Client Server Socket Example

Marcus Birkenkrahe (pledged)

April 18, 2025

Contents

1 Objectives

The following topics are covered:

- ☒ Understanding the importance of sockets in network programming
- ☒ Understanding the client socket workflow
- ☒ Coding a TCP client socket
- ☒ Understanding the server socket workflow
- ☒ Coding a TCP server socket
- ☒ Connecting client and server over TCP
- ☐ Use of `make` to build applications
- ☐ TCP vs. UDP program flow
- ☐ Header files for Windows and Unix-based systems
- ☐ Building real world program

2 Why sockets are important in network programming

- Sockets are the low level endpoint used for processing information across a network.
- Protocols like HTTP and FTP rely on sockets.
- Sockets are bidirectional so there isn't anything different about a "client" socket (on the client side) vs. a "server" socket on the server side.

3 Client Socket Workflow

On the client side, three function calls are involved: we need to open or create a `socket`, `connect` to a remote IP address and a port, and when the connection is accepted, we can receive data with `recv`:

```
+---------+
| socket()|
+---------+
     |
     v
+----------+
| connect()|
+----------+
     |
     v
+---------+
| recv()  |
+---------+
```

- The result of this call is stored in an integer `network_socket`, and this is the descriptor that we use to refer to the socket later on.
- We're going to use IP addresses (`AF_INET`) and TCP (`SOCK_STREAM`) as our protocol.
- Once the socket is created, we connected it to a remote address.
- If the connection is successful, we get a success signal back.
- Once we've received data, we can write it into a string and use it.
- The following function prototypes are involved:

```
<<network header files>>

  // IPv4 or IPv6, TCP or UDP, protocol type)
int socket(int domain, int type, int protocol);

// descriptor, ptr to destination address, length of address
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

// descriptor, ptr to buffer, max no. of bytes, msg flags)
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

4 Create TCP client socket

- Include standard header files:

```
#include <stdio.h>
#include <stdlib.h>
```

- Include several special header files

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
```

- Main program:

```
<<standard header files>>
<<network header files>>

#define PORT 9002
int main(void)
{
  // Redirect stderr to stdout
  dup2(1,2);

  // create a socket
  int network_socket;
  // Use IPv4, TCP, default protocol
  network_socket = socket(AF_INET,SOCK_STREAM,0);

  // specify an address for the socket
  struct sockaddr_in server_address;
  server_address.sin_family = AF_INET; // IPv4
  server_address.sin_port = htons(PORT); // Convert port number
  server_address.sin_addr.s_addr = INADDR_ANY; // Use 0.0.0.0

  // connect to socket
  int connection_status =  // return 0 if ok, or -1
    connect(network_socket,
            (struct sockaddr *) &server_address,
            sizeof(server_address));
  // check for error with connection
  if (connection_status == -1) {
    perror("Connection error");
```

```
  }

  // receive some data from the server
  char server_response[250]; // empty string to hold data
  recv(network_socket,&server_response,sizeof(server_response),0);

  // print the data that we get back
  if (connection_status == 0) {
    printf("The server sent the data: %s",server_response);
  }

  // close the socket - unistd.h
  close(network_socket);
  return 0;
}
```

```
Connection error: Connection refused
```

- To test, run this and you should get the output:

```
Connection error: Connection refused.
```

5 Server Socket Workflow

On the server side, we also need to create a `socket` but then we need to `bind` that socket to an IP address and port where it can `listen` for connections, `accept` a connection, and either `send` or `recv` data to those sockets it has connected to:

```
+---------+
| socket()|
+---------+
     |
     v
+----------+
| bind()   |
+----------+
     |
     v
```

```
+---------+
| listen()|
+---------+
     |
     v
+---------+
| accept()|
+---------+
     |
     v
+---------+
| send()  |
+---------+
```

- The following additional function prototypes are involved:

```
<<network header files>>

  // descriptor, ptr to address, size of address
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

// descriptor, max no of pending connections that can be queried
int listen(int sockfd, int backlog);

// descriptor, ptr to storage for address, length of address
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

// descriptor, ptr to the data to send, no of bytes to send, msg flags
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

6 Create TCP server socket

- We can reuse the header files from the client side program:

```
<<standard header files>>
<<network header files>>

#define PORT 9002
#define CONN 5
int main(void)
```

```
{
  // text sent to clients
  char server_message[250] = "You have reached the server!\n\n";

  // create socket
  int server_socket;
  server_socket = socket(AF_INET,SOCK_STREAM,0);

  // define server address
  struct sockaddr_in server_address;
  server_address.sin_family = AF_INET;
  server_address.sin_port = htons(PORT);
  server_address.sin_addr.s_addr = INADDR_ANY;

  // bind the socket to our specified IP and port
  bind(server_socket,
       (struct sockaddr*) &server_address,
       sizeof(server_address));

  // listen to the connection (max CONN)
  listen(server_socket, CONN);

  // accept connection with client
  int client_socket;
  client_socket = accept(server_socket,NULL,NULL); // local connection

  // send data (server message) to client
  send(client_socket, server_message, sizeof(server_message), 0);

  // close the socket
  close(server_socket);

  return 0;
}
```

7 Connecting client and server over TCP

- **Setup:**
  1. Tangle the client source code (`tcp_client.c`)

2. Tangle the server source code (`tcp_server.c`)
3. Make a directory `./TCPclient` with `mkdir -v`
4. Make a directory `./TCPserver` with `mkdir -v`
5. Move client source code into `./TCPclient` with `mv -v`
6. Move server source code into `./TCPserver` with `mv -v`
7. Open **two** command-line windows and put them on top of one another

- **Demo:**

  1. In both windows, run `ll` to see the files.
  2. In both windows, run `make` to build the machine code.
  3. In both windows, run `ll` again to see the machine code.
  4. In `./TCPclient` run `tcp_client` to get the error message.
  5. In `./TCPserver` run `tcp_server &` to start the server.
  6. In `./TCPclient` run `tcp_client` again to get the server message.

- Why does `tcp_server` end after the client runs?

  > This TCP server is a one-shot server: after calling `accept` and sending the message, the server closes the socket and returns - the server process exits.

- Home assignment:

  > To keep the server open and handle multiple client requests (sequentially), you can wrap the `accept` and `send` part in an infinite loop, and add a loggint `printf` message to monitor activity.

8 Byte order - Host or Network?

- Hexadecimal is a human-friendly shorthand for binary. Memory addresses, file contents, network packets, color codes etc.

- Example:

```
int i = 1;  // initialize integer variable i with the value 1
int *p = &i;  // pointer to the address of i
printf("The hex address of i = %d is %p.\n",i,&p);
```

