NETWORK_PROGRAMMING_MASTERY

> “Ctrl+Z is life’s greatest gift - if only it worked outside computers.”

- Author: Kintsugi-Programmer

> Disclaimer: The content presented here is a curated blend of my personal learning journey, experiences, open-source documentation, and invaluable knowledge gained from diverse sources. I do not claim sole ownership over all the material; this is a community-driven effort to learn, share, and grow together.

Table of Contents

1 Simple Web Client Implementation Using Sockets: Super Depth Notes

1. Introduction to Sockets and Networking

This content discusses sockets and networking, particularly in the context of inter-process communication (IPC).

1.1 Sockets Defined

- **Sockets** are another way that one program can talk to another program.
- Sockets allow you to create a **connection between two programs**.
- The unique feature of sockets is that the connected programs **do not have to be on the same machine**.
- The programs do not have to be in the same city, country, or continent.
- A connection can be established between programs running anywhere in the world, or wherever a computer can be placed and provided with an internet connection.

1.2 Project Goal: The Simplest Web Browser

- The goal is to examine what might be the world's simplest web browser.
- A typical web browser performs two actions:
    1. It goes out to web servers and **fetches HTML and other content**.
    2. It **displays** the content.
- This demonstration focuses solely on using sockets to go out to a web server and **pull down some content**; it will not include the display aspect (no fancy graphics).

2. Code Setup and Preliminaries

The example code requires including several header files to access socket functions.

2.1 Preprocessor Macros and Definitions

The following definitions are used for convenience and configuration:

- **Server Port:** A macro is used to record the port the server is listening on.
  - **HTTP (the standard for web pages)** servers listen on **port 80**.
- **Buffer Size:** A macro defines the size of the buffer used to read data back in.
- **Socket Address Struct:** The `struct sock adder` appears frequently in the code.
  - It is defined using a preprocessor directive (`#define`) as **`esé`** for convenience, making the code "a little bit less obnoxious".

2.2 Error Handling Function

A variadic function named `err_n_die` is defined at the top of the code.

- **Purpose:** It allows the printing of error messages and then exits the program.
- **Usage:** It is used repeatedly because a sockets program can produce errors in "a lot of different ways".
- **Note:** Using `err_n_die` is a choice made for this example; it is not required for general error handling.

2.3 Main Function Usage Check

- The program uses a usage check to ensure it is run correctly.
- This program is designed to take an **IP address** as an argument.
- The IP address is the address of the server on the internet, which the program uses to find the server.

3. Core Socket Programming Steps

The following steps detail how the client connects to the server and handles data transfer.

3.1 Step 1: Creating a New Socket

The first action performed is creating a new socket.

- **Socket Definition:** A socket is essentially an **endpoint**. It acts "kind of like a file". It is one end of a two-way connection for writing data into or reading data out of.

The `socket()` call involves three primary parameters:

1. **Address Family (`AF_INET`):** This signifies an **Internet socket**.
   - `AF` stands for Address Family, and `INET` stands for Internet.
2. **Socket Type (Stream Socket):**
   - The two most common varieties are stream sockets and datagram sockets.
   - **Datagram Sockets:** Used for sending a single packet chunk of data.
   - **Stream Sockets:** Used to create a connection, send a stream of data, and receive a stream back. This program uses stream sockets.
3. **Protocol Number (Zero):** The zero is for the protocol number.
   - If there were multiple protocols to choose from, they would be specified here.

- Using the default (zero) selects **TCP** (Transmission Control Protocol), which is the standard for stream sockets on most modern computers.

3.2 Step 2: Specifying the Address

Before connecting, the address structure must be set up:

- The address is first zeroed out.
- The address family is specified again (Internet address).
- The port must also be specified.

Key conversion functions are used to ensure network compatibility:

Function	Definition/Purpose	Details
`htons`	Host to Network Short	Converts a number from the host's local byte order (big-endian or little-endian) to the **network standard byte order**. This ensures communication even if the two computers use different byte orders.
`inet_pton`	IP Address String Conversion	Converts the string representation of the IP address (passed as a program argument) into a **binary representation** of the address. If this function fails, an error is printed, and the program is done.

3.3 Step 3: Connecting to the Server

- The program attempts to connect to the configured address using the `connect()` function.
- If the connection fails, an error is generated.
- If the connection is successful, the program continues.

3.4 Step 4: Sending the HTTP Request

A line is created that will be sent to the server in an HTTP request.

- The goal is to send the simplest possible request.
- The request line includes:
  - The `GET` **command** (saying "I want a page").
  - The **forward slash (`/`)**, which requests the root home page.
  - The **HTTP version** being used (`HTTP/1.1`).
  - The sequence `\r\n\r\n`, which signals the end of the request.

The request is written into the socket using the socket identifier created previously. This action sends the characters over the network to the server.

3.5 Step 5: Receiving and Displaying Data

If the request is successful, the response from the server comes back.

- **Preparation:** The received line (buffer) is zeroed out each time through the loop to ensure the string is **null terminated**, preventing potential trouble.
- **Reading:** The program calls the `read()` function, just like reading from a file. (Sockets are "kind of like files").

- **Output:** Whatever data is obtained is printed out to the standard output (`standardout`).
- **Alternative Uses:** The received HTML content could optionally be saved to a file or later rendered to display web pages.
- **Loop Termination:** The `while` loop continues as long as `read` returns a positive number.
    - The loop ends if `read` returns a value **less than or equal to zero**.
    - A return value of **zero (0)** means the connection is over.
    - A **negative number** means there was an error.

Finally, the program exits.

4. Demonstration and Conclusion

4.1 Running the Program

- The program can be compiled (in the example, using a quick make file on a Mac).
- To test, the IP address of Google's home page was determined using the `ping google.com` command.
- The program was run using that IP address.
- **Result:** The program worked and returned a "whole bunch of HTML".
- This basic structure will work for "just about any web server out there".

4.2 Summary of Client Basics

- While the program could be enhanced to specify the URL or page requested, the main point was to show the basics of how a **TCP client** works.
- This demonstration showed how to **connect** to a server and how to **send data** and **receive data**.

4.3 Future Development

- Future content will discuss the **server side**, which is acknowledged as "a little more complicated".

2 Building a Minimal Web Server in C (Sockets)

I. Introduction and Context

This video continues a networking theme focused on programming with sockets. The previous video covered building a stripped-down **web client**—a basic version of browsers like Firefox or Chrome—that could make a connection, send a request, and receive a response, without handling rendering or graphics.

This video focuses on building the **world's smallest web server**.

The objective is to focus exclusively on the **network aspects**:

1. Using sockets to **accept a connection** from a client.
2. Handling a connection and receiving a request.
3. Sending a response.

Note: This server will **not deal with file IO** or actually serving up web pages; it will only send a simple response.

II. Code Preparation and Utility Functions

In sockets programming, a large number of header files often accumulate. To simplify the program, headers and some constants are grouped into a file named `common.h`. This file also includes preprocessor macros and constants like line lengths used in the previous video.

A. Utility Functions in `common.h`

1. `error_and_die` **function**:
    - Prints out an error.
    - It is a variadic function, acting much like `printf`.
    - It terminates/dies after printing the error.
    - It is used frequently when checking for errors.
2. `bytes_to_hex` **function**:
    - A simple function useful for debugging.
    - It takes a string of bytes.
    - It converts these bytes to a hexadecimal representation and prints them out.
    - This helps determine if there are any unprintable characters that might interfere with the program.

III. Server Initialization and Setup

The server's `main` function is slightly more complicated than the previous client example.

A. Variable Definition

Variables are defined at the top for sockets, addresses, and buffers (used for sending and receiving data).

B. Creating the Listening Socket

The first step is allocating resources by creating a new socket.

- **Type:** It is an **Internet socket** (`AF_INET`).
- **Protocol:** It is a **TCP stream socket** (`SOCK_STREAM`).
- A TCP stream socket behaves like a file; data can be read and written to it, but the data travels to a client or comes from a client, rather than going to disk.

C. Setting Up the Listening Address

The address structure specifies the location the server is listening on. The server accepts connections rather than connecting to another machine.

1. **Family:** Specified as an Internet address.
2. **Address Option:** The `INADDR_ANY` option is used, meaning the server will respond to anything.
3. **Port Specification:** The port the server listens on is specified.

D. Port Selection (18,000)

- **Standard Port:** Port **80** is the standard port most web servers listen on, and it is the default port browsers try to find.

- **Custom Port:** This example uses **port 18,000**.
- **Reason:** Most operating systems are particular about which users get to listen on port 80. Using port 18,000 allows the program to run without needing super user privileges or operating system reconfiguration. (It *is* possible to use port 80, but it may require super user access).

E. Binding and Listening

1. **Bind:** The listening socket is bound to the set address. This informs the operating system that the socket will listen to this specific address.
2. **Listen:** The `listen` call is made.
    - *Design Note:* These steps (`bind` and `listen`) are separate calls, which offers flexibility, although they are typically performed sequentially.

IV. The Server Execution Loop

Once listening, the server enters an **infinite loop**. Servers typically operate this way, accepting connections and handling them repeatedly until the server is intentionally terminated.

A. Accepting Connections

The core functionality happens when the `accept` function is called.

```
// Example concept (not literal source code)
con_FD = accept(listening_file_descriptor, NULL, NULL);
```

1. **Arguments:** The listening file descriptor is passed in.
2. **Address Arguments:** The last two arguments, which allow retrieving the address of the client that connected, are set to `null`. This means the server does not care who is connecting.
3. **Waiting:** `accept` waits until a client connects.
4. **Return Value:** When a connection is accepted, `accept` returns **another socket**, referenced as `con_FD` (connection file descriptor).
5. **Socket Status:** The original listen socket remains active and is still listening on port 18,000.
6. **Interaction:** The `con_FD` socket is the socket used to interact (talk) with the newly connected client.

V. Handling the Client Request

Once the connection is established, the server reads from the socket.

A. Reading Data

- The server reads data in chunks, up to `MAX_LINE - 1` bytes.
- It prints both the **binary equivalent** and the **text equivalent** of the data read. This printing is for sanity checking and helps identify bizarre or non-printable characters.

B. Detecting the End of the Request

The server continues reading until it detects a newline (`\n`) at the very end of a chunk.

- **Standard HTTP requests** end in `\r\n\r\n` (carriage return, newline, carriage return, newline).
- *Caveat:* Using a simple detection of `\n` is described as a "really hacky way" and not a robust method for detecting the end of an HTTP request, but it is sufficient for this example.
- **Buffer Management:** Each time through the loop, the buffer is zeroed out to ensure that whatever is read next is null terminated.

C. Error Check

After the loop breaks, the program checks for errors, such as if the number of bytes read (`n`) was negative.

VI. Sending the Server Response

If no errors are found, the server constructs and sends a simple HTTP response.

A. Constructing the Response

A very simple string is written into a buffer:

```
HTTP/1.0 200 OK
hello
```

- **Response Line 1:** `HTTP/1.0 200 OK`.
  - `HTTP/1.0`: Indicates the server is using HTTP version 1.0.
  - `200 OK`: This is the code that signifies success—the request was received and looks good.
- **Response Content:** The standard response content (normally the webpage) is replaced by the simple string "hello".

Note: This is a "dumb web server" because it always responds "hello" regardless of the request. To serve actual web pages, the server would need to parse the request, obtain the file name, read the file from disk, and then write that content into the socket.

B. Finalizing the Connection

1. The response is written into the connection socket (`con_FD`).
2. The connection socket is closed.
3. The server then returns to the infinite loop to receive subsequent requests.

VII. Testing and Demonstration

A. Running the Server

1. The server is compiled using a `make` file.
2. The executable (named `TCPS` in the example) is run.
3. The server output indicates that it is waiting on a connection: "I'm waiting on connection".

B. Client Connection via Browser

A standard browser (like Google Chrome or Firefox) is used to connect to the new server.

- The address entered is: `localhost:18000`.
  - `localhost` specifies connecting to a web server running on the user's own machine.
  - The port `18000` must be specified, otherwise the browser defaults to port 80.
- **Result:** The browser successfully displays the word **"hello"**. This confirms that the connection was made and a response was received.

C. Analyzing the Server Log (The Request Content)

The log output shows what the browser actually sent.

1. **Binary Data:** The log first shows a large amount of binary data.
2. **Request End:** The end of the HTTP request is clearly visible in the binary log as `0D 0A 0D 0A`, which corresponds to `\r\n\r\n`.
3. **Text Request Structure:** The text version of the request shows several fields:
   - `GET / HTTP/1.1`: This is the starting line, requesting the root page (`/`) of the website, using HTTP 1.1.
   - `Host`: Specifies the connecting host.
   - `Connection: keep-alive`: This indicates the browser would like to send multiple requests over a single connection. (The current server closes the connection immediately, ignoring this request).
   - `Upgrade-Insecure-Requests`: The browser requests that insecure requests (like this one) be upgraded to HTTPS. (The server does not comply).
   - `User-Agent`: A string providing information about the browser and its version. This tells the server who is communicating with it and what capabilities they support, potentially allowing the server to adjust its interaction. (The user agent string can be falsified).
   - `Accept`: Lists the content formats the client accepts, such as HTML, text, XHTML, XML, and images. It also includes `*/*`, meaning the client will accept "just about anything".
   - `Accept-Encoding`: Informs the server that the client can understand data compressed using `gzip` encoding.
   - `Accept-Language`: States the preferred language (e.g., American English).

D. Subsequent Requests (Favicon)

After receiving the first request, the server continued running and immediately received another request.

- Chrome, upon loading the page, sent a second request for the **`favicon.ico`** file. This is the default icon the browser displays for the website.
- The server responded to this request with "hello" as well, because the server always responds with "hello".
- **Browser Outcome:**
  - The browser successfully processed the initial "hello" response (for the root webpage).
  - The browser did not receive an image file for the icon; it received the text "hello".
  - Because the browser did not know what to do with the "hello" text when expecting an image, it displayed the default empty page icon.

VIII. Conclusion

While the demonstrated client and server are simple—not sending canned HTTP request strings or responding "hello" every time (though there might be a use for that)—the concepts provide useful tools for:

- Building communication into applications.
- Building clients and servers.
- Sending useful information back and forth between them.

3 Socket Servers: Retrieving and Displaying Client Addresses

Introduction to Socket Programming Enhancement

This document details how to enhance a socket server to retrieve the address of the client making a connection.

In a previous simple web server video, connections from clients were allowed and data was received from them. However, that server did not know anything about the client that connected to it.

This material focuses on the `accept` call to make the server "a little bit smarter" so it can get the client address.

Using the `accept` Call Arguments

The `accept` call in the previous example had two arguments at the end that were specified as `null`. These two arguments are specifically used for getting the address from the client.

Steps for Implementation with `accept`

1. **Structure and Length:** A structure and a length variable are needed to hold the client's address.
2. **Passing Pointers:** The server must pass in the address (a pointer) of the structure and the address of its length.
    - The `accept` function will fill these variables with data.
    - A pointer is passed so that the variables will be filled by `accept`.
3. **Output Goal:** The retrieved addresses will then be printed out.

Address Conversion using `inet_ntop`

When the address is returned by `accept`, it is in a **network format**. This means it is a struct containing binary data. To print the address, conversion is necessary.

The `inet_ntop` Function

The function `inet_ntop` (Internet Network To Presentation) is used for this conversion.

- **Function Purpose:** `inet_ntop` takes an address in network format and converts it into presentation format (P).

