OST
Eastern Switzerland
University of Applied Sciences

Distributed Systems & Blockchain (DS1)

Protocols

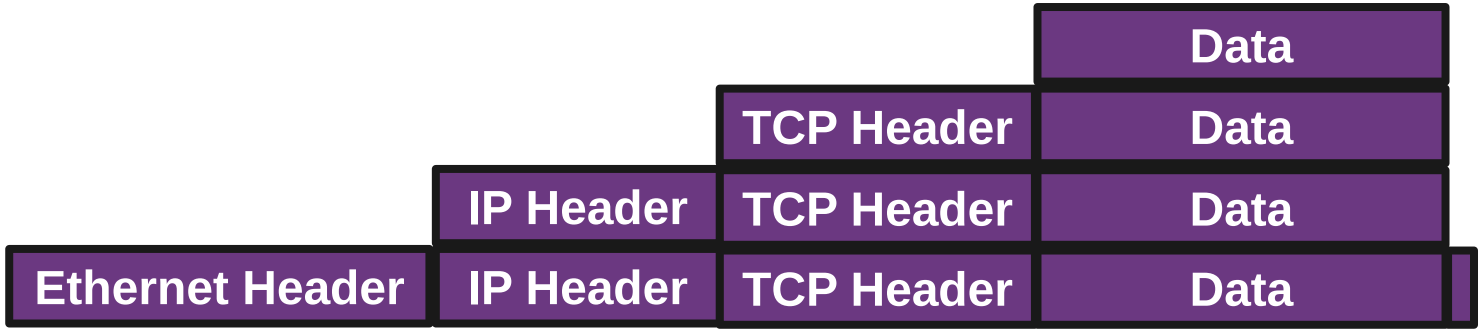
Thomas Bocek

6 March 2021

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	



Internet Model – Different Sources, Different Naming

RFC 1122, Internet STD 3 (1989)	Cisco Academy	Kurose, Forouzan	Comer Kozierok	Stallings	Tanenbaum	Arpanet	OSI model
Four layers	Four layers	Five layers	Four+one layers	Five layers	Five layers	Three layers	Seven layers
"Internet model"	"Internet model"	"Five-layer Internet model" or "TCP/IP protocol suite"	"TCP/IP 5-layer reference model"	"TCP/IP model"	"TCP/IP 5-layer reference model"	"Arpanet reference model"	OSI model
Application	Application	Application	Application	Application	Application	Application/Process	Application
							Presentation
							Session
Transport	Transport	Transport	Transport	Host-to-host or transport	Transport	Host-to-host	Transport
Internet	Internetwork	Network	Internet	Internet	Internet		Network
Link	Network interface	Data link	Data link (Network interface)	Network access	Data link	Network interface	Data link
		Physical	(Hardware)	Physical	Physical		Physical

source: https://en.wikipedia.org/wiki/Internet_protocol_suite

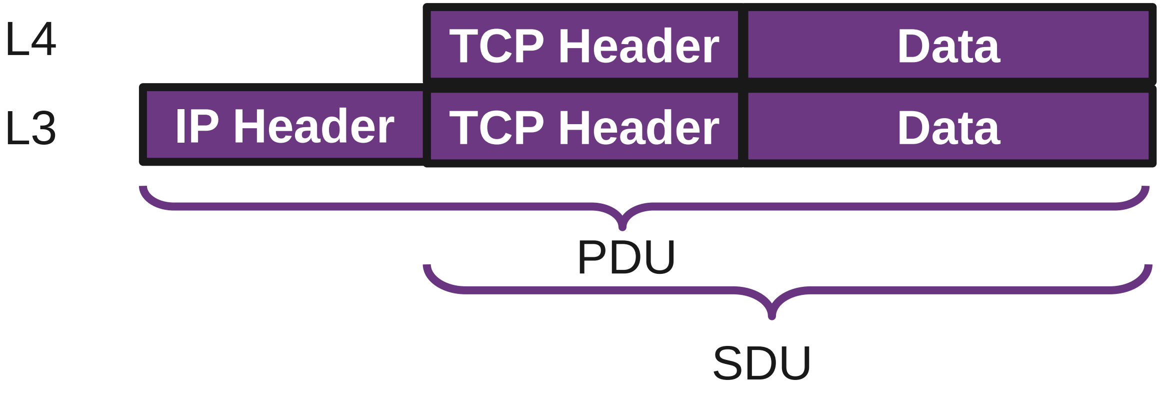
OST

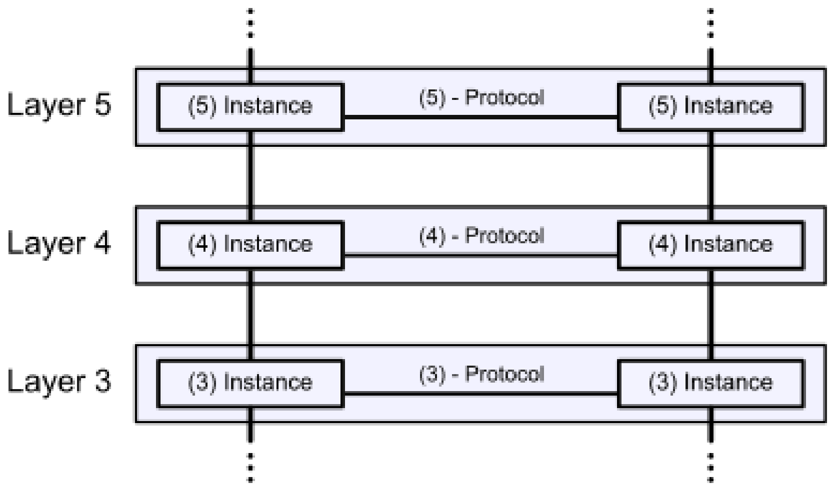
Layer Abstraction

- Protocols enable an entity/instance to interact with an entity/instance at the same layer in another host