```
The hex address of i = 1 is 0x7ffd9e83b3c0.
```

- Hex to decimal can be computed easily on the shell with `bc`:

```
echo "ibase=16; B34F" | bc
echo "obase=2;ibase=16; B34F" | bc


45903
1011001101001111
```

- How does the hexadecimal -> decimal result come about?

Pos	Hex	Decimal	16^{Pos}	Contribution
3	B	11	4096	$11 \times 4096 = 45056$
2	3	3	256	$3 \times 256 = 768$
1	4	4	16	$4 \times 16 = 64$
0	F	15	1	$15 \times 1 = 15$
				Total = **45903**

```
Decimal:  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
Hex:      0  1  2  3  4  5  6  7  8  9   A  B  C  D  E  F
```

- Two-byte hexadecimal numbers like `B34F` (two words), are stored in sequential bytes - B3 followed by 4F. This number, stored with the "Big End" (B3) first, is called "Big-Endian".

- Intel- or Intel-compatible computers store the bytes reversed: So in memory, B34F is stored as the sequential bytes 4F followed by B3. This is called "Little-Endian" (little end first).

- "Big-Endian" is also called "Network Byte Order".

- Your computer stores numbers in "Host Byte Order". If the CPU's an Intel 80x86, Host Byte Order is Little-Endian. If it's a PowerPC..who knows.

- So you need to make sure two- and four-byte numbers are in Network Byte Order. This is what `htons` does for `short` (two bytes - used for port numbers, TCP headers, checksums) and `htonl` for `long` (four bytes - used for IPv4 addresses, TCP SEQ and ACK numbers, and timestamps).

- You can use different combinations:

Function	Description
`htons`	host to network short
`htonl`	host to network long
`ntohs`	network to host short
`ntohl`	network to host long

- Example:

```
#include <stdio.h>  // IO functions
#include <arpa/inet.h> // contains ‘htons‘

#define PORT 9002

int main(void)
{
  printf("Port %hu = Host Byte Order | %X = Network Byte Order\n",
     PORT, htons(PORT));
  return 0;
}
```

```
Port 9002 = Host Byte Order | 2A23 = Network Byte Order
```

- Confirm using `bc`:

```
echo "obase=16; ibase=10; 9002" | bc
```

```
232A
```

9 Socket functions

In addition to the functions we already saw, there are a few more:

Function	Purpose
socket()	Creates and initializes a new socket.
bind()	Associates socket with a local IP address and port
listen() - server	Causes a TCP socket to listen for new connections.
connect() - client	SSets remote address and port, establishes a
accept() - server	CCreates new socket for an incoming TCP connection.
send(), recv()	Send and receive data with a socket.
sendto(), recvfrom()	Send and receive data from sockets in UDP
close(), closesocket()	Close a socket; also terminates a TCP connection.
shutdown()	Closes one side of a TCP connection
select()	Waits for an event on one or more sockets.
getnameinfo(),	PProtocol-independent way to work with
getaddrinfo()	hostnames and addresses
setsockopt()	Change socket options.
fcntl(), ioctlsocket()	Get and set some socket options.

10 Network programs as clients or servers

- Modern networking models stretch traditional client-server models. Network programs can be described as one of four types:
  1. A TCP server
  2. A TCP client
  3. A UDP server
  4. A UDP client
- In the classic **client-server paradigm**, a server listens for new connections at a published address. The client establishes the connection knowing the server's address.
- Once the connection is established, client and server can send and receive data. Either of them can terminate the connection. Think of a hotel (server) and the guests (clients).
- An alternative paradigm is the **peer-to-peer** model (e.g. BitTorrent): Each peer has the same responsibilities and acts both as a client and as a server.
- The sockets are not created equal here: For each peer-to-peer connection, one peer is listening, and the other connecting. A central "tracker" server stores a list of peer addresses.

- In the FTP protocol, an FTP server listens for connections until the FTP client connects. Then the FTP client issues commands to the server. When it requests a file, the server makes a new connection to the client to transfer the file - now the FTP client accepts connections like a TCP server.

11 TCP program flow

- Remember the basic client workflow:

  1. The TCP client program must first know the TCP's server address.
  2. The client creates a socket with `socket` and then establishes a connection with `connect`.
  3. When the server accepts, the client can exchange data using `send` and `recv`.

- So far, we have manually constructed `struct sockaddr_in` where we store the IP addresses:

```
struct sockaddr_in server_address;
server_address.sin_family = AF_INET;  // dest IP
server_address.sin_port = htons(PORT);  // port
server_address.sin_addr.s_addr = INADDR_ANY; // source IP
```

- We've used it e.g. when connecting to the server from the client:

```
int network socket;  // created on the client
network_socket = socket(AF_INET,SOCK_STREAM,0);
connect(network_socket,
        (struct sockaddr *) &server_address,
        sizeof(server_address));
```

- The function `getaddrinfo` replaces this construction: it resolves hostnames (`"localhost"`, `"example.com"`) and service names ("http", ":80") into address structures (linked lists) that you can use with sockets. You can then access the list elements with `->` :

```
struct addrinfo hints, *res; // res: points to head of address list
hints.ai_family = AF_INET;
```

```
hints.ai_socktype = SOCK_STREAM;

int status = getaddrinfo("localhost", "9002", &hints, &res);