Steps for Using `inet_ntop`

To use `inet_ntop`, the following information must be provided:

1. **Address Family:** The function must be told that the program is dealing with Internet addresses.
2. **Network Data Pointer:** A pointer to the struct that `accept` filled with data must be provided.
3. **Buffer for String Conversion:** A buffer is required for the presentation form. `inet_ntop` converts the address to a string.
4. **Destination Buffer:** A location, such as `client address`, is specified where the resulting string will be put.
5. **String Length:** The length of the string is given so the function knows when to stop. This length detail handles the case of a "really huge dress" (address), although this specific scenario won't occur here.

The function signature or conceptual use might be visualized as:

```
// Example parameters for inet_ntop
// AF_INET: Internet addresses
// struct_pointer: Pointer to struct filled by accept
// client_address: Buffer where the resulting string is placed
// length: Length of the buffer
inet_ntop(AF_INET, struct_pointer, client_address, length);
```

Results and Caveats

Connection Output

After implementing this change, the string containing the client address can be printed out.

When a connection is received:

- All information about the connection and all data from the client are still printed out.
- The client address is printed out at the top.

Information Provided by the Address

The printed address can provide information about the client that connected to the server. This information may indicate:

- If the client is on the local machine.
- If the client is on the local area network (LAN).
- If the client is located somewhere on the other side of the world.

Important Caution

While this information is generally reliable, there is one caution:

- **Spoofing:** There are ways to spoof IP addresses.
- Therefore, the address information is not foolproof.

This method provides another tool to make servers more capable.

4 Dealing with Endianness Issues in Programs

1. Introduction to Endianness and the Problem

Endianness refers to **big-endian** and **little-endian** byte ordering. This topic can be a **huge source of frustration** for students. It suggests that life would be much better if companies communicated more effectively with each other.

1.1 Case Example

A student was working on debugging a system that she built. She used two machines: a **laptop** and a small microcontroller-based device, the **Teensy 3.6**.

- Data was sent from the laptop to the Teensy.
- What was received on the other side did not match what she thought she was sending.
- The fundamental issue was **byte order**.

2. Multi-byte Integers and Storage

Programmers think of multi-byte integers (such as a short, a long, or an int) as having a single value.

However, these numerical values are actually stored as **individual distinct bytes**.

2.1 Standard Numerical Representation

When we write numerical values, we typically write them with the **most significant byte (MSB)** coming first and the **least significant byte (LSB)** coming last.

3. Types of Endianness

3.1 Big-Endian

- **Definition:** When a computer stores a numerical value, the **most significant byte** comes first.
- **Meaning:** The **big end** comes first. The most significant end is considered the **big end**.
- **Examples of Machines:** PowerPC and Sun SPARC machines use big-endian.

3.2 Little-Endian

- **Definition:** The computer stores the number the **other way**, with the **little end first**.
- **Examples of Machines:** If you are on an Intel processor, you are probably using little-endian.

3.3 Bi-Endian

- **Definition:** Machines that are **bi-endian** can actually work in **either mode**.
- **Mode Determination:** The mode the machine is in will depend on the software being used.
- **Examples of Machines:** ARM processors are bi-endian.

3.4 Why Byte Order is Often Forgotten

Most of the time, programmers do not have to think about byte order. This is because the **compiler and the architecture** generally take care of it "magically".

When byte order issues *do* become a problem, it is often annoying because programmers forget about them, and they are frequently the last thing thought of to check.

4. Scenarios Where Byte Order Matters

4.1 Network Communication

Byte order matters when sending data (such as an `int`, a `struct`, or an array) to another machine somewhere on the network.

Steps for Sending Data:

1. Copy the number into a byte array.
2. Send the byte array over the network as a packet or message.

Example of Byte Order Issue:

If the data is stored and sent in **little-endian order**, the bytes will be sent with the `six seven` coming first and the `zero one` coming last.

If the receiving computer uses **big-endian byte order**, that number is going to be interpreted **very differently**. This difference in interpretation was the problem experienced by the student.

4.2 Writing Binary Data to a File

Byte order must be monitored anytime you are writing **binary data to a file**. That file might be read on another machine that has a different byte order.

5. Standard Solutions for Endianness

The standard way to handle the byte order problem is to use a **standard byte order**. This standard is often called **network byte order**.

Rule: When communicating, all machines out there are going to use that network byte order so that they can all understand one another.

5.1 Current Network Standard

Currently, **big-endian** is the **standard network byte order for the internet**.

It is probable that this standard will not change.

5.2 C Programming Functions

If you are programming in C, most platforms provide a set of standard functions that help manage byte order switching.

These functions help switch:

- From your native host byte order to the network byte order.

- From network byte order back to your native host order.
- Specifically to big-endian or little-endian, if desired.

Examples of Standard Functions (Names may vary slightly, but indicate function):

Function Type (Mnemonic)	Description	Applies to
`htons` (Host to Network Long)	Converts from host byte order to network byte order.	Longs or four byte ints.
`ntohs` (Network to Host Short)	Converts from network byte order to the native host byte order for the machine.	Shorts (two byte ints).

5.3 Alternative: Manual Byte Flipping

It is always possible to **"roll your own"** solution (do it yourself) and flip the bytes around. However, the recommendation is to use the provided functions if available.

6. Alternative Data Handling: Text vs. Binary

6.1 Binary Data

- **Efficiency:** Binary data is more compact and likely more efficient.
- **Size:** Less bytes need to be sent over the network.
- **Protocols Affected:** Binary protocols, such as **DNS**, must worry about byte order issues.

6.2 Text Data

- **Interpretation:** Text is always interpreted the same way on all machines, provided the software handles it consistently.
- **Protocols Affected:** Text-based protocols, such as **HTTP**, typically do not have to worry as much about byte order issues.

7. Arguments for Both Varieties of Endianness

Programmers often ask why both varieties exist since there isn't one best way to implement byte order. It seems to be largely a **matter of preference**.

7.1 Advantages of Big-Endian

- **Readability:** People usually think big-endian is more readable.
- **Operational Simplicity:** It can make some operations simpler, such as:
  - Checking the sign of a number.
  - Comparing two numbers.
- **Network Benefit:** A big-endian machine does not have to swap bytes when it receives them from the network, which is a plus.

7.2 Advantages of Little-Endian

- **Operational Simplicity:** It may make things like parity checking simpler.

- **Mathematical Argument:** Some people argue that little-endian is **mathematically more natural**.
    - This is based on the fact that many mathematical operations (like addition) start with the **low order byte** first, and then move from low order to high order. Storing them this way aligns with that process.
- **Integer Interpretation:** A 2-byte integer, if interpreted as a 4-byte integer, will be the same number, which may be an advantage.

8. Conclusion and Final Advice

If there were one clear winner, everyone would be using it. The reality is that all chip makers could have saved programmers many headaches by simply getting together and deciding on one single method.

Since this did not happen, programmers must be careful when writing code that is going to **send, save, receive, and interpret binary data**.

5 Multithreaded Server in C (Threads and Sockets)

1. Introduction and Context

1.1 Setting the Scene

After fifty-three days in Africa, there was rain, which started as a sprinkle. It was described as a little bit of rain and water falling from the sky. The speaker noted that it was crazy, having not really seen more than one or two clouds in the sky, finding the event "kind of cool," though not a rain storm per se.

1.2 Video Focus and Prerequisites

This tutorial aims to bring together **threads and sockets** to show how to make a **multi-threaded server**.

- **Prerequisites:** This is an intermediate video. It is assumed that the viewer understands the basics of C. Viewers should have previously seen videos on sockets (simple client and simple server videos) and threads.
- **Source Code Structure:** The video starts with a quick example that is similar to a previous server example.

2. The Standard TCP Server Flow (Single-Threaded Example)

The example server follows the **fairly standard flow** seen with most TCP socket servers.

2.1 Server Flow Steps

The typical steps for setting up the server are:

1. Create a socket.
2. Bind that socket to the desired port.
3. Call `listen`.
4. Go into an infinite loop.

2.2 Key Server Concepts

- **Stream Sockets / TCP:** The server is using stream sockets, which means it is using **TCP**. Connections look like a **stream of bytes** coming in one after another, instead of having separate packets of data.
- **Server Port:** The server is listening on a server port, which is set to `8989` in this example, but it could be any available port that the operating system allows access to.
- **Error Handling Function (`check`):** The `check` function is an error handling function written by the speaker. It takes advantage of the fact that many C functions fail in the same way: they return a **negative one** if there is an error. This single function handles all errors in a standard way, avoiding cluttering the code with many `if` statements.

2.3 Handling Connections

Once the server is listening, it goes into an infinite loop where it:

1. **Accepts new connections**.
2. **Passes each connection off** to the `handle_connection` function.
3. The server keeps accepting and handling connections over and over until it is killed using Control-C.

2.4 Function of `handle_connection`

The `handle_connection` function is fairly straightforward for this example.

- **Server Functionality:** The server allows clients to **read files** from the server.
- **Process:** The client sends the file name. The server then reads the file and sends the contents back to the client.
- **Analogy:** This is similar to how a web server works, but without the extra HTTP parsing.

Security Warning: This code has one big security problem. In its current form, it allows the client to **read any file from your hard drive**. It is an effective example to show how things work, but it is **not meant to be used out in the wild on the big bad Internet**. Running this code on a publicly available server could cause trouble.

2.5 Detailed Steps within `handle_connection`

The connection handling involves the following specific steps:

1. Reads whatever the client sends, up to `buff size`, until a **newline character** is received.
2. Checks for errors.
3. Mil terminates the buffer.
4. Prints out the request, so the user can see what is going on.
5. Checks to make sure the path is a **valid file path** using `real path`.
6. Opens the requested file.
7. Reads the file's contents out.
8. Sends the contents back to the client.
9. Closes the socket.
10. Closes the file.

Once these steps are complete, the connection is finished, and the server goes back to wait for another connection.

3. Testing and Performance Issues (Single-Threaded)

3.1 Testing Setup

- **Client Script:** A quick Ruby script was made to act as the client.
- **Connection:** The script connects to port `8989`, the port the server is listening on.
- **Request:** The script requests a particular file name.
- **Test Files:** Test files were set up in `/temp`, initially all copies of the server source code.

3.2 Simulating High Traffic

The focus is on performance, especially when handling **many connections**.

- **Timing:** The client script execution time is measured. Output is piped to `/dev/null` to avoid timing delays caused by printing to the terminal.
- **Network:** The test involves connecting to the server on the same machine for ease of testing. Connecting over a network would likely introduce longer delays, and results may vary based on network setup.
- **High Traffic Script:** A shell script was created to run the client script **50 times at the same time** in separate processes. These processes hit the server roughly simultaneously; they do not wait for the previous one to complete. This simulates high traffic.

3.3 The Backlog Problem

When running the high traffic simulation, several connections **failed**.

- **Cause:** The issue was the `backlog` specified when `listen` was called.
- **Definition of Backlog:** The backlog is the number of connections that the system will **queue up** before it starts rejecting connections.
- **Initial Value:** The backlog was initially set to **one**. This meant that when all 50 connections came in, it would not queue them all up.
- **The Fix:** Changing the backlog to **100** allowed for 100 waiting connections. After this change, running the script again showed that it worked.

3.4 Performance Analysis (Sequential Bottleneck)

Running 50 connections successfully takes quite a bit longer than a single connection. The core question is: **Can we make things faster?**.

- **Network Constraint:** If the network connection is slow and the server is spending all its time moving data to and from the client, speedup is unlikely without fixing the connection. In this case, the process is waiting on the network, not the server.
- **Server Constraint:** Speedup is possible if the server is spending a lot of time handling connections. This is a reasonable assumption because **disk access is tending to be slow compared to processing**.
- **When to Use Threads:** If the server is spending a lot of time on **disk accesses** or on tasks that can be done on different processor cores, we can speed things up by **handling connections in separate threads**.

4. Converting to a Multi-Threaded Server

The simplest approach for using threads is adopted first.

4.1 Goal and Strategy

The goal is to change the connection handling code so that it handles connections in separate threads. **Each connection will be in its own thread**. If connection handling can be done in parallel, a speed up might be achieved.

4.2 Pthreads Implementation Steps

1. **Include Header:** The `pthread.h` header must be included to gain access to the Pthreads functions.

   ```
   #include <pthread.h>
   ```

2. **Thread Variable:** A new `pthread_t` variable is created to keep track of the thread.

3. **Create Thread:** The `pthread_create` function is called.

   - The address of the `pthread_t` variable is passed in.
   - A thread function is provided.
   - An argument is provided.

   Implementation Note: The thread function used is `handle_connection`, as that is what the new thread should do.

4. **Argument Passing (The Pointer Requirement):** The `client socket` needs to be passed to the thread, but Pthreads requires the argument to be a pointer.

   - To ensure the pointer is not messed with by any other thread, space is **allocated on the heap for an integer**.
   - The value of `client_socket` is stored there.
   - This pointer (`P client pointer`) is then passed as an argument to the thread.

4.3 Code Adjustments: `handle_connection`

The `handle_connection` function must be changed because thread functions have specific requirements:

- **Signature:** Thread functions must return a `void *` pointer.
- **Argument:** Thread functions must accept a pointer.

Specific modifications to `handle_connection`:

1. The argument is renamed to remind the programmer that it is a pointer, not an integer.
2. The pointer argument is copied to a local variable once the function starts.
3. The pointer is freed immediately because it is no longer needed.
4. Everywhere `return` was called previously, it is changed to `return null` to appease the compiler, as the function now must return a pointer.
5. The function prototype at the top must be fixed to match the new signature.

4.4 Resolving Compiler Warning

A compiler warning occurs because the `handle_connection` function accepts an `int pointer`, but `pthread_create` expects a thread function that takes a `void pointer`. Although a pointer is generally a pointer, and this difference shouldn't strictly matter to `pthread_create`, the code is fixed.

The Final Fix:

1. Change `handle_connection` to accept a `void pointer`.
2. Cast the `void pointer` to an `int pointer` inside the function, because that is what it truly should be.
3. Change the function prototype up top as well.

5. Performance Comparison (Sequential vs. Concurrent)

The multi-threaded server compiles and runs fine.

5.1 Initial Threaded Results

- Single request times are similar to what they were before.
- Running 50 clients at once takes about the **same amount of time** as before, which is described as "kind of a bummer".

5.2 Improved Testing Methodology

To better analyze performance, changes were made to the client and testing script.

1. **Client Timing:** Instead of printing the file content, the client is modified to time how long it took to get the file and print that out, focusing specifically on performance.
2. **Test File Diversity:** The assumption that all test files were copies of the source code was not true. **Test file number six** is quite a bit larger than the others.
3. **Round-Robin Access:** The high traffic script is changed to request each of the different test files in a **round-robin style**. This means one out of every six requests will request the large file.

5.3 Results Analysis: Slow Connections and Delays

Testing single accesses showed that short accesses were quick, and requesting file number six (the large file) took longer, which was expected.

When running all 50 concurrent requests:

Scenario	Observation	Citation
With Threads (Concurrent)	Some connections are very fast, and some take a lot longer. The slow connections are mostly the long number six connections. The super fast requests finished quickly and got out of the way. Handling multiple requests concurrently slowed down the big downloads a bit, but many clients got better service.	
Without Threads (Sequential)	All connections take about the same amount of time, and most of them take a lot longer than they did before. The **slow connections cause delays**, and other pending connections have to wait their turn until the one before them finishes. The long number six connections stall everybody that comes afterward.	

5.4 The Importance of Concurrency

If connections were handled sequentially (without threads), a client requesting a YouTube video might have to wait until everybody before them in line finished downloading their entire video. The server might still serve the same amount of video, but the client could be waiting for days.

Clients downloading large files (like the Lord of the Rings trilogy) are usually willing to tolerate a small slowdown, as they know it will take a while. An extra five minutes of download time is tolerable, especially if the video can start playing before the whole download is done.

However, an extra **10-second delay** on a website's home page could determine whether people use the site or leave and never come back.

