Linux System Programming

Part A. Introduction
Kernel space vs User space

Kernel space offers entry points to user space through system calls, there are around 250 system calls. User space program either :

- access kernel programs directly through system call, such as `write()`
- access kernel programs indirectly through standard library, such as `printf()`
- do not access kernel programs, operator entirely in user space without diving in kernel, such as `sqrt()`

System calls are direct request for service from kernel, they are fast as there is no wrapper in between user and kernel.

- system calls are available in C and in Python (`import os`)
- system calls usually involve bitwise OR among various options
- system calls do not throw exception, hence need to check return code (on every system call)

Shell command is not system call :

- shell command is just compiled program (executable) placed under `/usr/bin`
- shell command has nothing to do with kernel (unless it invokes system call internally)

Part B. Interprocess communication

We will try interprocess with :

1. file
2. database *(read postgres / sqlite3 doc)*
3. socket *(see next section, together with epoll)*
4. unnamed pipe and named pipe
5. shared memory
6. zero message queue

B4. Unnamed pipe and named pipe

We need to include these headers for pipe and shared memory :

```
#include <fcntl.h>
#include <unistd.h> // for getpid and fork
#include <sys/stat.h>
#include <sys/types.h>
```

Please dont forget to link with real time library `-lrt`

Unnamed pipes are created in a pair (`fd[0]` for reader and `fd[1]` for writer), copyied by fork, we then :

- close the unused `fd[0]` in producer
- close the unused `fd[1]` in consumer

```
void producer_pipe(int& fd, const std::string& comment)
{
    std::cout << "\nEnter command : ";
    while(true)
    {
        std::string str;
        std::cin >> str;
        ::write(fd, str.c_str(), str.size());
        if (str == "quit") break;
    }
    ::close(fd);
}

void consumer_pipe(int& fd, const std::string& comment)
{
    char buffer[1024];
    while(true)
    {
        auto bytes = ::read(fd, buffer, 1024);
        std::string str(buffer, bytes);
        std::cout << "\nreceive msg = " << str << std::flush;

        if (str == "quit") break;
        std::cout << "\nEnter command : " << std::flush;
    }
    ::close(fd);
}

// main function
int fds[2];
::pipe(fds); // create both pipes together

if (fork() > 0)
{
    ::close(fds[0]); // close unused fd
    producer_pipe(fds[1], "unnamed pipe"); // fds[1] is for writer
}
else
{
    ::close(fds[1]); // close unused fd
    consumer_pipe(fds[0], "unnamed pipe"); // fds[0] is for reading
}
```

The above code for unnamed pipe is tested and work properly. Hovever named pipe doesn't. Still debugging ...

```
if (is_producer)
{
//  ::mkfifo("~/my_pipe", S_IFIFO|0640); // 0777
    int fd = ::open("~/my_pipe", O_CREAT|O_WRONLY); // fd=-1, why?
    ipc::producer_pipe(fd, "named pipe");
}
else
{
//  ::mkfifo("~/my_pipe", S_IFIFO|0640); // 0777
    int fd = ::open("~/my_pipe", O_RDONLY); // fd=-1, why?
    ipc::consumer_pipe(fd, "named pipe");
}
```

B5. Shared memory

Shared memory is a little more complicated, as we need to use :

- first few bytes as counter of message (synchronization mechanism between producer and consumer)
- second few bytes as length of new message
- other bytes as message content itself