network_socket = socket(res->ai_family,
                        res->ai_socktype,
                        res->ai_protocol);
connect(network_socket, res->ai_addr,
                        res->ai_addrlen);
```

- With this function, the flow of TCP client & server looks like this:

12 UDP program flow

- TCP (Transmission Control Protocol) is called a **connection-oriented** protocol while UDP (User Datagram Protocol) is a **connectionless** protocol.
- In UDP, each data packet is addressed individually. It is completely independent and unrelated to any packets coming before or after it - like a **postcard**. There is no guarantee that it will arrive.
- A UDP client must know the address of the remote UDP peer in order to send the first packet.
- The UDP client uses `getaddrinfo` to resolve the address into a `struct addrinfo`.
- The difference to TCP: The UDP client cannot receive data first since the remote peer would not know where to send it (there's no handshake like in TCP). In TCP either client or server can send the first data.
- The program flow of a UDP client and server looks like this:

13 Header files for Windows and Unix-based systems

- "Winsock" short for Windows Sockets is Microsoft's implementation of the Berkeley Sockets API originally developed for UNIX systems. It lets you write TCP/UDP sockets.

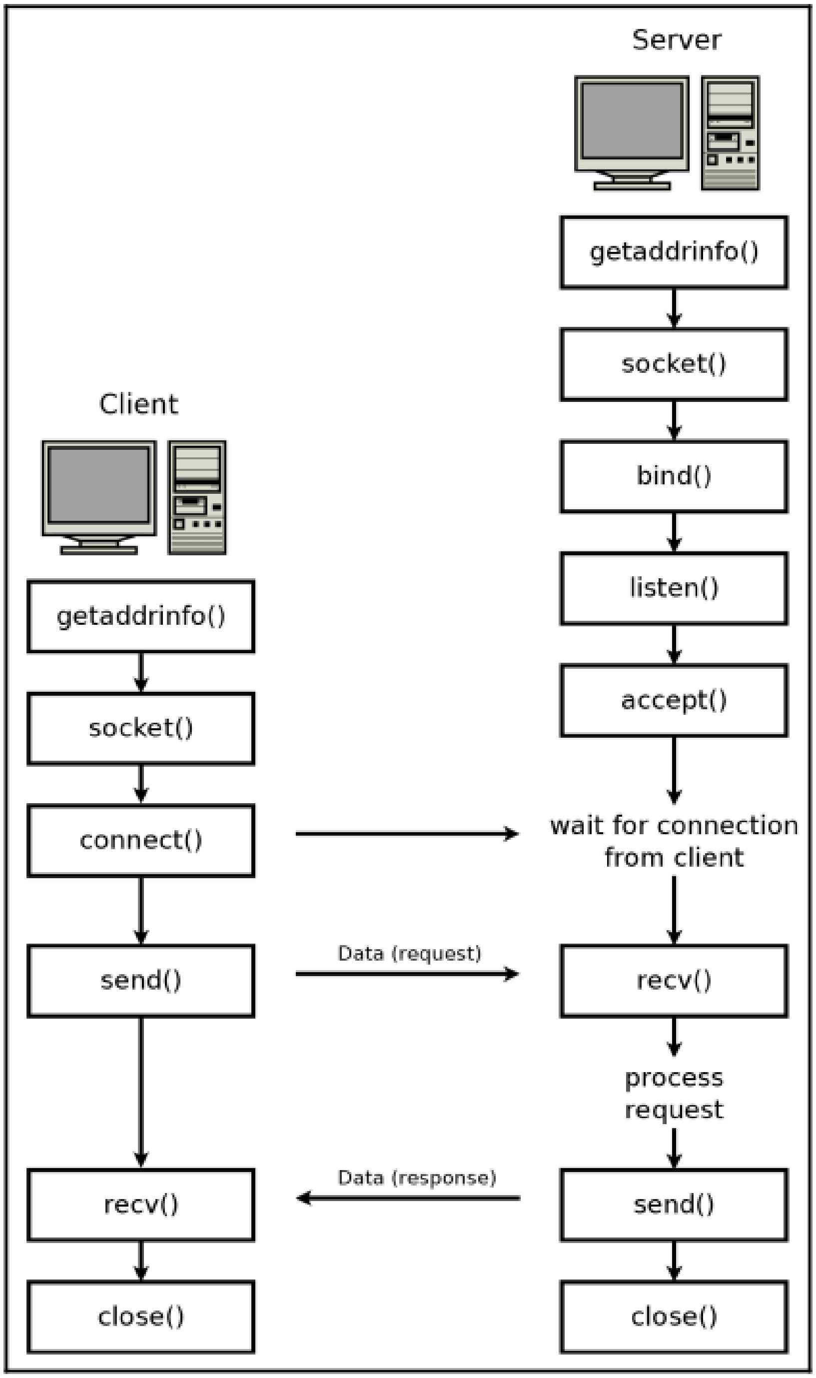

Figure 1: Source: van Winkle (2019)

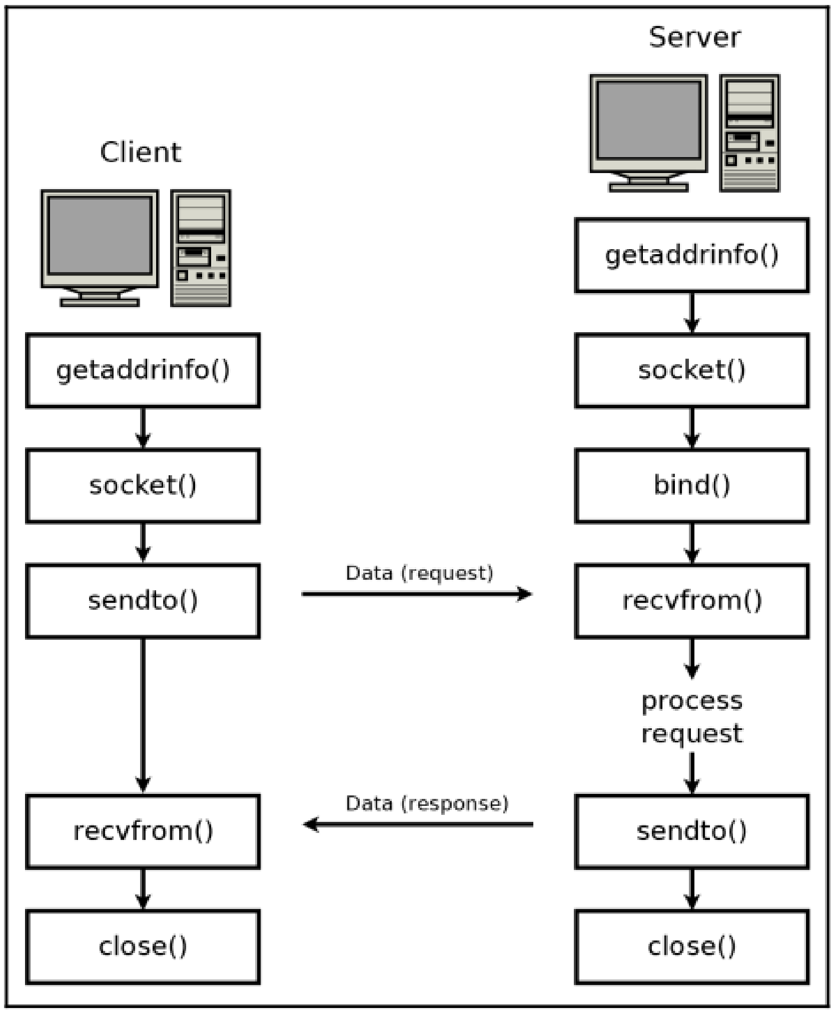

Figure 2: Source: van Winkle (2019)

- The required header files differ between implementations. The differences can be overcome with a **preprocessor** statement.
- In UNIX, a socket **descriptor** is a standard file descriptor (a small, non-negative number). `socket` returns an `int`.
- In Windows, a socket handle can be anything, and `socket` returns a `SOCKET`, a `typedef` for an `unsigned int` in the Winsock library headers.
- An example for how to deal with it:

