SYS1

Network programming

20xx

Version 1

Jules AUBERT <jules1.aubert@epita.fr>

Contents

1 Introduction

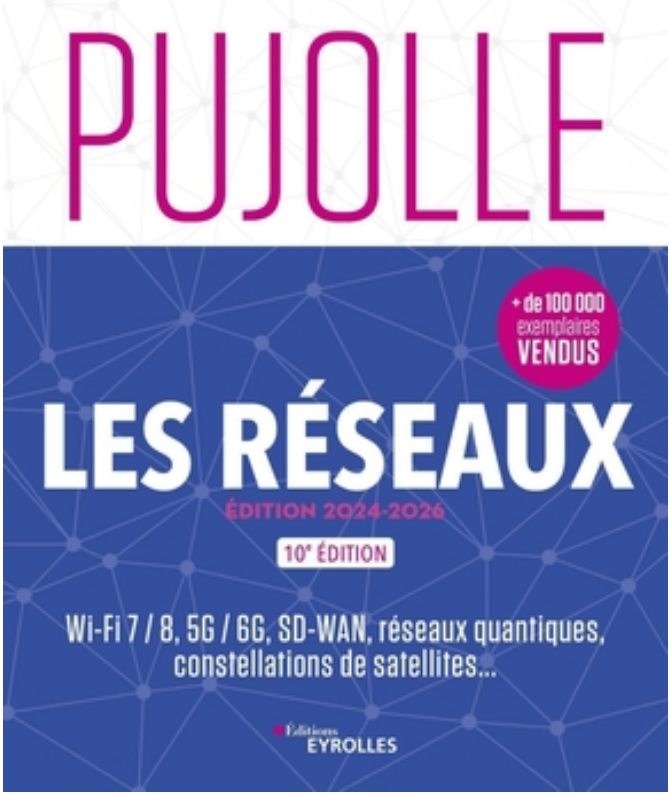
1.1 A finally good tutorial about network programming

The purpose of this course is to help you grasp the fundamental principles of network programming. You may have already experimented with online tutorials to build a client and a server, only to find yourself overwhelmed by code you don't fully understand. This course is your opportunity to change that. Together, we will build minimalist client and server programs in C, and take the time to demystify every function, every parameter, and every concept involved.

1.2 Resources

1.2.1 Book

If you get interested into IT networking, I recommend the French book **Les réseaux - Guy Pujolle**



ISBN: 978-2-416-01433-8

1.2.2 Beej's guide

Beej's guide on network programming is maybe the most well known guide over the Internet. But it is not that easy to understand, and this tutorial will help you dive into the Beej's guide.

https://beej.us/guide/bgnet

1.3 IP

An IP address is a unique identifier assigned to a device on a network. It allows devices to locate and communicate with each other over the networks.

There are two main versions: IPv4 and IPv6. In this tutorial, we will use IPv4 only.

1.4 Port

A network port is a logical endpoint for communication. It helps identify a specific process or service on a device connected to a network.

Each port is associated with a number, from 0 to 65535, and works alongside an IP address. Together, they form a socket. Most known services are reserved to some ports. Examples:

- Port 20/21: FTP
- Port 22: SSH
- Port 23: Telnet
- Port 25: SMTP
- Port 53: DNS
- Port 67/68: DHCP
- Port 80: HTTP
- Port 110: POP3
- Port 143: IMAP3
- Port 443: HTTPS

1.5 MAC

A MAC (Media Access Control) address is a hardware identifier, assigned to a network interface. It is a 48-bit address.

Unlike IP addresses, MAC addresses are fixed (burned into the hardware) and work as the **link layer** (Ethernet, Wi-Fi, ...).

Think of it like a serial number for your network device.

1.6 Socket

For this tutorial, a socket is an association of an IP address and a port number. In the code, a socket is just a file descriptor. You can **read(2)** it, **write(2)** on it and **close(2)** it. The only difference is that we will use **socket(2)** to create it and use **recv(2)** and **send(2)** on it, but I assure you that **read(2)** and **write(2)** work on it. I let you read the manpages about them and see for yourself.

How is that possible? Sockets are file descriptors, but do not worry, the kernel knows it is a resource for a network interface and not a storage interface.

1.7 Tools

1.7.1 ip

ip(1) is the tool on Linux to work with your network hardware. Execute **ip a** to have all the information about your network hardware. Remember to also **man** it.