Besides, we perform memory mapping to `fd`, so that read/write to the VAS memory is equivalent to read/write to shared memory. Recall that VAS is memory allocated to a process, shared memory lives outside a process, hence can be accessed by inter-process.

```
const std::string buffer_name = "ABC";
const int buffer_size = 1024;

void producer_shared_memory()
{
    auto fd   = ::shm_open(buffer_name.c_str(), O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    auto res  = ::ftruncate(fd, buffer_size); // truncate buffer size to fixed length
    auto addr = ::mmap(NULL, buffer_size, PROT_WRITE, MAP_SHARED, fd, 0);
    std::memset(addr, 0, buffer_size);
    std::uint8_t count = 0;

    while(true)
    {
        std::cout << "\nEnter command : " << std::flush;
        std::string str;
        std::cin >> str;
        std::uint8_t length = (std::uint8_t)str.size();

        ++count;
        std::memcpy(reinterpret_cast<std::uint8_t*>(addr)+2, str.c_str(), str.size());
        std::memcpy(reinterpret_cast<std::uint8_t*>(addr)+1, &length, 1);
        std::memcpy(reinterpret_cast<std::uint8_t*>(addr)+0, &count, 1);

        if (str == "quit") break;
    }
    res = ::munmap(addr, buffer_size);
    fd  = ::shm_unlink(buffer_name.c_str());
}

void consumer_shared_memory()
{
    auto fd   = ::shm_open(buffer_name.c_str(), O_RDONLY, S_IRUSR | S_IWUSR);
    auto addr = ::mmap(NULL, buffer_size, PROT_READ, MAP_SHARED, fd, 0);
    std::uint8_t count = 0;

    while(true)
    {
        if (reinterpret_cast<std::uint8_t*>(addr)[0] == count+1)
        {
            ++count;
            std::uint8_t length = reinterpret_cast<std::uint8_t*>(addr)[1];
            std::string str(reinterpret_cast<char*>(addr)+2, length);
            std::cout << "\nreceive msg" << (int)count << " : " << str << "\n" << std::flush;

            if (str == "quit") break;
            std::cout << "\nEnter command1 : " << std::flush;
        }
    }
}

// main function
if (fork() > 0)
{
    ipc::producer_shared_memory();
}
else
{
    ipc::consumer_shared_memory();
}
```

B6. ZeroMQ - Brokerless messaging system

Installation

Lets install `libzmq` and its C++ wrapper.

```
>> sudo apt install libzmq3-dev
>> find / -name libzmq* 2>/dev/null
/usr/lib/x86_64-linux-gnu/libzmq.a
/usr/lib/x86_64-linux-gnu/libzmq.so
```

Lets do some experiments.

Part C. From Hub / Switch / Router to Socket programming

Please read https://www.pwnthebox.net/packet/overflow/2019/01/27/packet-overflow.html

Here is the TCP stack (in 5 layers). Each layer serves a special purpose. Lets go through them in bottom up approach.

- Physical layer and data link layer are sometimes collectively known as Network access layer, served by hub and switch.
- Allocation of IP addresses for various network service providers can be found by keywords `ipinfo + Hong Kong ASN summary`.
- Application programming interface API lies in between transport layer and application layer

Layer	Hardware / Software	Protocol	Message unit
application layer	user space	HTTP, FTP, SSH, OCG	
transport layer	*OS* kernel space	TCP, UDP	
network layer	*OS* kernel space, router	IP	packet
data link layer	*NIC* (network interface card), hub and switches	Ethernet, Wifi	dataframe
physical layer	*NIC* (network interface card), hub and switches		

Layer	Purpose	Address
application layer	application specific purpose	-
transport layer	inter process communication across internet	port *(65536 ports in a machine)*
network layer	inter machine communication across internet	4 bytes *IP* address *(4G machines in internet)*
data link layer	inter machine communication within intranet	6 bytes *MAC* address *(unique physical address)*
physical layer	-	-

C1. Hub / Switch / Router

When two machines talk together :

- common protocols are needed (like ethernet, wifi, *IP*, *TCP*, *UDP*, *HTTP*, *FTP* etc)
- intermediate connecting devices are needed (like hubs, switches and routers) forming some topologies (star / ring / bus ...)

The simplest way to connect multiple machines to form a local network is use ***hubs*** as the intermediate connecting device. Hub is a repeater, which repeats the signal it received from any machine to all other machines indifferently. It results in 2 problems, namely *(1)* breaking confidentiality and *(2)* unnecessary traffic for the network.