```
#if !defined (_WIN32)
   #define SOCKET int  // define Windows-type socket
#endif

#if defined(_WIN32)
   #define CLOSESOCKET(s) closesocket(s) // Windows
#else
   #define CLOSESOCKET(s) close(s)  // Unix
#endif
```

- Comparison Unix/Winsock

Concept	Unix / POSIX	Windows (Winsock)
Socket library	Built-in (`unistd.h`, etc.)	Must load `ws2_32.dll`
Startup required?	No	call `WSAStartup()`
Shutdown required?	No	call `WSACleanup()`
Close socket	`close(socket_fd)`	`closesocket(socket_fd)`
Error codes	`errno` and `perror()`	`WSAGetLastError()`
Headers	`<sys/socket.h>`, etc.	`<winsock2.h>`

14 Networking a real world program

- We're building a web service that tells you what time it is right now. Users could navigate to our web page and get the time.
- You should always work out the local version of a program before networking it.
- The local, console version of the time-telling program:

```c
#include <stdio.h>  // for printf
#include <time.h>   // for time-related functions

int main()
{
  time_t timer; // declare a variable of type time_t for  calendar time

  time(&timer); // get the current time and store it in 'timer'

  // convert the calendar time to a human-readable string and print it
  printf("Local time is: %s", ctime(&timer));

  return 0;           // indicate that the program finished successfully
}
```

```
Local time is: Fri Apr 11 11:46:48 2025
```

- When you pass `&timer`, you pass a memory location where the function can write the current time. For more, check `man time(2)`. Then `ctime` takes the `time_t`-formatted time and converts it to a string to print - see `man ctime(3)`.

15 Analysis of the C Program for Displaying Local Time

The program retrieves the current system time and prints it in a human-readable format (e.g., "Sun Apr 13 12:34:56 2025"). It uses the C standard library and the `time.h` library to handle time-related operations.

15.1 Detailed Analysis

15.1.1 1. Preprocessor Directives

```c
#include <stdio.h>
#include <time.h>
```

- **#include**: A preprocessor directive that includes the contents of a library or file.

- **<stdio.h>**: The Standard Input/Output library, providing functions like `printf` for output.
- **<time.h>**: The Time library, providing functions and types like `time` and `ctime` for handling dates and times.
- Libraries are pre-written code collections. Including them lets you use their functions without writing the code yourself.

15.1.2 2. The main Function

```
int main()
```

- **int**: The return type, indicating `main` returns an integer to signal program status.
- **main**: The program's entry point where execution begins.
- **Parentheses ()**: Indicate `main` takes no arguments here.
- **Curly braces {}**: Enclose the body of `main`, defining the program's actions.
- Every C program needs a `main` function as its starting point. The `int` return type is standard, and we return `0` to indicate success.

15.1.3 3. Variable Declaration

```
time_t timer;
```

- **time**$_t$: A data type from `time.h`, typically an integer (e.g., `long`), representing time as seconds since the Unix epoch (January 1, 1970, 00:00:00 UTC).
- **timer**: The variable name, which will store the current time.
- Think of `time_t` as a container for time values. `timer` is the label for that container, to be filled with the current time.

15.1.4 4. Getting the Current Time

```
time(&timer);
```

- **time**: A function from `time.h` that fetches the current system time.

- **&timer**: Passes the address of `timer` (a pointer) to `time`, which stores the current time in `timer`.
- **What it does**: Updates `timer` with the current time as seconds since the epoch.
- `time` asks the computer for the current time and stores it in `timer`. The `&` is needed because `time` modifies `timer` directly by accessing its memory location.

15.1.5 5. Printing the Time

```
printf("Local time is: %s", ctime(&timer));
```

- **printf**: A function from `stdio.h` for formatted console output.
- **"Local time is: %s"**: The format string:
  - "Local time is: ": Printed as-is.
  - %s: Placeholder for a string.
- **ctime(&timer)**:
  - **ctime**: A `time.h` function that converts a `time_t` value into a human-readable string.
  - **&timer**: Passes the address of `timer`.
  - **Output**: Returns a string in the format `"Www Mmm dd hh:mm:ss yyyy\n"`, where:
    * `Www`: Day of the week (e.g., "Sun").
    * `Mmm`: Month (e.g., "Apr").
    * `dd`: Day of the month (e.g., "13").
    * `hh:mm:ss`: Time in 24-hour format (e.g., "12:34:56").
    * `yyyy`: Year (e.g., "2025").
    * `\n`: Newline character.
  - Example: `"Sun Apr 13 12:34:56 2025\n"`.
- **How it works**: `ctime(&timer)` generates the time string, which `printf` inserts into `%s`, producing output like:

  ```
  Local time is: Sun Apr 13 12:34:56 2025
  ```

- `ctime` turns seconds into a readable format. `printf` combines the phrase with the time and displays it.

15.1.6 6. Returning from main

```
return 0;
```

- **return**: Ends `main` and sends a value to the operating system.
- **0**: Indicates successful execution (non-zero values signal errors).
- After `return 0`, the program terminates.
- `return 0` is like saying, "I'm done, and everything worked!" It signals the computer that the program completed successfully.

15.2 How the Program Works as a Whole

1. Execution starts at `main`.
2. Declares a `time_t` variable `timer`.
3. `time(&timer)` fetches the current time and stores it in `timer` as seconds since 1970.
4. `ctime(&timer)` converts the time to a readable string.
5. `printf` prints "Local time is:" followed by the time string.
6. `return 0` ends the program successfully.

15.3 Key Concepts

- **Libraries**:
  - `stdio.h` and `time.h` provide functions to simplify coding.
  - `stdio.h` handles input/output; `time.h` manages time.
- **Data Types**:
  - `time_t` stores time values.
  - Types define what data a variable holds.
- **Functions**:
  - `time`, `ctime`, and `printf` perform specific tasks.
  - Functions take arguments (e.g., `&timer`) and may return values.

- **Pointers**:
  - `&` gives a variable's memory address.
  - Introduces pointers, a core C concept.
- **Program Flow**:
  - Executes line by line in `main`.
- **Output Formatting**:
  - `printf` uses `%s` to insert values.
  - `ctime` formats time automatically.

15.4 Example Output

Running the program on April 13, 2025, at 12:34:56 might produce:

