Introduction to Sockets

Marcus Birkenkrahe

February 26, 2025

Contents

1 Objectives

The following topics are covered:

- ☐ What are port numbers?
- ☐ How to find port numbers on Linux?
- ☐ What is the client-server model?
- ☐ What's a socket?

☐ What's your IP address?

☐ How can you list network adapters using C?

2 Port numbers

- How does a packet know, which application is responsible for it once it arrives at a specific system?
- To direct a packet to the right application, **port** numbers are used. If IP addresses are phone numbers, then port numbers are like phone extensions.
- Applications on your system with specific ports:
  - multiple web browsers
  - an email client
  - a video-conferencing client
- When a packet arrives, the OS looks at the destination port number, and it is used to determine which application could handle it.
- Port numbers are stored as unsigned 16-bit integers - what's their range therefore?

  ```
  2^16
  ```

- Some port numbers for common protocols:

Port Number		Protocol
20, 21	TCP	**File Transfer Protocol (FTP)**
22	TCP	**Secure Shell (SSH)**
23	TCP	**Telnet**
25	TCP	**Simple Mail Transfer Protocol (SMTP)**
53	UDP	**Domain Name System (DNS)**
80	TCP	**Hypertext Transfer Protocol (HTTP)**
110	TCP	**Post Office Protocol, Version 3 (POP3)**
143	TCP	**Internet Message Access Protocol (IMAP)**
194	TCP	**Internet Relay Chat (IRC)**
443	TCP	**HTTP over TLS/SSL (HTTPS)**
993	TCP	**IMAP over TLS/SSL (IMAPS)**
995	TCP	**POP3 over TLS/SSL (POP3S)**

- The listed port numbers are assigned by the IANA (Internet Assigned Numbers Authority).

3 Linux example

- To see some of these in Linux:

```
netstat -tulnp

Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       Pi
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN      -
```

```
tcp6       0      0 ::1:631                 :::*                    LISTEN      -
tcp6       0      0 :::1716                 :::*                    LISTEN      22
tcp6       0      0 :::60000                :::*                    LISTEN      -
udp        0      0 224.0.0.251:5353        0.0.0.0:*                           44
udp        0      0 224.0.0.251:5353        0.0.0.0:*                           44
udp        0      0 0.0.0.0:5353            0.0.0.0:*                           -
udp        0      0 0.0.0.0:39386           0.0.0.0:*                           -
udp        0      0 127.0.0.53:53           0.0.0.0:*                           -
udp        0      0 127.0.0.1:323           0.0.0.0:*                           -
udp6       0      0 :::5353                 :::*                                -
udp6       0      0 ::1:323                 :::*                                -
udp6       0      0 fe80::3411:8f1:989f:546 :::*                                -
udp6       0      0 :::1716                 :::*                                22
udp6       0      0 :::35705                :::*                                -
```

On my machine at home (PIDs not listed):

```
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp6       0      0 :::60000                :::*                    LISTEN
tcp6       0      0 ::1:631                 :::*                    LISTEN
udp        0      0 127.0.0.53:53           0.0.0.0:*
udp        0      0 127.0.0.1:323           0.0.0.0:*
udp        0      0 224.0.0.251:5353        0.0.0.0:*
udp        0      0 224.0.0.251:5353        0.0.0.0:*
udp        0      0 224.0.0.251:5353        0.0.0.0:*
udp        0      0 224.0.0.251:5353        0.0.0.0:*
udp        0      0 0.0.0.0:5353            0.0.0.0:*
udp        0      0 0.0.0.0:46510           0.0.0.0:*
udp6       0      0 ::1:323                 :::*
udp6       0      0 fe80::3411:8f1:989f:546 :::*
udp6       0      0 :::5353                 :::*
udp6       0      0 :::47923                :::*
```

- Web server example (do this on the shell):
  1. start a web server with `python3` on port `8080`
  2. `grep` for `8080` in the `netstat -tulnp` output

```
python3 -m http.server 8080 &
netstat -tulnp | grep 8080
```