The solution is to replace the intermediate device by a ***switch***, then connect each machine to the switch via a *NIC* (network interface card), which is installed or embeded in each machine. Each *NIC* has a 6 bytes globally unique number, called *MAC* address (media access control address). On setting up the switch, it must be able to create a table like :

MAC address of machine	Port number on the switch
00:15:5d:93:4f:fe	00
00:12:a3:04:d7:1e	01
00:27:3b:b3:e5:3a	02
...	...

On top of hardware setup, we build a data link layer. Here is the journey of a message from machine A to B in the network :

- machine A appends header to the data, the header should contain the *MAC* address of source A and destination B
- the header together with the data is known as a data frame in the data link layer
- the data frame is sent to the switch
- the switch extracts the destination *MAC* address from the data frame
- the switch then finds the destination port from the above table and sends it to target port, ***NO*** unnecessary traffic created
- the switch needs to solve other problems such as collision, i.e. when multiple machines try to send simultaneously ...

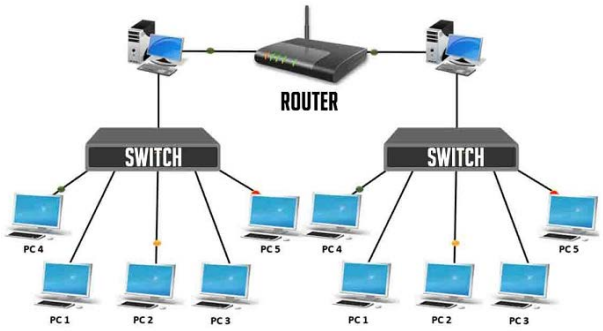
Two common data link layer protocols include ...

- ethernet and wifi (they are two different protocols in the same layer)
- one machine may have multiple *NICs*, hence multiple *MAC* addresses
- *NIC* is simply called interface, my PC has 3 basic interfaces : one software interface (`127.0.0.1`), two hardware interface :

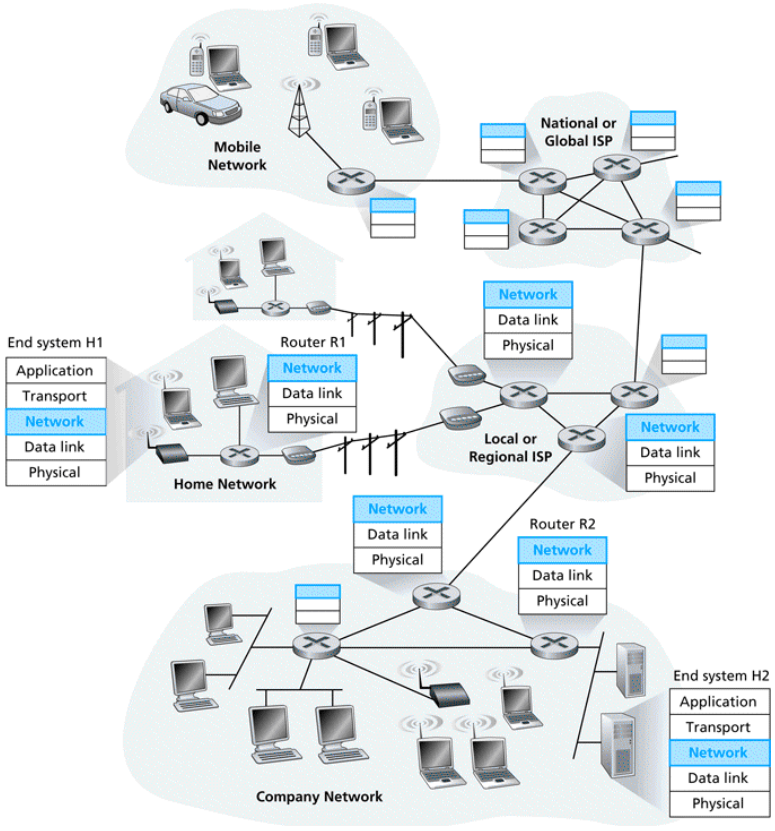
```
>> ifconfig

eth3:   inet  172.17.85.97      inet6 fe80::50e3:8e:c232:e179      ether 00:15:5d:6a:4f:3e (Ethernet)
wifi0:  inet  192.168.8.8       inet6 fe80::a8a4:afcc:5aa9:437     ether c0:a5:dd:f6:f3:ab (Ethernet)
local:  inet  127.0.0.1         inet6 ::1                          loop  (Local Loopback)
```