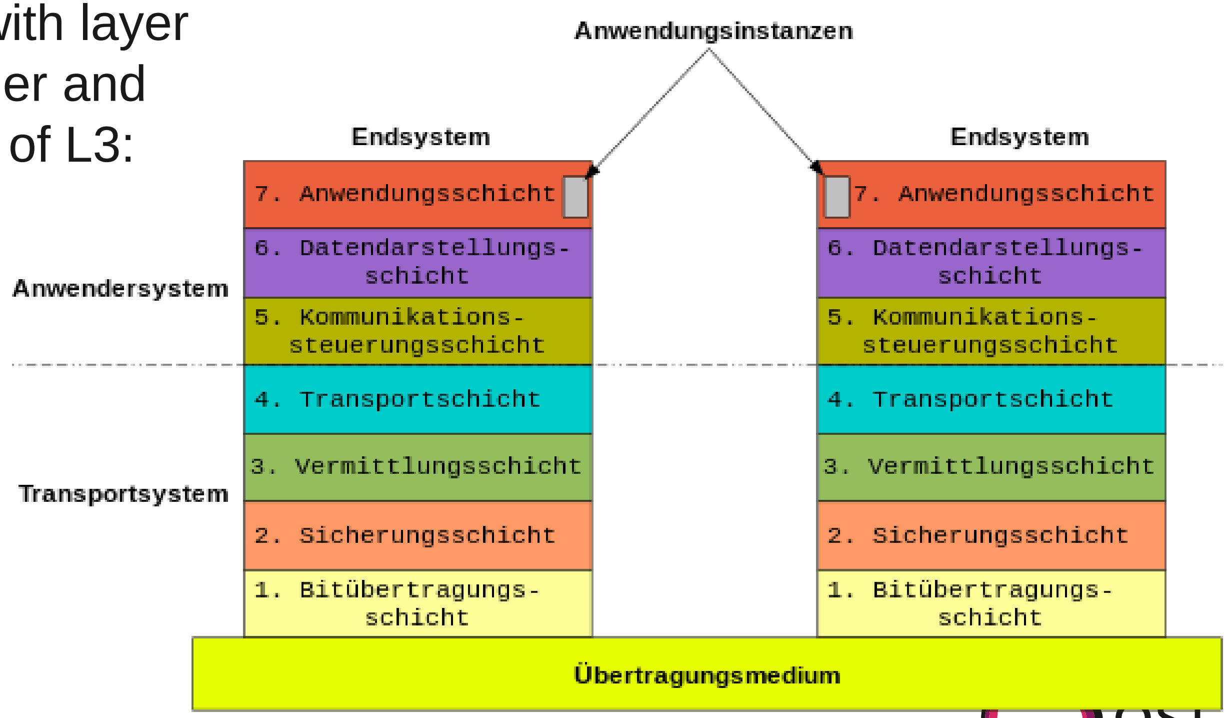
- Service definitions: provide functionality to an (N)-layer by an (N-1) layer

- Layer N exchange protocol data units (PDUs) with layer N protocol. Each PDU contains a protocol header and payload, the service data unit (SDU). E.g. PDU of L3:





source: https://en.wikipedia.org/wiki/OSI_model



source: https://de.wikipedia.org/wiki/OSI-Modell

Layer 4 - Transport

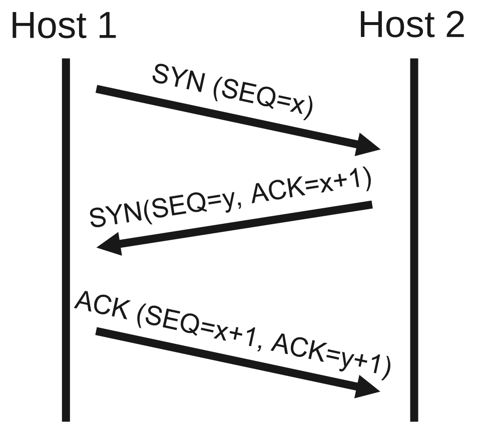
- TCP (Transmission Control Protocol)
  - Reliable (retransmission)
  - Ordered
  - Window – capacity of receiver
  - Checksum – 16bit (crc16)
  - TCP overhead: 20bytes
    - IP overhead: 20bytes
    - Ethernet frame: 18bytes (crc32)
- TCP tries to correct errors; you don’t need to worry…
  - Sometimes, you need to worry…