```
Local time is: Sun Apr 13 12:34:56 2025
```

The output depends on the system's local time and time zone.

15.5 Potential Questions

- **Why use `&timer` instead of `timer`?**
  - `time` and `ctime` need a pointer to `timer`'s memory address. `&` provides that address.
- **What if I remove `#include <time.h>`?**
  - Errors occur because `time_t`, `time`, and `ctime` are undefined without `time.h`.
- **Can I change the time format?**
  - Yes, using `strftime` and `localtime` (advanced topic).
- **What's in `timer`?**
  - Seconds since January 1, 1970 (e.g., 1,744,099,296 on April 13, 2025).
- **Why does `ctime` add a newline?**
  - It's part of `ctime`'s standard output for tidy printing.

15.6 Suggestions for Experimentation

Try these to learn more:

1. **Print raw `timer` value**:

   ```
   #include <time.h>
   printf("Seconds since epoch: %ld\n", time);
   ```

   ```
   Seconds since epoch: 125674404552736
   ```

   Shows seconds since 1970.

2. **Add a loop to update time**:

   ```
   #include <stdio.h>
   #include <time.h>
   #include <unistd.h>

   int main() {
     time_t timer;
     for (int i = 0; i < 5; i++) {
       time(&timer);
       printf("Local time is: %s", ctime(&timer));
       sleep(1);
     }
     return 0;
   }
   ```

   ```
   Local time is: Sun Apr 13 22:33:21 2025
   Local time is: Sun Apr 13 22:33:22 2025
   Local time is: Sun Apr 13 22:33:23 2025
   Local time is: Sun Apr 13 22:33:24 2025
   Local time is: Sun Apr 13 22:33:25 2025
   ```

   Prints time every second for 5 seconds (`sleep` needs `unistd.h` on Unix-like systems).

3. **Use `gmtime` for UTC**:

```c
#include <time.h>

time_t timer;
struct tm *gmt = gmtime(&timer);

printf("UTC time: %d-%d-%d %d:%d:%d\n",
       gmt->tm_year + 1900, gmt->tm_mon + 1, gmt->tm_mday,
       gmt->tm_hour, gmt->tm_min, gmt->tm_sec);
```

```
UTC time: 1970-1-1 0:1:40
```

Shows UTC time and introduces `struct tm`.

15.7 Common Errors to Watch For

- Forgetting `#include <time.h>`:
  - **Error**: `time_t` or `ctime` undefined.
  - **Fix**: Add `#include <time.h>`.
- Using `timer` instead of `&timer`:
  - **Error**: Type mismatch in `time` or `ctime`.
  - **Fix**: Use `&timer` for the address.
- Not returning 0:
  - May work but returning `0` is best practice.

15.8 Why This Program is Useful

- **Real-world application**: Time display is used in logs, interfaces, and scripts.
- **Learning opportunity**: Teaches libraries, functions, pointers, and types.
- **Foundation**: Basis for exploring time formatting, differences, or dates.

15.9 Summary for Students

This program teaches you to:

- Use libraries (`stdio.h`, `time.h`).
- Declare `time_t` variables.
- Fetch time with `time`.
- Format time with `ctime`.
- Print with `printf`.
- Structure a C program with `main` and `return`.

16 Using `make` to build files from multiple sources

- Tangle the source code and build it with `make` using the built-in `makefile`:

  ```
  ls -lt time_console.c
  make time_console
  ```

- Run it:

  ```
  ./time_console
  ```

- Write a `makefile` ‘maketime‘ that builds the executable `time_console`.

  ```
  time_console: time_console.c
  gcc time_console.c -o time_console
  ```

- Run it:

  ```
  touch time_console.c
  make -f maketime
  ./time_console
  ```

- Now split the file `time_console.c` into two parts:

  1. `time_console2.h` (with the header files only)
  2. `time_console2.c` (with the `main` program only)

3. Add `#include "time_console2.h"` before the `main` program.

- Header file:

```
#include <stdio.h>  // for printf
#include <time.h>   // for time-related functions
```

- Main program:

```
#include "time_console2.h"

int main()
{
  time_t timer; // declare a variable of type time_t for  calendar time

  time(&timer); // get the current time and store it in 'timer'

  // convert the calendar time to a human-readable string and print it
  printf("Local time is: %s", ctime(&timer));

  return 0;           // indicate that the program finished successfully
}
```

- Create a suitable `makefile` called `maketime2`:

```
time_console2.o: time_console2.c time_console2.h
gcc -c time_console2.c -o time_console2.o

time_console2: time_console2.o
gcc time_console2.o -o time_console2

clean:
rm -f *.o time_console2
```

- Check all files were created:

```
ls -lt maketime2 time_console2*
```

- Test the `makefile`:

```
make -f maketime2 clean
make -f maketime2
```

- Run the executable:

```
pwd
```

17 Network header files - library files and macros

- This header file will only work on Unix-like systems.
- We're putting all header files into `time_server.h`:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
```

- Explanation:

Header File	Purpose and Use (Max 72 chars)
`sys/types.h`	Types (e.g., `pid_t`) for socket vars in time server.
`sys/socket.h`	Socket funcs (`socket`, `bind`) to send time to clients.
`netinet/in.h`	Structs (`sockaddr_in`) for IP/port in time server.
`arpa/inet.h`	Funcs (`inet_pton`, `htons`) for IP/port in time server.
`netdb.h`	Hostname resolution for network time requests.
`unistd.h`	Close socket after sending time to clients.
`errno.h`	Error handling for network issues in time server.
`time.h`	Time funcs (`time`, `ctime`) to get/format time for client.
`stdio.h`	`printf` to format time string for network transmission.
`string.h`	String manipulation functions such as `strlen`

18 Building the networked time-telling program

- This is the flow that we've got to implement:

