UCLA Samueli School of Engineering

CS 35L Software Construction

Lecture 5 – Client Server & Node.js

Tobias Dürschmid
Assistant Teaching Professor
Computer Science Department

We are Starting to Take off the "Training Wheels" It's Time to Start Independently Studying Node.js

This lecture:

- Will give a **short intro** to JavaScript & Node.js
- Will teach you the **most important concepts of client-server interactions** that Node.js implements
- Include some **examples code snippets** of how this is implemented in Node.js
- Will **not replace going through a thorough tutorial of JavaScript & Node.js (This is not 31 with JavaScript / Node.js)**

We recommend a set of tutorials. Please select some of these and go through them in enough detail for you to become familiar with Node.js. **This might take some time. That's why we recommend you start with this now!**
If you struggle with Node.js, please ask on Piazza and/or LA/TA office hours

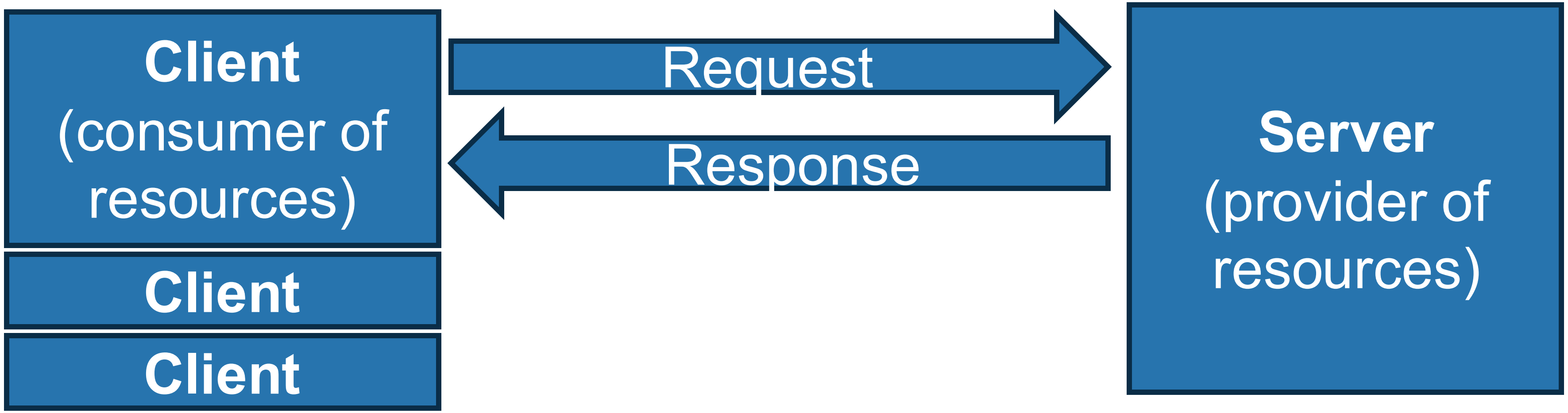
Self-Study Tips

- Don’t just watch video tutorials. Code yourself.
- Build a **small demo app** yourself by following one of the tutorials.
- Web searches & AI can help you understand unfamiliar concepts

We are happy to help you, particularly if you have not have not had much programming experience yet. If you need help, just please come to us directly

Client Server Architectures Define Two Roles

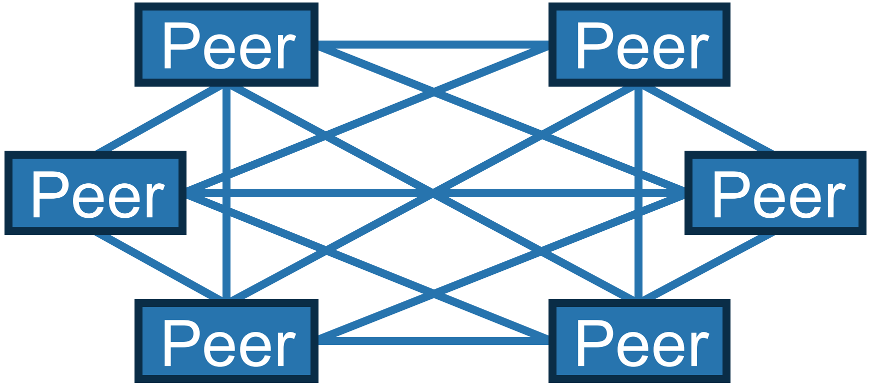
- **Multiple clients** can use the same server
- Connections are **initiated by the client**, not the server
- **Centralized architecture** (all communication flows via the server)



Client (consumer of resources)
Client
Client
Request
Response
Server (provider of resources)

Alternative: In Peer-to-Peer Architectures (P2) Clients Directly Interact with other Clients

- **Decentralized** Architecture
- Peers are **equally privileged** participants in the network
- Peers are both **suppliers** and **consumers** of resources
- In practice, there are more hybrid architectures than pure P2P architectures. Hybrid means that communication happens via client-server, and some communication happens via P2P



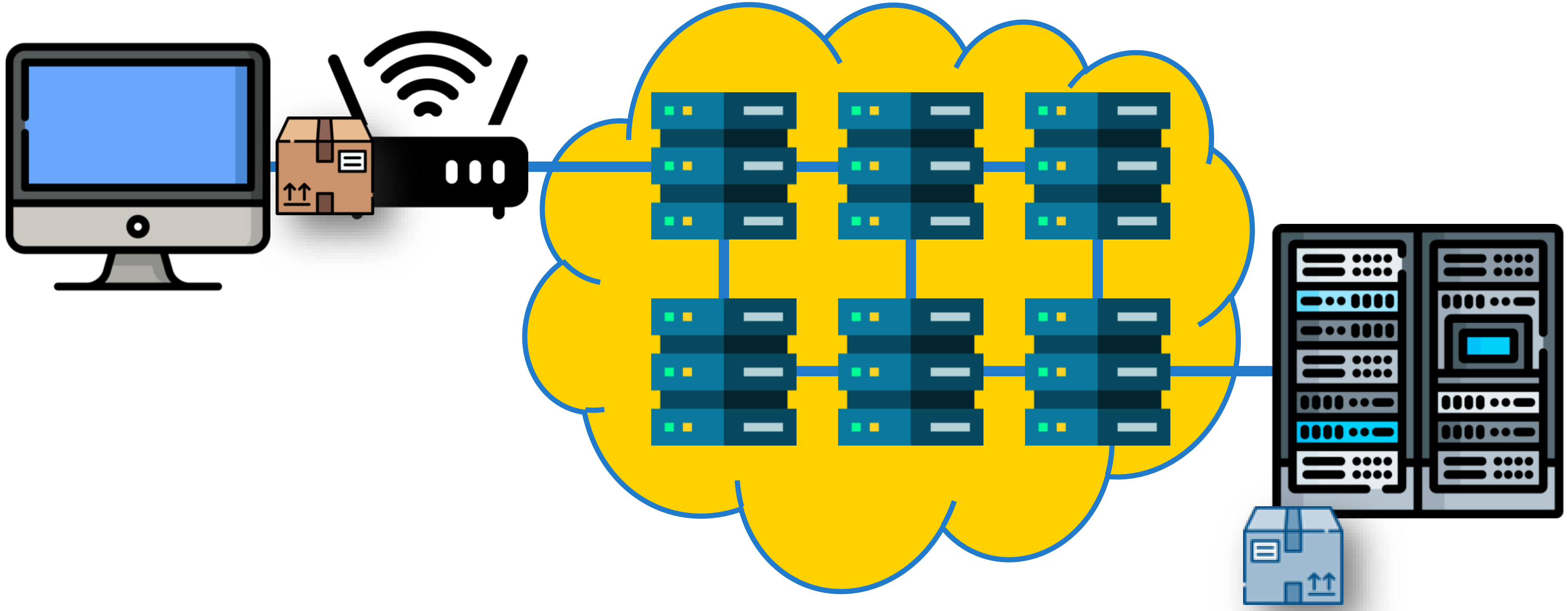
Would you implement Zoom via Client-Server or via P2P?