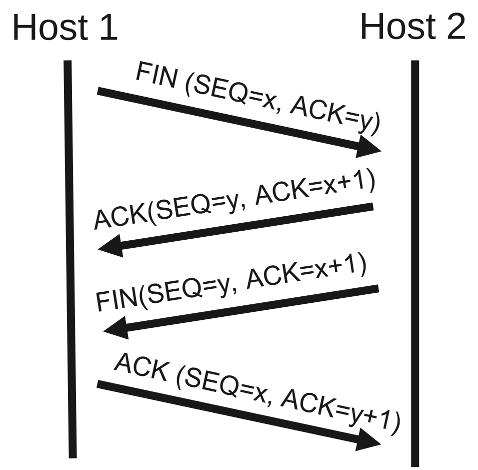
```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Data |           |U|A|P|R|S|F|                               |
| Offset| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

source: http://freesoft.org/CIE/Course/Section4/8.htm

Layer 4 - TCP

- Connection establishment
  - SYN, SYN-ACK, ACK (three way)
  - Initiates TCP session: initial sequence number is ~random
- Connection termination
  - FIN, ACK + FIN, ACK (three/four way)
  - 3-way handshake, when host 1 sends a FIN and host 2 replies with a FIN & ACK
- Sequences and ACKs
  - Identification each byte of data
  - Order of the bytes → reconstruction
  - Detecting lost data: RTO, DupACK:

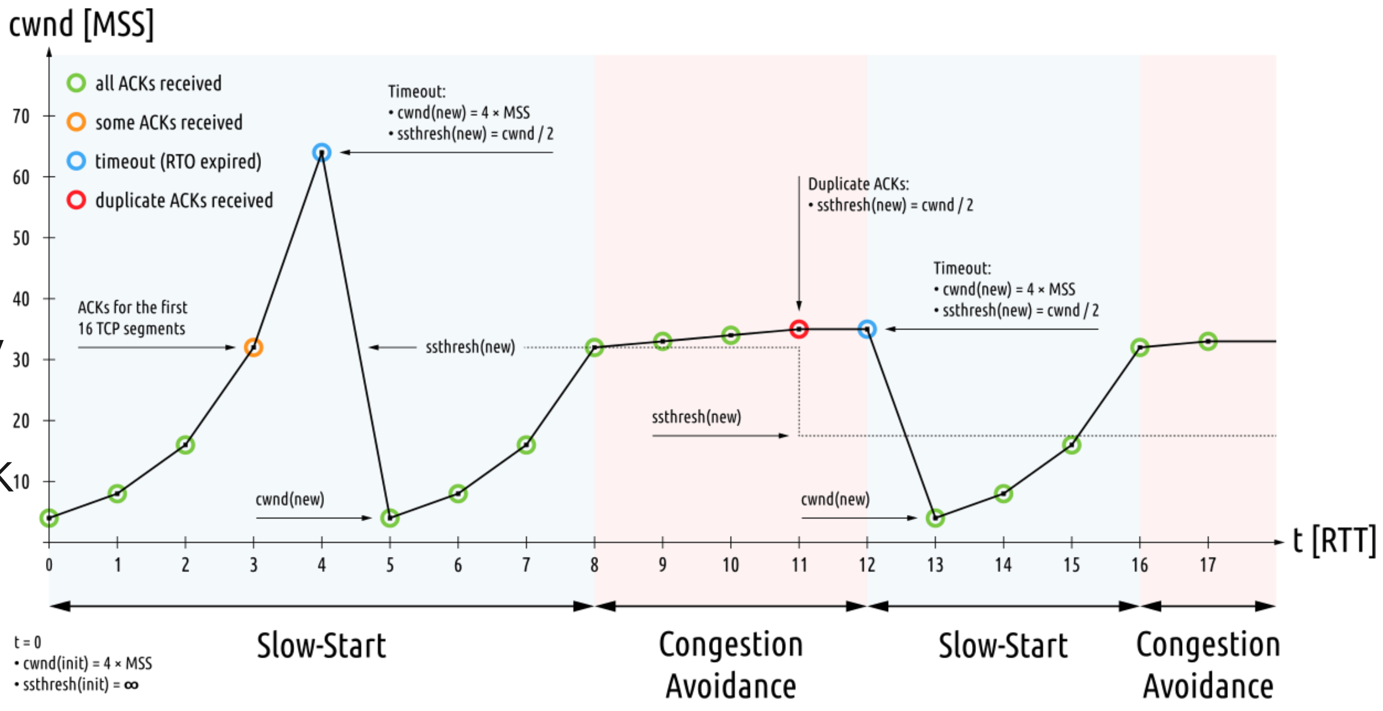




- Retransmission timeout
  - If no ACK is received aftert timout (e.g. 2xRTT), resend.
- Duplicate cumulative acknowledgements, selective ACK
  - ACKs for last consecutive packets
  - 3 times same ACK → retransmit missing packets (fast retransmit)

Layer 4 - TCP

- Flow control
  - Sender is not overwhelming a receiver
  - Back pressure
  - Sliding window:
    - Receiver specifies the amount of additionally received data in bytes that can be buffered
    - Sender up to that amount of data before ACK
- Congestion control
  - slow-start
  - congestion avoidance
- Difference flow/congestion control



source: https://upload.wikimedia.org/wikipedia/commons/thumb/2/24/TCP_Slow-Start_and_Congestion_Avoidance.svg/1280px-TCP_Slow-Start_and_Congestion_Avoidance.svg.png

TCP/IP from an Application Developer View

- Server in golang (repo)
  - git clone https://github.com/tbocek/FS21.git
  - Download GoLand, or others
  - go build server.go → server
  - GOOS=linux GOARCH=amd64 go build server.go
  - Running on alpine → glibc/musl
    - Install glibc if you cross-compile on your machine (v2 and v3)
    - Or compile in Alpine…
- Listening on TCP port 8081
  - Return string in uppercase
- Node.js version: apk add nodejs (+12MB)
  - Download WebStorm, or other