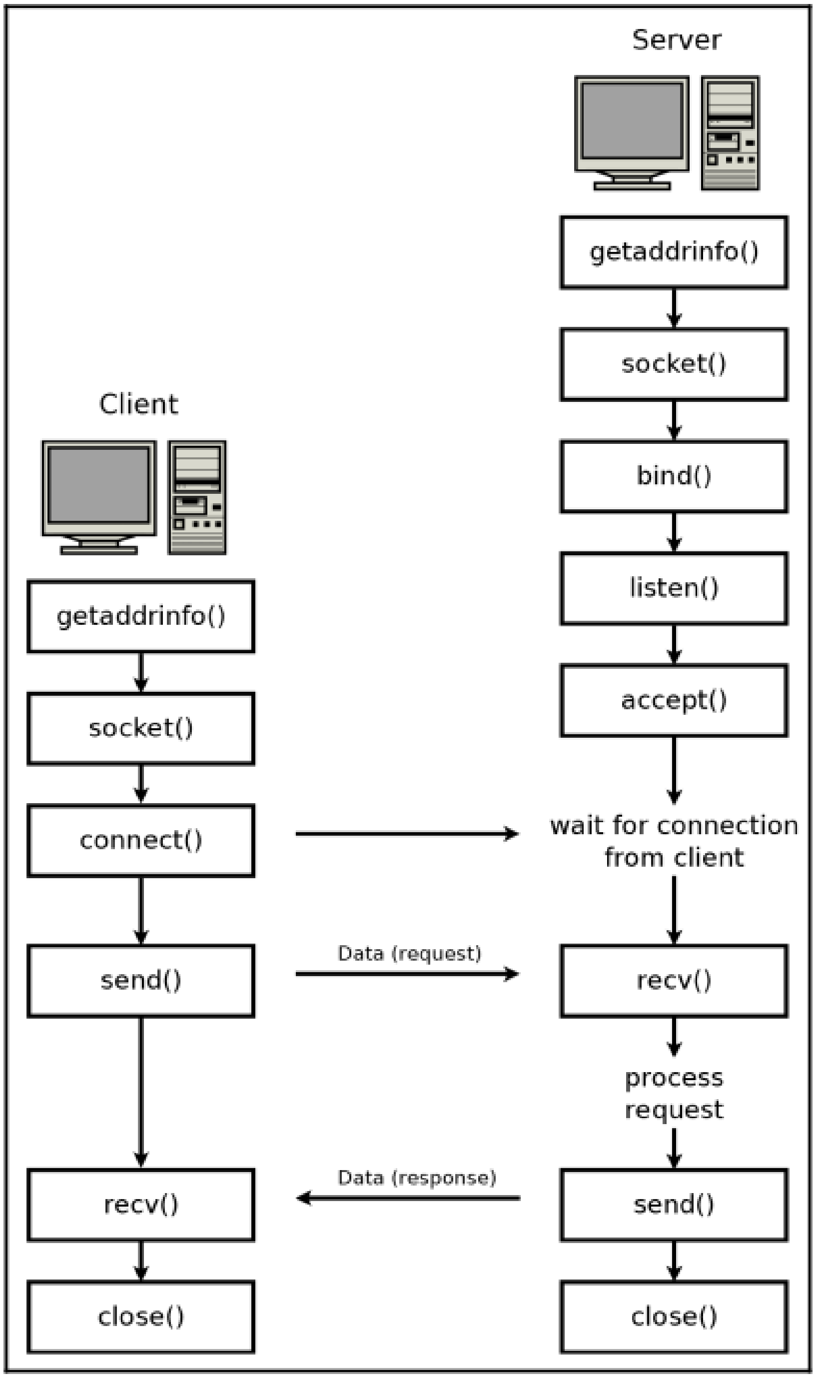

Figure 3: Source: van Winkle (2019)

- Most of our functions are `man` level 2 (system calls) that interact directly with the OS kernel for time retrieval and networking.
- A couple of our functions are `man` level 3 (library functions).
- We're going to assemble the file using `noweb` chunks.

```
#include "time_server.h"
#define PORT "8080" // for the ‘service‘ parameter of ‘getaddrinfo‘
#define CONN 10 // max number of connections to ‘listen‘ to

int main(void)
{
  <<configure local address>>
  <<create socket>>
  <<check that socket is valid>>
  <<bind address to socket>>
  <<listen to connections>>
  <<accept incoming connection>>
  <<print client address to console>>
  <<receive client request>>
  <<send server response>>
  <<send time>>
  <<close client connection>>
  <<close server connection>>

    return 0;
}
```

18.1 Pick address for the server to `bind(2)` to with `getaddrinfo(3)`

- Figure out the local address that the web server should bind to:

```
printf("Configuring local address...\n");
struct addrinfo hints;
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;

struct addrinfo *bind_address;
getaddrinfo(0, PORT, &hints, &bind_address);
```

- Explanation:

  1. `struct addrinfo hints`: Declares `hints struct` for address configuration options.
  2. `memset(&hints, 0, sizeof(hints))`: Clears `hints struct` to avoid garbage values (check the `memset man` page).
  3. `hints.ai_family = AF_INET`: Sets address family to IPv4 for the server.
  4. `hints.ai_socktype = SOCK_STREAM`: Specifies TCP (stream) socket type.
  5. `hints.ai_flags = AI_PASSIVE`: Allows binding to any local IP (e.g., 0.0.0.0).
  6. `struct addrinfo *bind_address`: Declares pointer for storing address info.
  7. `getaddrinfo(0, "8080", &hints, &bind_address)`: Gets address for port 8080, using `hints` and 0. The prototype:

     ```
     int getaddrinfo(const char *node, const char *service,
                     const struct addrinfo *hints,
                     struct addrinfo **res);
     ```

     `service` can be a `PORT` (like `"8080"`) or a service like (`="http"`), which is why it's got to be a string.
  8. The `man` page of `getaddrinfo` explains what that does:

     > If the `AI_PASSIVE` flag is specified in `hints.ai_flags`, and node is NULL, then the returned socket addresses will be suitable for `bind(2)` -ing a socket that will `accept(2)` connections. The returned socket address will contain the "wildcard address" ($\text{INADDR}_{\text{ANY}}$ for IPv4 addresses [...]). The wildcard address is used by applications (typically servers) that intend to accept connections on any of the host's network addresses.

     `getaddrinfo` is protocol-independent: we only need to change `AF_INET` to `AF_INET6` to make it work on IPv6.

18.2 Create the socket descriptor by calling `socket(2)`

- Once we've got the local address info, we can create the socket.

```
printf("Creating socket...\n");
int socket_listen;
socket_listen = socket(bind_address->ai_family,
                       bind_address->ai_socktype,
                       bind_address->ai_protocol);
```

- Explanation:

  1. `int socket_listen`: Declares variable for the listening socket. On Windows, this would be the `SOCKET` type defined just for sockets.
  2. `socket_listen = socket(...)`: Creates socket using address family, type, and protocol from $\text{bind}_{\text{address}}$.
  3. The reason we used `getaddrinfo` is that we can now pass parts of `bind_address` as the arguments to `socket`.
  4. It is common to see programs that call `socket` first. This complicates things since socket family, type, and protocol must be entered multiple times.

18.3 Check that the call to `socket` is successful

- We check that the call to `socket` was successful:

```
if (!(socket_listen >=0)) {
  fprintf(stderr, "socket() failed, (%d)\n", errno);
  return 1;
 }
```

- Explanation of Socket Validation Code for Time Program

  1. `if (!(socket_listen >` 0))=: Checks if `socket_listen` is valid (non-negative).
  2. `fprintf(stderr, "socket() failed, (%d)\n", errno)`: Prints error to `stderr` with `errno(3)` if socket creation fails. Check `man errno`.
  3. `return 1`: Exits program with error code 1 on socket failure.

