

- FTP
  - Uses a command port and data port over TCP
- TFTP
  - Copies files over UDP
- BitTorrent
  - Copies chunks from multiple sources simultaneously
  - More of a mesh of file servers
  - http://wiki.theory.org/BitTorrentSpecification
- SFTP (SSH)
  - Command and data use same port, plus both commands and data are encrypted (this is really the modern standard)
- Others (XMODEM, YMODEM, Kermit)?

- Same or different?

Data security

- So, what data ports do we let in?
- What data ports do we let out?

- How do we know what data flows over that connection?
- Can we check it to make sure that it's right?

- Does it depend on the protocol? (think: HTTP, FTP)

- Which protocols are more secure?

- Does it matter to where? (good servers vs bad servers)

This is the realm where the network engineer comes in and specialized hardware, generally firewalls (hardware or software-defined) come into the picture. There are also a variety of additional services we will touch on later that help in this space – intrusion detection, bot detection, behavioral analysis, denial of service hardening, etc

Socket Programming

- Sockets are a protocol independent method of creating a connection between processes. Sockets can be either TCP or UDP (for now)
- And then we can modify the defaults via options (setsockopt)

- The TCP and UDP protocols use ports to map incoming data to a particular process/thread on a computer.

- We start by establishing our Internet Endpoint (socket) and then determining who to talk to (their socket) and then connect the two together.

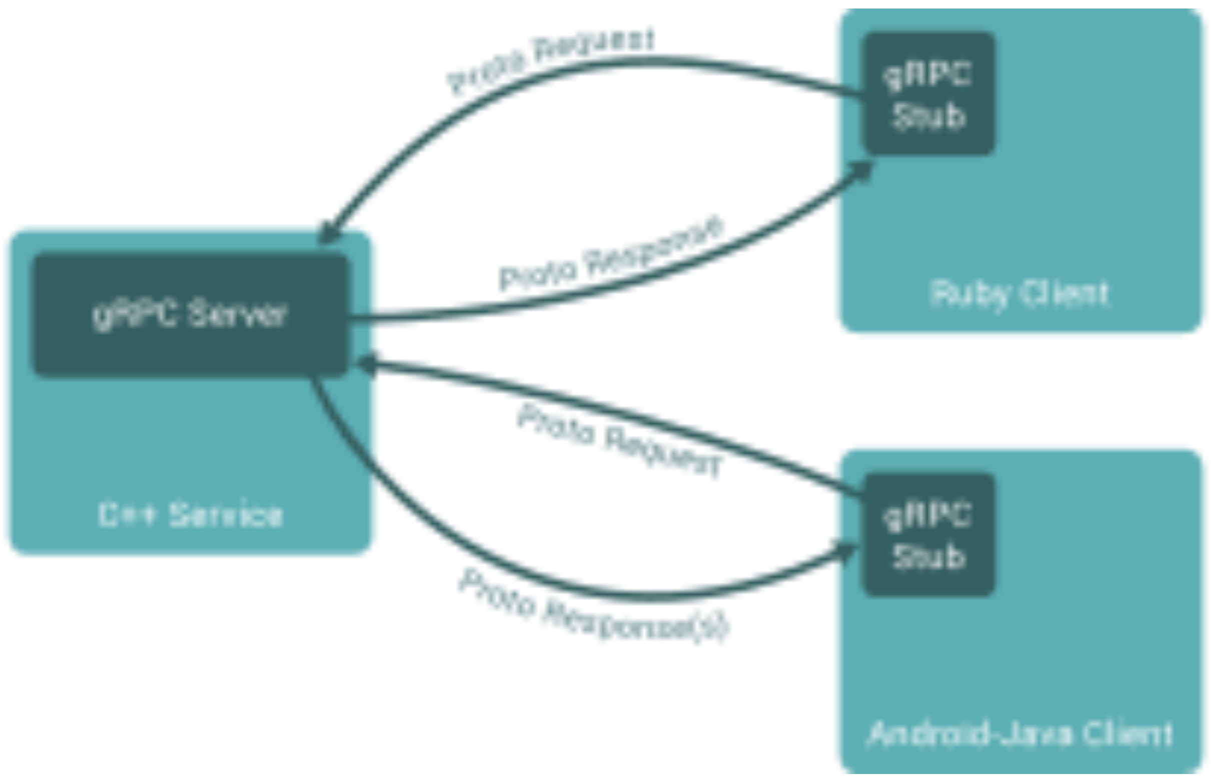
Remote Procedure Call

- Another method of talking over a network – old way (COM/DCOM), new way GRPC
- Program split into two or more modules and modules can be split across the network.
- Procedure call turns into message over the network and program waits for result
- Return message gets copied into procedure variables and/or function return value.