It's hybrid, starting with client-server. For video & audio of 1-on-1 calls, Zoom attempts peer to peer commutation and uses client server as fallback

Throughput & Latency are important Quality Attributes for Client-Server Systems

- **Throughput** measures the volume of work, data, or messages processed by a system, network, or process within a specific period of time *(e.g., the system processes 100 requests per second)*
- **Latency / Response time** measures the time it takes for a single request to receive a reply *(e.g., the average response time is 10ms)*
- Latency can be improved by making the implementation more efficient
- Throughput can be improved by improving latency and/or duplicating servers so that more requests can be processed at the same time

How do Requests reach the Server?



TCP/IP Stack

Messages from higher layer protocols are wrapped in messages from lower layer protocols

Application Layer
(e.g., HTTP, HTTPS, SSH, DNS, POP,TLS/SSL)

Provides **an interface for applications to access network services** and handles actual data exchange between applications

Transport Layer
(e.g., TCP, UDP)

Provides **end-to-end communication between applications** running on different hosts

Internet Layer
(e.g., IPv4, IPv6)

Enables communication between networks through **addressing** and **routing data packets**

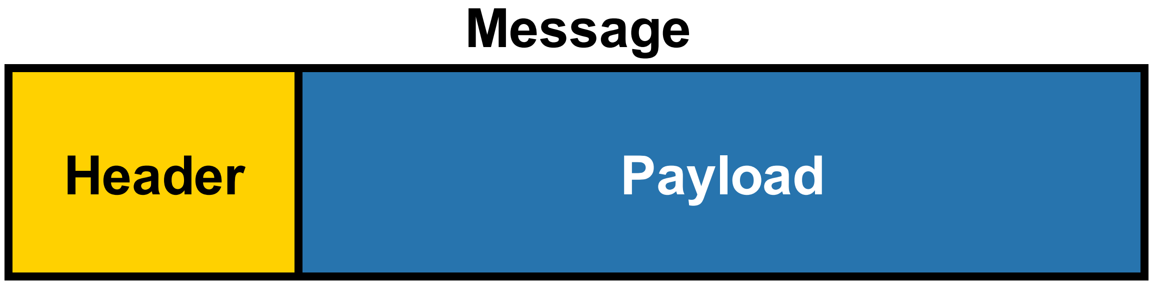
Link Layer
(e.g., Ethernet, Wifi, MAC)

Handles the **physical transmission** of data over underlying local network hardware

Each Message has a Header and a Payload

- Headers contain meta information

  (e.g., destination, origin, content type, index number, checksum, …)
- The **payload** portion contains the content of the message
- Different protocols define different headers



Package Wrapping

- **Higher-layer protocols use the protocols directly below them** to send messages.
- Whenever this happens, the higher-layer **message might get split up**. Then it the individual message get placed in the payload portion of the lower-layer messages

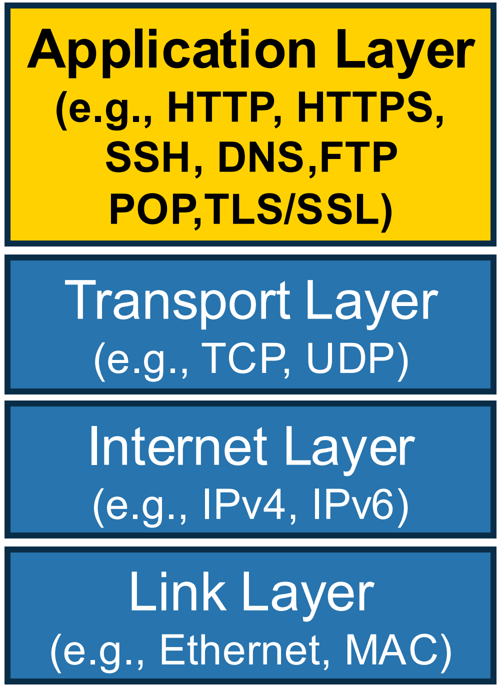
Application Layer (e.g., HTTP, HTTPS)
Transport Layer (e.g., TCP, UDP)
Internet Layer (e.g., IPv4, IPv6)
Link Layer (e.g., Ethernet, MAC)

Message

Ethernet Header	**IP Header**	**TCP Header**	**HTTP Header**	**Payload**

Your Applications use Protocols in the Application Layer

- Your application can use:
  - **HTTP/HTTPS** to **request files documents a server**
  - **FTP** to **request static files from a server**
  - **POP/SMTP** to send emails
  - …



HTTP (Hypertext Transfer Protocol)

- The foundation of **data communication on the World Wide Web**
- **Stateless** protocol
  - *Each request is independent*
  - the server doesn't remember any information about previous requests from the same client

www.ucla.edu
HTTP REQUEST
HTTP RESPONSE

HTTP Requests can use different Verbs

- **GET**
  - Requests a resource (e.g., a web page, data entry, image, file, …)
  - Response: The content of the resource & status code
- **POST**
  - Sends data to the server to create or update a resource (e.g., a direct message, file upload, complex request objects, …)
  - Response: status code
- **PUT:** Updates an existing resource on the server
- **DELETE**: Deletes a resource on the server
- **HEAD**: Retrieves only headers of a resource, not body

A URL (Uniform Resource Locator) is the web address of a resource on the internet

http://localhost:8080/users/1

{protocol}://{domain}(:{port})?(/{resource})?

https://myapp.com/about.html

Common HTTP Status Codes

- **2xx Successful responses** (the request was successfully received, understood, and accepted)
  - **200 OK: The request was successful.**
  - 201 Created: request fulfilled, and new resource created
- **4xx Client error responses** (client's request contains an error)
  - **400 Bad Request: invalid request (e.g., malformed syntax)**
  - **404 Not Found: The server cannot find the requested resource.**
  - Others: 401 Unauthorized, 403 Forbidden, …

Read more here: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status

Common HTTP Status Codes

- **5xx Server error responses** (the server failed to fulfill a valid request)
  - **500 Internal Server Error: A generic error message indicating an unexpected condition on the server.**
  - Others: 502 Bad Gateway, 503 Service Unavailable, …

Read more here: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status

Common HTTP Header Fields

- **Content-Type** (the media type of the resource)
  - "`text/html; charset=utf-8`" for HTML files in UTF-8 encoding
  - "`text/plain`" for plain text
  - "`application/json`" for json files (used for API requests / responses)

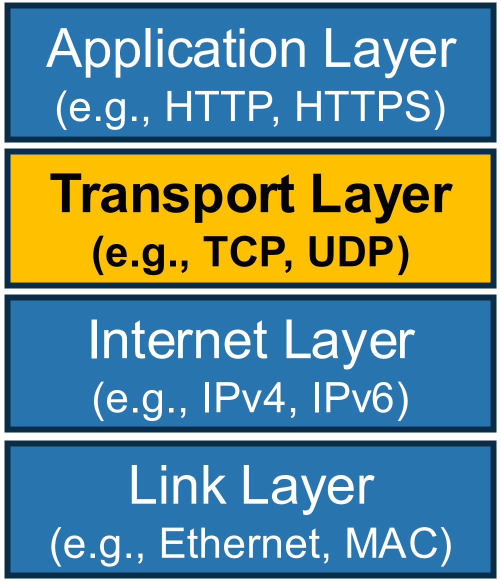
Read more here: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Content-Type