1.7.2 ipcalc

This course is not about networking but really network programming. I will not dive into networking fundamentals. If you want a good tool to do network computing, you have **ipcalc(1)**.

```
$ ipcalc 192.168.69.42/14
Address:   192.168.69.42        11000000.101010 00.01000101.00101010
Netmask:   255.252.0.0 = 14     11111111.111111 00.00000000.00000000
Wildcard:  0.3.255.255          00000000.000000 11.11111111.11111111
=>
Network:   192.168.0.0/14       11000000.101010 00.00000000.00000000
HostMin:   192.168.0.1          11000000.101010 00.00000000.00000001
HostMax:   192.171.255.254      11000000.101010 11.11111111.11111110
Broadcast: 192.171.255.255      11000000.101010 11.11111111.11111111
Hosts/Net: 262142                Class C, In Part Private Internet
$
```

2 Network programming

2.1 Goal

The goal here is for you to code a client and a server in C, both acting kinda like **netcat(1)**.

2.2 Server

2.2.1 The code

Here is the complete code. We will unwrap it in the following pages. Call it **server.c**.

```c
#include <err.h>
#include <netdb.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    if (argc != 3)
        return 1;

    int ret = 0;
    int sockfd = 0;
    int clientfd = 0;
    char buff[4096] = { 0 };
    ssize_t len = 0;
    struct addrinfo hints = {0};
    struct addrinfo *res = NULL;
    struct addrinfo *r = NULL;

    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if ((ret = getaddrinfo(argv[1], argv[2], &hints, &res)) != 0)
        errx(EXIT_FAILURE, "getaddrinfo: %s", gai_strerror(ret));

    for (r = res; r; r = r->ai_next)
    {
        if ((sockfd = socket(r->ai_family, r->ai_socktype, r->ai_protocol)) == -1)
            continue;

        if (bind(sockfd, r->ai_addr, r->ai_addrlen) == 0)
            break;

        close(sockfd);
    }

    if (r == NULL)
        errx(EXIT_FAILURE, "Cannot bind to the service.");

    freeaddrinfo(res);

    listen(sockfd, SOMAXCONN);

    clientfd = accept(sockfd, NULL, NULL);

    while (true)
    {
        len = recv(clientfd, buff, sizeof (buff) - 1, 0);
        if (len <= 0)
            break;

        for (ssize_t i = 0; i < len; ++i)
        {
            if ('a' <= buff[i] && buff[i] <= 'z')
                buff[i] -= 32;
        }

        buff[len] = '\0';
        printf("%s", buff);
```

```
    }

    close(sockfd);
    close(clientfd);

    return 0;
}
```

The Makefile:

```
CPPFLAGS = -D_POSIX_C_SOURCE=200112L
CFLAGS = -std=c99 -pedantic -Wall -Wextra -Wvla -Werror
OBJ = server.o
BIN = $(OBJ:.o=)

all: $(BIN)

$(BIN): $(OBJ)

clean:
  $(RM) $(OBJ) $(BIN)

# $ // Do not mind this line, it is for my LaTeX document
```

The **CPPFLAGS** is used for the **getaddrinfo** function. (**man getaddrinfo**).

It is a simple server, it will wait for a unique client and print the data on stdout until the client disconnect.

Let's unwrap this!

```
#include <err.h>
#include <netdb.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

A new header has appeared! It is **netdb.h(0P)**. You can read its manpage. Thanksfully on Linux, in includes more headers we need, like **sys/socket.h**.

```
int main(int argc, char *argv[])
{
    if (argc != 3)
        return 1;

    int ret = 0;
    int sockfd = 0;
    int clientfd = 0;
    char buff[4096] = { 0 };
    ssize_t len = 0;
    struct addrinfo hints = {0};
    struct addrinfo *res = NULL;
    struct addrinfo *r = NULL;
```

You will use two parameters:

1. argv[1]: the network interface(s) to listen to
2. argv[2]: the port to connect to

The following variables are present:

- **ret**: used to check the return values of some functions

- **sockfd**: the socket of the server
- **clientfd**: the socket of the client
- **buff**: the buffer to receive data
- **len**: the data len received from the client
- **hints**: hints to use to connect and listen on the network
- **res**: the results to connect to our network interface(s)
- **r**: a sentinel to iterate on the results and try to create a socket from them

```
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
```

Here, we say that we feed the server with *hints* to know what kind of network communication we want to use.

AF_UNSPEC means we are ready to use IPv4 and IPv6. The version we use is **UNSPEC**IFIED.

SOCK_STREAM means we will use **TCP**.