6. Demonstrating Major Speedup (Non-CPU Intensive Tasks)

The existing server is fast because it accesses a small set of files from a Solid-State Drive (SSD), making requests fast and uniform.

6.1 Simulating Slow Tasks

Threads provide major differences when the server must do anything slow that **doesn't tie up the processor**. Examples include:

- Accessing a slower magnetic disk.
- Accessing another server to handle the request.

For illustration, a **one-second** `sleep` is added to the `handle_connection` function. This simulates a slow but **not CPU intensive** task.

6.2 Results Comparison with Slow Task

1. Sequential Handling (No Threads):

- The 50 concurrent connections stack up, and things are **super slow**.
- It takes **over 50 seconds** to run all of them through, as each one waits for the one that arrived first to finish.

2. Concurrent Handling (With Threads):

- All of that waiting is done **concurrently**.
- A ton of time is saved.
- All accesses finish in just a **few seconds**.

7. Downsides of the Simple Threading Approach

While simple (spinning up a new thread for more work), this approach has drawbacks:

1. **Resource Consumption:** If there are 10,000 or a million connections at once, each thread requires **memory and CPU time**.
2. **Performance Killing:** At some point, adding more threads will not help; it will start killing performance, using up all memory, and making things extremely slow.

3. **Thread Creation Time:** A new thread is created for **every connection**. Creating new threads takes a little time. This time could potentially be saved by reusing created threads for future connections.

The following super depth notes draw exclusively from the provided source transcript excerpts.

6 Multithreaded Server Part 2: Thread Pools

1. Overview and Problem Statement

1.1 Context and Prior Server Model

This content continues a discussion on networking and threads, focusing on using threads in a multi-threaded server to improve performance.

The prior server model was simple:

1. The server accepts connections.
2. Clients provide a file name.
3. The server reads the file and sends its contents back to the client.

1.2 Security Note

The simple server is not particularly secure because it allows a malicious client to read almost any file on the machine. The user should not run this type of server on a publicly available platform.

1.3 The Performance Issue

The main issue with the prior approach is handling high connection volumes.

- If there is a flood of thousands of connections, the server creates a new thread for each connection.
- This uses a lot of memory.
- This overuse of memory can cause the server's performance to slow down ("grind to a really sad slow crawl").

1.4 Goal

The goal is to upgrade the code to guarantee that the server does not get into the specific situation where an unbounded number of threads are created, which degrades performance.

2. Solution: Thread Pools

2.1 The Concept of a Thread Pool

The idea behind a thread pool is simple:

- Rather than creating a new thread for every connection that comes in, a set of threads is created at the beginning.
- These threads wait ("hang out") and are available.
- As work (connections) comes in, the work is handed to one of the threads.
- If there are no threads available, that work waits until a thread becomes available.

2.2 Benefits

- It still allows the server to work on multiple things at once.
- It ensures that an **unbounded number of threads** are not created.
- The size of the thread pool is pre-selected, and the system sticks to that number.

2.3 Implementation: Defining and Creating Threads

1. **Define the size of the pool:** The number of threads to have in the pool must be defined.
    - Example used: **20** (This number is completely arbitrary).
    - The correct number depends on the hardware, the amount of memory affordable for the server, and the expected workload.
    - Experimenting and playing around with different numbers is necessary to see how performance is affected.
2. **Declare the pool:** An array of `P thread underscore T's` is declared to hold the pool of threads.
3. **Create the threads:** At the start of the server, the threads are created.
    - A loop runs from 0 to the size of the pool minus 1.
    - The function `pthread create` is called for each thread.
    - The `thread function` is passed into `pthread create`.

> **Note on Thread Function:** The threads must be reused, so their behavior is different than before, where they previously handled a connection and then died.

3. Managing Work with a Queue

3.1 Requirement for Sharing Data

To enable the thread pool structure, connection information must be put somewhere where the reusable threads can find it once they become available. This requires a shared data structure.

3.2 Introduction to the Queue

A special type of linked list called a **queue** is used for this purpose.

Queue Definition and Operations

- A queue is like any other linked list, but specific rules apply for adding and removing nodes:
    - Nodes are always added to one end.
    - Nodes are always removed from the other end.
- A queue is a **first-in first-out (FIFO)** type data structure.

Operation	Action
Enqueue (NQ)	Adds a node to the queue.
Dequeue (DQ)	Removes the first node from the queue.

DQ Implementation Detail

In the specific queue implementation used in the example, `DQ` returns `null` if the queue is empty and there is nothing to pull off the queue. This allows checking if there is "nothing to do here".

4. Server Implementation: Connecting the Queue and Threads

4.1 Main Program Logic (Adding Connections)

1. The queue header file must be included in the main program.
2. Prototypes for `NQ` and `DQ` must be included in the header file.
3. When the main server code receives a new connection, it takes the connection (as done before).
4. The connection is then added to the queue.
5. The connection stays in the queue until one of the threads comes along to retrieve it.

4.2 Thread Function Logic (Worker Threads)

The threads are never intended to die, so they are placed in an **infinite loop**. (A termination condition could be added, but for simplicity, the infinite loop is used, relying on Ctrl+C to terminate the process).

Inside the loop, the thread performs these actions perpetually:

1. **Call `DQ`:** Check if a node is available on the queue.
2. **Check for Work:** If the result (`P client`) is *not* `null`, the thread has work to do.
3. **Handle Connection:** The thread calls `handle connection`.
4. **Loop:** Once finished, the thread loops around again to check the queue for more work. The thread keeps doing this forever, over and over again.

5. Identifying and Fixing the Race Condition Bug

5.1 Initial Testing and Compilation Notes

Initial tests showed the server compiled and ran, waiting for connections. When a script firing off 50 clients was run, it worked and provided comparable performance to the original version.

Compilation required fixing several issues:

1. Quotes were needed in include statements so the compiler would search the current directory and standard directories for header files.
2. The Q code needed to be compiled.
3. The makefile required adding a rule to build files.
4. A new rule for building the server was needed.
5. The `-C` flag needed to be added to the compile rule.

5.2 The Major Bug: Race Condition

Running the server again revealed a crash ("rear its ugly head").

The problem is that the queue is a **shared data structure** and is **not thread safe**.

- This scenario is known as a **race condition**.
- A race condition occurs when:
    - Two threads try to remove work (`DQ`) from the queue at the same time.

    - One thread tries to `DQ` while the main thread is calling `NQ`.
- A race condition can leave the data structure in a bad state, resulting in **bad pointers** or **double frees**.

5.3 The Fix: Protecting the Queue with Mutex Locks

The fix is straightforward: protect the calls to `NQ` and `DQ`. This is achieved using a **mutex lock**.

Implementation Steps

1. **Create the Lock:** A mutex lock is created (example name: `mutex`).
2. **Protect NQ (Main Server):**
    - The mutex is locked before calling `enqueue`.
    - The mutex is unlocked afterwards.
    - This prevents anyone else from messing with the queue while a connection is being added.
3. **Protect DQ (Worker Threads):**
    - The mutex is locked before the `DQ` call.
    - The mutex is unlocked afterwards.

Result of Mutex Fix

- **Performance:** Grabbing and releasing the lock adds a small amount of time, making the server "a tad slower," but not significantly.
- **Stability:** The server no longer crashes and can be run for a long time.

The resulting server is a multi-threaded server that uses a thread pool to ensure millions of threads are not created when traffic is high, thus preventing performance degradation.

6. Remaining Issue: CPU Consumption

6.1 Observation of High CPU Use

If the `top` utility is run, the server is observed consuming "every last CPU cycle" on the machine.

6.2 The Reason: Busy Waiting

The worker threads are constantly checking ("busily checking") to see if there is more work, even when there is no traffic whatsoever.

- The threads are constantly asking: "hey is there more work is there more work is there more work is there more work".
- This behavior burns CPU cycles repeatedly until new work arrives.

6.3 Consequences of Busy Waiting

Although the server is functional, this continuous checking leads to negative consequences:

- The machine runs "really hot".
- A lot of energy is wasted.
- The server does not "play well with other programs" running on the same machine.

6.4 Future Solution

This issue is planned to be fixed using **condition variables** in the next video.

7 Multi-Threaded Web Server: Finishing Implementation with Condition Variables

Background and The Thread Pool

This video is the third in a series focusing on building a multi-threaded web server.

The Purpose of the Thread Pool Previously, a thread pool was added to the server. The thread pool ensures that if tons of requests come in, millions of threads are not produced, which would cause the server's performance to grind to a halt.

How the Thread Pool Works The thread pool consists of a bunch of threads that sit and watch a **work queue**.

1. As a connection comes in, it is stuck onto the work queue.
2. When a thread is available, it grabs that connection (or any piece of work) off the queue.
3. The thread then goes to work on the item.

The Problem with Constant Checking

Where the implementation left off, each thread was constantly checking the queue to see if there was new work to be done.

- This constant checking is problematic.
- It means the server uses up **all the available CPU cycles** that the machine has, even when no connections are coming in.
- This is definitely not desirable.

The 'Sleep' Option and Its Drawbacks

One obvious option suggested was to simply add a `sleep`.

Implementation of Sleep Any time a thread tries to get work from the queue and doesn't find any, it would just sleep (e.g., for a second, or less time).

The Consequences of Sleeping

- Connections start taking longer (between the time they were taking before and a second).
- When new work comes in, all threads may be sleeping, and the connection has to wait up to a second for one of the threads to wake up and handle it.
- Sometimes handling is fast (if a thread just woke up), and sometimes it is slow.

The Trade-Off If a shorter sleep time is used, the server becomes more responsive. However, the more responsive the server is made, the more CPU cycles are consumed. There is a fundamental trade-off between how much CPU overhead is acceptable and how responsive the server needs to be.

Adding sleeps is not considered a satisfying solution.

Solution: Using Condition Variables

The sleep approach is removed. Instead, a **condition variable** is used.

Definition and Operations

Students often mistakenly think that the name "condition variable" implies it has something to do with an `if` statement or some kind of logical condition, but it does not.

Condition Variable Purpose All a condition variable does is allow threads to wait for some event (a condition) to occur.

- When threads are not needed, they can be suspended.
- They jump into action when the time comes for them to actually do something. This is exactly what is needed in the web server.

Two Things You Can Do with a Condition Variable

1. **Wait:** A thread can wait on the condition variable. The thread will wait, not doing anything, until another thread calls `signal`.
2. **Signal:** Another thread can call `signal` on the condition variable. The waiting thread will wait until another thread calls `signal` on that condition variable.

Interaction with Mutex Locks

Condition variables are designed to work closely with mutex locks. When using the wait operation, the mutex must be passed in.

The Wait Mechanism When a thread calls `wait`:

1. It waits.
2. It also **releases the lock**.
    - If the lock was not released, other threads could not grab the lock to put work on the queue, and they would be stuck.
    - Releasing the lock allows other threads to access that critical section (to put work on the queue).
3. When it is time for the thread to stop waiting, it will **reacquire the lock** before `wait` returns.

Initial Implementation (Attempt 1)

The condition variable is added to the server.

Steps for Initial Implementation

1. **Waiting:** Before any thread tries to take work off the queue, the thread calls `wait`:

```
pthread_cond_wait
```

This is the pthreads version of waiting on a condition variable, which makes the thread wait until it is signaled.

2. **Signaling:** Up where work is added to the queue (where the `enqueue` function is called), `signal` is called.
    - Calling `signal` lets one of the currently waiting threads stop waiting and jump back in.

Initial Results (CPU Usage)

After compiling and running the server with this new logic:

- The server is no longer consuming all the CPU cycles.
- It often does not even show up near the top of the `top` list.
- All the server's threads are suspended, waiting for work to come in.
- These suspended threads are not using any CPU.
- Connections are once again handled very quickly.

The Issue: Condition Variables are Stateless

Although CPU consumption is solved, the server still has an issue. This issue appears when a large number of connections arrive simultaneously (e.g., 150 connections all at once).

When 150 connections are sent, the server starts off okay but then gets stuck.

The Problem Explained

The initial implementation was too simplistic: threads were waiting each time through the loop, and the main thread was signaling each time a new connection arrived.

The Stateless Nature of Condition Variables Condition variables are **stateless**.

- This means that when `wait` is called, the thread waits.
- It does not matter how many times `signal` was called before the `wait`.

The Failure Scenario

1. A lot of connections arrive and are added to the queue, and `signal` is called.
2. Let's assume the threads are not keeping up.
3. At some point, no new work is added to the queue, so `signal` stops getting called.
4. There is still a bunch of work left on the queue.
5. Worker threads finish their current tasks, call `wait`, and wait, **even though** there is work still to be done.
6. The waiting threads will not grab any more work until another connection arrives and calls `signal`.

The Final Fix

This problem is easily sorted out. The whole point is that threads should only wait if they cannot get work from the queue.

Steps for the Final Implementation

1. The thread should first attempt to call `DQ` (dequeue).
2. Only if `DQ` returns null should the thread call `wait` on the condition variable.

Result of the Final Fix The server no longer runs into this trouble. The server can now be compiled and run, and it successfully processes all connections.

The result is a working multi-threaded server that:

- Uses a thread pool (preventing a million threads from piling up).
- Does not use all CPU cycles while waiting for new connections (due to condition variables).

Remaining Server Issues (Future Work)

This multi-threaded server example concludes here, but the server is not without its issues and could still be improved.

Vulnerability: Denial of Service (DoS) Attack

The server is still vulnerable to a particular kind of Denial of Service attack.

DoS Scenario

1. A malicious client connects to the server and takes a long time to send its message (data comes in very slowly).
2. One thread from the thread pool will be occupied, waiting and waiting, trying to get the data off this slow connection.
3. If a malicious user creates a bunch of these really slow connections, they will occupy all the threads in the thread pool.
4. The server becomes occupied with these long, slow connections and cannot handle anybody else. This could bring the server to its knees.

Future Solutions Ways to handle this type of vulnerability include using event-driven models and asynchronous I/O. These concepts will be covered in future videos.

8 Depth on the `select` Function in C for Asynchronous I/O

1. Introduction to `select`

The topic for discussion is the `select` function. This function is being introduced as part of a longer journey into asynchronous I/O and event-driven software design. The goal is to help learning developers thrive.

The source code for the video demonstration is available through Patreon.

2. Review of the Simple Web Server Design

The starting point is simple web server code, similar to code previously written for a basic socket server in C. The modified code is designed to be easier to follow.

2.1 Server Operation Cycle

1. The code initially sets up the server, instructing it to listen on a specific port.

2. It then waits for connections.
3. When a connection is received, it handles that connection using the `handle connection` function.
4. Once the connection is handled, the server returns to wait for the next connection, repeating this cycle forever (rinse and repeat).

2.2 Core Server Functions

The server relies on several key functions:

- `set up server`: This function calls `bind` and `listen` to establish the server socket, which is then returned for use elsewhere.
- `accept connection`: This function calls `accept` and includes error checking.
- `check`: This function performs common error checking and is designed to end the program if anything goes awry.
- `handle connection`: This is where most of the server's heavy-lifting occurs. It performs the following steps:
    - Reads the request from a socket (which is the path to a file).
    - Reads the contents of that specified file.
    - Sends those contents over the network back to the client.

Note on Security: This illustrative code is helpful for educational purposes but is **not very secure**. It should not be posted on a public web server.

2.3 Analysis of the Simple Design

Aspect	Detail	Source
Upside	Simplicity. The code is only about 128 lines of C code, including comments (around 40 lines are probably includes).	
Downside (Slowness)	The server is fairly slow because it only handles one connection at a time.	
Stalling Risk	If one connection stalls (due to slow internet or a malicious user being intentionally slow), it negatively affects service for everyone else.	