HTTPS (Hypertext Transfer Protocol Secure)

- Uses Secure Sockets Layer (**SSL**) / Transport Layer Security (**TLS**) to **encrypt communication**
- **Very important whenever sensitive data** is transferred (e.g., passwords, personally-identifiable information, private data, private messages, …)
- Has become the **default for all public web pages,** even for non-sensitive data

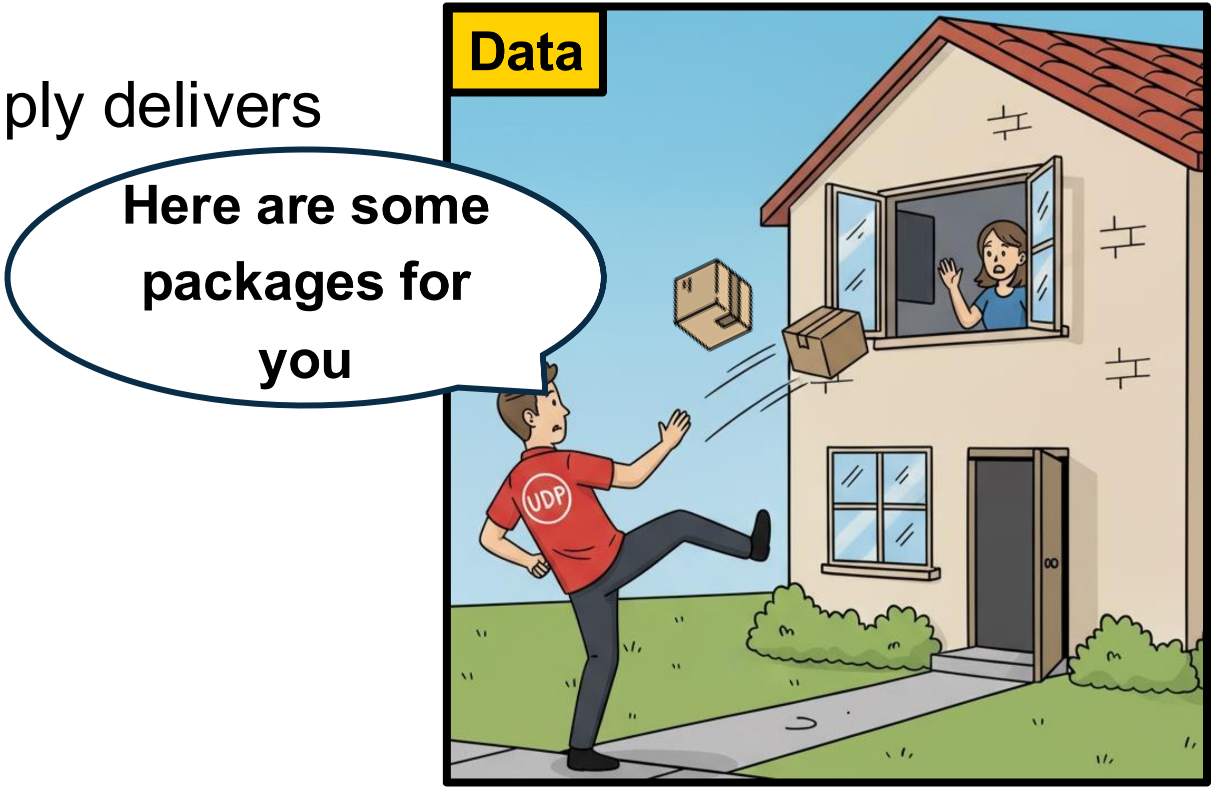
Application Layer Protocols use Transport Layer Protocols to Transmit Messages

- **Higher-layer protocols use the protocols directly below them** to send messages.
- Whenever this happens, the higher-layer **message might get split up**. Then it the individual message get placed in the payload portion of the lower-layer messages



UDP (User Datagram Protocol) Just “throws” Messages at the Receiver

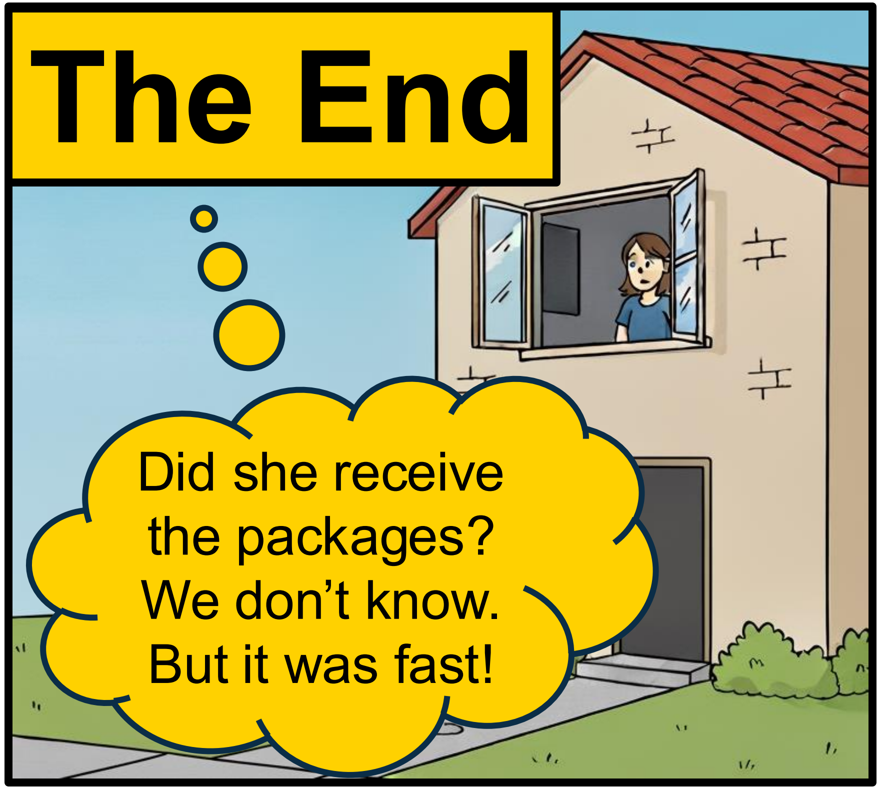
In UDP, the Bob simply delivers packages to Alice’s address without waiting for a response.