- Procedure calls defined in IDL (Interface Definition Language) as well as definition of servce (service #)
- Marshalling of data variables done in XDR (eXternal Data Representation) which is Big Endian.
- Used for Microservices!

- How do we do that in sockets?

Remote Procedure Call – GRPC is a modern RPC protocol

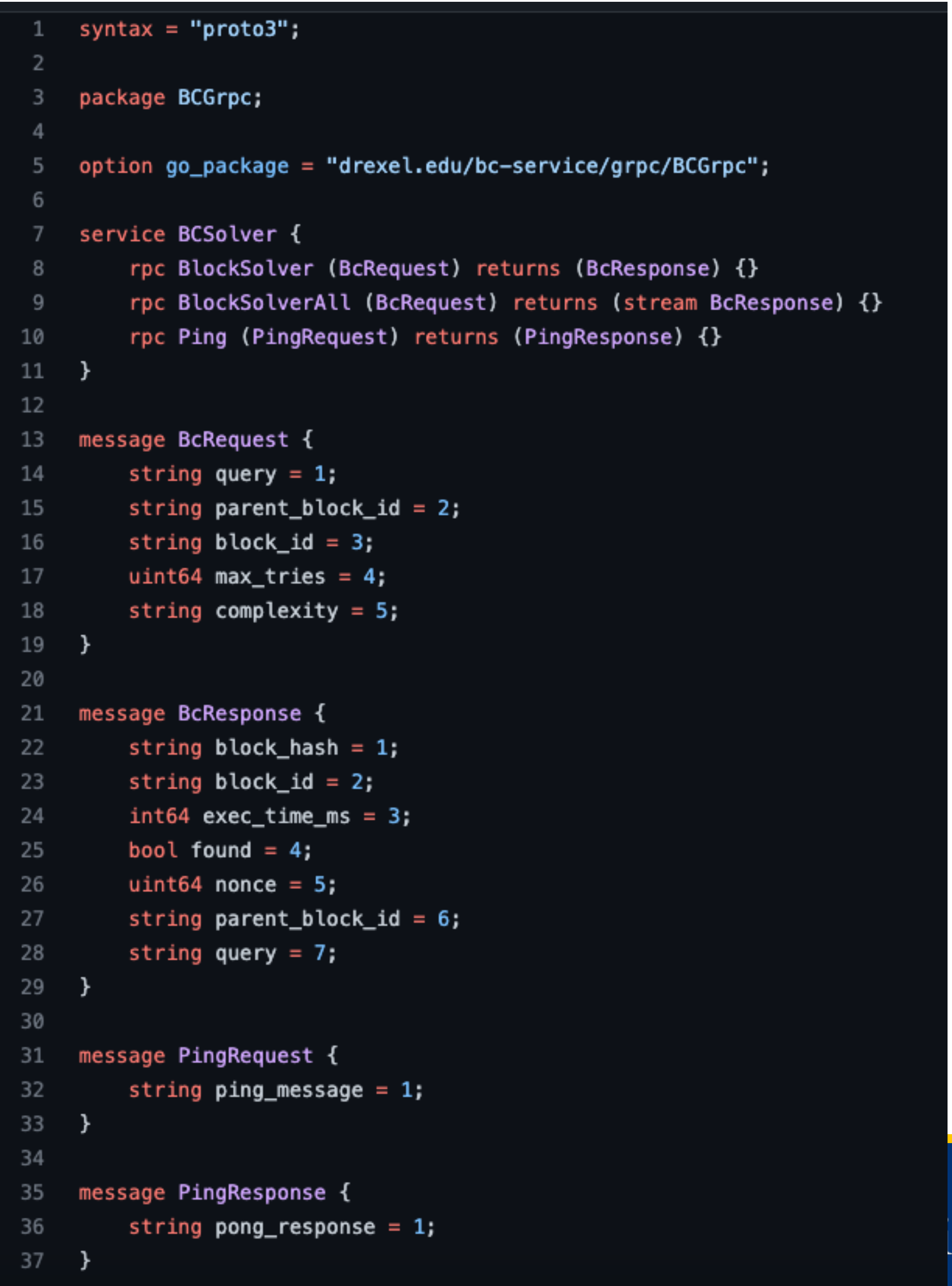


Features

- Typesafe, stubs and clients are generated from interface definition files
- Fast, encoding is binary, serialization is fast
- Polyglot, clients and servers can be written in different languages
- Interoperable, wire format is over HTTP/2 so it works well with existing network infrastructure