2.4 Threads as a Previous Solution

Threads are one way that was previously discussed to handle the stalling issue.

Issues with Threads:

1. **Memory Use:** Using threads takes up a decent amount of memory, as each new thread created requires a significant amount.
2. **Vulnerability:** A slow connection can still mess up one thread. A pool of 100 threads can still be clogged by 100 intentionally slow connections.

3. Alternative: `select` and I/O Models

`select` is an alternative approach that allows for **some degree of concurrency** without requiring the creation of new threads.

3.1 I/O Approach Definitions

Most standard I/O functions use a blocking approach.

- **Blocking/Synchronous Calls:** When a function (like reading from a file) is called, it **blocks or pauses the current thread** until the operation is complete. Once the data returns, the function returns, and the computation resumes.
- **Non-blocking/Asynchronous Calls:** These calls make a request and then rely on an **event, a signal, a callback function, or an interrupt** to indicate when the request finishes. This allows the program to get other work done while waiting.

3.2 `select` Position

The `select` function is described as "sort of asynchronous". It still blocks, but it attempts to achieve asynchronous behavior.

4. How `select` Works

`select` takes a group of **file descriptors (FDs)** and reports when there is something ready to read on any of them.

- File descriptors can include: open files, open network sockets, or anything "file-like" (since nearly everything is treated like a file in a UNIX system).
- The function also works for writing, but the example focuses on reading.

Example: A server with 10 current connections can instruct `select` to "watch these 10 connections and let me know when any of them is ready for reading".

4.1 Setting up File Descriptor Sets (`fd_sets`)

1. **Declaration:** Two `fd_sets` must be declared. `fd_sets` is short for file descriptor sets. For now, they can be thought of as a collection of file descriptors, though they are technically a bit field.
2. **Initialization Macros:** `select` provides macros for working with `fd_sets`:
    - `FD_ZERO`: Used to zero out or initialize the set (e.g., the set of current sockets).
    - `FD_SET`: Used to add one socket (e.g., the server socket) to the current set.

4.2 Handling the Destructive Nature of `select`

`select` **is destructive**. It changes the set passed into it.

Because of this:

- A temporary copy (the second declared `fd_set`) is required.
- Each time through the loop, the current set of sockets must be copied to the ready socket set (the temporary copy). This prevents losing the master list of descriptors being watched.

4.3 Calling `select`

The `select` function requires multiple arguments:

Argument	Description	Example Usage in Source	Source

Argument	Description	Example Usage in Source	Source
FD Range	The range of file descriptors to check. This is **not** the number of descriptors in the set, but the **maximum possible file descriptor**.	`FD_SETSIZE` is used initially.	
Read Set	The set of FDs to check for reading.	The ready socket set (temporary copy) is passed in.	
Write Set	The set of FDs to check for writing.	Passed as `NULL` and ignored for this example.	
Error Set	The set of FDs to check for errors (e.g., on a socket or file).	Passed as `NULL` and ignored for this example.	
Timeout	An optional value to specify how long `select` should wait for changes.	Passed as `NULL`, meaning `select` will wait forever (or until an FD has something ready to read).	

If `select` returns an error, the example program prints the error and exits, although applications should handle errors more robustly.

When `select` returns, it signals that **one of the file descriptors has work**.

4.4 Processing the Results

After `select` returns, the `FD set` that was passed in **contains only the file descriptors that are ready for reading**.

To identify which FD is ready, the program must iterate and check.

1. **Iteration:** Loop through the range of possible file descriptor values, starting at 0 and going up to `FD_SETSIZE` (which is the largest numbered file descriptor that can be stored in an `FD set`).
2. **Checking:** Use the `FD_ISSET` macro to check if the current file descriptor value (`I`) is set. If it is set, `I` is a file descriptor with data ready to read.

```
// Example logic for checking FDs after select returns
for (int i = 0; i < FD_SETSIZE; i++) {
    if (FD_ISSET(i, &ready_socket_set)) {
        // Handle I, which is a ready file descriptor
    }
}
```

4.5 Handling Ready File Descriptors

There are two primary cases when a file descriptor is ready:

Case	Condition	Action	Macro/Function Used	Source

Case	Condition	Action	Macro/Function Used	Source
1. New Connection	The ready FD (`I`) is the **server socket**. This means there is a new connection to accept.	Call `accept new connection`. This call will return immediately because `select` has already guaranteed data is ready. Then, add the new client socket to the list of watched sockets.	`FD_SET` (to add the new socket).	
2. Data Ready	The ready FD is one of the **client sockets**.	Read its data and handle the connection. Once handling is complete, remove the socket from the list of FDs being watched.	`FD_CLR` (to remove the socket).	

The server loop continues this process forever.

5. Benefits and Criticisms of `select`

5.1 The Good Aspects (Benefits)

- **Concurrency:** `select` provides a mechanism for achieving some concurrency without having to create new threads.
- **Efficiency:** A single thread can use its time more efficiently by avoiding "dead waiting". `select` informs the thread exactly when connections actually have data ready to read.
- **Portability:** `select` is highly portable, meaning it is "basically available everywhere," unlike some of its more modern replacements.

5.2 The Bad Aspects (Gripes)

5.2.1 Limitation in the Current Example

If a client connects, sends a small amount of data, and then stalls, the program still handles the entire connection in one shot. The thread will still have to wait/stall.

- **Potential Fix:** This limitation could be fixed by only handling a single `read` call each time through the loop after `select` returns. However, this requires more serious modification and restructuring.

5.2.2 Inefficient Iteration and `FD_SETSIZE`

The main criticism is the requirement to iterate through the whole range of possibilities to find which FD is ready.

- **Size of Check:** On the machine used in the example, `FD_SETSIZE` is **1024**.
- **Overhead:** If a server only has two active sockets (the server socket and one connected client), it still must check **1024 different possibilities** every time the loop executes after `select` returns.

5.2.3 Mitigating the Iteration Overhead

The overhead can be slightly improved by tracking the **largest socket number seen so far**.

- If the server socket is 3 and there is one client at 4, the program only needs to loop up to 4, improving performance.

5.2.4 The Server "Gets Slower with Age"

Tracking the maximum socket number is not a complete fix.

- If the server handles 500 simultaneous connections, the maximum socket number becomes large.
- From that point on, the loop must check that larger number of socket numbers until the server is restarted.
- This causes the server to get slower as it ages, which is "not very satisfying".

5.2.5 Connection Limit

Due to the fixed size of `FD_SETSIZE` (1024 on the speaker's machine), the server cannot handle more than 1024 active connections. This size may vary depending on the machine.

6. Future Topics

Future videos plan to cover:

- Improvements to the current `select` example.
- Further discussions on asynchronous I/O and event-driven programming.
- Bit fields and bit masking, to explain what is happening "under the hood" with `FD sets`.

9 libcurl: Easy Networking in C

1. Introduction and Rationale

This guide demonstrates how to add **networking capabilities** to programs the easy way using the libcurl library. We will also cover **callback functions**.

While low-level network programming using sockets is valuable, sometimes time constraints demand a quicker solution.

libcurl is beneficial when:

- You need to **get the job done quickly**.
- You are short on time (you don't have a week or a month).
- Your program needs to interact with a standard server, such as a **web server, FTP server, or mail server**.

The alternative—implementing these protocols from scratch, perhaps using inline assembly—is a great educational opportunity but too time-consuming when the priority is getting something working today.

2. libcurl Availability and Naming Convention

Availability: libcurl is a library available on **just about any computing platform** out there. It reportedly works on Windows.

API Convention: libcurl frequently includes the word **"easy"** in its API calls, such as `curl easy init`, `curl easy cleanup`, `curl easy set opt`, and `curl easy perform`. This reflects the library's focus on simplifying network programming.

3. Basic Program Setup and Initialization

To use libcurl, the following steps are required:

3.1. Inclusion and Initialization

1. **Include the Header:** Include the necessary curl header at the top of the program.
2. **Initialize the Library:** Call `curl easy init`.

```
#include <curl/curl.h>

CURL *curl_handle = curl_easy_init();
```

3.2. Handle and Cleanup

- **Handle:** The pointer returned by `curl easy init` (e.g., `curl_handle`) is a **handle** (a pointer to a `curl` struct). This handle is used in subsequent calls to the library.
- **Cleanup:** Perform cleanup at the end by calling `curl easy cleanup`.

3.3. Error Checking

Although initialization failure is rare (the speaker has never seen it fail), it is possible. Error checking should be included:

```
CURL *curl_handle = curl_easy_init();
if (curl_handle == NULL) {
    // Print out a little error message
    // return failure
}
```

4. The Basic Pattern: Set Options and Perform

The fundamental structure for using libcurl in a program is:

1. **Set options**.
2. **Perform the action**.

4.1. Setting Options (URL Example)

The function used to set options is `curl easy set opt`.

The simplest example is setting the URL:

1. Call `curl easy set opt`.

2. Pass the curl handle.
3. Specify the option: `CURL OPT URL`. This tells libcurl the URL to download.
4. Provide the URL string.

```
curl_easy_setopt(curl_handle, CURLOPT_URL, "https://www.personal-webpage.com");
```

Protocol Detection: The library automatically detects the protocol (e.g., HTTPS) based on the URL prefix, which simplifies usage.

4.2. Performing the Request

The action is executed by calling `curl easy perform`. This takes the current set of options and executes the request.

Default Behavior: By default, libcurl connects to the specified URL, downloads the contents, and **dumps the output to standard out** (`stdout`).

4.3. Handling Errors During Performance

The return code from `curl easy perform` should be saved.

If the return code (`result`) is not equal to `CURLE OK`, an error message should be printed to standard error (`stderr`):

```
CURLcode result = curl_easy_perform(curl_handle);

if (result != CURLE_OK) {
    // Print out error message to standard error
}
```

Standard error is used for error messages because standard out may be used for downloaded data.

4.4. Compilation

To successfully compile the program, it must be linked with the **libcurl library**. Failing to link will result in unresolved symbol errors.

5. Custom Data Handling using Callback Functions

The default behavior of dumping content to the terminal is often not useful; typically, a program needs to **save data to a file, analyze it, or otherwise process it**.

5.1. Introducing Callback Functions

To customize how data is handled, a **callback function** is used.

Definition: A callback function is a function that is called by the library whenever a specific event occurs (e.g., data arrival, completion of an action). We provide the library function (like `curl set opt`) with a **function**

pointer, asking it to call that function when the event happens.

5.2. Setting the Write Function Option

To process the incoming data:

1. Set the option: `CURL OPT WRITE FUNCTION`.
2. Pass a **function pointer** (e.g., `got_data`) instead of a string.

This instructs Curl that when data arrives from the server, it should perform the write operation to the specified function, rather than writing to standard out.

```
curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, got_data);
```

5.3. Implementing the Callback (`got_data`)

The callback function must adhere to a **very particular look** determined by the library. This function is called whenever a **chunk of data** arrives from the server.

Data Arrival:

- Data is received in **chunks**.
- The chunks are **not all the same size**.
- The size variability depends on how data was broken into network packets and how the network stack handed the data up; programmers cannot control or make assumptions about chunk size.

Function Signature (Required Structure):

Parameter	Type	Description
Return Value	`size_t`	The number of bytes the function processed.
1st Argument	`char *buffer`	The buffer holding the received data.
2nd Argument	`size_t item_size`	The size of each item (this is **always going to be one**).
3rd Argument	`size_t number_of_items`	The number of items.
4th Argument	`void *pointer`	A pointer that can be used to pass an additional object (ignored in this example).

Example Calculation of Bytes: The total number of bytes currently being worked with is computed by multiplying the two sizes:

```
size_t bytes = item_size * number_of_items;
```

5.4. Crucial Return Value

The function **must return the number of bytes processed (`bytes`)**.

Termination Condition: If the function returns **less than the number of bytes in the buffer**, Curl assumes an error, terminates, and will not continue pulling down subsequent chunks of data. Returning the full calculated `bytes` signals that **all bytes in the buffer were processed** and data transfer should continue.

6. Practical Example: Adding Line Numbers

This example demonstrates processing the incoming data chunk-by-chunk to print the content to standard out with line numbers.

The `got_data` function implementation:

```
size_t got_data(char *buffer, size_t item_size, size_t number_of_items, void
*pointer) {
    size_t bytes = item_size * number_of_items;

    // Declare and initialize line number counter
    // NOTE: This counter resets with every new chunk unless made static or
global.
    int line_number = 1;

    // Announcing the new chunk (for debugging/visibility)
    printf("New chunk received. Length: %zu\n", bytes);

    // Print the first line number
    printf("%d: ", line_number);

    // Loop through the data byte by byte (from beginning to end)
    for (size_t i = 0; i < bytes; i++) {
        char character = buffer[i];

        // Print the current character to standard out
        putchar(character);

        // Check if the character just printed was a new line
        if (character == '\n') {
            // If it was a new line, increment the line number
            line_number++;
            // Print the next line number, effectively sticking it at the
beginning of the next line
            printf("%d: ", line_number);
        }
    }

    // Add vertical separation after each chunk for easier viewing
    printf("\n\n");

    // Return the number of bytes processed to ensure subsequent chunks are
received
```

```
    return bytes;
}
```

Line Number Scope: In this specific implementation, the line numbers **reset after each chunk**. To make the line counter global across all chunks, the counter would need to be made **static or global**.

7. Protocol Flexibility (FTP Example)

One of the most useful features of libcurl is its flexibility in handling different protocols.

By simply changing the URL string, the program can use a totally different protocol.

Example: Switching from an HTTPS server to an FTP server: The entire URL is changed to specify the FTP protocol, including login credentials and file path.

Original (HTTPS): `https://www.personal-webpage.com`

New (FTP):

```
ftp://demo:password@test.rebex.net/readme.txt
```

(Note: `test.rebex.net` is a publicly available server used for demo purposes. Username: `demo`, Password: `password`).

By only changing this URL, the program is able to pull information from the FTP server using a totally different protocol. This is considered **super easy and super cool** and is highly useful for communicating with standard servers using standard protocols.

10 Progress Bars in Terminal Programs: Accuracy and Implementation

Introduction to Progress Bar Issues

Progress bars are essential components in software when operations, such as processing or downloading, take a significant amount of time. Despite their ubiquity, progress bars are **inaccurate**.

They often fail to provide an accurate view of the process location or remaining time:

- A bar might indicate 50% completion when the task is almost finished.
- A bar might indicate 50% completion when a large amount of work remains.

This inaccuracy is a common observation, even asked about by people outside the programming field. This discussion aims to build a progress bar in C for terminal programs and explore these issues.

Building a Simple Terminal Progress Bar

Initial Program Structure

A simple starter C program is used, along with a typical `make` file for compilation. The initial goal is to periodically update a status bar.

The `main` loop structure iterates from 0 to 100 (inclusive):

```
for (int i = 0; i <= 100; i++) {
    update_bar(i);
    // Wait a little while to see the progress
    usleep(20000); // 20 milliseconds
}
```

- `update_bar` is the function called to update the progress bar, passing in the current percentage (`i`).
- `usleep(20000)` slows things down so the process can be observed.

`update_bar` Function Implementation (Initial Draft)

The goal is to stick a progress bar on the current line that grows until the task is completed. Simplicity is prioritized, avoiding complex features like colors or using `curses` for fixed placement.

A constant is defined for the length of the bar:

```
const int BAR_LENGTH = 30; // 30 characters long
```

The function signature for `update_bar` is defined:

```
void update_bar(int percent_done) {
    // Function implementation details
}
```

The function calculates how many characters should be marked as "done" or filled in based on the percentage.