18.4 Call `bind` to associate the socket with `bind_address`

- Once the socket has been created, we call `bind` to associate the socket with the address generated by `getaddrinfo`:

  ```
  printf("Binding socket to local address...\n");
  if (bind(socket_listen,
           bind_address->ai_addr,
           bind_address->ai_addrlen)) {
    fprintf(stderr, "bind() failed. (%d)\n", errno);
    return 1;
   }
  freeaddrinfo(bind_address);
  ```

- Explanation:

  1. `if (bind(socket_listen, ...))`: Binds `socket_listen` to the address and port in `bind_address`. When `bind` succeeds (0) the condition is false.
  2. `fprintf(stderr, "bind() failed. (%d\n", errno)`: Prints error with `errno` if binding fails. `bind` returns 0 on success. It fails (-1) if the port we are binding to is already in use.
  3. `return 1`: Exits program with error code 1 if binding fails.
  4. `freeaddrinfo(bind_address)`: Frees memory allocated for `bind_address`.

18.5 Start listening for connections with `listen(2)`

- Once socket has been created and bound to a local address, we can start listening for connections:

  ```
  printf("Listening...\n");
  if (listen(socket_listen, CONN) < 0) {
    fprintf(stderr, "listen() failed. (%d)\n", errno);
    return 1;
   }
  ```

- Explanation:

  1. `if (listen(socket_listen, CONN) < 0)`: Sets `socket_listen` to listen for up to `CONN` incoming connections.

2. `fprintf(stderr, "listen() failed.  (%d)\n", errno)`: Prints error with `errno` if `listen` fails.
3. `return 1`: Exits program with error code 1 if `listen` fails.

18.6 Start accepting incoming connection with `accept(2)`

- Once the socket has begun listening for connections, you can `accept` incoming connections:

```
printf("Waiting for connection...\n");
struct sockaddr_storage client_address;
socklen_t client_len = sizeof(client_address);
int socket_client = accept(socket_listen,
                       (struct sockaddr*) &client_address,
                       &client_len);
if (!(socket_client >= 0)) {
  fprintf(stderr, "accept() failed, (%d)\n", errno);
  return 1;
 }
```

- Explanation:

1. `struct sockaddr_storage client_address`: Declares storage for client address info.
2. `socklen_t client_len = sizeof(client_address)`: Sets size of `client_address struct`.
3. `int socket_client = accept(socket_listen, ...)`: Accepts client connection, returns new socket.
4. `if (!(socket_client` >0))=: Checks if accept failed (`socket_client < 0`).
5. `fprintf(stderr, "accept() failed, (%d)\n", errno)`: Prints error with `errno` if accept fails.
6. `return 1`: Exits program with error code 1 if `accept` fails.

- `accept` will block your program until a new connection is made: it will `sleep` until a client connects to `socket_listen`. Then `accept` will create a new socket `socket_client` used to send and receive data, and `address` also fills in address info of the client.

18.7 Print the client's address to the console with `getnameinfo(3)`

- A TCP connection to a remote client has now been established, and we're waiting for the client to make a request.
- This step is optional but it is good practice to log network connections, and we're using `getnameinfo` for that.

```
printf("Client is connected... ");
char address_buffer[100];
getnameinfo((struct sockaddr*) &client_address,
        client_len,
        address_buffer,
        sizeof(address_buffer),
        0,0,NI_NUMERICHOST);
printf("%s\n", address_buffer);
```

- Explanation:

  1. `char address_buffer[100]`: Declares array to store client's IP address as string.
  2. `getnameinfo(...)`: Converts client address to numeric IP string.
  3. `printf("%s\n", address_buffer)`: Prints the client's IP address from buffer.

- The `getnameinfo(3)` prototype:

```
int getnameinfo(const struct sockaddr *addr, socklen_t addrlen,
                char *host, socklen_t hostlen,
                char *serv, socklen_t servlen, int flags);
```

The function takes the client's address `addr` and address length `addlen` (it can work with both IPv4 and IPv6 addresses). The hostname output is written to `host`. The service name and its length is output to `serv` and `servlen` (we don't care about this here). The `NI_NUMERICHOST` flag means that we want to see the hostname as an IP address.

18.8 Read client request with `recv`

- We read a client request using the `recv(2)` function:

```
printf("Reading request...\n");
char request[1024];
int bytes_received = recv(socket_client,
                          request,
                          1024,0);
printf("Received %d bytes.\n", bytes_received);
```

- Explanation:

  1. `char request[1024]`: Declares array to store client's request up to 1024 bytes.
  2. `int bytes_received = recv(socket_client, request, 1024, 0)`: Receives data from client into `request`, returns bytes read.
  3. `printf("Received %d bytes.\n", bytes_received)`: Prints number of bytes received.

- If nothing has been received yet, `recv` blocks until it has something. If the connection is terminated by the client, `recv` returns 0 or -1 (one should check `recv > 0`).

- A real web server would need to parse the request and look at which resource the web client is requestion (e.g. HTTP, FTP etc.). Here, we can ignore the `request` altogether but you can print it to console.

18.9 Send response from server back to the client with `send(2)`

- Once the web browser has sent the client request, the server can send its response back:

```
printf("Sending response...\n");
const char *response =
  "HTTP/1.1 200 OK\r\n"
  "Connection: close\r\n"
  "Content-Type: text/plain\r\n\r\n"
  "Local time is: ";
int bytes_sent = send(socket_client,
                  response,
                  strlen(response),
                  0);
printf("Sent %d of %d bytes.\n", bytes_sent, (int) strlen(response));
```

- Explanation:

    1. `const char *response = "HTTP/1.1 200 OK\r\n..." ...`: Defines HTTP response header with time prefix.
    2. `int bytes_sent = send(socket_client, response, strlen(response), 0)`: Sends response string to client, returns bytes sent.
    3. `printf("Sent %d bytes.\n", bytes_sent)`: Prints number of bytes sent.

- We set `response` to a standard HTTP response header followed by the beginning of our message (`"Local time is: "`). It tells the browser:

    1. Your request is OK.
    2. The server will close the connection when all data is sent.
    3. The data you receive will be plain text.

- HTTP requires line endings to take the form of a carriage return followed by a new line: `\r\n` is a blank line in your response.

- You should generally check that the number of bytes sent with `send(2)` was as expected, and you try to send the rest if it's not.

18.10 Send the time with `send(2)`

- Once the HTTP header and the beginning of the message is sent, we can send the actual time. We get it as before in `time_console.c`, and we send it using `send(2)`:

```
time_t timer;  // declare a variable to hold the time
time(&timer);  // get the time
char *time_msg = ctime(&timer);
bytes_sent = send(socket_client,
              time_msg,
              strlen(time_msg),
              0);
printf("Sent %d of %d bytes.\n",bytes_sent, (int)strlen(time_msg));
```

- Explanation:

    1. `time_t timer`: Declares variable to hold current time.