```go
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
    }
}
```

```javascript
const net = require('net');
const rl = require('readline');
const server = net.createServer(onClientConnected);

server.listen(8081, 'localhost', function() {
    console.log('Launching server...');
});

function onClientConnected(sock) {
    const i = rl.createInterface(sock, sock);

    i.on('line', function(line) {
        console.log('Message Received: %s', line);
        sock.write(line.toUpperCase());
    });
};
```

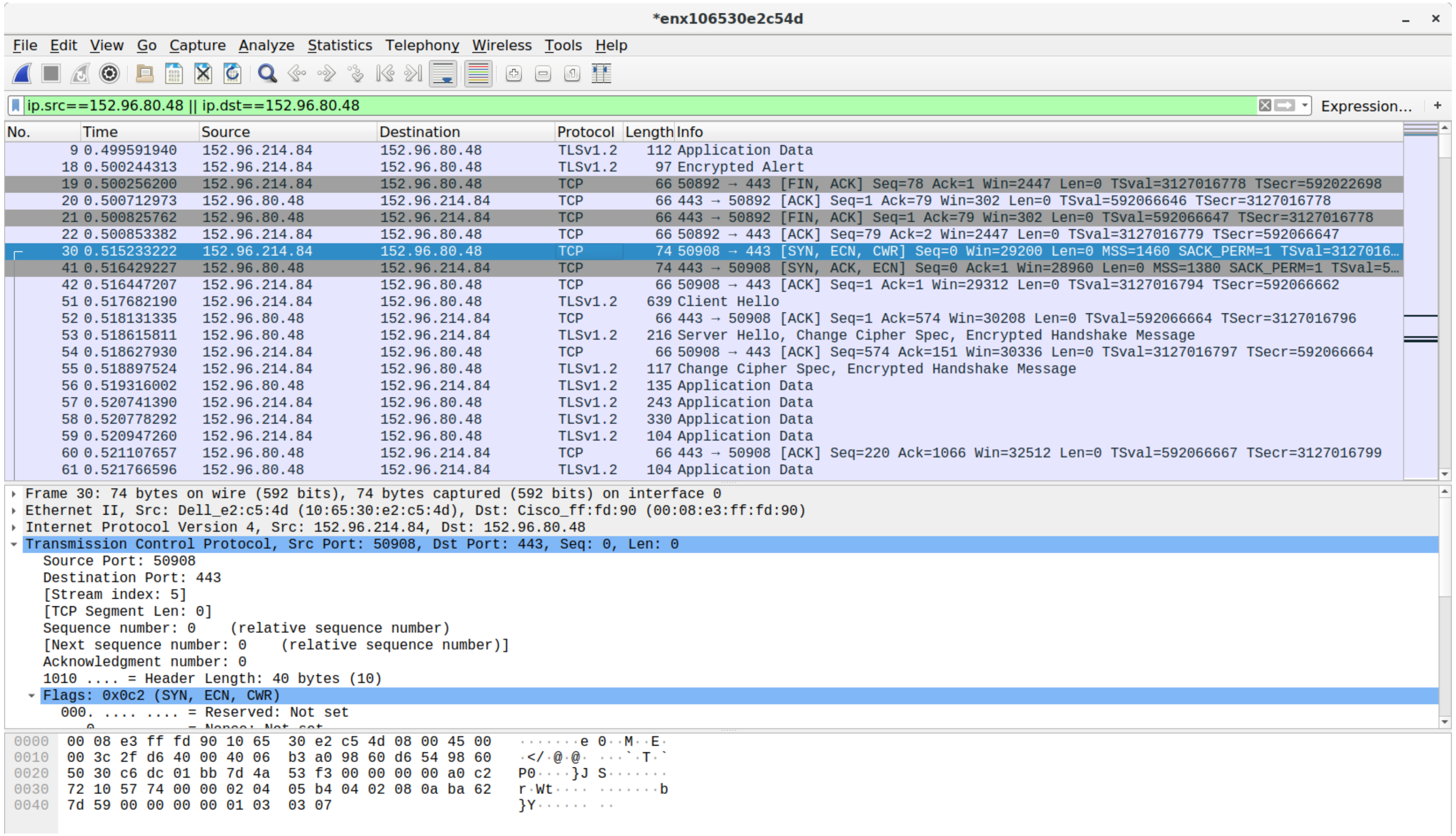
TCP/IP from an Application Developer View

- Client in Java, IntelliJ IDEA, Eclipse, Netbeans
- Golang, node.js, Java in same directory
  - Bad idea, for demo only
- TCP/UDP is supported in any major language and interoperable with each other.
- Netcat
  - Networking utility for reading from and writing to network connections using TCP or UDP
  - More complex scenarios (e.g., SCTP): socat
  - nc localhost 8081
- Challenge Task: user outside of VM
  - Port mapping, 8888 to 80
- TCP tries to correct errors; you don't need to worry… Sometimes, you need to worry…