Calculation of Done Characters (`num_cares`): To avoid integer division flooring the result to zero (if division by 100 happens first), multiplication must occur before division:

```
int num_cares = (percent_done * BAR_LENGTH) / 100;
```

Printing the Progress Bar Elements:

1. **Start Bracket:** The line starts with an open square bracket.
2. **Filled Characters:** A loop prints the filled characters (pound signs) up to `num_cares`.
    - The pound sign (`#`) is used because it looks like a bar and fills in the space.
3. **Unfilled Characters:** A second loop prints spaces for the remainder of the bar length.

```
// 1. Open bracket
printf("[");

// 2. Print filled characters
for (int i = 0; i < num_cares; i++) {
    printf("#");
}

// 3. Print unfilled characters (spaces)
for (int i = 0; i < (BAR_LENGTH - num_cares); i++) {
    printf(" ");
}
```

4. **End Bracket and Percentage:** The line is closed with a square bracket, and the actual percentage is printed.

```
// 4. Close bracket and print percentage
printf("] %d%% done", percent_done);
// Note: This initial version implicitly prints a newline later or relies on the
loop structure.
```

Making the Bar Grow In Place

The initial compilation and run show the progress bar printing on a bunch of different lines. To make the bar grow in place on the same line, three changes are required.

1. **Remove Newline:** Remove the newline character (`\n`) from the output of the progress bar.
2. **Add Carriage Return:** Add a **carriage return** (`\r`) to the very beginning of the progress bar output.
    - The carriage return means "return the carriage/cursor to the start of the line". This allows the new bar to print over the top of the last line.

    The updated print statement structure:

    ```
    printf("\r["); // Added \r at the start
    // ... bar content ...
    // Note: No \n at the end of the line
    ```

3. **Flush Standard Output:** After implementing the carriage return, running the program might show nothing until the end. This happens because **standard out is buffered** for speed reasons. The terminal waits until it sees a new line character before pushing the content to the screen.

    To fix this, `fflush` is called with `stdout` after printing the bar:

    ```
    fflush(stdout); // Forces the buffered content to write immediately
    ```

After the loop completes, a final newline character (`\n`) must be added to ensure the terminal prompt appears correctly afterward, avoiding an "odd percent character".

Transitioning to a Realistic Example (Downloads)

The simple progress bar example works well because it grows steadily over time, but this is only because the program is not doing anything real.

The example is changed to simulate a **series of downloads** of different sizes.

Defining URLs and Tracking

An array of character pointers (URLs) is used, including images, videos, and a large Debian CD ISO image.

The number of URLs (`num_urls`) is calculated dynamically using array size divided by element size.

Structuring Status Information

To avoid using global variables, a `struct` called `status_info` is created to track information.

Initial `status_info` structure:

```
typedef struct {
    size_t total_bytes; // Tracks total bytes downloaded
    // ... (More fields added later)
} status_info;
```

The `download_url` Function

The `download_url` function handles fetching a single URL using `libcurl`.

Function signature (initial): `bool download_url(char *url, status_info *s_info)`.

Libcurl Setup Steps:

1. Initialize curl object handle: `curl_easy_init`.
2. Clean up at the end: `curl_easy_cleanup`.
3. Set options (`curl_easy_setopt`):
    - Set the URL: `CURLOPT_URL`.
    - Define a write function (where data goes): `CURLOPT_WRITEFUNCTION` (e.g., `got_data`).
    - Define the data (pointer) to pass to the write function: `CURLOPT_WRITEDATA` (passing `s_info` pointer).
    - Enable redirects: `CURLOPT_FOLLOWLOCATION` set to `1` (This prevents needing to manually track redirects).
4. Perform the operation: `curl_easy_perform`.
5. Return `true` or `false` based on success (`result == CURLE_OK`).

The Data Handler Function (`got_data`)

This function is called by libcurl when new data is received.

Function signature: `size_t got_data(char *buffer, size_t item_size, size_t num_items, void *st_info)`.

- `st_info` (void pointer) is the status info pointer passed via `CURLOPT_WRITEDATA`.

Inside `got_data`, the void pointer is cast so it can be used:

```
status_info *status = (status_info *)st_info;
```

The size of the received data is calculated:

```
size_t bytes = item_size * num_items;
```

The function must return the total number of bytes processed to tell curl to keep getting data.

Key Action: Keep track of total received bytes:

```
status->total_bytes += bytes;
return bytes;
```

Initial Inaccurate Progress Tracking

In `main`, the loop structure is changed to iterate over the number of URLs.

```
// Initialize status info
status_info s_info;
s_info.total_bytes = 0;

// Loop through downloads
for (int i = 0; i < num_urls; i++) {
    download_url(urls[i], &s_info);

    // Update bar based on percentage of URLs done
    int percent = (i + 1) * 100 / num_urls;
    update_bar(percent, &s_info); // Assuming update_bar now accepts s_info
}
```

This initial tracking approach is simple: it calculates the percentage of URLs completed.

Observation of Inaccuracy: When run, the program jumps quickly through the small downloads (e.g., 20%, 40%) but then gets stuck at 60% completion for a long time due to the large Debian ISO download.

Problems with Initial Progress Bar:

1. **Infrequent Updates:** The progress bar is only updated when a download completes. This fails to provide hope or assurance that the program is still working.
2. **Unequal Weighting:** All downloads are treated the same. The large download causes the bar to sit stuck for a long time at 60%, then suddenly zoom to 100% when it finishes.

Improving Progress Tracking

Updating More Often

The first improvement is to update the bar every time new data is received, not just when a download completes. This means moving the `update_bar` call into the `got_data` function.

In `update_bar`, the total bytes downloaded is displayed to give the user a better idea of what is happening.

To make the byte count cleaner, it is displayed in megabytes (MB) by dividing by 1,000,000:

```
// Inside update_bar
printf("%d MB", s_info->total_bytes / 1000000);
```

The Problem of Unknown Size

When calling `update_bar` from `got_data`, a `percent_done` value must be passed. The major problem is that the program has **no idea how many total bytes to expect**.

Potential Solution (Rejected for Complexity): A head-only request could be performed for each URL to get the download sizes (content length from HTTP headers).

- *Downsides:* This requires extra connections and work, slowing the program down.

Chosen Solution: Make educated guesses and update those guesses as the download proceeds. The predictions will likely be inaccurate, relying only on past data.

Refining the `status_info` Structure for Prediction

Several members are added to `status_info` to support prediction:

```
typedef struct {
    size_t total_bytes;         // Total bytes downloaded overall
    size_t total_expected;      // Total bytes expected for all remaining work
    double expected_bytes_per_url; // Prediction for the average bytes per URL
    size_t current_bytes;       // Bytes in the current download
    int urls_so_far;            // Number of downloads completed
    int total_urls;             // Total number of URLs in the list
} status_info;
```

Prediction using Exponentially Weighted Moving Average (EWMA)

A simple EWMA technique is chosen because it is simple and fast, despite not being the most accurate predictor.

Weight Definition: A weight, sometimes called "alpha," is used:

```
const double PREDICT_WEIGHT = 0.4;
```

`predict_next` **Function:** This function predicts the next value based on the previous prediction and the actual observed value.

Function signature: `double predict_next(double last_prediction, size_t actual)`.

EWMA Calculation:

```
double predict_next(double last_prediction, size_t actual) {
    // Combine the old prediction and the actual result using the weight
    return (last_prediction * (1.0 - PREDICT_WEIGHT)) +
           (actual * PREDICT_WEIGHT);
}
```

- If `PREDICT_WEIGHT` is larger (closer to 1), the prediction relies more heavily on the most recent measurement (new data).
- If `PREDICT_WEIGHT` is lower, the prediction relies more on older predictions.
- If `last_prediction` and `actual` are the same, the prediction does not change.

Updated Logic in `got_data` (Prediction and Estimation)

In `got_data`, the logic handles updating the current progress and estimating the total remaining work.

1. **Update Current Bytes:** The bytes received in the current chunk are added to `current_bytes`:

   ```
   status->current_bytes += bytes;
   ```

2. **Calculate Remaining URLs:**

   ```
   int urls_left = status->total_urls - status->urls_so_far;
   ```

3. **Estimate Current Download Size:** Start the estimate using the average prediction.

   ```
   size_t estimate_current_download = status->expected_bytes_per_url;
   ```

If the actual `current_bytes` already exceeds the expected average, the prediction is known to be wrong. A pessimistic assumption is made that the download is only 75% complete, meaning the total size is estimated to be 1.33 times the current bytes received:

```
if (status->current_bytes > status->expected_bytes_per_url) {
    // Assume we are 3/4 done (pessimistic estimate)
    estimate_current_download = status->current_bytes * 4 / 3;
}
```

4. **Guess Next Prediction:** Use the predictor function based on the potentially updated estimate for the current download:

```
double guess_next_prediction = predict_next(
    status->expected_bytes_per_url,
    estimate_current_download
);
```

5. **Estimate Total Bytes for All Downloads:** This combines the actual downloaded bytes with the estimate for the remaining work.

```
size_t estimated_total = status->total_bytes;

// Add remaining bytes in current download
estimated_total += (estimate_current_download - status->current_bytes);

// Add estimated bytes for future downloads (urls_left minus the current
one)
estimated_total += (urls_left - 1) * guess_next_prediction;
```

6. **Calculate Percentage Done:**

```
int percent_done = (status->total_bytes * 100) / estimated_total;
update_bar(percent_done, status);
```

Final Adjustments in `main`

The status information needs to be initialized before the loop.

Initialization:

- `urls_so_far` set to `0`.
- `total_urls` set to `num_urls`.
- `expected_bytes_per_url` (initial prediction) is set to a pessimistic starting value, e.g., 100 megabytes (100,000,000 bytes).

In-Loop Updates:

1. Before each download, `current_bytes` must be reset to zero.
2. After each download, two values must be updated:
    - Increment `urls_so_far`.
    - Update the global prediction (`expected_bytes_per_url`) using the actual bytes downloaded in the finished URL:

```
s_info.expected_bytes_per_url = predict_next(
    s_info.expected_bytes_per_url,
    s_info.current_bytes
);
```

Conclusion on Progress Bar Accuracy

When running the improved program, the progress bar updates continuously and adjusts itself. It starts slow due to the pessimistic initial prediction but speeds up. During the large download, the percentage may move forward, then "come back a little" as the predictor discovers the file is much larger than estimated, constantly adjusting.

The fundamental difficulty in creating accurate progress bars stems from two key truths:

1. **Unknown Completion Time:** Developers never truly know how long a task will take to complete.
    - In the download example, the uncertainty is the unknown download size.
    - Even if the size is known, external factors like server speed, network traffic, or local computer resource contention (e.g., CPU time used by other tasks) can cause major slowdowns or speedups.
2. **Resource Constraints:** Spending significant time making progress bars perfect is usually not justifiable.
    - The main purpose of a progress bar is to provide **comfort** that the program is still working.
    - Extremely accurate prediction methods often require more processor time or network bandwidth.
    - For most applications, the progress bar becomes an **exercise in good enough**; it must keep moving and try not to be too misleading.

This document provides comprehensive notes based on the supplied source transcript excerpts, presented in Markdown format.

11 Code Review: Highlighting a Student Programming Project

1. Introduction and Project Context

The video focuses on conducting a code review and highlighting a project created by a student named **Tomas**. The project inspiration came from a conversation with Tomas, who is a good friend, a former student, and a supporter of the channel on Patreon.

Code Availability

Typically, all code from the instructor's videos is hosted on Patreon. However, in this specific case, the code belongs to Tomas, and it is **publicly available on GitHub**. A link to the code will be provided in the description.

2. Educational Backstory

Tomas took an online class taught by the instructor last summer. This was the first online class the instructor had ever taught.

Topics Covered in the Course

The three-week course covered various strategies for improving programming skills:

- Strategies for learning to become a better programmer.
- Common mistakes that student programmers make.
- Modular programming.
- Software testing.
- Setting up projects.
- Design patterns.
- A bunch of other stuff.

The instructor notes that a three-week course cannot solve all the world's problems or provide everything necessary to succeed in the field.

Related Course Offering

The original class was a **once-in-a-lifetime chance** and will never be taught again. However, the instructor is currently repackaging and retooling some of that content into a different class.

- **Availability:** This new course is being **pre-released on Thinkific**.
- **Status:** It is still under construction; the instructor is still working out kinks and adding content, so it is not completely finished.
- **Pricing:** To address concerns about the unfinished state, the course is being offered at **half off** while it is being finished. This current price will not last forever.

3. Tomas's Programming Project

The project is an **active project** and may change over time, hopefully getting better.

Project Concept: Scratching Your Own Itch

The project is highlighted as a great example of **"scratching your own itch"**.

It is recommended that when looking for projects to work on, programmers should choose something they care about, something they are going to use, or something that matters to them.

Tomas's specific motivation:

1. He spent a lot of time translating between German and English.
2. He did not want to constantly jump between his terminal and his web browser.
3. **The Solution:** He created a **terminal frontend** for the dictionary web page he uses, which is `dick.cc`.

Value as a Learning Tool

This project holds high learning value because:

- He will actually use it.
- He is building a tool for himself.
- He will test it day by day.
- He will be motivated to fix issues and add features.
- It is less likely to be something he implements and forgets, or, more commonly, starts and never finishes.

4. Key Project Highlights

4.1. Integrated Learning

Tomas used this project to bring together many different concepts he learned both in the course, on the channel, and even a few things the instructor had not covered.

Integrated concepts and skills include:

- Project setup.
- Unit testing.
- Modular design.
- Curl.
- Curses (seen on the channel).
- Regular expressions (seen in the code).

Crucially, the student extended what he had learned, exploring new features and solidifying his understanding, instead of just repeating what he had seen.

4.2. Professional Documentation and Project Management

Tomas is taking the project seriously, treating it like a serious professional or community project.

- **Documentation:** He is documenting his process.
- **Tracking:** He keeps track of the work to date and maintains a to-do list for future needs.
- **Method:** He is utilizing the **README file**, which integrates well with GitHub.

This contrasts with many student programming projects, which are often "really sloppy" or "super sloppy" because students believe no one will see them.

Building Habits: Tomas is learning how to create, document, and manage a project, which helps build good habits that will serve him well in the future. Taking documentation and management seriously also makes it easier to share the project or potentially turn it into something bigger.

5. Code Review Details

5.1. Makefile

The Makefile is solid and pretty straightforward.

Features utilized include:

- Variables.
- Pattern substitution. (These were discussed in the course and on the channel).

Testing Structure:

- He is keeping his **tests separate** from his source files, which is praised as "really great".
- The fact that he is doing testing at all needs to be celebrated.
- Testing is automated, making running tests very easy.

5.2. Modular Design

Tomas uses a strong modular design within the source folder.

- **Contrast:** Many self-driven student projects end up as one massive `.c` or `.cpp` file containing all the code.
- **Benefit:** The modular structure is much nicer, clearly showing the project pieces and how they fit together. Individual modules are clearly named, indicating their function.

General Principle: Regardless of the programming language used, code should be broken down into **small, understandable, testable chunks**.

5.3. Data Center Module (`data_center.h` and `.c`)

`data_center.h` (Header File)

This module manages and stores the translations pulled down from the website.

- **Interface:** The header file contains the interface to the rest of the program—the elements that anyone else is going to work with.
- **Guards:** Includes nice `ifdef` guards.
- **Struct and Typedef:** He defines a `struct` and then uses a `typedef` to give it a better name.

Comment/Suggestion	Detail
Typedef alternative	You can merge the `struct` and `typedef` into one definition by placing the `typedef` before the `struct` and the name down below.
Naming Issues (Struct Members)	Some members are unclear, such as `nr`, `vl`, and `vr`. The purpose of `id` and `is_marked` are guessable.
Recommendation	The instructor suggests making these names a little more readable.