- Output:

```
tcp  0  0 0.0.0.0:8080 0.0.0.0:* LISTEN  274074/python3
```

4 Clients and servers

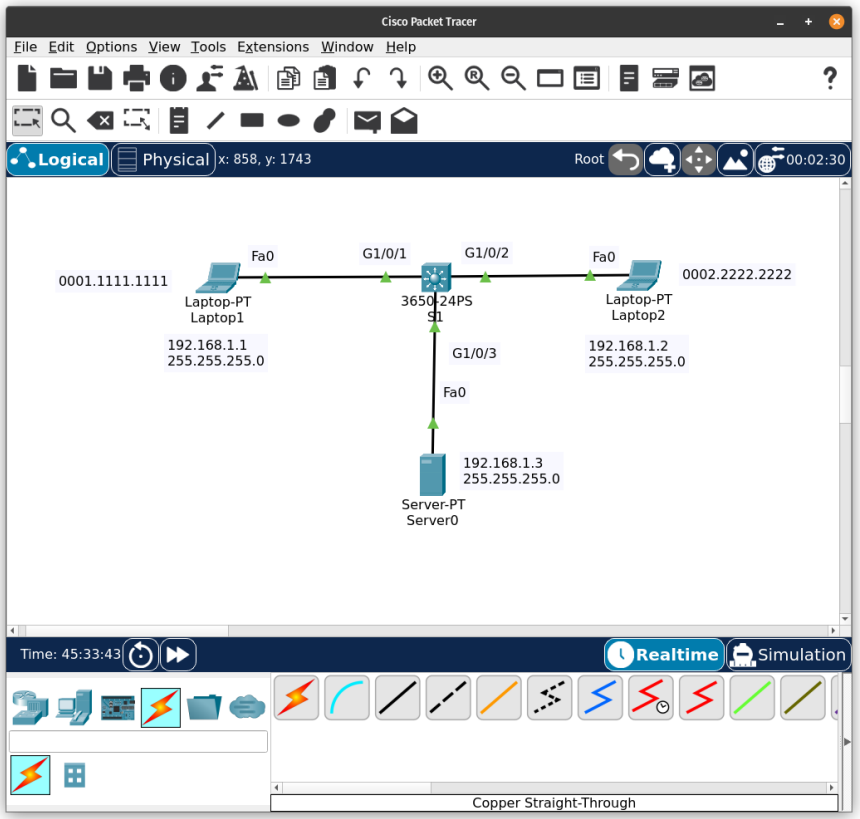

Figure 1: 2-client-1-server network in Cisco Packet Tracer

- When you use a telephone, a call must be initiated by one party. The initiating party dials the number for the receiving party, and the receiving party answers.

- This describes the **client-server** model paradigm in networking:
  1. A server listens for connections.
  2. The client knows the address and port number that the server is listening to and establishes connection by sending the first packet.
- Example: Web server
  1. The web server at `example.com` listens on port `80` (HTTP) and on port `443` (HTTPS) for connections.
  2. A web browser (client) establishes the connection by sending the first packet to `http://example.com:80`.

5 Sockets

- A **socket** is one end-point of a communication link. It enables sending and receiving data over the network.
- An open socket is defined by a quintuple (5-tuple):
  1. Local IP address
  2. Local port
  3. Remote IP address
  4. Remote port
  5. Protocol (UDP or TCP)
- With this information, the OS knows which application is responsible for any packets received.
- Example: You have two web browsers (Chrome and Firefox) open simultaneously, both connecting to `example.com` on port `80`. How can the OS keep the connections separate?

  > The OS keeps the connections separate by looking at the socket data. In this case, local and remote IP addresses, remote port and protocol are identical but the local port is different. The local (aka **ephemeral**) port was chosen to be different by the OS for the connection.

- The router NAT (Network Address Translation) also stores the socket 5-tuple to know how to route received packets back into the private network.

- Take a look at the `socket` man page on Linux. What does it do?

  The function `socket()` creates an endpoint for communication and returns a **file descriptor** that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

- What is a "file descriptor"?

  For the OS (any OS), everything is a "file". The file descriptor is a small non-negative integer that uniquely identifies a socket within a process. When calling `socket`, you get a descriptor that you can use to perform operations on that socket (binding, listening, accepting, reading, and writing).

- How does a `socket` call look like?

```
// socket: establish network connection to listen on
// returns: file descriptor (int)
// params: domain (protocol family), type (data stream),
//         protocol (TCP or UDP or another)
int socket(int domain, int type, int protocol);
```

  For example:
  - `domain = AF_STREAM` (Address Family - Internet: using IPv4 address)
  - `type = SOCK_STREAM` (Socket Type - TCP)
  - `protocol = 0` (automatically select the protocol - TCP)

- Where is the 5-tuple?

  The 5-tuple is completed when a client connects to a server, and a server accepts a client (functions `connect` and `accept`).

- Before looking at socket programming, let's look at a similar abstraction, **file** handling.

6 Hands-on tutorial: `hello world`

6.1 Simple hello world program

- Problem: Create a "hello world" program in C.

- Open Emacs to `hello.c` (C-x C-f hello.c <RET>)
- Solution:

```
/*****************************************************************/
// hello.c: print "Hello, World!" to stdout. Input: NONE. Output:
// Hello, World! (string). Author: Marcus Birkenkrahe. Date:
// 02/24/2025
/*****************************************************************/
// include libraries
#include <stdio.h>

// main: main program
// returns: integer (int)
// params: none (void)
int main(void)
{
  puts("Hello, world!");
  return 0;
}
```

- Compile and run the file on the command-line. The object file (executable) should be called `hello`:

```
gcc ../src/hello.c -o hello
./hello

Hello, world!
```