```
import java.io.*;
import java.net.*;

class TCPClient {
    public static void main(String argv[]) throws Exception {
        Socket clientSocket = new Socket("localhost", 8081);
        DataOutputStream outToServer = new
DataOutputStream(clientSocket.getOutputStream());
        BufferedReader inFromServer = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
        outToServer.writeBytes("5Anybody there?\n");
        String modifiedSentence = inFromServer.readLine();
        System.out.println("Client received from server: " +
modifiedSentence);
        clientSocket.close();
    }
}
```

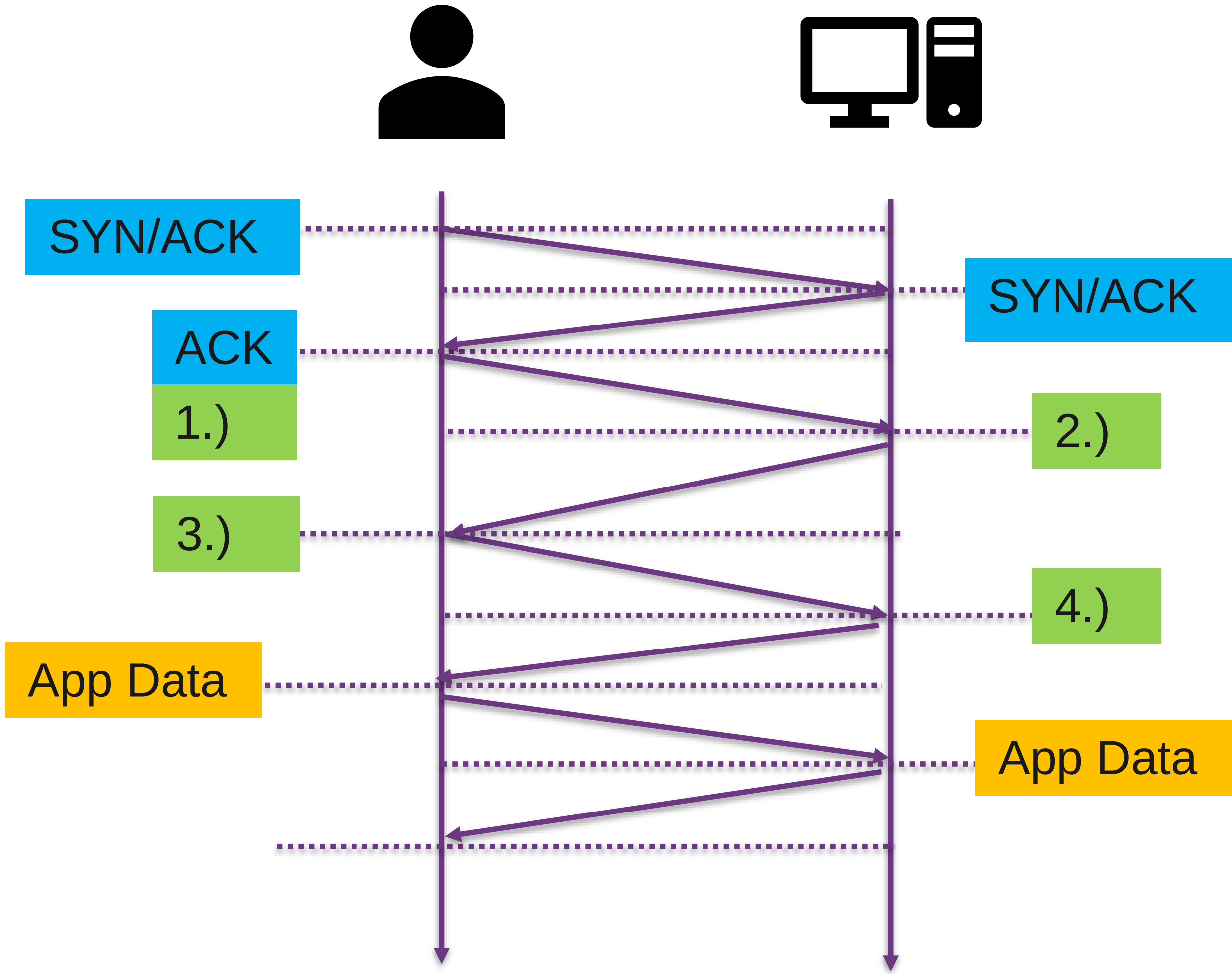

OST

Wireshark – sometimes needed when designing protocols



Layer 4 – TCP + TLS

- Security: Transport Layer Security (TLS)
    1. "client hello" lists cryptographic information, TLS version, ciphers/keys
    2. "server hello" chosen cipher, the session ID, random bytes, digital certificate (checked by client), optional: "client certificate request"
    3. Key exchange using random bytes, now server and client can calc secret key
    4. "finished" message, encrypted with the secret key
- 3 RTT until first byte

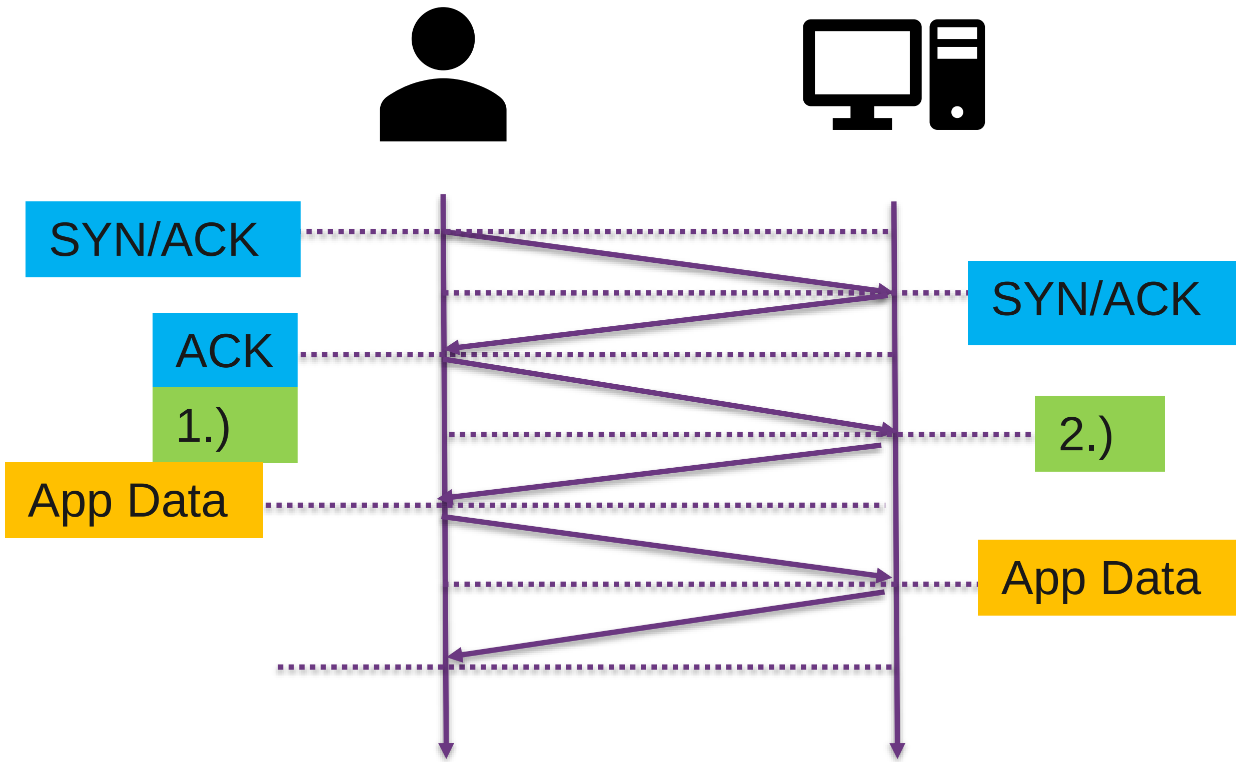


https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_8.0.0/com.ibm.mq.sec.doc/q009930_.htm

Layer 4 – TCP + TLS