```
    if ((ret = getaddrinfo(argv[1], argv[2], &hints, &res)) != 0)
        errx(EXIT_FAILURE, "getaddrinfo: %s", gai_strerror(ret));
```

Here we are using **getaddrinfo** to get the info on how to *connect* to the resources we need. Here the server will *connect* to its network interface using the IP and the port.

If you want to listen to **all** interface, you can use the address **0.0.0.0**. If you want to listen to the localhost address, you can use **127.0.0.1** or **localhost**.

For the port, you need to use a port **above** 1023. Why? Because the ports from 0 to 1023 are reserved and usually[1], only root can use them freely.

Finally, we send the **hints** to tell **getaddrinfo** how to connect to the resource and also **the address of res** to have a **linked linst** we will use later to create the socket.

Do you know what returns **getaddrinfo** in case of error? Try it yourself:

```
$ ./server error 4242
server: getaddrinfo: Name or service not known
$
```

Do you feel the familiarity of the message?
No?
Try to **ping error** and see for yourself. :)

[1] unless you play with **capabilities(7)**

```
for (r = res; r; r = r->ai_next)
{
    if ((sockfd = socket(r->ai_family, r->ai_socktype, r->ai_protocol)) == -1)
        continue;

    if (bind(sockfd, r->ai_addr, r->ai_addrlen) == 0)
        break;

    close(sockfd);
}

if (r == NULL)
    errx(EXIT_FAILURE, "Cannot bind to the service.");

freeaddrinfo(res);
```

This part is big. After getting a list of results, we iterate on it using **r** as a sentinel. We first try to create the **socket**. If it fails, we loop again, if it works we try to **bind** the socket to the resource (the network interface(s)). If if fails, we close the socket and try again. If it works, we get out of the loop and free the linked list. In case of error, we display an error message.

```
listen(sockfd, SOMAXCONN);

clientfd = accept(sockfd, NULL, NULL);
```

This is straightforward. We listen our socket. **SOMAXCONN** is a constant that defines the maximum number of pending connections in a server's listen queue.

If you want to see the value of SOMAXCONN, just **cat(1)** on **/proc/sys/net/core/somaxconn**.

Once we have a connection, we use **clientfd** as a socket for the client to accept the connection.

The last two parameters are used to get the client's IP and port, and to get the size of the address structure to hold it (IPv4 has a different size than IPv6).

```
while (true)
{
    len = recv(clientfd, buff, sizeof (buff) - 1, 0);
    if (len <= 0)
        break;

    for (ssize_t i = 0; i < len; ++i)
    {
        if ('a' <= buff[i] && buff[i] <= 'z')
            buff[i] -= 32;
    }

    buff[len] = '\0';
    printf("%s", buff);
}
```

Here, we are looping on the **recv** syscall. If you look closely at its manpage, you can read that it works the same as **read(2)** syscall. How is that possible? Because sockets are just **file descriptors**.

If the len is zero, then the client disconnected. If the len is lower than 0, then an error occured.

Then, all we do is **uppercase** the buffer and display it on the stdout of the server.

```
close(clientfd);
close(sockfd);

return 0;
```

Finally, once the client disconnects, we get out of the loop and close the file descriptors.

2.2.2 Test

Let's test our server! If you do not have your client yet, you can use **netcat**. Is is the swiss army knife of network tools. You can use it as a client and also as a server.

First, compile and execute your server.

```
# Shell 1
$ ./server 127.0.0.1 4242

# Shell 2
$ nc 127.0.0.1 4242
hello apping

# Shell 1
HELLO APPING

# Shell 2
CTRL^C
$

# Shell 1
$
```

2.3 Client

2.3.1 The code

Here is the complete code. We will unwrap it in the following pages. Call it **client.c**.

```c
#include <err.h>
#include <netdb.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    if (argc != 4)
        return 1;

    int ret = 0;
    int sockfd = 0;
    struct addrinfo hints = {0};
    struct addrinfo *res = NULL;
    struct addrinfo *r = NULL;

    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if ((ret = getaddrinfo(argv[1], argv[2], &hints, &res)) != 0)
        errx(EXIT_FAILURE, "getaddrinfo: %s", gai_strerror(ret));

    for (r = res; r; r = r->ai_next)
    {
        if ((sockfd = socket(r->ai_family, r->ai_socktype, r->ai_protocol)) == -1)
            continue;

        if (connect(sockfd, r->ai_addr, r->ai_addrlen) == 0)
            break;

        close(sockfd);
    }

    if (r == NULL)
        errx(EXIT_FAILURE, "Cannot connect to the service.");

    freeaddrinfo(res);
```

```
    send(sockfd, argv[3], strlen(argv[3]), 0);

    close(sockfd);

    return 0;
}
```

It is a simple client, it will send a message and disconnect. You need to add yourself the code to have a better client, but it will be pure C and not network programming, you should be able to add this logic yourself.

Let's unwrap this!