6.2 Simple hello world program with function

- Problem: Modify the "hello world" program with a function `hello` that prints the string `"Hello, world!"`. Include a function prototype. Tangle the code to `hello2.c`
- Solution:

```
/*****************************************************************/
// hello2.c: print "Hello, World!" to stdout use a void function
//           hello() to print Input: NONE Output: Hello, World!
//           (string) Author: Marcus Birkenkrahe Date: 02/24/2025
```

```
/*****************************************************************/

// include libraries
#include <stdio.h>

// prototype functions

// hello: print hello world
// returns: nothing (void)
// params: none (void)
void hello(void);

// main: main program
// returns: integer (int)
// params: none (void)
int main(void)
{
  hello(); // function call
  return 0;
}

// function definition
void hello(void)
{
  puts("Hello, world!");
}
```

- Compile and run the file on the command-line. The object file (executable) should be called `hello2`:

```
gcc ../src/hello2.c -o hello2
./hello2

Hello, world!
```

6.3 Hello world with command-line argument

- Modify the "Hello, World!" program to accept a name as a command-line argument and print:

  1. "Hello, [Name]!" if a name is provided.

2. "Hello, World!" if no name is provided.

- To check if an argument is provided, check if `argc` (the number of command-line arguments) is greater than 1.
- The command-line argument itself (if one was given), is stored in the array `argv[]`. Its first element, `argv[0]` is the name of the program itself (e.g. `hello3`), the next elements are the command-line arguments.
- The format specifier to print a string in C is `%s`, for example:

  ```
  printf("Hello, %s\n", "Jack");
  ```

  prints: `Hello, Jack`.
- Starter code:

  ```
  /*****************************************************************/
  // hello3.c: command-line version of "hello world" with input.  Input:
  // [name] (string) or none. Output: "Hello, [name]!" or "Hello,
  // world!". Author: Marcus Birkenkrahe Date: 02/24/2025
  /*****************************************************************/
  #include <stdio.h>

  // main: main program
  // returns: integer (int)
  // params: argument count (argc), argument vector (argv)
  int main(int argc, char *argv[])
  {
    // TODO: Check if an argument is provided

    // TODO: Print "Hello, [Name]!" if a name is given

    // TODO: Otherwise, print "Hello, World!"

    return 0;
  }
  ```

- Test the code on the command-line:

  ```
  gcc hello3.c -o hello3
  ./hello3 Bob
  ./hello3
  ```

```
Hello, Bob!
Hello, world!
```

- Expected output:

```
Hello, Bob!
Hello, world!
```

- Solution:

```
/*****************************************************************/
// hello3.c: command-line version of "hello world" with input.  Input:
// [name] (string) or none. Output: "Hello, [name]!" or "Hello,
// world!". Author: Marcus Birkenkrahe Date: 02/24/2025
/*****************************************************************/
#include <stdio.h>

// main: main program
// returns: integer (int)
// params: argument count (argc), argument vector (argv)
int main(int argc, char *argv[])
{
  // TODO: Check if an argument is provided
  if (argc > 1) {
    // TODO: Print "Hello, [Name]!" if a name is given
    printf("Hello, %s!\n", argv[1]);
  } else {
    // TODO: Otherwise, print "Hello, World!"
    printf("Hello, world!");
  }
  return 0;
}
```

7 IN_PROGRESS Files

- A similar abstraction is a **file**, which enables applications to read and write data using a file handle.

- In C for example, the `FILE *` type in the standard library is a file handle used with functions like `fopen`, `fread`, `fwrite`, and `fclose`.

- A lower-level system call `open` returns a file descriptor (an integer handle), which can be used with `read`, `write`, and `close`.
- Code along!
- Example 1: Using `FILE *` (C Standard Library)
  1. `fopen` opens `file` (the 'handle') in write mode.
  2. `file` is a `FILE` pointer to the beginning of the file.
  3. `fprintf` writes text to the file.
  4. `fclose` close the handle.

```
#include <stdio.h>

int main() {
  FILE *file = fopen("example.txt", "w");  // Open file for writing

  if (file == NULL) {
    perror("Error opening file"); // print error to stderr
    return 1;                     // if no file found
  }

  fprintf(file, "Hello, File!\n");  // Write to the file
  fclose(file);  // Close the file handle

  return 0;
}
```

- Testing example 1:

```
gcc fileh.c -o fileh
./fileh
ls -lt example.txt fileh.c fileh
cat example.txt
```

- Example 2: Using a file descriptor (`open`, `write`, `close`)
  1. `open` opens the file with write-only access, creating it if necessary.
  2. `write` writes raw bytes (`char` array/string) to the file.

3. `close` closes the file descriptor.

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

int main() {
  int fd = open("example2.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

  if (fd == -1) {
    perror("Error opening file");
    return 1;
  }

  const char *text = "Hello, File Descriptor!\n";
  write(fd, text, 25);  // Write raw bytes to the file

  close(fd);  // Close the file descriptor

  return 0;
}
```

- Testing example 2:

```
gcc fileh2.c -o fileh2
./fileh2
ls -lt example2.txt fileh2.c fileh2
cat example2.txt
```