GRPC Site: https://grpc.io/

Code Demo (in go): https://github.com/ArchitectingSoftware/se577-webservices-demo/tree/main/grpc

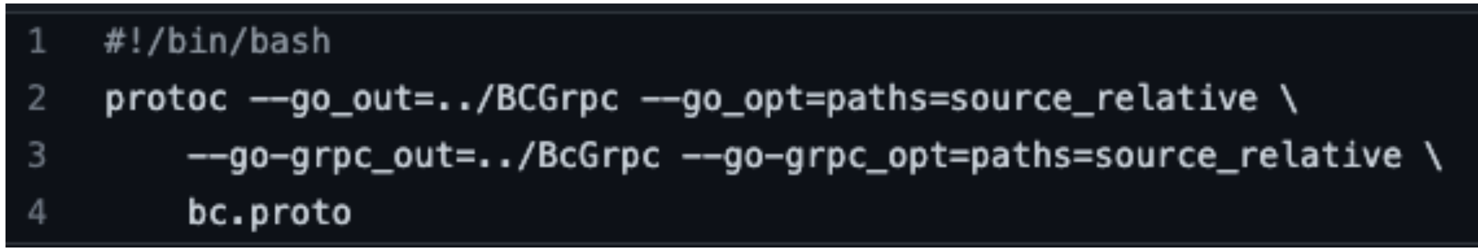


```
syntax = "proto3";

package BCGrpc;

option go_package = "drexel.edu/bc-service/grpc/BCGrpc";

service BCSolver {
    rpc BlockSolver (BcRequest) returns (BcResponse) {}
    rpc BlockSolverAll (BcRequest) returns (stream BcResponse) {}
    rpc Ping (PingRequest) returns (PingResponse) {}
}

message BcRequest {
    string query = 1;
    string parent_block_id = 2;
    string block_id = 3;
    uint64 max_tries = 4;
    string complexity = 5;
}

message BcResponse {
    string block_hash = 1;
    string block_id = 2;
    int64 exec_time_ms = 3;
    bool found = 4;
    uint64 nonce = 5;
    string parent_block_id = 6;
    string query = 7;
}

message PingRequest {
    string ping_message = 1;
}

message PingResponse {
    string pong_response = 1;
}
```

Remote Procedure Call – GRPC example Protocol Buff Definition

Features

- Service Definition – What are the APIs?
- Message Definition – What is the structure and data types of the messages



```
#!/bin/bash
protoc --go_out=../BCGrpc --go_opt=paths=source_relative \
    --go-grpc_out=../BcGrpc --go-grpc_opt=paths=source_relative \
    bc.proto
```

The protoc compiler generates the client and server stubs in the identified programming language, the example above uses the go programming language, but nearly every programming language is supported

Socket Programming - TCP Client Algorithm

- Create a socket
- Tell OS to use any port for this socket (Bind port 0)
- Determine who to connect to (utilities, cmd line)
- Connect to remote (CONNECT)

- /* DO PROTOCOL (depends on protocol which order)*/
    - BEGIN
        - WRITE request
        - READ answer
        - Multiple times depending on algorithm
    - END
- CLOSE

Socket Programming - TCP Server Algorithm

- Create a socket
- Tell OS to use specified port for this socket (Bind)
- Tell OS that this process is listening for new requests (Listen)
- WHILE (TRUE)
- BEGIN
    - SOCK = Get next connection (ACCEPT)
    - /* DO PROTOCOL (depends on protocol which order)*/
    - BEGIN
        - READ request
        - WRITE answer
    - END
    - CLOSE SOCK
- END;

TCP Socket issues

- So, how does socket know where the message begins and ends?
- TCP is a stream of bytes, so it is up to the application to make sure that the program blocks together enough data to make up the application's message (PDU)
- UDP is message-oriented, so we don't have this problem.
- Is this a client or server problem? Or both?

Socket programming should be thought of as network programming without training wheels, these days most people use libraries that make this stuff a lot easier. Sockets are also really close to the transport layer, newer frameworks work directly with higher layer protocol constructs (e.g., HTTP)

Example for HTTP: Fastify, Golang Gin, etc

TCP Socket issues

- It is both since both agree on the PDU definition (one or more)
- It is a **GREAT IDEA (hint, hint)** to have a routine which reads from a socket and constructs the PDU (for each PDU that you have)

```
        Integer get_next_pdu( sock, buffer, maxlen );
```

- Your TCP server/client may work – but if TCP fragments in the middle, you're sunk.
- **Recv(2048) is NOT ENOUGH!**
- **At the socket level, as mentioned, you are responsible for managing all flow control. What did I do for my socket tutorial? What are other approaches?**

Types of servers

- Iterative = server handles one client at a time
  - Not really useful in most cases (to serve as many clients at once)
  - Can control the backlog queue, the second parameter to `listen(sock,backlog)` is basically the queue depth.
  - Where would they be useful?