UDP Is a Simple, Unreliable, but Fast & High Throughput Protocol

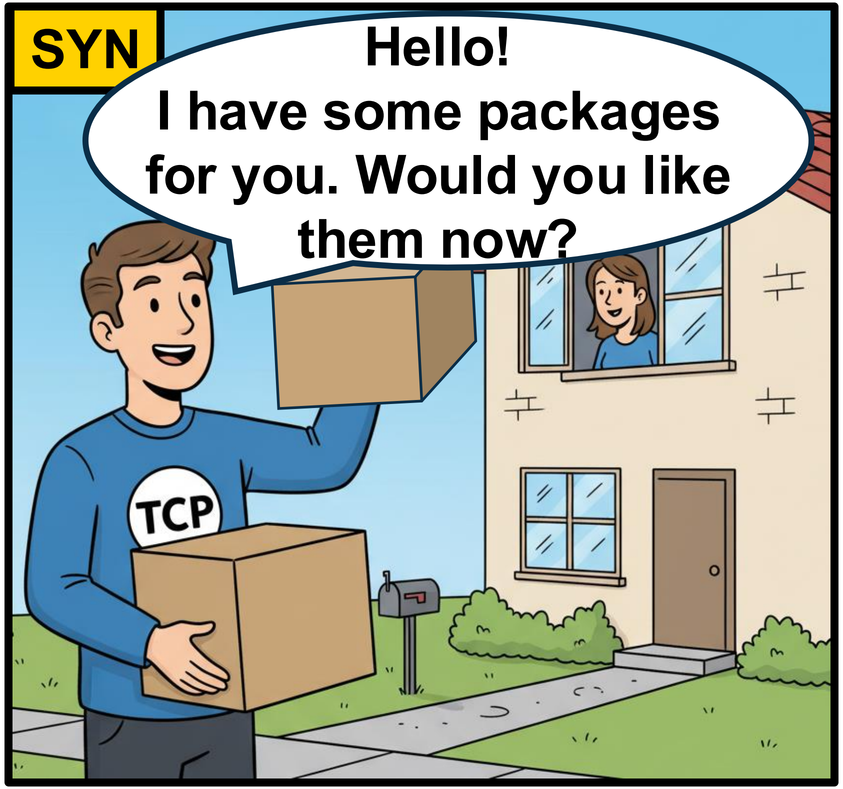
UDP provides **fast**, **connectionless**, and **lightweight** communication.

It **does not guarantee delivery**, **order**, or **error checking**.


The End
Did she receive the packages? We don't know. But it was fast!

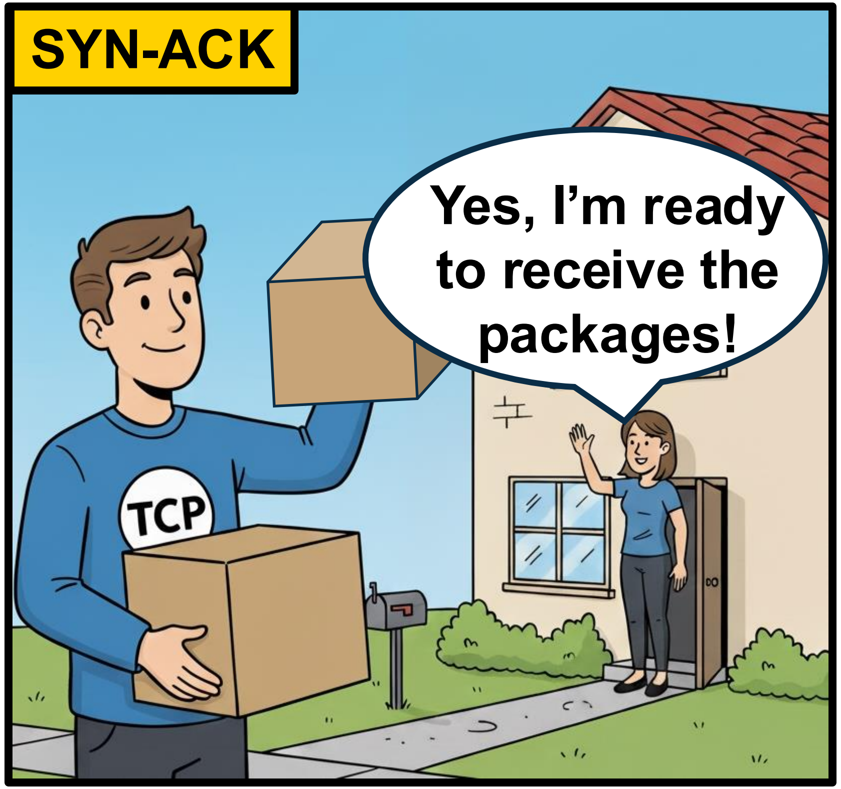
TCP (Transmission Control Protocol) is a more Complicated, but Reliable Protocol

- To **initiate the connection**, Bob sends a synchronize message (SYN).


SYN
Hello! I have some packages for you. Would you like them now?
TCP

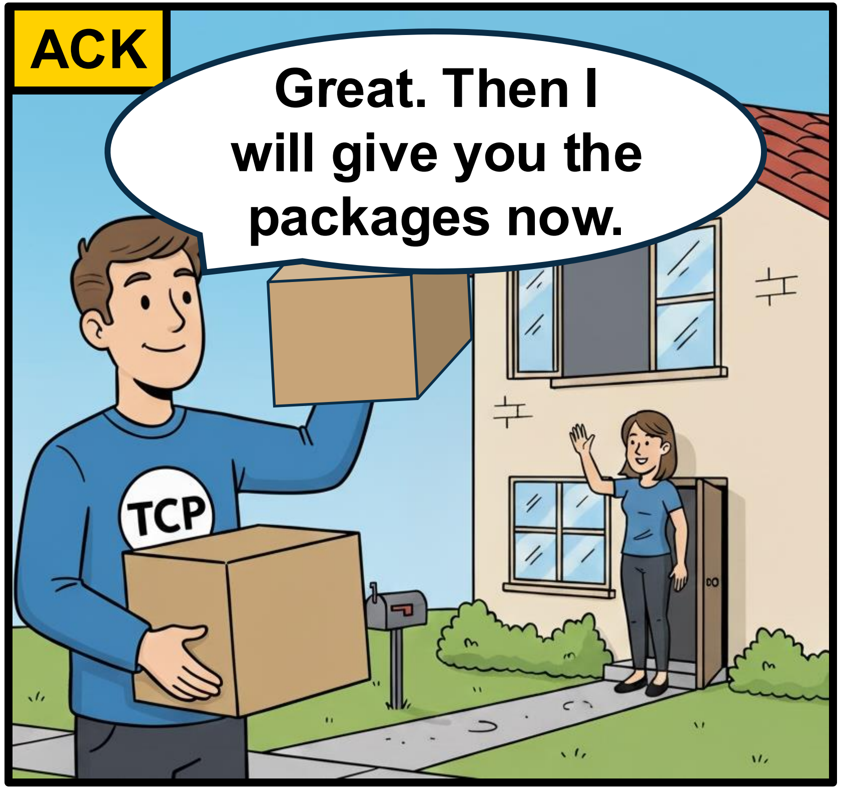
TCP (Transmission Control Protocol) is a more Complicated, but Reliable Protocol

- Alice **confirms the connection** by sending a SYN-ACK message to Bob.



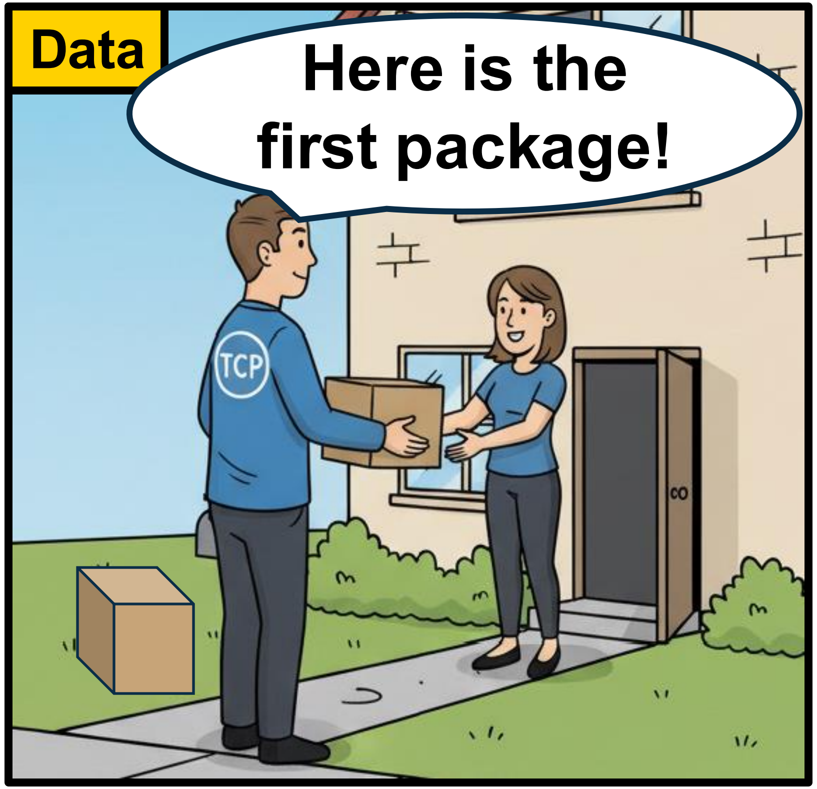
TCP (Transmission Control Protocol) is a more Complicated, but Reliable Protocol

- Bob **confirms that the connection has been established** by sending an ACK (acknowledgment) message to Alice.