```
#include <err.h>
#include <netdb.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```
int main(int argc, char *argv[])
{
    if (argc != 4)
        return 1;

    int ret = 0;
    int sockfd = 0;
    struct addrinfo hints = {0};
    struct addrinfo *res = NULL;
    struct addrinfo *r = NULL;
```

You will use three parameters:

1. argv[1]: the IP to connect to
2. argv[2]: the port to connect to
3. argv[3]: the message to send to the server

The following variables are present:

- **ret**: used to check the return values of some functions
- **sockfd**: the socket of the client to communicate with the server
- **hints**: hints to use to connect and listen on the network
- **res**: the results to connect to our network interface to communicate to the server
- **r**: a sentinel to iterate on the results and try to create a socket from them

```
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
```

Same logic we used on the server, but on the client this time.

```
    if ((ret = getaddrinfo(argv[1], argv[2], &hints, &res)) != 0)
        errx(EXIT_FAILURE, "getaddrinfo: %s", gai_strerror(ret));
```

Same logic we used on the server, but on the client this time.

```
    for (r = res; r; r = r->ai_next)
    {
        if ((sockfd = socket(r->ai_family, r->ai_socktype, r->ai_protocol)) == -1)
            continue;

        if (connect(sockfd, r->ai_addr, r->ai_addrlen) == 0)
            break;

        close(sockfd);
    }

    if (r == NULL)
        errx(EXIT_FAILURE, "Cannot connect to the service.");

    freeaddrinfo(res);
```

It is **almost** the same logic we used on the server. Can you spot the difference?

Instead of using **bind**, we use **connect**, to try to connect to the server. The logic is the same apart of that.

```
    send(sockfd, argv[3], strlen(argv[3]), 0);
```

We send the **message** to the server.

```
    close(sockfd);

    return 0;
```

We close the socket and stop the program.

2.3.2 Test

Let's try the client! First without our server, using **netcat**, and then with our server.

First of all, execute **netcat** as a server, then execute the client.

```
# Shell 1
$ nc -lnvp 4242 # Listen on the port 4242
Listening on 0.0.0.0 4242

# Shell 2
$ ./client 127.0.0.1 4242 "hello apping"
$

# Shell 1
Connection received on 127.0.0.1 37790
hello apping
$
```

You may have a slightly different ouput. It is fine.

Time to get some fun, we will use both of our server and client!

```
# Shell 1
$ ./server 127.0.0.1 4242
```

```
# Shell 2
$ ./client 127.0.0.1 4242 "hello apping"
$

# Shell 1
HELLO APPING
$
```

2.4 Going further

A client can **recv(2)** and **send(2)**, aswell for a server. It is not because a server is called a server that it has to be only recv'ing. The same logic applies to a client.

With these information, can you try to make a better client looping and exchanging data with the server? Begin simple: make the server an **uppercase echo server**, sending back the data to the client in uppercase. Then try to add more complexity and interests into it.

3 More

3.1 Goal

The goal here is to give you more resources about network programming. I will not dive into details, as this is just a SYS1 course and not a **TCOM** nor **GISTRE** course. If you are interested, you are invited to search for yourself information about it (manpages are a good start).

3.2 Multiplexing

The real problem about having a server is: How do I accept multiple clients? In fact, there is a well known problem named the **C10k problem**: How do I accept 10 thousands clients on my server?

https://en.wikipedia.org/wiki/C10k_problem

People used to use **threads** to handle several clients. This only confuses the code and make it a spaghetti code. Some people used to use **forks** to handle several clients. This works for 10.. 100... 1000 clients? Anyway, it is not a good option.

An implementation for multiplexing was **select(2)** but was not that good.

A standardized version came, **poll(2)** but some system did not really implemented it and use their own way to multiplexing. And is was not that great either.

3.3 Epoll

To address **poll**'s limitation, Linux introduced **epoll(7)**, a high-performance API for monitoring large number of file descriptors.

3.4 io_uring

io_uring(7) is a modern Linux interface introduced in kernel version 5.1 that enables high-performance, fully asynchronous[2] I/O operations with minimal system call overhead.

3.5 Data Plane Development Kit

DPDK is a set of high-performance libraries and drivers that allow **user-space programs** to bypass the Linux kernel and direcly access network interfaces for extremly fast packet processing.

[2] do you want a joke about asynchrone? Knock-knock. An asynchronous joke. Who is it?