- Concurrent = server handles multiple clients at a time
  - Three methods
  - Process/thread = server maintains main thread/process which processes new requests and creates a separate thread/process for each client.
  - Threadpool = server maintains a pool of threads and then an available thread from the pool picks up the workload
  - Select = server maintains list of outstanding sockets and asks OS (through select() API) if any have any activity or errors

- Why would you pick one or the other?

TCP Server Algorithm (CONCURRENT)

```
•   Create a socket
•   Tell OS to use specified port for this socket (Bind)
•   Tell OS that this process is listening for new requests (Listen)
•   WHILE (TRUE)
•   BEGIN
    •   SOCK = Get next connection (ACCEPT)
    •   OPERATING SYSTEM MAGIC START – THREADS, ETC
    •   BEGIN
        •   READ request
        •   WRITE answer
    •   END
    •   CLOSE SOCK
    •   OPERATING SYSTEM MAGIC END – THREADS, ETC
•   END;
```

See a simple threaded example here:
https://github.com/ArchitectingSoftware/CS472-Class-Files/blob/main/socket-tutorial/server3.c

UDP Server Algorithm

- Create a socket
- Tell OS to use specified port for this socket (Bind)
- WHILE (TRUE)
- BEGIN
    - /* DO PROTOCOL */
    - READ request
    - WRITE answer
- END;

UDP Client Algorithm

- Create a socket
- Tell OS to use specified port for this socket (Bind)
- BEGIN
  - /* DO PROTOCOL */
  - WRITE request
  - READ answer
- END;

DNS (Domain Name System)

- Takes names and translates to IP addresses (v4 and v6)
  - www.cs.drexel.edu => 129.25.203.108 or IPv6 address (FECO::42F4:800:BFF:FE3A:211C)
- Can do other things:
  - Host aliasing – mailserver.drexel.edu and webserver.drexel.edu might be the same machine (point to same IP address)
  - bob@drexel.edu = bob@mailrelay.irt.drexel.edu (simplify naming)
  - www.drexel.edu (or www.ebay.com) may be more than one IP address (load balancing)

- Are there problems with #3?

How DNS works

- Client sends request to his name server
- DNS server address is either hard coded, or provided when a IP address is assigned over DHCP
- If his name server doesn't know it, the name server asks it's name server
- If it reaches the root before being resolved, then it asks the root (TLD) of the request (for example .edu)
- If the root doesn't know it, it is sent down the tree to the request's name server.

- Recursive – each server saves any address and a timestamp
- Answers are marked as "authoritative answer" or "non-authoritative answer"
- AA means that a server who is responsible for a domain answered.

DNS example

- Client tux.cs.drexel.edu asks for IP address for www.espn.com.
- Cs.drexel.edu doesn't know it, it goes to drexel.edu
- Drexel.edu doesn't know it, it goes to .edu root.
- .edu root doesn't know it, it goes to .com root.
- .com root goes to espn.com.
- Answers are propagated back to the requestor, and cached by all in the middle.

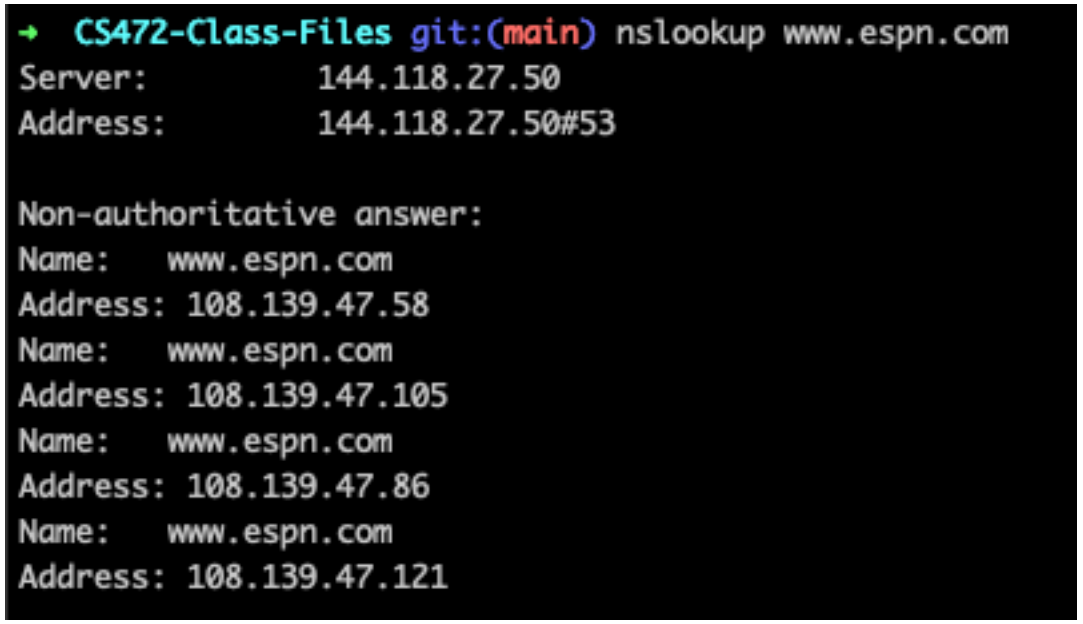
DNS record types (PDUs)