`data_center.c` (Implementation File)

This file also looks pretty good, including a couple of linked lists.

- **Access Control:** The code shows separation into **private variables** and **private methods/functions**. This helps keep track of elements that stay within the module versus those that are exposed outside.
- **Method Grouping:** Grouping methods by public and private is praised, as it makes it easy to identify the public-facing interface.

Comment/Suggestion	Detail
Private Functions Improvement	The private functions are correctly excluded from the header file.
Suggestion: `static`	They could be made `static`, which would strictly place them in the translation unit. This means they could not be called from outside this particular module.

5.4. Security and Buffer Handling

The code uses standard C string functions like `string copy` and `sprintf`.

- **The Problem:** Using these functions without checking string lengths can lead to trouble, potentially allowing a program to be hacked.
- **Current Risk:** In this project, the risk is assessed as low because the student is accepting input from a website (`dick.cc`) that he trusts (assuming he truly trusts it).
- **Potential Failures:** If received words are longer than the defined strings, problems like crashes or buffer overruns could occur.
- **Recommendation:** It is recommended to move to safer functions like `strncpy` or `snprintf`. These functions implement **bounds limitations** on buffers to ensure buffer overruns do not crash the program or open up an exploitable vulnerability.

5.5. Web Client Module (`webclient.c`)

This module handles the interaction with the dictionary website.

- **Networking:** The student is using **curl** for easy networking.
- **Searching:** The student is using **regular expressions (regex)** for searching.
- **Advanced Networking:** The code demonstrates functionality beyond the instructor's previous examples by showing how to perform:
  - `POST` requests.
  - Open sessions.
- **Specific Need:** The ability to post information is necessary because some interactions with the webpage require the posting of a form or similar data.

```
// Example demonstrated in webclient.c (Functionality discussed)
// Open session logic
// Post information logic (e.g., posting a form)
```

The instructor suggests he may do a future video breaking down the individual pieces of this web client implementation.

6. Conclusion and Further Resources

The project is helpful for others looking for ideas on projects, or seeking examples of how someone is teaching themselves how to program.

For individuals who are frustrated, lost, or unsure where to start in programming, the previously mentioned new course is targeted specifically at how one approaches learning to program and computing topics. As noted, it is currently half off while under construction.

12 Checksums: Detecting Message Corruption

1. Introduction and Context

This discussion covers **checksums**: what they are, how they are useful, and how to implement them in programs.

1.1 The Problem of Corruption

The topic is **message corruption**, not political corruption.

When sending a message (which is a series of bits, or ones and zeros, perceived as characters) from one machine to another—usually across a wire or over the air (like Wi-Fi)—there is always a chance the message will get messed up along the way. This means some bits might get corrupted (e.g., a one becomes a zero, or a zero becomes a one).

The key question is: **How does the person who received the message know that the corrupted version is not what was originally sent?**. This is where checksums come in.

1.2 Checksums as a Detection Method

Checksums represent **step one** in a series of ideas designed to help detect problems and corruption, whether accidental or malicious.

A checksum is a value computed based on the bits of a message.

The idea is that the checksum provides:

- A summary.
- A hash-like way to represent that message.
- A method to detect upon receipt whether the message "all makes sense".

2. Setting Up the Programming Example (C)

The demonstration uses C programming, starting with a simple "hello world type program" and a standard `make file`.

For control, the demonstration **does not** involve actually sending Wi-Fi packets over the air. This is done because of time constraints and the need to be able to control the corruption (otherwise, one might try for a long time without getting any corruption).

The basic process involves:

1. Creating an initial message string (a series of characters).
2. Printing the original message.
3. Corrupting the message using a function.
4. Printing the corrupted message.

2.1 Memory Management

The string holding the original message is not allocated on the heap, so it does not need to be freed.

A corrupted message (`c_message`) is created by duplicating the original string. This duplicated string is allocated on the heap and **must be freed** using `free(c_message)`.

3. Demonstrating Message Corruption

A function named `corrupt_message` is used to introduce controlled bit corruption.

3.1 `corrupt_message` Function Details

The function flips a specified number of random bits.

```
char *corrupt_message(const char *message, int num_bits)
```

- **Return Type:** `char *` (character pointer).
- **Inputs:** `const char * message` and `int num_bits` (the number of bits to mess with).

Steps for Corruption:

1. **Duplicate the String:** Duplicate the input `message` using `str_dupe` and store it in `result`. This is done to ensure the memory is writable, as attempting to change the memory where the original string is stored might cause problems.
2. **Loop:** Iterate through the loop based on the `num_bits` specified.
3. **Pick Random Byte:** A random byte (`r_byte`) in the message is picked using the random number generator, modulo the string length.
    - This picks a value from zero to `length of the string - 1`.
    - Code example uses `rand() % strlen(message)`.
4. **Pick Random Bit:** A random bit (`r_bit`) within that chosen byte is picked using the random number generator, modulo 8.
    - This picks a number between zero and seven.
    - Code example uses `rand() % 8`.
5. **Flip the Bit using XOR:** The chosen bit in the chosen byte is flipped using the XOR operator.
    - **Mechanism:** XORing a bit with one flips it (1 XOR 1 = 0; 0 XOR 1 = 1).
    - The value `1` is shifted left by `r_bit` places.
    - When this shifted value is XORed with the existing value (`result[r_byte]`), it flips only the desired bit.

```
// Step 5: Flipping the bit
result[r_byte] ^= (1 << r_bit);
```

Note: The random number generator must be seeded (e.g., using `time`) to produce different forms of corruption upon successive runs.

3.2 Observations on Corruption

- Corrupting one bit often changes the case of a character (lowercase to uppercase, or vice versa). This is an "interesting product of the way that ascii and utf-8 actually work".
- Corrupting many bits (e.g., 10) causes the message to look "totally different".
- Corruption may produce a zero byte, which can terminate the string early.

4. Checksum Implementation Examples

4.1 Checksum 1: Addition-Based Checksum

This is a common way to do checksums.

The function `checksum1` returns a single byte.

```
unsigned char checksum1(const char *message, int length)
```

- **Return Type:** `unsigned character`.
- **Inputs:** `const char * message` (made `const` as a good idea) and `int length`.

Steps for Calculation:

1. Initialize `result` to zero.
2. Loop from 0 up to, but not including, `length`.
3. Add up the values of the bytes in the message to `result`.

```
// Step 3: Summing the bytes
result += message[i];
```

Note on Overflow: If the message is long, this addition will "definitely gonna overflow" (unless all bytes are zero), but the result is still a product/sum of all the bytes being sent.

4.2 Verifying the Checksum

The checksum is computed for the original message and then for the corrupted message. The results are often printed in hex.

Verification Process:

1. The sender sends the original message and its checksum value (e.g., 40).

2. The receiver computes the same value on the received message.
3. If the computed value does not match the sent checksum (e.g., if the received checksum is 7c instead of 40), the receiver knows "there's something wrong".

4.3 Weaknesses of Checksums (Collisions)

Checksums are effective if **one bit** gets wrong.

However, a major weakness is the possibility of **collisions**.

- It is possible to get two bits wrong in such a way that they cancel each other out.
- This results in the sum being exactly the same for both the original and corrupted messages (e.g., both checksums resulting in 40).
- If the program is run long enough, this outcome becomes a probability.

4.4 Checksum 2: XOR-Based Checksum

XOR is another way to combine things for a checksum.

The function `checksum2` uses XOR instead of addition.

Comparison between Addition and XOR:

- The two checksums are not always the same.
- They may or may not get confused (collide) in the same way.
- In one observed example, the XOR checksum ended up with a collision, while the addition checksum did not.

5. Advanced Alternatives

If the issue of collisions is a problem, other, more robust options are available.

These alternatives include:

1. **CRCs** (Cyclic Redundancy Checks).
2. **Cryptographic hashes** (if the concern is malicious modification of messages).

These methods vary in complexity and computational intensiveness.

6. Conclusion

Checksums provide a **really easy and very useful way** to perform an easy first check on data to see if it is valid.

13 Unix File Abstraction: Sockets and Pipes Look Like Files

1. Core Concepts and Overview

1.1 The Unix Philosophy

In a **Unix style OS** (like Linux or Mac OS), a core philosophy is often casually stated as "everything's a file".

More accurately, in a Unix style OS:

- **Everything looks like a file**.
- **Everything behaves like a file**.
- This similarity is **intentional**.

Unix style file systems attempt to make things look as much like files as possible.

1.2 Benefit of File Abstraction

Making different components look like files allows very different things to be handled in a very similar way. This means a programmer can write **one piece of code** that can handle inputs from:

1. **Pipes**.
2. **Sockets**.
3. **Files**.

The goal of the demonstration program is to process data from a file, a pipe, and a socket using the same processing code.

1.3 Prerequisites

This discussion assumes prior knowledge of:

- **Sockets** (understanding their basic function).
- **File I/O** (knowing how to open, read, and write to a file).

2. Program Setup: The `count lines` Application

The example program is called `count lines`. The purpose of this program is to **read data from a bunch of different sources and count the number of lines in the output**.

2.1 Utility Functions

The source code includes several helpful components:

Function	Purpose / Description
`HTTP get`	A simple socket client. It opens a socket connection, connects to a server (using an IP address and port), sends a standard HTTP request, and returns the **socket file descriptor** for the connection, allowing the response to be read. It is almost exactly the same as a previous socket tutorial for writing a web client.
`error and die`	A function used to print out an error message. It acts like `printf` because it has variatic stuff. It is used throughout `HTTP get` to exit if an error occurs.
`print usage`	A function defining the intended way the program should be run.

2.2 Intended Usage Scenarios

The program is designed to be run in three different ways, based on command-line arguments (the source `SRC`):

1. **Standard In (Pipe):**

```
countlines standard in
```

   - This reads whatever is coming in from standard in (from that pipe).

2. **File Input:**

```
countlines file <file name>
```

   - This reads from the specified file name (`ARG V2`).

3. **HTTP Request (Socket):**

```
countlines HTTP <IP address>
```

   - This makes an HTTP request to a web server (`ARG V2` is the IP address) and downloads the output.

The goal is to process the output from all three sources in the **same exact way**, avoiding duplicate code.

3. Implementing Unified Input Processing

3.1 Initial Setup and Variables

The program first performs a quick argument check, requiring two or three arguments.

Variables defined include:

Variable	Type/Definition	Purpose
`SRC`	Character pointer (`char *`), set to `ARG V1`.	Used to make the code more readable.
`input source`	File pointer (`FILE *`), started at `null`.	This variable is intended to be the single source that all subsequent reading code will use.
`buffer`	Character array, size `max line` (defined as 4096).	Used to hold lines read from the source.
`num lines`	Integer, started at `zero`.	Used to count the number of lines read.

3.2 The Common Reading Loop

The code designed to process the input source is identical regardless of where the data originates (assuming `input source` is correctly connected):

```
// Assuming input_source is connected to a file/pipe/socket
while (!feof(input_source)) {
    // Read a line into the buffer
    if (fgets(buffer, max_line, input_source) != NULL) {
        num_lines++;
    }
}
fclose(input_source);
printf("We read %d lines\n", num_lines);
```

This processing loop relies only on standard file I/O functions (`feof`, `fgets`, `fclose`) operating on the `FILE *input source`.

4. Connecting Different Sources to a File Pointer

The main challenge is ensuring that the `input source` (`FILE *`) is correctly set up for each of the three sources.

4.1 Case 1: Standard In (Pipe)

Standard in is the easiest case:

- **Check:** Compare `SRC` (ARG V1) to the string `"standard in"`.
- **Action:** Standard in is already a pipe pointed to by a file pointer.

```
if (strcmp(SRC, "standard in") == 0) {
    input_source = stdin;
}
```

- **Result:** The assignment works because standard in already looks like a file and can be assigned pointer to pointer.

4.2 Case 2: File

Opening a local file is also straightforward:

- **Check:** Compare `SRC` to the string `"file"`.
- **Action:** Use `f open` with the file name (`ARG V2`) in read mode (`"r"`).

```
else if (strcmp(SRC, "file") == 0) {
    input_source = fopen(argv, "r");
}
```

4.3 Case 3: HTTP Request (Socket)

This is where things become interesting because the initial result is not a file pointer.

- **Check:** Compare `SRC` to the string `"HTTP"`.
- **Action (Step 1 - Open Socket):** Call the `HTTP get` function. The server port is defined as 80 (standard HTTP port).

```
int socket_fd;
else if (strcmp(SRC, "HTTP") == 0) {
    socket_fd = http_get(argv, SERVER_PORT);
// ... continue to step 2 below
}
```

File Pointers vs. File Descriptors

The issue arises because:

- `standard in` and `f open` return a `FILE *` (a file pointer).
- A socket, as returned by `HTTP get`, is an **integer** (`int`).

Component	Type	Role
File Pointer (`FILE *`)	A convenience `struct`.	Helps manage buffering and related tasks.
File Descriptor (`int`)	An integer.	What the operating system actually uses to keep track of open files. **Every file pointer contains a file descriptor**.

Using `fdopen`

To convert the socket integer (the file descriptor) into a `FILE *` (file pointer), the function `fdopen` is used.

- `fdopen` is similar to `f open`, but instead of taking a file name, it takes an **open file descriptor**.
- **Action (Step 2 - Conversion):** Pass the `socket_fd` and the read mode (`"r"`) to `fdopen`.

```
// ... continued from step 1
    input_source = fdopen(socket_fd, "r");
}
```

- **Result:** This single extra line of code successfully takes a socket and converts it into a file pointer. Now, all sources—pipe, file, and socket—are represented by the same `FILE *input source` and can be processed identically by the line counting code.

4.4 Error Handling

The program includes checks for invalid input and opening errors.

1. **Invalid Source Check:** If `SRC` does not match `"standard in"`, `"file"`, or `"HTTP"`, an error message is printed, and the program returns `EXIT FAILURE`.

```
else {
    // print error invalid source
    return EXIT_FAILURE;
}
```

2. **Open Failure Check:** After determining the source, if `input source` is still `null` (e.g., a non-existent file name was passed in), an error is printed, and the program returns `EXIT FAILURE`.

```
if (input_source == NULL) {
    // print error not open source
    return EXIT_FAILURE;
}
```

5. Demonstration and Results

The completed program, `count lines`, works and is flexible, handling input from various sources.

5.1 Test Case 1: Standard In

Input is typed directly into the terminal, ending with Control-D.

Input Lines	Result
hello	
my name is Jacob	
and I'm testing	
my program	
(6 lines total)	**Read 6 lines**.

5.2 Test Case 2: Local File

Using the `make file` as input.

Input Source	Result
`make file` (10 lines long)	**Read 10 lines**.

5.3 Test Case 3: Network Server (Socket)

Using an IP address obtained by pinging `google.com`.

Input Source	Result
HTTP request to Google's IP address (Port 80)	**Read 91 lines**.

6. Conclusion

The demonstration confirms that many things in Unix style operating systems look like files. This abstraction is helpful because it allows a short, flexible program to handle inputs from many different sources (sockets, pipes, files) **without having to write source-specific code**.

14 Getting an IP Address from a Host Name in C using `getaddrinfo`

1. Introduction and Context

This document explains how to obtain the IP address associated with a domain name within a C program.

1.1 IP Addresses and Domain Names

- Users are accustomed to working with **domain names** (e.g., `jacobsorber.com`) when referring to destinations on the web.
- Network programming, particularly when dealing with sockets, often requires the **IP address**.
- The goal is to transition from a domain name to an address that can be used in sockets examples.
- While command line tools exist for this lookup, this guide focuses on implementing the lookup directly within a C program.

2. The Programming Task

The task is to produce source code where a user types in a domain name, and the program is able to figure out the corresponding IP address (or addresses). The resulting program is compiled as `get IP`.