TCP (Transmission Control Protocol) is a more Complicated, but Reliable Protocol

- Bob sends data to Alice in multiple **ordered** messages that contain a **checksum** to allow Alice to detect corrupted data and request retransmission if needed.



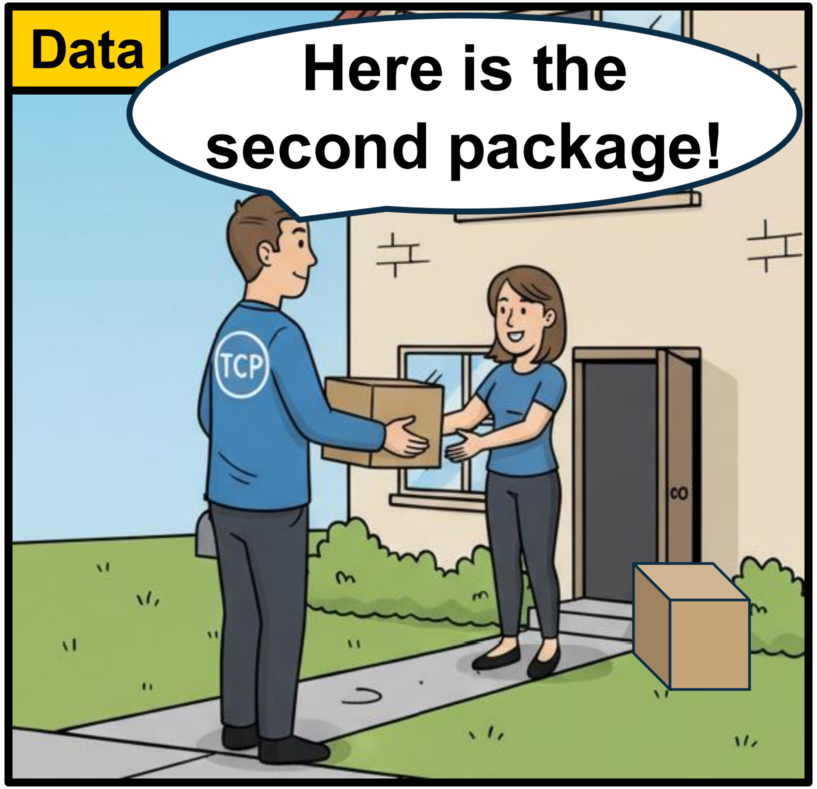
TCP (Transmission Control Protocol) is a more Complicated, but Reliable Protocol

- If Alice has received the message. Since it does not contain errors, she sends an **acknowledgement** (ACK) to Bob.
- If Bob does not receive the ACK until a given **timeout**, he **attempts to** send the message **again**.

ACK
I received the first package!
TCP

TCP (Transmission Control Protocol) is a more Complicated, but Reliable Protocol

- Bob sends data to Alice in multiple **ordered** messages that contain a **checksum** to allow Alice to detect corrupted data and request retransmission if needed.



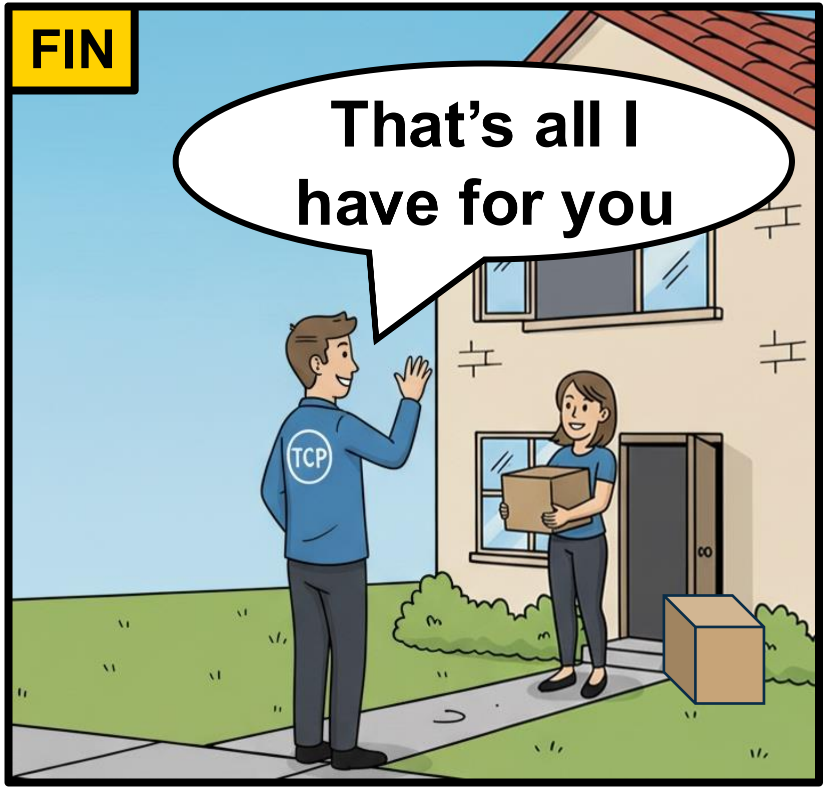
TCP (Transmission Control Protocol) is a more Complicated, but Reliable Protocol

- If Alice has received the message. Since it does not contain errors, she sends an **acknowledgement** (ACK) to Bob.
- If Bob does not receive the ACK until a given **timeout**, he **attempts to** send the message **again**.

ACK
I received the second package!
TCP

TCP (Transmission Control Protocol) is a more Complicated, but Reliable Protocol

- To notify Alice that **Bob does not have any more messages to send**, he sends a FIN message.