- A = IPv4 address record
- AAAA = IPv6 address record
- CNAME = Canonical Name (alias, processing continues)
- MX = mail server record
- PTR = name record (processing stops)
- LOC = geographic location

- Defined by RFC 1035 (IPv4), 3596 (IPv6).

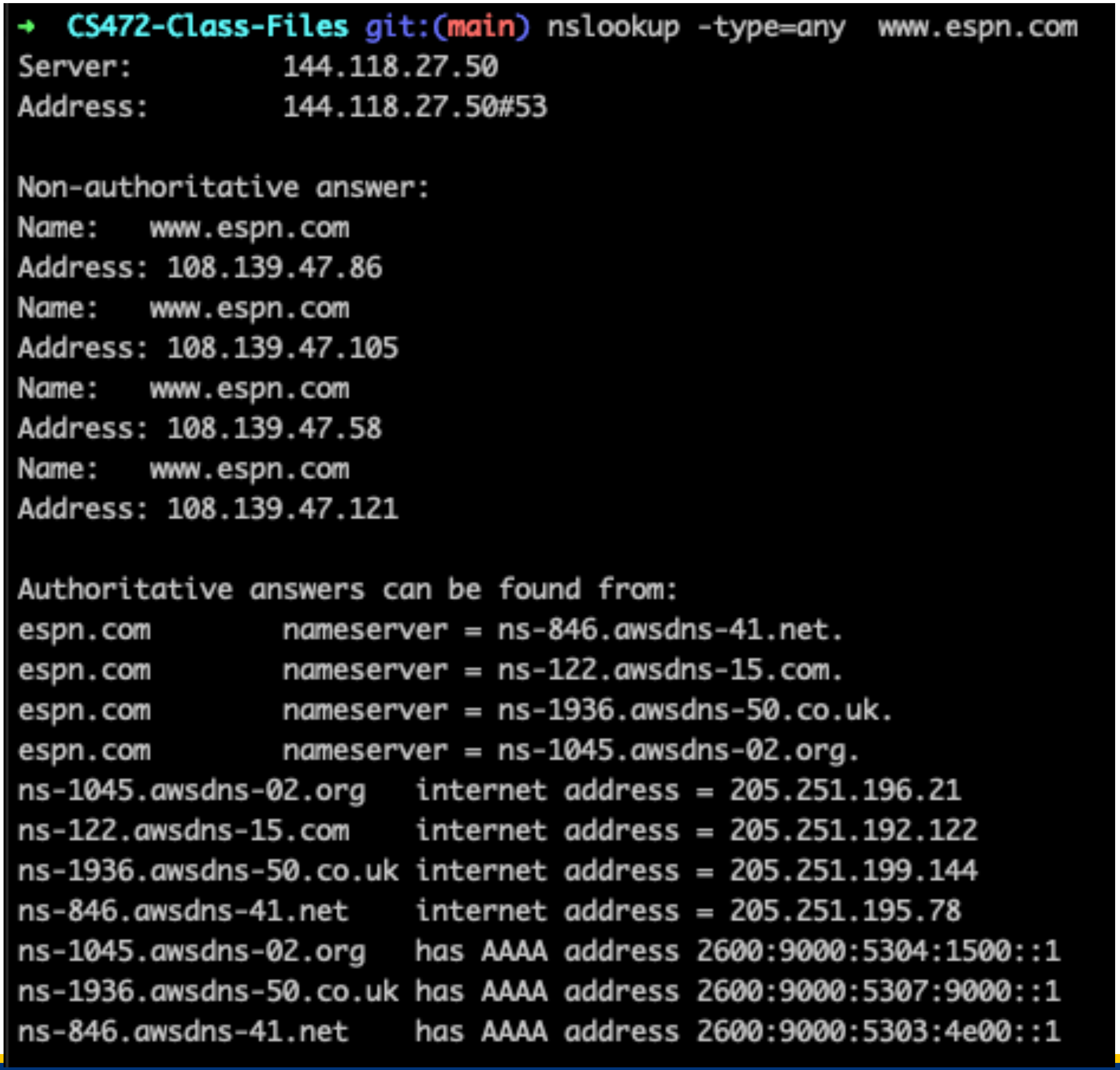
Problems?