3. Initial Program Structure and Argument Handling

The program starts as a simple, empty structure with necessary header files.

3.1 Checking Program Usage

The program first checks if the user ran it correctly by verifying the argument count (`argc`).

```
if (argc != 2) {
    // Error handling block
}
```

If `argc` is not equal to 2, the program prints the usage instructions and returns failure:

```
printf("usage %s hostname or a domain name\n", argv);
return EXIT_FAILURE;
```

- `argv` is used for the name of the program.

3.2 Storing the Hostname

If the input is correct, the hostname (the first argument) is saved for readability:

```
char *hostname = argv;
```

- Example of running the program: `get IP jacobsorber.com`.

4. Setting up `struct addrinfo`

The process utilizes the `struct addrinfo`. This structure is used both to specify search criteria (hints) and to store the results.

4.1 Struct Variables

Two variables of type `struct addrinfo` are required:

1. **`hint`**: This structure is the input; it tells the function performing the lookup what the program is looking for.
2. **`result`**: This is a pointer that will hold the result (what comes back from the function).

5. Using the `getaddrinfo` Function

The core function for performing the lookup is `getaddrinfo`.

5.1 Function Call and Status

The `getaddrinfo` function is called, and its return value is captured in an integer variable, `status`:

```
int status;

status = getaddrinfo(hostname,
                     NULL, // Server name ignored for now
                     &hint,
                     &result);
```

- The second argument (server name) is passed in as `NULL`.
- The fourth argument is the address of the `result` pointer (`&result`).
- Crucially, **the caller does not allocate space for `result`**; this allocation happens internally inside the `getaddrinfo` function.

6. Configuring the `hint` Structure

Before calling `getaddrinfo`, the `hint` structure must be specified.

6.1 Initializing `hint`

First, the memory associated with `hint` is set to zero using `memset`:

```
memset(&hint, 0, sizeof hint);
```

6.2 Specifying Address Family (`ai_family`)

The `ai_family` field specifies the desired address family (`AF`):

- `AF_INET`: Specifies only IPv4 addresses.
- `AF_INET6`: Specifies only IPv6 addresses.
- `AF_UNSPEC`: Specifies an unspecified family, meaning the program will take whatever addresses are given (e.g., all available addresses).

6.3 Specifying Socket Type (`ai_socktype`)

The socket type can also be specified.

- `SOCK_STREAM`: Specifies a stream socket, typically used for TCP connections.
- If this is left unspecified, the function will return all different types.

7. Error Checking

After calling `getaddrinfo`, the `status` variable must be checked.

- Many standard C functions return **zero for success**.
- If `status` is non-zero, an error occurred.

```
if (status) {
    printf("getaddrinfo failed\n"); // Specific error information could also be
printed
    return EXIT_FAILURE;
}
```

If the program reaches past this check, a successful result has been obtained.

8. Handling the Results (The Linked List)

The result returned by `getaddrinfo` is a **linked list**.

- The `result` pointer points to the **head** of this linked list.
- A linked list is used because `getaddrinfo` may find many entries that match the specified hostname.

8.1 Memory Cleanup (`freeaddrinfo`)

Since space was allocated inside `getaddrinfo`, it must be freed. A special function, `freeaddrinfo`, is used to free the entire linked list:

```
freeaddrinfo(result);
```

- **Warning**: Calling `free(result)` would only free the head of the list, leading to memory leaks.

8.2 Iterating the Linked List

To process all addresses, a temporary pointer (`temp`) of type `struct addrinfo *` is created and used to iterate through the list:

```
struct addrinfo *temp = result;

while (temp != NULL) {
    // Process the current node

    // Move to the next record
    temp = temp->ai_next;
}
```

- The loop continues as long as `temp` is not `NULL`.
- The pointer to the next record is stored in the `ai_next` field.

9. Extracting and Printing Address Details

Within the loop, information about each address entry is extracted and printed.

9.1 Printing General Information

Information such as the socket type and address family can be accessed directly from the current node (`temp`):

```
printf("Entry:\n");
printf("\tType: %d\n", temp->ai_socktype); // Socket type (e.g., 1 for
SOCK_STREAM, 2 for datagrams)
printf("\tFamily: %d\n", temp->ai_family); // Address family (e.g., 2 for IPv4)
```

9.2 Converting the Address to a Printable String

The raw IP address is not directly available as a simple string in the `struct addrinfo`. The way the address struct is interpreted depends on the `ai_family`.

9.2.1 Preparation (Address String Buffer)

A character array is needed to store the printable IP address string.

- The size constant `INET6_ADDRSTRLEN` is used to ensure enough space is reserved, as IPv6 addresses produce the longest strings.

```
char address_string[INET6_ADDRSTRLEN];
```

9.2.2 Identifying Address Location

A `void *` pointer, `address`, is used to point to the actual address bytes. Logic is required to correctly cast the generic `ai_addr` struct based on the family.

1. **If IPv4 (`AF_INET`)**: The `ai_addr` must be cast to a pointer to `struct sockaddr_in *`. The IPv4 bytes are located in the `sin_addr` field.

   ```
   void *address; // Placeholder for address bytes

   if (temp->ai_family == AF_INET) {
       // Cast to IPv4 struct pointer
       address = &((struct sockaddr_in *)temp->ai_addr)->sin_addr;
   }
   ```

2. **If IPv6 (or other)**: The `ai_addr` must be cast to a pointer to `struct sockaddr_in6 *`. The IPv6 bytes are located in the `sin6_addr` field.

   ```
   else {
       // Cast to IPv6 struct pointer
       address = &((struct sockaddr_in6 *)temp->ai_addr)->sin6_addr;
   }
   ```

9.2.3 Using `inet_ntop`

The socket library provides the function `inet_ntop` (Network to Printable) to convert the network representation of the address bytes into a printable string.

```
inet_ntop(temp->ai_family, // 1. Address Family (V4 or V6)
          address,         // 2. Pointer to address bytes
          address_string,  // 3. Destination buffer (where the string is placed)
          sizeof address_string // 4. Size of destination buffer
          );
```

- **Note on `sizeof`**: If `address_string` were a pointer allocated on the heap instead of an array, `sizeof` would return the pointer size (e.g., 8 bytes), and the size constant (`INET6_ADDRSTRLEN`) should be used

instead.

9.2.4 Printing the Address String

After `inet_ntop` is called, `address_string` contains the human-readable IP address, which is then printed:

```
printf("\tAddress: %s\n", address_string);
```

10. Example Results and Observations

10.1 `jacobsorber.com` Example (Unspecified `hint`)

When `ai_family` is set to `AF_UNSPEC` and `ai_socktype` is unspecified, running the program for `jacobsorber.com` yielded two entries:

Entry	Type (`ai_socktype`)	Family (`ai_family`)	Address Type
1	2 (Datagram)	2 (IPv4)	IPv4
2	1 (Stream)	2 (IPv4)	IPv4

- **Family 2** corresponds to IPv4.
- **Type 1** corresponds to a Stream Socket (TCP).
- **Type 2** corresponds to a Datagram Socket (UDP).

10.2 Specifying Socket Type

- If the `hint` requested `SOCK_STREAM` (Type 1), only the address for Type 1 was returned.
- If the `hint` requested Datagram Sockets (Type 2), only the address for Type 2 was returned.

10.3 `google.com` Example (Unspecified `hint`)

When the `ai_family` was unspecified, running the program for `google.com` yielded four entries:

- Two IPv6 addresses (one for datagrams, one for streams).
- Two IPv4 addresses (one for datagrams, one for streams).
- Observation: Some hostnames are associated with both IPv4 and IPv6 addresses.

11. Conclusion

This method enables a domain or hostname lookup to retrieve the corresponding IP address, which makes the client more user-friendly by allowing users to input names instead of requiring them to look up IP addresses. The addresses can be obtained either in network format (bytes) or in string (human-readable) format.

11.1 Related Topics Suggested for Further Study

- DNS (Domain Name System) resolution, including writing custom code that interacts with DNS.
- UDP (User Datagram Protocol) and Datagram sockets.

15 The Two Main Types of Network Socket

1. Defining a Network Socket

A socket is an important concept in network programming.

1.1 What a Socket Is

- A socket is **not really a data structure**.
- It is more accurately described as an **abstraction**.
- It is an **interface** that the operating system gives you.
- This interface allows a process to perform network communications in a somewhat standard way.

1.2 Socket Functionality and Interface

- A socket works a lot like a file.
- When a program requests a new socket, the system gives it a **socket descriptor**.
- The socket descriptor is an integer that represents that specific socket.
- The use of this integer descriptor feels a lot like working with files.
- However, rather than representing a file, that number represents a **network endpoint**.
- This endpoint can be used to send data to another process or to receive data from some other process.
- The other process may be on the same machine or on the other side of the world.
- Sockets are probably the closest thing available to a **standard network interface** when working in C, though they are also seen in other languages.

2. The Two Main Types of Sockets

You will typically run into two different main kinds of sockets:

1. **Stream Sockets**
2. **Datagram Sockets**

> **Note on other types:** Unix domain sockets or raw sockets exist, but this discussion focuses only on the main types and the interfaces they provide.

3. Stream Sockets

Stream sockets are commonly used in network programming.

3.1 Communication Characteristics

- A stream socket makes communication look like a **stream of data**, specifically a stream of bytes.
- Data can be put into the stream or pulled out of the stream.
- Crucially, when data is pulled out of the stream, the bytes will be in the **same order that they went in**.
- Although a large block of data sent across the network is broken up arbitrarily into small chunks called packets, when working with stream sockets, the application does not see this chunking.
- The application simply sees a nice, **continuous stream of data** where data goes in and data comes out.

3.2 Reliability (Reliable Sockets)

Stream sockets are also referred to as **reliable sockets**.

- This designation does not mean that the connection itself cannot be broken. Connections between a client and a server can still be disconnected, for example, if:
  - The Wi-Fi goes down.
  - A network cable is unplugged.
  - The network becomes too congested, preventing packets from getting through.
- What "reliable" means is that messy network issues are handled magically by the **transport layer protocol**, which is usually **TCP** (Transmission Control Protocol).
- **Packet Handling by TCP:**
  - If some packets get lost during transmission, the lost packet is going to be **resent** (retransmitted).
  - If packets are received out of order (e.g., Packet A was sent before B, but B is received first), the packets will be **put back in the right order** before they are delivered to the application.
- The stream socket connection handles all this messy stuff under the hood, and the application does not see any of it.

4. Datagram Sockets

Datagram sockets are used when the cost of reliability is unnecessary or unwanted.

4.1 Communication Characteristics

- With datagram sockets, the application is sending **individual packets** or **datagrams**.
- Datagram sockets do **not** provide the reliability features of stream sockets.

4.2 Lack of Reliability Features

Datagram sockets offer no built-in reliability features:

- No reordering.
- No retransmissions.
- No congestion control.

4.3 Data Fate and Network Effort

- Packets or datagrams sent using this method could get lost on the way.
- They might also be received out of order.
- If packets are lost, they are lost ("life's tough").
- The network still makes a **best effort attempt** to get the data through.

4.4 Congestion Control

- There is no congestion control with datagram sockets.
- The system will not throttle back packets if the user is sending too many or if there are too many users talking on the network.
- The decision on how much data and how fast to send is left entirely up to the application.
- **Risk:** If too many users attempt to send too much data, they can clog up the network and cause various problems.

- **Best Practice:** When using datagram sockets, users should act responsibly and make sensible decisions to avoid disrupting the network for others and their own application.
- **Advantage:** If used responsibly ("play nice"), datagram sockets can sometimes achieve higher data throughput than when using TCP and stream sockets.
- The protocol used by datagram sockets is **UDP** (User Datagram Protocol).

4.5 Example Use Case: Real-time Audio/Video (Video Chat)

Using datagram sockets is beneficial in scenarios where speed and low latency are prioritized over guaranteed delivery, such as video chat:

Feature	Stream Socket (TCP)	Datagram Socket (UDP)
Data Loss Event	If an audio/video packet gets lost, TCP tries to **resend** the packet.	If a small chunk of data is lost, the application can skip the resend.
Resulting Experience	Resending adds lag, making everything wait for the lost data. This results in a very noticeable, annoying glitch in the audio/video.	Skipping the data means one frame might be slightly messed up (e.g., 1/30th or 1/24th of a second), which is **hardly noticeable**.
Underlying Principle	The reliability features of TCP come at a cost.	The application decides it does not care about ordering or loss in this context.

16 Super Depth : How to Send and Receive UDP Packets (in C)

I. Introduction to UDP Sockets

This content focuses on how to send and receive **UDP packets** or **datagrams** in C.

Key Concepts

- **Socket Types:** Network programming utilizes two main types of sockets: stream sockets and datagram sockets.
- **Protocol:** This demonstration uses the **UDP protocol** (User Datagram Protocol).
- **Source Code:** The source code discussed in the video is available through Patreon.

II. Part 1: Implementing the UDP Sender Program

The sender program is designed to perform one action: send a packet to a specified IP address and port.

A. Program Structure and Arguments

The program utilizes starter code including necessary header files and a `main` function.

The program requires the following command-line arguments (after the program name):

1. **IP address:** The destination address to send the packet to.

2. **Port:** The port number to send to.
3. **Message:** The message content to be sent.

B. Argument Handling

Variables are created to simplify referring to the input arguments:

Variable	Type	Assignment (Example)	Notes
`peer_IP`	`const char *`	`argv`	Represents the peer's IP address (as a string).
`port`	`int`	`atoi(argv)`	The port number, converted to an integer. (Error checking for integer conversion is omitted in this example).
`message`	`const char *`	`argv`	The message content.

Important Note on IP Format: The IP address (`peer_IP`) is currently in **human readable format** (printable format). Socket functions require a different format, necessitating conversion.

C. Address Preparation and Conversion

The IP address must be converted from printable format to binary format.

1. Declaring the Address Structure

A structure of type `sockaddr_in` is declared to hold the peer's address:

```
struct sockaddr_in peer_Adder;
```

2. Initializing `peer_Adder`

Specific fields in the structure must be set:

- **Address Family:** Set to use IPv4 addresses.

  ```
  peer_Adder.sin_family = AF_INET; // AF_INET = Address Family Internet
  ```

  (For IPv6, `AF_INET6` would be used).
- **Port:** The port number must be converted to network byte order.

  ```
  peer_Adder.sin_port = htons(port);
  ```

  - `htons` (Host to Network Short) converts the native host byte order to the network byte order.

- This is crucial because not all machines use the same byte order.
- `Short` is used because ports are typically two bytes.

3. Converting the IP Address String

The printable IP address string is converted to binary format using `inet_pton`.

- **`inet_pton`** stands for Internet Printable To Network format.

```
// Arguments: AF_INET (family), peer_IP (string), &peer_Adder.sin_addr
(destination)
inet_pton(AF_INET, peer_IP, &peer_Adder.sin_addr);
```

Error Checking for Conversion: If `inet_pton` returns a value less than or equal to zero, an error has occurred ("something wrong with the IP address"), and the program returns `EXIT_FAILURE`.

D. Socket Creation

A socket is created using the `socket` function:

```
int udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
```

Argument	Value	Purpose
Address Family	`AF_INET`	Specifies the protocol family (IPv4).
Socket Type	`SOCK_DGRAM`	Specifies that this is a **datagram socket**. (Contrast with `SOCK_STREAM` used for stream sockets).
Protocol Number	`0`	In this context, often specifies that the default or only available protocol (UDP) should be used.

Error Checking: If `udp_socket` is less than zero, an error is printed ("couldn't create the socket"), and the program returns `EXIT_FAILURE`.