Now we are going to build a bigger network and eventually the internet by :

- connecting multiple switches to a router, then ...



- connecting millions of routers together to form a graph, the *graph-of-routers* is the INTERNET



where  means router

In order to send messages from one machine to another machine within the same local network hosted by a switch, we need *MAC* address of the source and destination, the transmission can be done purely inside the data link layer. However if we send messages from one machine to another machine connected to different switches under the same router, or even two machines under different routers, we need to escalate to one layer higher, known as the network layer, running *IP* protocol, using 4 bytes unique *IP* address to identify machines. With special routing algorithm, given the source and destination *IP* address, the router-network can route the message to the target destination regardless how big the internet is, but this generalizability comes with costs, the *IP* protocol is not reliable, it provides best effort service only, it does not guarantee 100% successful rate, thus it is the responsibility of higher layer to implement error detection.

[Remark 1] Different scenarios :

- Send data to another machine under same switch : data link layer is fine, no router is involved
- Send data to another machine under diff switches of same router : network layer needed, only parent router is involved
- Send data to another machine under diff routers : network layer needed, part of router-network is involved

[Remark 2] Why do we use *IP* address (instead of *MAC* address) for network layer?

- *MAC* address is called physical address, hard coded in NIC
- *IP* address is called logical address it is assigned by *ISP* (Internet Service Provider), can be changed from time to time
- *MAC* address is unknown to the routing algorithm, *MAC* address is thus non-routeable
- *IP* address is known to the routing algorithm, *IP* address is thus routeable

[Remark 3] Does *IP* address enough?

- No, it supports 4G machines only.
- Thus there are public *IP* address (the one we mention above) and private *IP* address (in local network scope).
- There are 3 classes (3 ranges) of private *IP* address, the most common one starts with `192.168.x.x`

C2. TCP layer and application layer

Consider the scenairo, when we have multi-process running in same machine, all making connections to internet. If these processes send queries to different sites simultaneously, when those replies return, how does the computer knows which process should each reply be forwarded to? This problem is solved by adding transport layer on top, using port as the unique identifier for each process in the computer (hence it supports 65536 processes having connections simultaneously). Therefore, we need to bind a port to socket of a server process (but not a client process).

How does the special routing algorithm work? *[My speculation only]*

- When routers are connected together, each router knows its direct neighbours (it does not know about the whole graph).
- With *IP* protocol and *IP* addresses, each router learns something from its neighbours, for the sake of routing algorithm.
- When machine A sends message to machine B in the other side of the world, go top-down way first, then bottom-up :

`machine_A::OS::transport_layer`	append transport header (includes *A's port num* and *B's port num*) to the data
`machine_A::OS::network_layer`	append network header (includes *A's IP addr* and *B's IP addr*) to the data
`machine_A::NIC::datalink_layer`	append datalink header (includes *A's MAC addr* and *routerA's MAC addr*) to the data
`router_A`	extract *B's IP addr* and *routerA's MAC addr* on finding that *B* is not under *routerA*, then find next-router according to routing algo replace *routerA's MAC addr* by *next-router's MAC addr*
`next_A`	extract *B's IP addr* and *next-router's MAC addr* on finding that *B* is not under *next-router*, then find next-next-router replace *next-router's MAC addr* by *next-router's MAC addr*
`repeat until reaching router_B ...`	
`router_B`	extract *B's IP addr* and *routerB's MAC addr* on finding that *B* is under *routerB*, then forward to machine *B*
`machine_B::NIC::datalink_layer`	extract *B's MAC addr* and verify
`machine_B::OS::network_layer`	extract *B's IP addr* and verify
`machine_B::OS::transport_layer`	extract *B's port* and forward to corresponding process

Please note that source machine does not know *MAC* address of final destination machine, therefore the datalink layer fills with the *MAC* address of next router instead. Common transport layer protocol includes TCP and *UDP*. *TCP* offers error detection feature to deal with the unreliability of network layer.