- Do you think that DNS is perfectly done?
  - Not one server with the “answer” but a hierarchy.
  - Is that a good or bad thing?
- Redundancy?
- What is the conversation of the protocol?
- How about the security of the protocol?
- How do I trust the next person in the chain?
- Security of DNS – how do you know DNS is returning the right information?
  - http://isc.sans.edu/diary/How+do+you+monitor+DNS%3F/16661
  - How do you know who’s asking is the right person?



```
➜  CS472-Class-Files git:(main) nslookup www.espn.com
Server:         144.118.27.50
Address:        144.118.27.50#53

Non-authoritative answer:
Name:   www.espn.com
Address: 108.139.47.58
Name:   www.espn.com
Address: 108.139.47.105
Name:   www.espn.com
Address: 108.139.47.86
Name:   www.espn.com
Address: 108.139.47.121
```

Using nslookup to learn more



```
➜  CS472-Class-Files git:(main) nslookup -type=any  www.espn.com
Server:         144.118.27.50
Address:        144.118.27.50#53

Non-authoritative answer:
Name:   www.espn.com
Address: 108.139.47.86
Name:   www.espn.com
Address: 108.139.47.105
Name:   www.espn.com
Address: 108.139.47.58
Name:   www.espn.com
Address: 108.139.47.121

Authoritative answers can be found from:
espn.com        nameserver = ns-846.awsdns-41.net.
espn.com        nameserver = ns-122.awsdns-15.com.
espn.com        nameserver = ns-1936.awsdns-50.co.uk.
espn.com        nameserver = ns-1045.awsdns-02.org.
ns-1045.awsdns-02.org   internet address = 205.251.196.21
ns-122.awsdns-15.com    internet address = 205.251.192.122
ns-1936.awsdns-50.co.uk internet address = 205.251.199.144
ns-846.awsdns-41.net    internet address = 205.251.195.78
ns-1045.awsdns-02.org   has AAAA address 2600:9000:5304:1500::1
ns-1936.awsdns-50.co.uk has AAAA address 2600:9000:5307:9000::1
ns-846.awsdns-41.net    has AAAA address 2600:9000:5303:4e00::1
```

Looks like espn.com is hosted by AWS.

The AWS DNS System is called Route53

Other problems?

- How do you know what is correct traffic?
- What about "good" traffic vs "bad" traffic?
  - Mirai
  - Kaminsky
- What about security of the data?

- Newer ideas
  - DoT (DNS over TLS) – RFC 7858
  - DoH (DNS over HTTPS) – RFC 8484

  - How do they change the conversation?
  - Better/worse? "IT depends"

The Big Picture - Trust

- Do we trust information that we get on the Internet?
- "If it came from the Internet, it must be right."

- Do you trust any one source (Wikipedia, Google)?

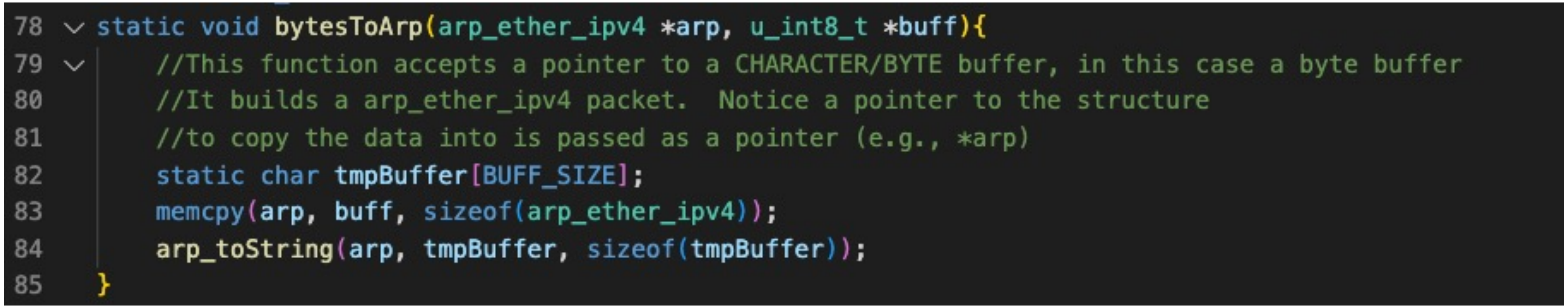
- What about trust of people who have access to the Internet?
  http://chronicle.com/article/The-Shadow-Scholar/125329/
  http://theory.stanford.edu/~aiken/moss/