2. `time(&timer)`: Fetches current system time into timer.
3. `char *time_msg = ctime(&timer)`: Converts timer to human-readable string.
4. `bytes_sent = send(socket_client, time_msg, strlen(time_msg), 0)`: Sends time string to client, returns bytes sent.
5. `printf("Sent %d of %d bytes.\n", ...)`: Prints bytes sent vs. time string length.

18.11 Close client connection with `close(2)`

- We now must `close` the client connection to indicate to the browser application that we've sent all of our request data:

  ```
  printf("Closing requesting (client) connection...\n");
  close(socket_client);
  ```

- If we don't close the connection, the browser will wait for more data until it times out (~300s for Chrome - browser dependent).

18.12 Close server connection with `close(2)`

- We could call `accept` on `socket_listen` to accept additional connections: That's what a real server would do.

  ```
  printf("Closing listening (server) connection...\n");
  close(socket_listen);
  printf("Finished.\n");
  ```

19 Compile and run the program

- Compile the program:

  ```
  make time_server
  ```

- Start the server on the terminal (`M-x term /usr/bin/bash`) or open another fully functional terminal outside of Emacs.
- Start the server with `./time_server`
- When you see the `"Waiting for connection..."` message, open a web browser to the address `127.0.0.1:8080` or `localhost:8080`.

- This is what you should see:

```
$ ./time_server
Configuring local address...
Creating socket...
Binding socket to local address...
Listening...
Waiting for connection...
Client is connected... 127.0.0.1
Reading request...
Received 660 bytes.
Sending response...
Sent 79 of 79 bytes.
Sent 25 of 25 bytes.
Closing requesting (client) connection...
Closing listening (server) connection...
Finished.
$
```

- If you restart the server immediately after this, you may get a bind() failed. (98) error. This is the EADDRINUSE error code from <errno.h>.

- For the location of <errno.h> in /usr/include (std header files):

```
grep -r "#define.*\b98\b" /usr/include | grep errno.h


/usr/include/asm-generic/errno.h:#define EADDRINUSE 98 /* Address already in use
```

- The port 8080 was still in use because that port is still in TIME_WAIT state for a short period to make sure any lingering packets are handled, about 30-60 seconds.

- Fix this by calling setsockopt(2) after socket but before bind:

```
int opt = 1; // flag to enable socket option
setsockopt(socket_listen, // socket on which the server listens
           SOL_SOCKET, // change options at the sockets API level
           SO_REUSEADDR, // option name
           &opt, // option value (switch on)
           sizeof(opt)); // length of option value
```

- Insert this into the `main` program before the `<<bind...>>` code:

  ```
  <<enable immediate socket reuse>>
  ```

- Then tangle `time_server.c`, `make time_server` again, start the server, get the time, restart the server and try to get the time immediately and it should work now.

- Alternative check: Use `netcat` (`nc`) if you have it:

  ```
  nc localhost 8080
  ```

- Output:

  ```
  HTTP/1.1 200 OK
  Connection: close
  Content-Type: text/plain

  Local time is: Mon Apr 14 20:24:22 2025
  ```

20 TODO Networking with `inetd`

- On Unix-like systems like Linux or MacOS, the `inetd` service (a system daemon program) can be used to turn console-only applications into networked ones. This service only runs when a client connects.

- You can configure `inetd` with `/etc/inetd.conf` with the program's location, port number, protocol, and the user you want it to run as.

- Steps to do this:

  1. Check if `inetd` exists: `which inetd`
  2. Install it: `sudo apt-get install openbsd-inetd`
  3. Modify the program to use `stdin` and `stdout` instead of sockets.
  4. Configure `inetd` to launch the program on a port (e.g. `8080`).

20.1 Modify the Program

- Remove socket code (`socket`, `bind`, etc.).
- Read client request from `stdin` (optional, for HTTP).
- Write HTTP response and time to `stdout`.
- Add `#include <time.h>` for time functions.
- `fgets` reads HTTP request (e.g. from browser).
- `fputs` write response to `stdout`, sent to client.
- `fflush(stdout)` ensures data is sent before exit.
- Code: You can run this file since it writes to `stdout`, but you should tangle it as `time_inetd.c` and `make time_inetd` it.

```
#include <stdio.h>
#include <string.h>
#include <time.h>

int main(void) {
  // Read request from stdin (optional, to handle HTTP clients)
  char request[1024];
  if (fgets(request, sizeof(request), stdin) == NULL) {
    fprintf(stderr, "No input received\n");
  }

  // Prepare HTTP response
  const char *header =
    "HTTP/1.1 200 OK\r\n"
    "Connection: close\r\n"
    "Content-Type: text/plain\r\n\r\n"
    "Local time is: ";
  fputs(header, stdout);

  // Get and send time
  time_t timer;
  time(&timer);
  char *time_msg = ctime(&timer);
  fputs(time_msg, stdout);
```

```
  fflush(stdout); // Ensure output is sent

  return 0;
}
```

```
HTTP/1.1 200 OK
Connection: close
Content-Type: text/plain

Local time is: Mon Apr 14 20:08:35 2025
```

- Build it:

```
make time_inetd
```

```
cc     time_inetd.c   -o time_inetd
```

- Move the executable to `/usr/local/bin`

```
mv -v time_inetd /usr/local/bin/time_inetd
which time_inetd
```

20.2 Configure inetd and service

- To update `/etc/inetd.conf`, use `update-inetd(8)`. Or you can write this to the file in the service category `#:OTHER`:

```
time_service stream tcp nowait nobody /usr/local/bin/time_inetd time_inetd
```

- Fields:
  - `time_service`: Service name (defined in `/etc/services`).
  - `stream`: TCP socket type.
  - `tcp`: Protocol.
  - `nowait`: Spawn new process per connection.
  - `nobody`: Run as low-privilege user.
  - `/usr/local/bin/time_inetd`: Program path.

    - `time_inetd`: Program name (`argv[0]`).

- Update `/etc/services`: Map service name to port in /etc/services.

  ```
  time_service 8080/tcp # Time service
  ```

- Restart `inetd` daemon: `sudo service inetd restart`.

- Test the program:

    1. Ensure program is executable: `chmod +x /usr/local/bin/time_inetd`

- Test with browser: `http://localhost:8080`.

- Or use `netcat: nc localhost 8080`.

- Expected output:

  ```
  HTTP/1.1 200 OK
  Connection: close
  Content-Type: text/plain

  Local time is: Mon Apr 14 20:00:00 2025
  ```

21 Sources

- van Winkle, Hands-on Network Programming with C (Packt);

- Eduonix, Learn Socket Programming from Scratch (Udemy).