TCP (Transmission Control Protocol) is a more Complicated, but Reliable Protocol

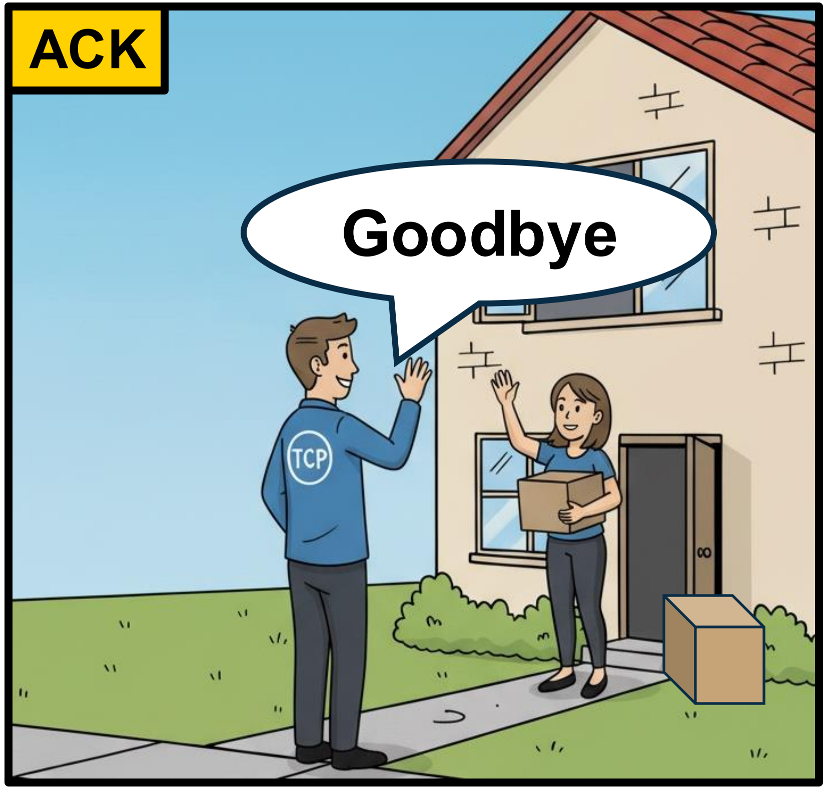
- Alice **acknowledges** Bob's message (ACK). **She also does not have any more messages to send**.

  So, she sends a FIN message too.

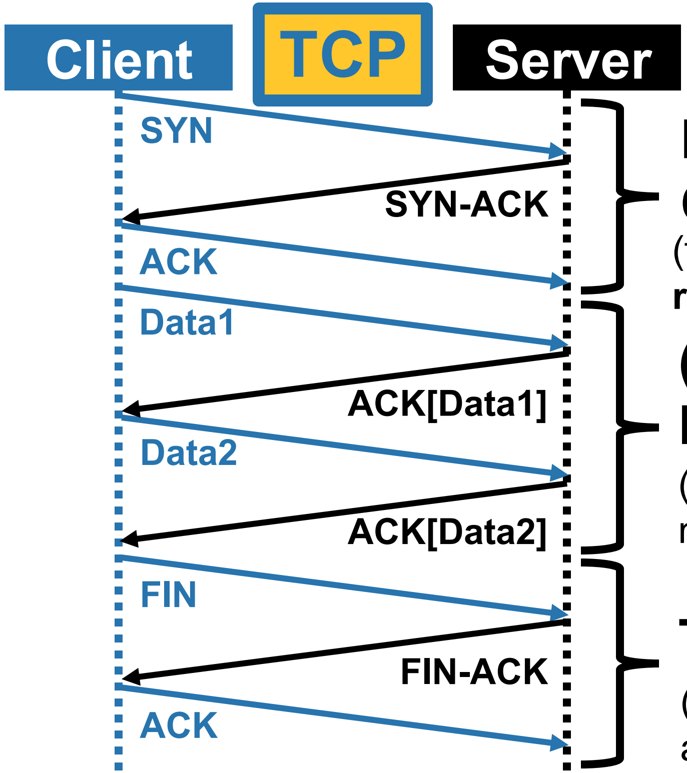
FIN ACK
Okay. Goodbye
TCP

TCP (Transmission Control Protocol) is a more Complicated, but Reliable Protocol

- Bob **acknowledges** Alice's FIN message with an ACK message.



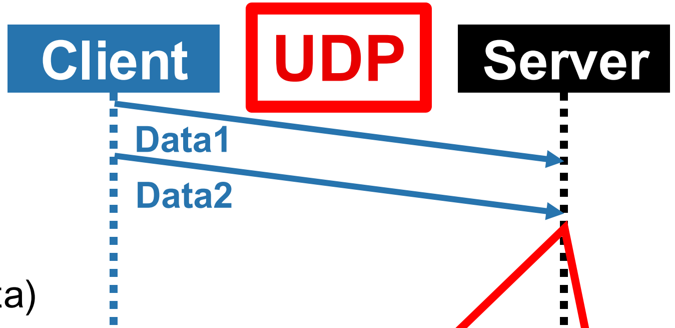
To achieve reliability, TCP sends at least 6+2N messages for N messages of data



Establishes the Connection
(to ensure both parties are **ready** to send / receive data)

(Full Duplex) Data Transfer
(both can send messages at any time)

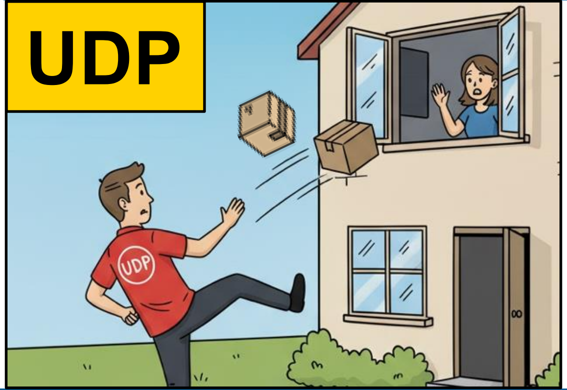
Terminates the Connection
(to **free up resources** on both the client and server systems and to **ensure reliable closure** of the communication session)



UPD completes the transfer before TCP even sends the first data package

TCP and UDP Offer Different Trade-Offs

- ✓ Message **Order** is preserved
- ✓ **Error-Detection** included
- ✓ **Lost messages** get re-sent
- ✗ Protocol adds additional overhead



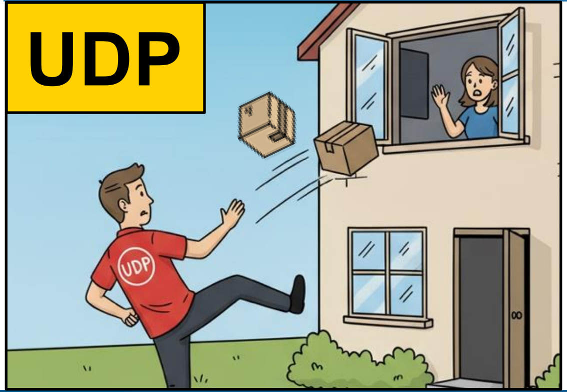
- ✗ Messages can arrive in **Any Order**
- ✗ No **Error-Detection** included
- ✗ Some messages **Might Get Lost**
- ✓ **No Additional Delay**

TCP and UDP Work Well For Different Use Cases

Text Messaging (e.g., Slack, Discord)

Web Browsing

File Transfer



Live Video Streaming (e.g., Webcasts)

Real-Time Voice Chat (e.g., Audio Call)

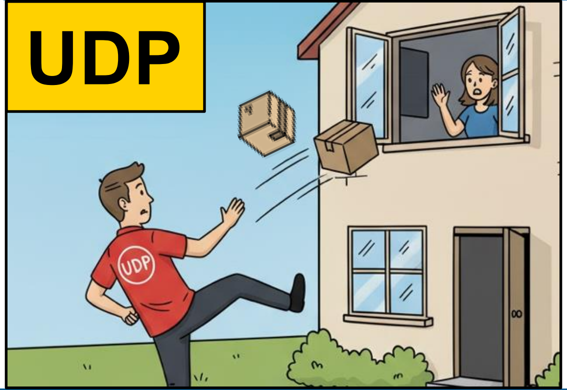
What about online gaming? Talk to your Neighbor(s)!