```
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
```

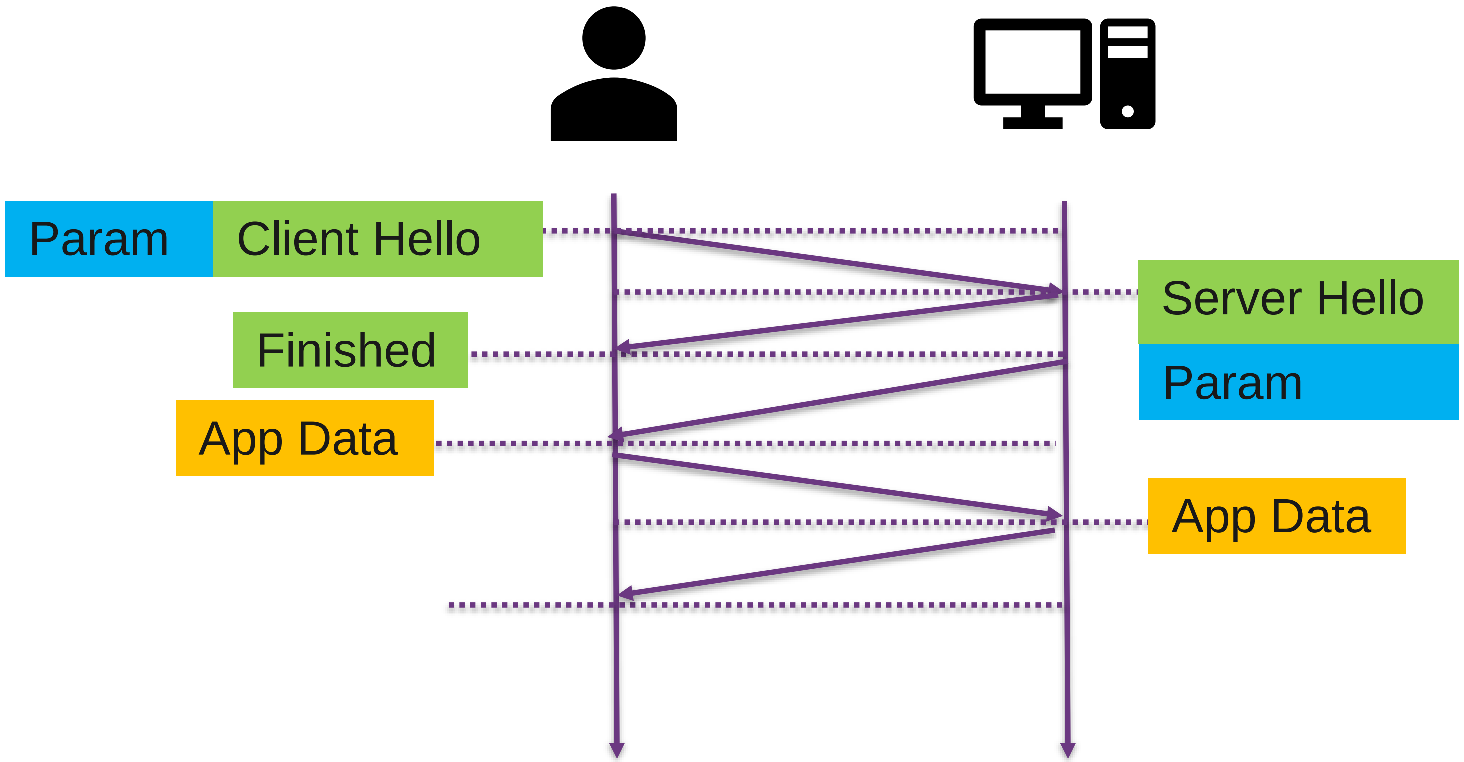
- Ping to Australia: 329ms
  - One way ~ 165ms
- TCP + TLS handshake:
  - 3RTT = 987ms! No data sent yet
- TLS 1.3, finished Aug 2018
  - 1 RTT instead of 2
    - 1.) Client Hello, Key Share
    - 2.) Server Hello, key Share, Verify Certificate, Finished
  - 0 RTT possible, for previous connections,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



OST

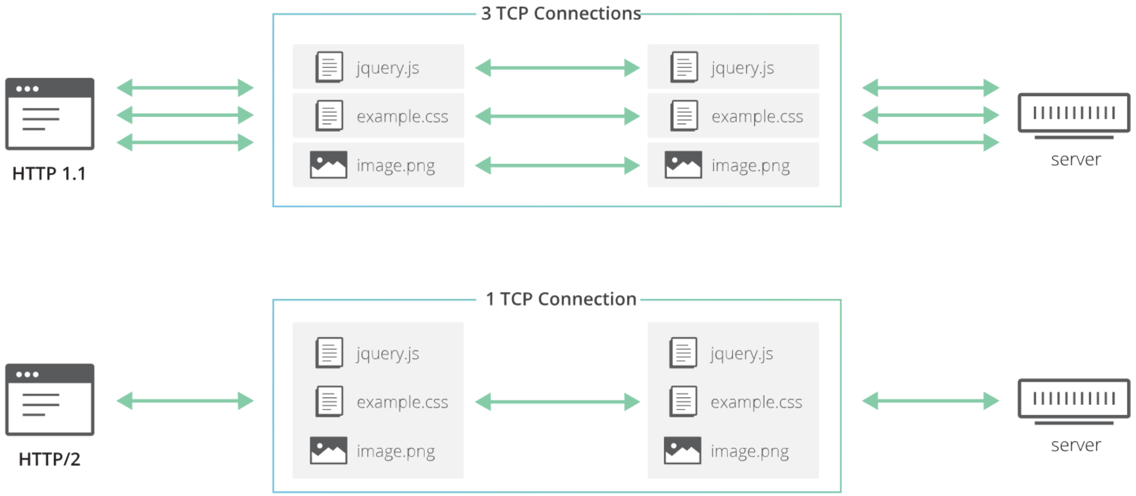
QUIC

- QUIC: 1RTT (chrome example)
  - For known connections: 0RTT
  - Built in security
  - "between 2.6 per cent and 9.1 per cent of traffic (5%, 9%)"
    - Facebook
    - Cloudflare
  - Can I use
- Example Australia: TTFB from 987ms to 329ms

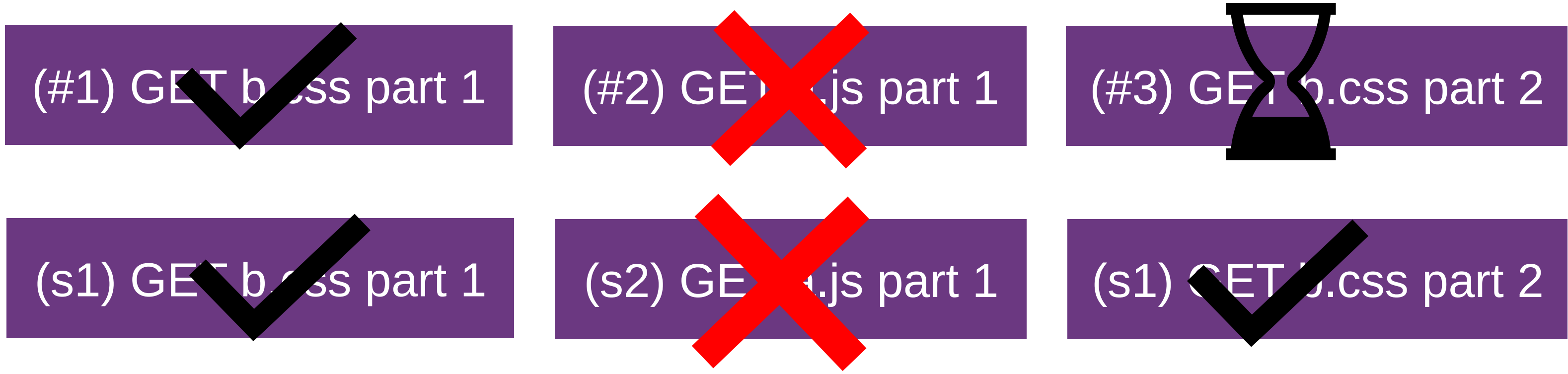


QUIC

- 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/



Layer 4 - Transport

- User Datagram Protocol (UDP)
  - UDP is used for DNS, streaming audio and video
  - Simple connectionless communication model
  - No guarantee
    - Delivery
    - Ordering
    - Duplicate protection

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port            |        Destination Port         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Length               |           Checksum              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                data                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- SCTP (Stream Control Transmission Protocol)
  - Message-based
  - Allows data to be divided into multiple streams
  - Syn cookies - SCTP uses a four-way handshake with a signed cookie.
  - Multi-homing multiple IP addresses of endpoints
  - Not widely used: "We have been deploying SCTP in several applications now, and encountered significant problem with SCTP support in various home routers."
    - E.g., OpenWRT – not enabled by default
    - E.g., UFW - Uncomplicated Firewall – not supported
  - SCTP used by WebRTC, but tunneled over UDP

UDP example

- UDP Server (Java)