On top of transport layer, we have application layer, with which various application protocols. The network API is an interface for user space program to communicate with kernel space.

Socket binding

When we do socket programming, we need to bind the server socket to an address plus a port. What does that mean?

For machines in trading firm, there are multiple network interface cards (with different IP) in one machine, we need to select one of them for a socket (some NICs may access datafeed, some may not, some may access OCG, or OAPI etc). We also need to select one port as the process identifier.

C3. Socket programming

Lets do some socket programming in Python. We need to build a *TCP* echo server and a client.

- A socket becomes a passive socket if we invoke `listen` on it, otherwise it is an active socket.
- Passive socket cannot read / write, it is for accepting incoming connection only.
- Active socket can read / write.

Here are the steps for server :

1. instantiate a socket
2. bind the socket to a port
   (1) assigning a network interface card and
   (2) assigning a port to this server process
3. listen on the socket
4. accept incoming connection, new active socket is spawned

Here are the steps for client :

1. instantiate a socket
2. connect to specific server with *IP* and *port*

```
#! /usr/bin/python
import sys
import socket

def server(port):
    listen_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = ('localhost', port)
    listen_sock.bind(server_address)
    listen_sock.listen(1) ### Queue size is 1, only first client is connected, extra clients are dropped. ###

    try:
        while True: # Outer-loop
            sock, client_address = listen_sock.accept()
            print('receive connection from ', str(client_address))

            while True: # Inner-loop
                data = sock.recv(100) ### Buffer size is 100 bytes, extra bytes are read in next iteration ###
                print('received:', data.decode('utf-8'))
                if data:sock.sendall(data)
                else:
                    print('disconnection from ', client_address)
                    break

    finally: sock.close()

def client(ip, port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = (ip, port)
    sock.connect(server_address)

    try:
        while True : # Outer-loop
            message = input('Enter message : ')
            sock.sendall(bytes(message.encode('utf-8'))) ### Note (1) encode str to utf-8, (2) convert str to bytes ###

            amount_expected = len(message)
            amount_received = 0
            while amount_received < amount_expected: # Inner-loop
                data = sock.recv(16)
                amount_received += len(data)
                print('received:', data.decode('utf-8')) ### Note (1) decode bytes to utf-8 before printing ###

    finally: sock.close()

########################################
# Run server >> tcp_echo 23456
# Run client >> tcp_echo 127.0.0.1:23456
########################################
def main():
    if len(sys.argv) == 2:
        index = sys.argv[1].find(':')
        if index>=0 :
            ip   = sys.argv[1][:index]
            port = sys.argv[1][index+1:]
            client(ip, int(port))
        else : server(int(sys.argv[1]))

if __name__ == '__main__' : main()
```

C4. Epoll - from standard input

Please read : https://suchprogramming.com/epoll-in-3-easy-steps

Epoll offers an effective way to poll multiple IO file descriptors asychronously. It involves 3 steps.