TCP and UDP Work Well For Different Use Cases

- Ideal for round-based games (e.g., Chess, Go)

Many games use a hybrid approach:

- **Reliable**, **non-time-sensitive data** where loss is fatal (e.g., chat messages, logging into the server, inventory transactions, scoring)



- **High-speed real-time events** (e.g., player positions, physics)
- Game updates should not require previous messages, so they need to include the absolute positions of all movable objects

IP Addresses

Challenge: Identification

How to identify a particular sender/receiver out of the billions of computers connected to the internet?

Solution Idea: IP Addresses

Have a **portion** of the address that represents the ***network*** (like a "city") and a portion that represents the ***host*** (like a "street address")

IPv4 addresses range from `0.0.0.0` to `255.255.255.255`

There are some special IP addresses, e.g., `127.0.0.1` is "local host" (your current machine)

After running out of all IP addresses, a new standard (IPv6) was created

What is Funny about this Meme?

It implies that the developers used AI to create a webpage and **didn't know that 127.0.0.1 is localhost**, so only accessible from your current machine

Introducing VibeCon – the world's largest vibe coding conference.

Make sure you register today: https://127.0.0.1:8080/register

TCP/IP Stack

Messages from higher layer protocols are wrapped in messages from lower layer protocols

Application Layer
(e.g., HTTP, HTTPS, SSH, DNS, POP,TLS/SSL)

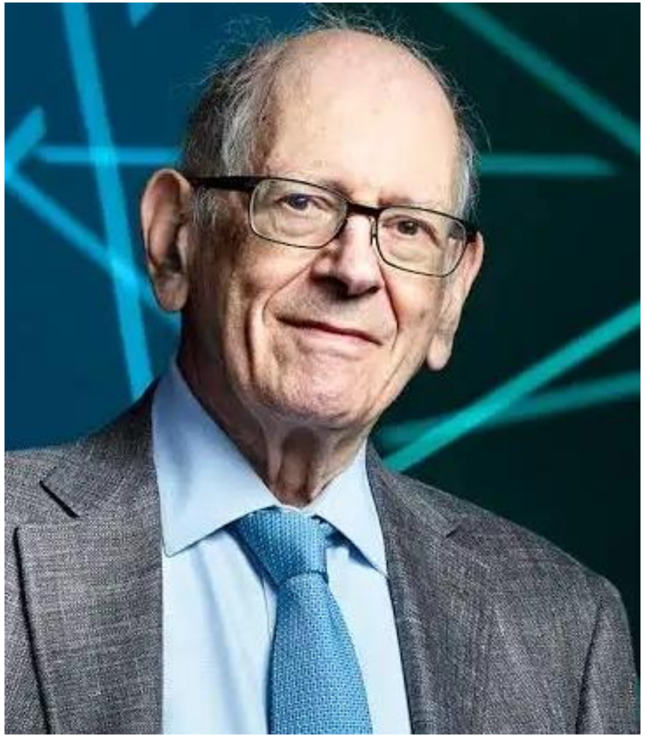
Transport Layer
(e.g., TCP, UDP)

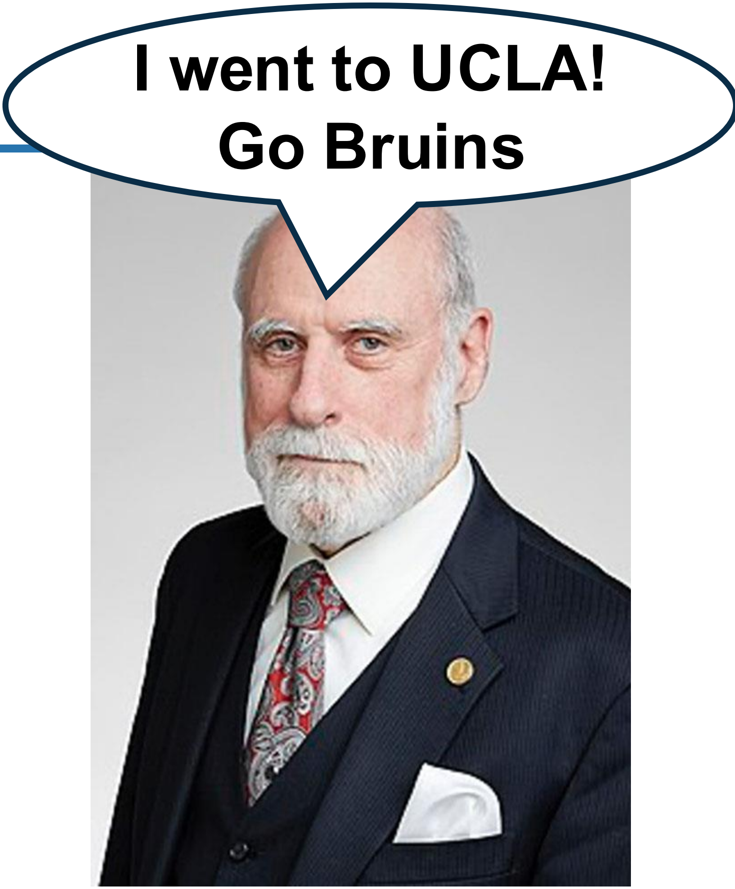
Internet Layer
(e.g., IPv4, IPv6)

Link Layer
(e.g., Ethernet, Wifi, MAC)

- TCP/IP is an instance of a **layered architecture** in which higher-level layers use only lower-level layers
- This increases **reusability** of lower-layer implementations
- It also allows **flexibility** as you can simply replace one layer with a different implementation to get different results

TCP-IP was invented by Robert Kahn & Vinton Cerf





Read more about TCP here: https://www.geeksforgeeks.org/computer-networks/what-is-transmission-control-protocol-tcp/

HTML (Hyper Text Markup Language)

HTML is the most common **markup language used to create web pages.**

It supports structured content. HTML is **not** a programming language

```
<!DOCTYPE html>
<html>
<head>
<title>Page Title</title>
</head>
<body>

<h1>This is a Heading</h1>
<p>This is a paragraph.</p>

</body>
</html>
```

Head contains metadata read by the browser
Body contains the content to be rendered

h1 indicates the top heading (others: h2 – h6)

Java Script (JS)

- Interpreted, dynamically typed programming language (similar to Python)
- Can be hooked into HTML

JavaScript

```
// A function that takes two arguments
function add(num1, num2) {
  let sum = num1 + num2;
  return sum; // Returns the result
}

let result = add(5, 7); // result is now 12
console.log(result); // Output: 12
```

JSON (JavaScript Object Notation)

human-readable, lightweight data-interchange format used to transfer data between a server and a web page, or between applications.

```
{
  "address-dict": {
    "city": "New York",
    "residential": true,
    "postal_code": 10023
  },
  "phone_numbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [
    "Catherine",
    "Thomas",
    "Trevor"
  ]
}
```

Node.js

Recommended Tutorial:
https://www.youtube.com/watch?v=32M1al-Y6Ag

- **JavaScript runtime** for asynchronous events
- Allows you to run JavaScript outside of the browser
- Is **not** a programming language

  (you are writing code in Java Script to be run within the Node.js environment)
- Fundamentally **single-threaded** for its main execution of JavaScript code
- Works via **callbacks** and **event handlers**

Blocking vs. Non-Blocking Code

`require` imports the **`fs`** (file system) module from node standard library (similar to **`import fs as fs`**)

```
const fs = require('node:fs');
```

`const` declares a constant (cannot be changed after initialization)

```
const data = fs.readFileSync('/file.md');
// blocks here until file is read
```

Read more here: https://nodejs.org/en/learn/asynchronous-work/overview-of-blocking-vs-non-blocking

Blocking vs. Non-Blocking Code

```
const fs = require('node:fs');
```

Blocking Code

```
const data = fs.readFileSync('/file.md′);
// blocks here until file is read
```