Homework #1 Review

- The intent was to get reacquainted with systems programming in C, especially how to work with byte aligned data structures.
- It also covered things like network byte order (big endian)

QUESTION: Why do you think network byte order was defined as big endian when most of our computers these days use little endian

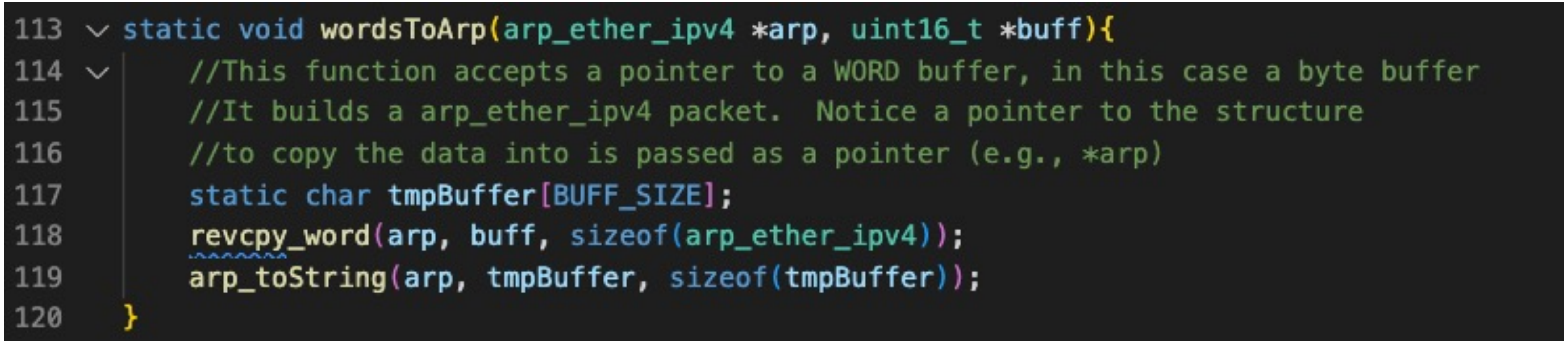
Homework #1 Solutions – Working with the byte array



```
static void bytesToArp(arp_ether_ipv4 *arp, u_int8_t *buff){
    //This function accepts a pointer to a CHARACTER/BYTE buffer, in this case a byte buffer
    //It builds a arp_ether_ipv4 packet.  Notice a pointer to the structure
    //to copy the data into is passed as a pointer (e.g., *arp)
    static char tmpBuffer[BUFF_SIZE];
    memcpy(arp, buff, sizeof(arp_ether_ipv4));
    arp_toString(arp, tmpBuffer, sizeof(tmpBuffer));
}
```

Since Big/Little endian is only a concern with byte ordering on data types larger than one byte (eg, 16 bit, 32 bit, 64 bit), when working with bytes you can just copy them over.