E. Sending the Packet (`sendto`)

Since UDP datagrams do not use connections, the `sendto` function is used, which requires specifying the destination address for each individual packet.

```
sendto(
    udp_socket,
    message,
    strlen(message) + 1, // Message length (+1 for null terminator)
    0,
    (struct sockaddr *)&peer_Adder,
```

```
    sizeof(peer_Adder)
);
```

Function Parameters Details:

1. **Socket:** The `udp_socket` created.
2. **Payload:** The `message` content.
3. **Length:** The length of the message string plus one (`+1`) to ensure the **null Terminator** is sent.
4. **Flags:** Set to `0` (the default).
5. **Address:** The address of `peer_Adder`, which is the destination address structure.
    - **Casting:** The address is cast to a generic `(struct sockaddr *)` pointer to prevent compilation warnings. The generic `sockaddr` structure can be used interchangeably for many different protocols.
6. **Size:** The size of the `peer_Adder` structure.

Error Checking for Sending: If `sendto` returns a negative value, `perror` is called ("failed to send message"), the `udp_socket` is closed, and `EXIT_FAILURE` is returned.

F. Success and Cleanup

If the packet is sent successfully, a status message is printed confirming the message, destination IP, and destination port. The `udp_socket` is then closed to release the network endpoint, and the program exits successfully.

G. Sender Example

The program is compiled and run using the loopback address and a selected port (9800):

```
./UDP_send 127.0.0.1 9800 "Here is a UDP datagram"
```

The output confirms the program ran correctly.

III. Part 2: Implementing the UDP Receiver Program

The receiver program (`UDP_receive.c`) is designed to listen for and receive a UDP packet.

A. Helper Function for Error Checking

A custom helper function named `check` is used to handle repetitive error checking.

- Most socket functions return **-1** (often referred to as `socket_error`) upon failure.
- The `check` function takes the function result, checks if it is equal to `socket_error` (`-1`), prints a message using `perror`, and exits if an error occurred.

B. Program Arguments (Receiver)

The receiver program requires only one argument:

1. **Port Number:** The port the program will listen on.

The IP address is not specified because the program listens on the local address. The input port is converted to an integer and stored in `my_port`.

C. Variable Declarations

The receiver declares variables for the socket, addresses, and data storage:

Variable	Purpose
`udp_rx_socket`	The socket used for receiving (RX is shorthand for receive).
`peer_Adder`	`struct sockaddr_in` to store the address of the person sending the data.
`my_adder`	`struct sockaddr_in` to specify the local address and port the program will listen on.
`buffer`	An array to hold the incoming packet data.
`BUFFER_SIZE`	Defined as `1024` bytes.

D. Initializing `my_adder` (Local Listen Address)

The structure specifying the receiving address must be initialized:

1. **Family:** `my_adder.sin_family = AF_INET`.
2. **IP Address:** Set to listen on all available local addresses.

   ```
   my_adder.sin_addr.s_addr = INADDR_ANY;
   ```

   - `INADDR_ANY` specifies that the machine will list on **any IP address** it has.
3. **Port:** The port is set after converting it to network byte order.

   ```
   my_adder.sin_port = htons(my_port);
   ```

E. Socket Creation (Receiver)

Socket creation is identical to the sender process:

```
udp_rx_socket = socket(AF_INET, SOCK_DGRAM, 0);
```

Error checking is performed (e.g., if result ≤ 0, exit with "fail to create socket").

F. Binding the Socket (`bind`)

The socket must be **bound** to the specific local address and port (`my_adder`) the program intends to receive on.

```
int result = bind(
    udp_rx_socket,
    (struct sockaddr *)&my_adder,
    sizeof(my_adder)
);
```

`bind` Arguments:

1. **Socket:** `udp_rx_socket`.
2. **Address:** The address of `my_adder`, cast to a `struct sockaddr *` pointer.
3. **Address Size:** `sizeof(my_adder)`.

Error checking uses the helper function: `check(result, "could not bind socket to address")`.

G. Receiving the Packet (`receivefrom`)

The `receivefrom` function is used to wait for and receive the datagram. This function will **block** (wait) until a packet arrives.

Preparation for `receivefrom`: The function needs a pointer to the address length.

```
socklen_t address_length = sizeof(peer_Adder);
```

- `address_length` is initialized to the expected size of `peer_Adder`.
- The type `socklen_t` is used to avoid casting the variable, although an `int` could technically be used.

Calling `receivefrom`:

```
int bytes_received = receivefrom(
    udp_rx_socket,
    buffer,
    BUFFER_SIZE,
    0,
    (struct sockaddr *)&peer_Adder,
    &address_length
);
```

`receivefrom` Arguments:

1. **Socket:** `udp_rx_socket`.
2. **Buffer:** `buffer` (stores the incoming data).
3. **Buffer Size:** `BUFFER_SIZE` (maximum capacity).
4. **Flags:** `0` (default, no flags specified).
5. **Sender Address:** The address of `peer_Adder`, cast to `struct sockaddr *`. This structure will be **filled** by the function with the originating address of the packet.

6. **Address Length Pointer:** The address of `address_length`. This allows the function to potentially change the value of the length variable.

The return value (`bytes_received`) indicates the number of bytes received. This result is error-checked using `check(bytes_received, "sorry receive from failed")`.

H. Output and Cleanup

Upon successful receipt, a status message is printed:

```
Received a packet from [IP address] and [port], message = [received string]
```

To display the IP address and port, conversions are needed because they are received in network byte order:

1. **IP Address Conversion:** `inet_ntoa` is used.
   - **`inet_ntoa`** (Network To Printable address) converts the network representation of the address (`peer_Adder.sin_addr`) into a printable string.
2. **Port Conversion:** `ntohs` is used.
   - **`ntohs`** (Network To Host Short) converts the port from network byte order (`peer_Adder.sin_port`) back to the host byte order.
3. **Message:** The `buffer` content is printed, assuming the received data is a well-formatted printable string.

Finally, the socket is closed.

I. Receiver Testing Example

1. The receiver program is run and starts blocking, waiting for a packet.

   ```
   ./UDP_receive 9800
   ```

2. The sender program is run in a separate terminal targeting the receiver:

   ```
   ./UDP_send 127.0.0.1 9800 "hello this is a message sent over UDP"
   ```

3. The receiver successfully processes the packet and outputs the result:
   - It confirms receipt from the local machine (127.0.0.1).
   - It shows the randomly assigned port that the sender used.
   - It displays the message: "hello this is a message sent over UDP".

IV. Future Topics

The source material suggests several related topics that could be explored further:

- Network communication dynamics and how UDP works.

- Setting socket **timeouts**.
- Exploring different **flags** used in socket functions.
- How to use a **packet sniffer** like Wireshark.
- Programmatic ways to look up **protocol numbers**.

17 Super Depth: Using `connect` with UDP Sockets

1. Overview and Context

This content addresses questions related to the previous video on sending and receiving UDP packets in C.

The previous video demonstrated two C programs:

1. `UDP send. C`: Designed to send a single, particular packet.
2. `UDP receive`: Designed to receive that packet.

Two specific questions raised in the comments are addressed here.

2. Question 1: What Happens to Unreceived UDP Packets?

2.1 Test Example

An example test involved running `UDP send`. This required specifying three components:

1. An **IP address** (e.g., the loopback address).
2. A **port** (e.g., 9800).
3. A **message** (e.g., "hello world").

The speaker demonstrated sending multiple packets (e.g., "hello world 2," "hello world 3") while no process was listening. When attempting to run `UDP receive 9800`, nothing was received if a packet was not available. The question asked where these unreceived packets went.

2.2 The Nature of UDP

The clarification regarding these lost packets is based on the characteristics of UDP:

- UDP operates on a **best effort** basis.
- There is **no reliability** in UDP.
- If a packet is sent and no process is receiving it, the packet is just going to get dropped.
- The packet is simply going to get lost.

3. Question 2: Can `connect` be Used with UDP Sockets?

3.1 Initial Approach: Using `sendto`

In the original `UDP send` program, the `connect` function was **not** used.

Instead, the socket was created, and then the **`sendto`** function was used.

The `sendto` function differs from the standard `send` function because it requires specifying the destination address.

The parameters for `sendto` include:

- The socket.
- The message.
- The length of the message.
- The **address that you are sending to**.

Specifying the address to send to makes `sendto` appropriate for standard UDP sockets.

3.2 The Alternate Approach: Using `connect`

Although the speaker previously mentioned not using `connect` with UDP, it is actually possible to use it.

An alternate version of the program, which is otherwise exactly the same as the last program, demonstrates this use.

In this alternate version, the program calls **`connect`**.

The `connect` call involves passing in:

1. The UDP socket.
2. The peer address.

This setup looks very much like what would be done with a **stream socket** (like TCP). After calling `connect`, the program can then use the standard **`send`** function.

3.3 The Role of `connect` in UDP

Using `connect` in UDP might be confusing because UDP sockets deal with datagrams and have **no concept of a connection**.

When used with UDP sockets, the `connect` function operates differently than it does with stream sockets:

- **Action:** This `connect` call is **only saving the address**.
- **Purpose:** It is telling the system, "hey, this is the address I'm going to use".
- **Process:** There is **no connection process** like there is with a stream socket using TCP.
- **Meaning:** Using `connect` with UDP does **not mean the same thing** as it does with TCP.

3.4 Implications for Code

Using `connect` with UDP offers certain conveniences:

- It makes the code a little bit **smaller**.
- The address does **not have to be specified every time** the program sends a packet.
- The system will remember that address.
- This can be convenient for some programs.

This situation highlights that in programming, there are often **multiple ways to solve the same problem**.

3.5 Design Question: Is This a Good Idea?

Using `connect` with UDP raises an interesting design question.

Some people may prefer this method because the `send` call is **more compact** than a `sendto` call.

However, the speaker expresses a personal dislike for this approach.

Reasons why using `connect` with UDP might be confusing:

1. **Mental Shift:** If a programmer reads code containing `connect` and `send`, their brain immediately starts to think about **connections** and **stream sockets**.
2. **Clarity:** While this method provides a parallel structure with stream sockets, it is considered **more likely to be confusing** than helpful.

The speaker points this out so that viewers know that using `connect` with UDP is possible if they encounter it. It is also noted that this method might be helpful for a future project.

18 Socket Timeout Implementation using `setsockopt` in C

Introduction and Context

This content focuses on setting up **timeouts for sockets**. The discussion is related to **UDP and datagram sockets**. Previous material provided a simple example demonstrating how to **send a packet and how to receive it** using a datagram socket and the UDP protocol.

The current goal is to add a specific feature: a **timeout**, which is **really useful** in many programs. Timeouts are necessary when programmers do not want to **wait forever** on a call, such as a receive call.

The example utilizes simple UDP programs from a recent tutorial, specifically focusing on the **receive program**.

The Default Problem: Indefinite Blocking

In the receive program, there is a `receive from` **call**. This function is designed to **wait for a packet to come in**.

By default, the `receive from` call is **just going to wait forever** until a packet arrives. While this might be acceptable sometimes, programmers often want to **limit how long they will wait**. The ultimate goal is to **give up** waiting after a certain time (e.g., a few seconds or minutes) and allow the **current thread to do something else**.

Implementing the Timeout using `setsockopt`

The objective is to add a timeout to the receive program, making it wait for a specific duration and then **have it give up**.

Location for Timeout Setup

The code to set up the timeout is inserted in the receive program **after the** `bind` **call** but **before the** `receive from` **call**.

The `setsockopt` Function

To set up the timeout, the **`set sock opt` function** is called.

- **Purpose:** `setsockopt` is a **general function** that allows setting different socket options.
- **Options:** The man page for `setsockopt` lists various options that can be set.
- **Timeout Options:** The relevant options for timeouts are the **Receive Timeout** and the **Send Timeout**. The process for setting the send timeout would be **exactly the same** as the receive timeout.

Step 1: Defining the Timeout Duration

The required timeout duration must be specified using a **struct**.

- **Structure Name:** The structure used to represent time values or durations is `struct timeval`.
- **Structure Members:** This struct has **two members**:
    1. `tv_sec`: Represents the **number of seconds**.
    2. `tv_usec`: Represents the **number of microseconds** (used for very small sleeps).

Example Structure Definition and Initialization

For the example, a timeout of 5 seconds is chosen.

```
struct timeVal timeout;

// Set the number of seconds
timeout.tv_sec = 5;

// Set the number of microseconds
timeout.tv_usec = 0;
```

This configuration specifies a timeout of **5 seconds and no microseconds**.

Step 2: Calling `setsockopt`

The `setsockopt` function is called using the following arguments:

1. **Socket:** The first argument is the socket being worked with, which is the **UDP RX socket** (the receiving socket).
2. **Level:** The second argument specifies the protocol level. For this task, the **socket level (`SOL_SOCKET`)** is used.
3. **Option Name:** The third argument is the option being set, which is the **receive timeout option (`SO_RCVTIMEO`)**.
4. **Pointer to Struct:** The fourth argument is a **pointer to the struct** (`&timeout`) that specifies the timeout duration.
5. **Size:** The final argument specifies the **size of that thing** (`sizeof(timeout)`).

Example Call to `setsockopt`

```
// Assume UDP_RX_socket is defined and active
// Option name for receive timeout is typically SO_RCVTIMEO
// Level is typically SOL_SOCKET
setsockopt(UDP_RX_socket, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout));
```

Error Handling for `setsockopt`

The function call to `setsockopt` **could return errors** if a mistake is made. A `check` function is typically used to handle potential errors.

- **Error Message Example: "setting timeout failed"**.

Testing and Initial Result

After compiling the code, running the program requires specifying a port (e.g., 9800).

- The program will **wait**.
- After the specified duration (5 seconds), the `receive from` function **returns an error**.
- The error returned indicates that the **resource is temporarily unavailable**.
- This demonstrates that the timeout has been successfully created, and the **behavior of `receive from` has changed**.

Differentiating Timeout Errors

When setting a timeout, it is often desirable to **treat timeouts slightly differently** than other socket errors (e.g., network failure or connection broken). The initial error checking detects **any sort of error** (anything that is not a packet arriving).

Checking for a Timeout

To differentiate a specific timeout from other errors, two conditions must be checked:

1. The result of the receive operation (`bytes received`) happens to equal **`socket error`**.
2. The **Global variable `errno` is set to `EWOULDBLOCK`**.

Definition of `EWOULDBLOCK`

The value **`EWOULDBLOCK`** specifically signifies that if the timeout had not occurred, the call **would still block**. However, because the timeout is set, the system gives up, resulting in this specific error code.

Revised Error Handling Logic

This logic allows the program to differentiate between a timeout and a normal socket error.

```
// Check if the operation resulted in a socket error
if (bytes_received == socket_error) {
    // Check if the specific error is EWOULDBLOCK (indicating a timeout)
    if (errno == EWOULDBLOCK) {
        // Timeout specific action
```

```
            // Example: Print "Our socket timed out"
            // Example Correction: return exit_failure; to prevent further processing
        } else {
            // Normal error checking (e.g., network down, connection broken)
        }
    }
```

If the logic is implemented correctly (e.g., by returning `exit_failure` in the timeout case), the desired behavior is achieved: the program will print **"our request timed out"** and **not say "we received a packet"**.

This procedure demonstrates how to use the **`set sock opt` function** to set a socket timeout, choosing whatever duration is desired. This is a useful technique in network programming examples.

End-of-File

The god-stack repository, authored by Kintsugi-Programmer, is less a comprehensive resource and more an Artifact of Continuous Research and Deep Inquiry into Computer Science and Software Engineering. It serves as a transparent ledger of the author's relentless pursuit of mastery, from the foundational algorithms to modern full-stack implementation.

> Made with ♥ Kintsugi-Programmer