Non-Blocking Code

```
fs.readFile('/file.md', (err, data) => {
  if (err) {
     …
  }
});
// following code is executed instantly
```

Defines an **asynchronous callback** that executes after the data has been completely stored in the `data` variable

Node.js philosophy: use **non-blocking code** for all non-instant tasks to maintain high performance and scalability

Define Callbacks Functions for Long Non-Blocking Code

```
function handleMarkdownRead(err, data) {
  if (err) {
     …
  }
}


fs.readFile('/file.md', handleMarkdownRead);
// following code is executed instantly
```

Callbacks Functions allow you to **reuse** non-blocking code, give it a **name,** and make your code **easier to read**

See: https://www.w3schools.com/nodejs/nodejs_http.asp

What does it do? Talk to your Neighbor(s)!

```
const http = require('http');
const PORT = 3000;

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, World!\n');
});

server.listen(PORT, 'localhost', () => {
  console.log(`Server running at http://localhost:${PORT}/`);
});
```

Imports http module

Creates an HTTP server object by defining an callback to be called whenever the server receives an HTTP requires (req)

Sets HTTP response header to status code 200 (success) and plain text content

Writes 'Hello, World!' into the HTTP response body and sends the message

Starts the server on localhost to listen on port 3000

Like f-strings in Python

See https://developer.mozilla.org/en-US/docs/Learn_web_development/Extensions/Server-side/Express_Nodejs/Introduction

What does it do? Talk to your Neighbor(s)!

```
const express = require('express');
const app = express();
const port = 8080;

app.get('/users/:userId', (req, res) => {
  res.send(`GET request to user ${req.params.userId}`);
});

app.post('/', (req, res) => {
  res.send('POST request to the homepage');
});

app.get('/about', (req, res) => {
  res.send('About page');
});

app.all('*', (req, res) => {
  res.status(404).send('404 - Page not found');
});
```

Imports the super helpful **`express`** module that helps you with routing URLs

Route for all GET requests to **`localhost:8080/users/.*`**

Retrieves the user id from the GET request URL

Route for all POST requests to **`localhost:8080/`**

Route for all GET requests to **`localhost:8080/about`**

Route for all other requests

See https://nodesource.com/blog/an-absolute-beginners-guide-to-using-npm

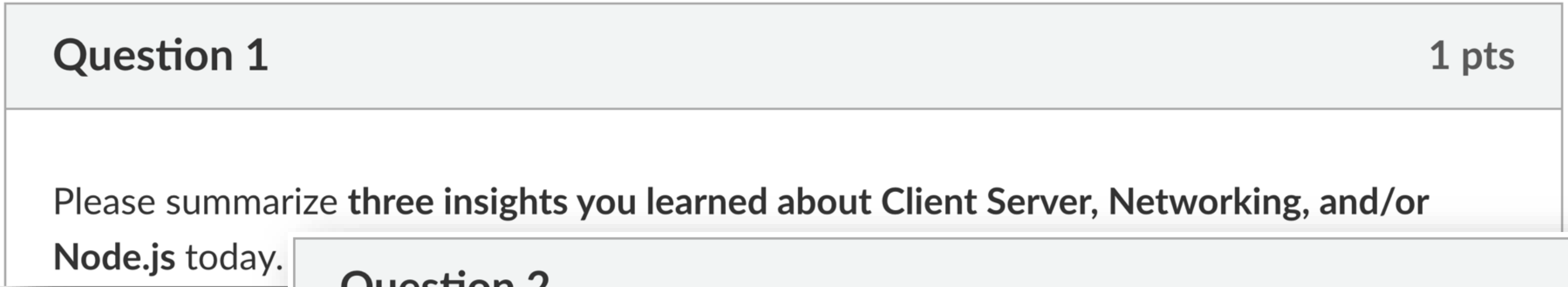
NPM – The Node Package Manager

- NPM allows you to install node packages and keep the version listed in your package.json
- Node has a large ecosystem of useful package

Common npm cpmmands

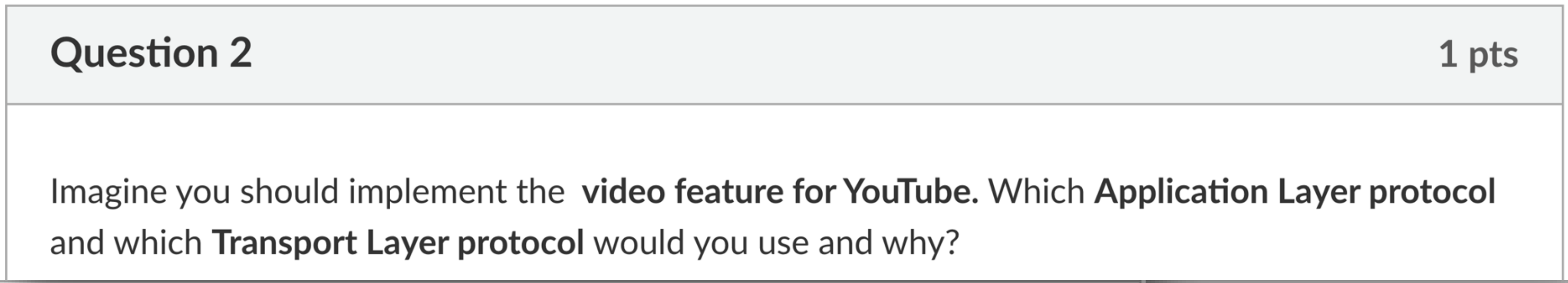
```
$ npm init (creates a new NPM package from your code)
$ npm install express (installs the express package)
$ npm install express@4 (installs version 4 of express)
$ npm install --save (creates a package.json)
$ npm install (installs all packages listed in the package.json)
```

Please fill out your Exit Tickets on Bruin Learn!



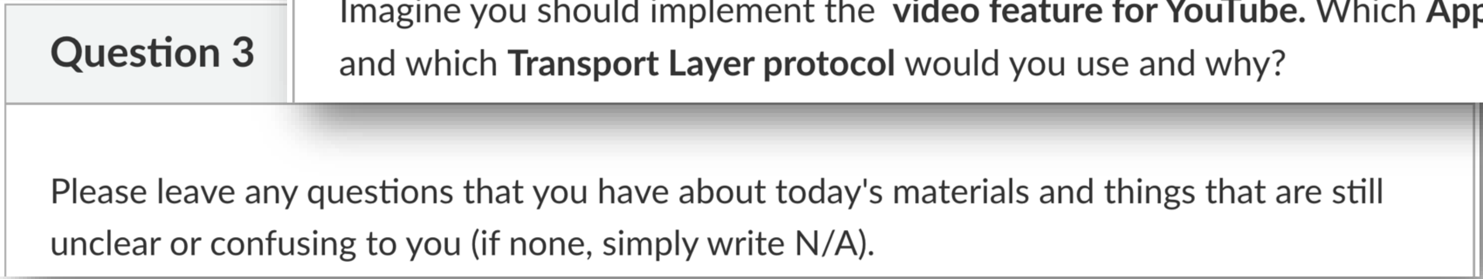
Question 1 1 pts

Please summarize **three insights you learned about Client Server, Networking, and/or Node.js** today.



Question 2 1 pts

Imagine you should implement the **video feature for YouTube.** Which **Application Layer protocol** and which **Transport Layer protocol** would you use and why?



Question 3

Please leave any questions that you have about today's materials and things that are still unclear or confusing to you (if none, simply write N/A).

Credits: These slide use images from Flaticon.com (Creators: Freepik, syafii5758, kerismaker)