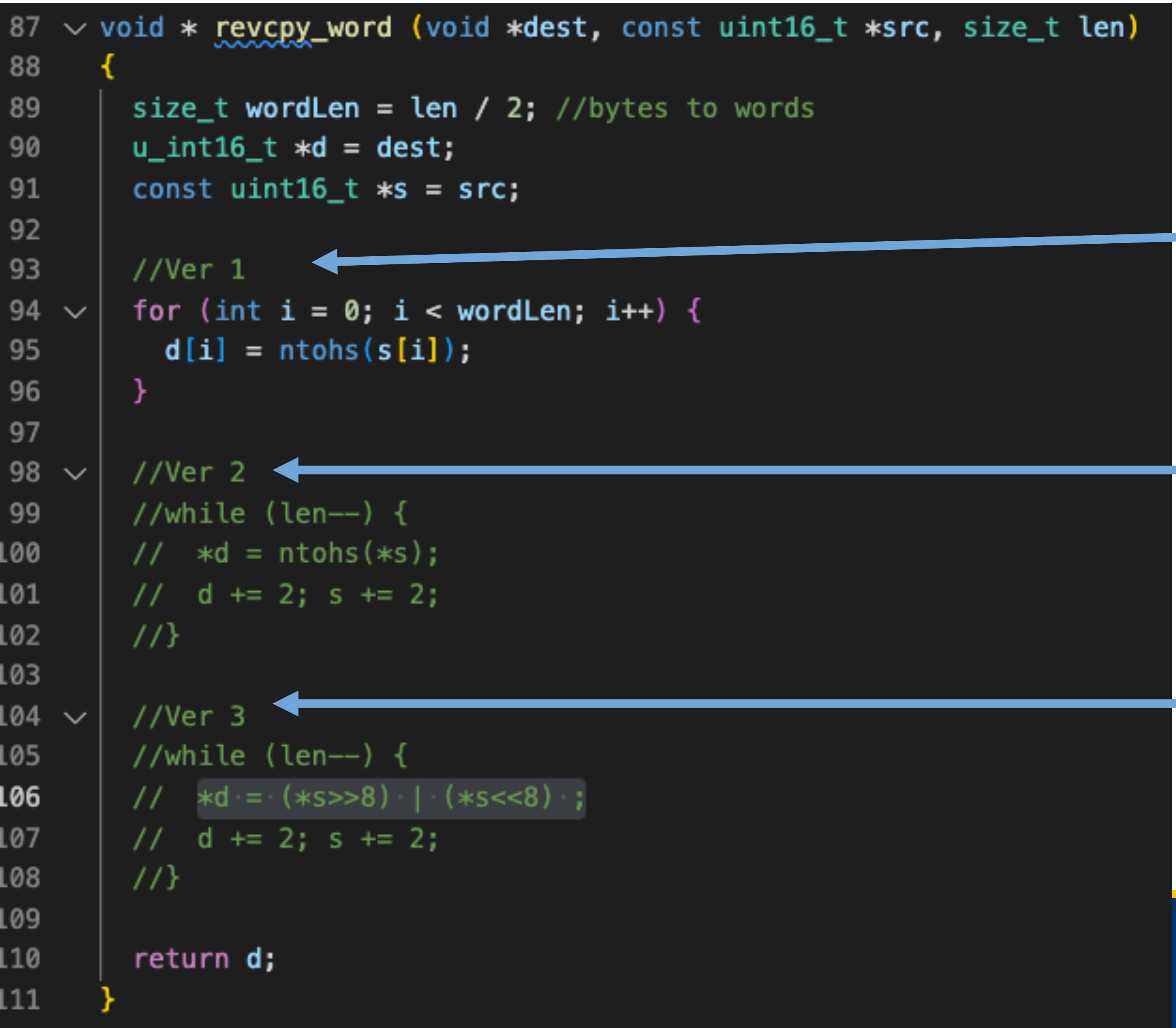
Homework #1 Solutions – Working with the 16 bit word array



```
static void wordsToArp(arp_ether_ipv4 *arp, uint16_t *buff){
    //This function accepts a pointer to a WORD buffer, in this case a byte buffer
    //It builds a arp_ether_ipv4 packet.  Notice a pointer to the structure
    //to copy the data into is passed as a pointer (e.g., *arp)
    static char tmpBuffer[BUFF_SIZE];
    revcpy_word(arp, buff, sizeof(arp_ether_ipv4));
    arp_toString(arp, tmpBuffer, sizeof(tmpBuffer));
}
```

I took a similar approach for the word array, but instead of memcpy, I wrote a helper routine called revcpy_word() ….

Homework #1 Solutions – Working with the 16 bit word array



```
void * revcpy_word (void *dest, const uint16_t *src, size_t len)
{
  size_t wordLen = len / 2; //bytes to words
  u_int16_t *d = dest;
  const uint16_t *s = src;

  //Ver 1
  for (int i = 0; i < wordLen; i++) {
    d[i] = ntohs(s[i]);
  }

  //Ver 2
  //while (len--) {
  //  *d = ntohs(*s);
  //  d += 2; s += 2;
  //}

  //Ver 3
  //while (len--) {
  //  *d = (*s>>8) | (*s<<8) ;
  //  d += 2; s += 2;
  //}

  return d;
}
```

Solution 1: traverse the raw data as a 16 bit array and use the ntohs() helper to flip the bytes

Solution 2: traverse the raw data using pointer arithmetic and use the ntohs() helper to flip the bytes

Solution 3: same as solution 2 but use bit shifting vs a helper function to shift the bits