```
import java.net.*;

class Server
{
    public static void main(String args[]) throws Exception
    {
        DatagramSocket serverSocket = new DatagramSocket(8081);
        byte[] receiveData = new byte[1024];
        while(true)
        {
            DatagramPacket receivePacket = new
DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);
            String s = new String( receivePacket.getData());
            System.out.println("Message Received: " + s);
        }
    }
}
```

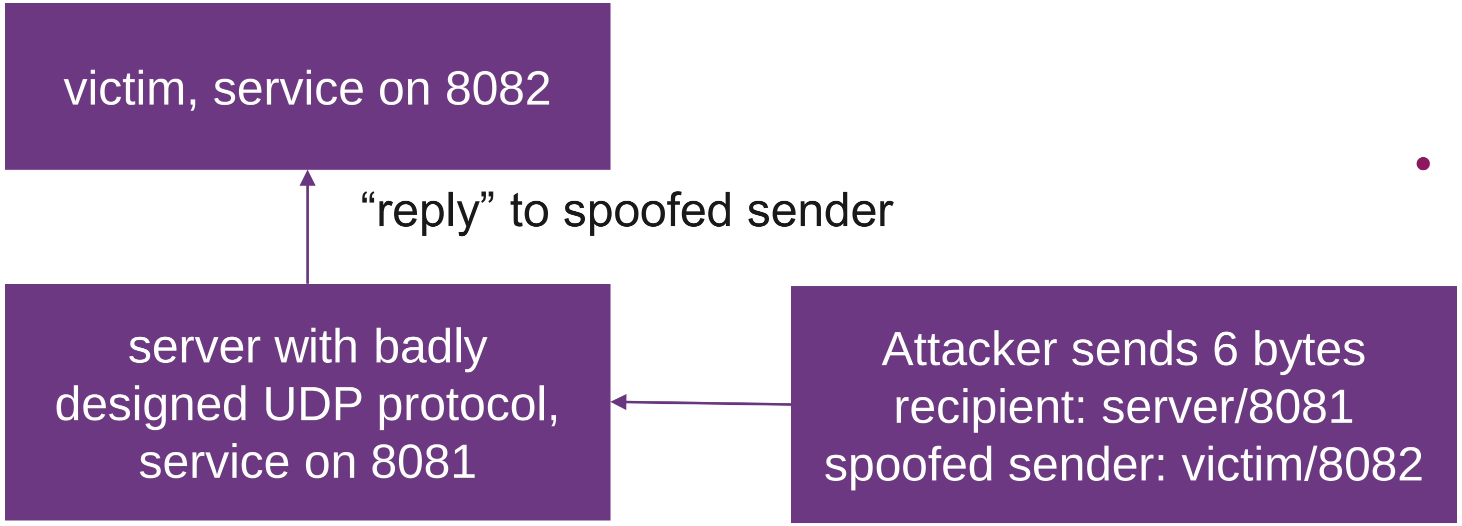
- UDP Client (golang)

```
package main

import (
        "net"
)

func main() {
        srv, _ := net.ResolveUDPAddr("udp","127.0.0.1:8081")
        local, _ := net.ResolveUDPAddr("udp", "127.0.0.1:0")
        conn, _ := net.DialUDP("udp", local, srv)
        defer conn.Close()
        conn.Write([]byte("5Anybody there?"))
}
```

- nc –u localhost 8888
  - From outside: port mapping!

Layer 4 - Transport

- DDoS Amplification Attacks
  - Request 10 bytes, reply 100 bytes → factor 10
- Local demo with server-ra/victim, and hping3
  - hping3 --udp IP -p 8081 -E test.tmp -d 6 -s 8082 -c 1



- Attacker in go/Java/node/c#
  - You need to spoof UDP packets, typically not supported in those languages
  - Go: `func DialUDP(network string, laddr, raddr *UDPAddr) (*UDPConn, error)`
    - laddr: we need to set here the victims IP/port
    - But go tries to bind to that
    - Not yours: “bind: cannot assign requested address”
- Hping3: Pen test tool
  - hping3 is a command-line oriented IP, TCP, UDP, ICMP and RAW-IP packet assembler

Comparison – Transport Layer

TCP *	UDP *	SCTP *
Transport layer	Transport layer	Transport layer
Connection oriented	Connection less	Connection oriented
Reliable transfer	Unreliable transfer	Reliable transfer
Streams	Messages	Messages
Guaranteed order	Unordered	User can choose
Widely used – HTTP/1, HTTP/2	Widely used – DNS, HTTP/3	WebRTC
Flow and congestion control	No flow, congestion	Flow and congestion control
Heavyweight	Lightweight	Heavyweight
Header size is 20 bytes	Header size is 8 bytes	Common header is 12 bytes
Error checking and recovery	Error checking, no recovery	Error checking and recovery

OST