1. create epoll handle (which is also a file descriptor)
2. register one or multiple file descriptors (sockets or pipes) to epoll handle
3. invoke `epoll_wait` to wait until one of the file descriptors, the return `struct` will specify the file descriptor to be handle

- unlike `select` model, which requires user to scan through all file descriptors, `epoll` is *O(1)* while `select` is *O(N)*

Control function `epoll_ctl` is a common pattern in Linux programming, in which we need to issue a command :

- `EPOLL_CTL_ADD` for adding file descriptor
- `EPOLL_CTL_MOD` for modifying file descriptor
- `EPOLL_CTL_DEL` for deleting file descriptor

```
#include<unistd.h>       // for close(), read()
#include<fcntl.h>        // for open() options
#include<sys/epoll.h>    // for epoll_create1(), epoll_ctl(), struct epoll_event

#define MAX_EVENTS 5
#define READ_SIZE 10

void test_epoll()
{
    // ***************************** //
    // *** Step 1 : Create Epoll *** //
    // ***************************** //
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) { std::cout << "Failed to create epoll"; return; }

    // ***************************** //
    // *** Step 2 : Registration *** //
    // ***************************** //
     auto fd = open("./temp", O_CREAT | O_TRUNC | O_RDWR); // fopen() returns FILE*, while open() returns FD
     if  (fd == -1) { std::cout << "Failed to open file"; return; }

    epoll_event acquire_event0;
    epoll_event acquire_event1;
    acquire_event0.events  = EPOLLIN; // READY-TO-READ to event
    acquire_event0.data.fd = 0;       // 0 = std::cout
    acquire_event1.events  = EPOLLIN; // READY-TO-READ to event
    acquire_event1.data.fd = fd;

     // We can ADD/MOD/DEL
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, 0, &acquire_event0))
    {
        close(epoll_fd);
        std::cout << "Failed to add fd"; return;
    }

    // ****************************** //
    // *** Step 3 : Wait for event *** //
    // ****************************** //
    epoll_event received_events[MAX_EVENTS];

    bool running = true;
    while(running)
    {
        // When events are detected, they are copied ...
        // from kernel-event-table (in kernel space)
        // to received_events (in user space)
        auto event_count = epoll_wait(epoll_fd, received_events, MAX_EVENTS, 30000); // 30000 ms waiting

        for(std::uint32_t n=0; n!=event_count; ++n)
        {
            char buffer[READ_SIZE + 1];
            size_t bytes = read(received_events[n].data.fd, buffer, READ_SIZE); // no null char in buffer

            std::string str(buffer, bytes);
            std::cout << bytes << " bytes from fd " << received_events[n].data.fd << " : " << str;

            if (str == "stop\n") running = false; // Remark : newline is included
        }
    }

    // *** Clear *** //
    close(epoll_fd);
}
```

C5. Epoll - from TCP socket

Now, let's apply epoll to read/write of BSD socket to build a TCP echo server that listens to multiple clients.

Here are the steps for server, there are more steps (step A-F) as compared with python (step 1-4 in part C3) :

A. hints for addrinfo query
B. query for addrinfo results
C. iterate list of addrinfo results
1. create a socket
2. bind socket to a port
D. set socket as non blocking socket
3. listen on the socket
4. accept incoming connection, new active socket is spawned

```
int create_and_bind(std::uint16_t port)
{
    addrinfo hints;
    addrinfo *result;

    // Step A. initial address hints
    memset(&hints, 0, sizeof(addrinfo));
    hints.ai_socktype = SOCK_STREAM;   // request TCP socket
    hints.ai_family   = AF_UNSPEC;     // request both IPv4 and IPv6
    hints.ai_flags    = AI_PASSIVE;

    // Step B. get address results
    std::string port_str = std::to_string(port);
    if (auto status = getaddrinfo(NULL, port_str.c_str(), &hints, &result) != 0) return -1;

    // Step C. iterate address results
    for(const auto* result_ptr = result; result_ptr!=NULL; result_ptr = result_ptr->ai_next)
    {
        // Step 1
        int fd = socket(result_ptr->ai_family,
                        result_ptr->ai_socktype,
                        result_ptr->ai_protocol);
        if (fd == -1) continue;

        // Step 2
        if (bind(fd, result_ptr->ai_addr,
                     result_ptr->ai_addrlen) == 0)
        {
            freeaddrinfo(result);
            return fd;
        }
        close(fd);
    }
    freeaddrinfo(result);
    return -1;
}

// *** Step D *** //
int set_socket_non_blocking(int fd)
{
    int flags = fcntl(fd, F_GETFL, 0); // Get flags
    if (flags == -1) return -1;

    flags |= O_NONBLOCK;
    if (fcntl(fd, F_SETFL, flags) == -1) return -1; // Set flags
    return 0;
}

// *** Step A-C,1-2,D,3 *** //
int create_passive_socket(std::uint16_t port)
{
    int FD_passive = create_and_bind(port);
    if (FD_passive == -1)                          return -1;
    if (set_socket_non_blocking(FD_passive) == -1) return -1;
    if (listen(FD_passive, SOMAXCONN) == -1)       return -1; // *** Step 3 *** //

    return FD_passive;
}

// *** Helpers *** //
int register_fd_to_epoll(int FD_epoll, int FD)
{
    struct epoll_event event;
    event.data.fd = FD;
    event.events = EPOLLIN | EPOLLET;
    return epoll_ctl(FD_epoll, EPOLL_CTL_ADD, FD, &event);
}
```

Here are the steps for epoll, same as (step 1-3 in part C4) :

1. create epoll handle
2. register one or multiple file descriptors (sockets or pipes) to epoll handle
3. invoke `epoll_wait` to wait until one of the file descriptors, the return `struct` will specify the file descriptor to be handle

There are 3 possible types of events in this echo server :

- new client connects in *then add the corresponding file descriptor to epoll handle*
- existing client disconnected *then close the corresponding file descriptor*
- existing client sending messages *then echo the message*

```
// *** Testing *** //
void test_epoll_socket(std::uint16_t port)
{
    int FD_passive = create_passive_socket(port);
    if (FD_passive == -1) return;
    int FD_epoll = epoll_create1(0); // Step 1
    if (FD_epoll == -1) return;
    if (register_fd_to_epoll(FD_epoll, FD_passive) == -1) return; // Step 2

    // *** The event loop *** //
    epoll_event events[MAXEVENTS];
    while(true)
    {
        // Blocked until ready ...
        int num_events = epoll_wait(FD_epoll, events, MAXEVENTS, -1); // Step 3
        for(int n=0; n!=num_events; ++n)
        {
            // *** Event 1 : Disconnection *** //
            if (events[n].events & EPOLLERR || events[n].events & EPOLLHUP || !(events[n].events & EPOLLIN))
            {
                close(events[n].data.fd); // continue ...
            }
            // *** Event 2 : New connection *** //
            else if (FD_passive == events[n].data.fd)
            {
                sockaddr sock_addr;
                socklen_t sock_len = sizeof(sock_addr);

                int FD_active = accept(events[n].data.fd, &sock_addr, &sock_len);
                if (FD_active == -1)
                {
                    std::cout << "\nError in acceptng new connection, next event ... ";
                }
                else
                {
                    std::cout << "\nAccepted connection from " << get_client_ip(sock_addr, sock_len);

                    if (set_socket_non_blocking(FD_active) == -1)
                    {
                        std::cout << "\nFail to set socket non blocking";
                        return;
                    }

                    if (register_fd_to_epoll(FD_epoll, FD_active) == -1)
                    {
                        std::cout << "\nFail to register fd to epoll";
                        return;
                    }
                }   // continue ...
            }
            // *** Event 3 : Message from existing connection *** //
            else
            {
                char buf[512];
                size_t count = read(events[n].data.fd, buf, sizeof(buf));
                if (count == -1)
                {
                    std::cout << "\nConnection read failure";
                    close(events[n].data.fd);
                    break;
                }
                else if (count == 0)
                {
                    std::cout << "\nDisconnection detected [EOF]";
                    close(events[n].data.fd);
                    break;
                }

                write(events[n].data.fd, buf, count); // echo back
            }   // continue ...
        }
    }
    close(FD_passive);
}
```