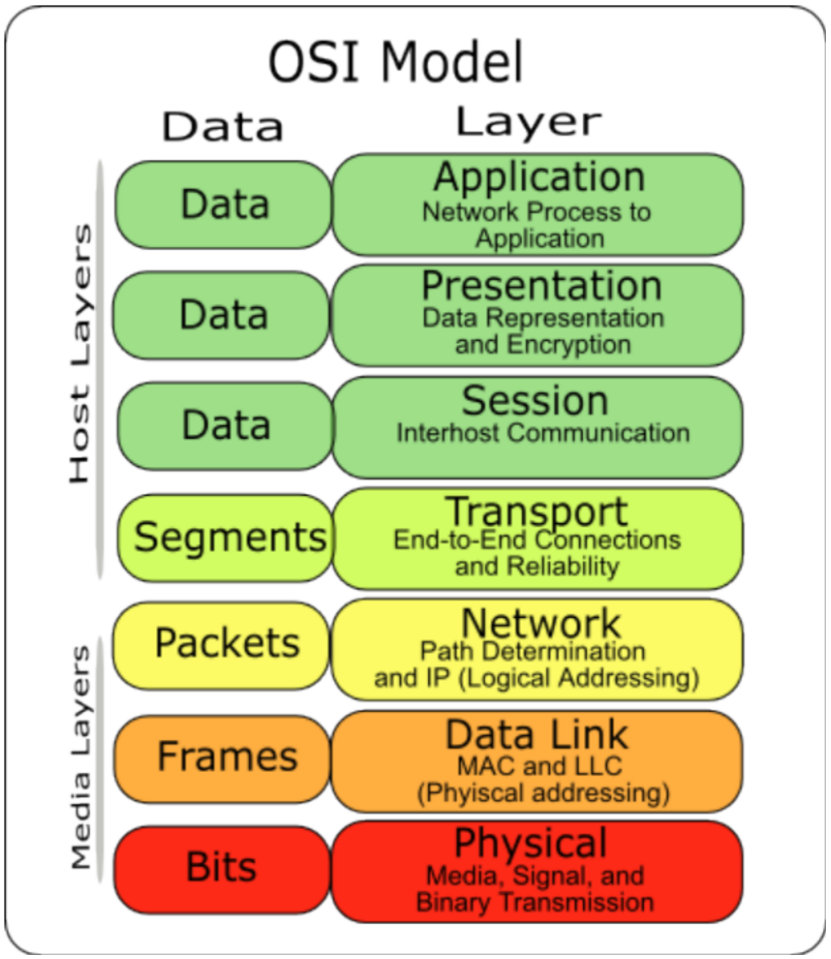
Реферат підготував студент 2-го курсу комп’ютерної математики
Нікітчин Назарій

Клієнт/Сервер на C++/C (Linux)

Основи роботи з мережею

Перш ніж перейти до конкретно програмування клієнт серверної архітектури на C/C++, варто ознайомитись з основними поняттями роботи мережі.
Очевидно, що клієнт/серверний застосунок складається з клієнта і сервера, де завдання клієнта це забезпечити інтерфейс взаємодії віддалено з сервером, а на сервері виконується основна логіка обробки даних. Для реалізації цього існує модель взаємодії між різними пристроями.

Модель OSI



Модель OSI (EMBBC) (базова еталонна модель взаємодії відкритих систем, англ. Open Systems Interconnection Basic Reference Model, 1978 р.) — абстрактна мережева модель для комунікацій і розроблення мережевих протоколів. [Wikipedia]

Такий рівневий підхід дозволяє розробникам ефективніше розробляти ПО, адже вони можуть абстрагуватись від принципів роботи інших рівнів і працювати лише на тому, який дозволяє вирішити поставлену задачу, та користуватись вже готовим API інших рівнів.

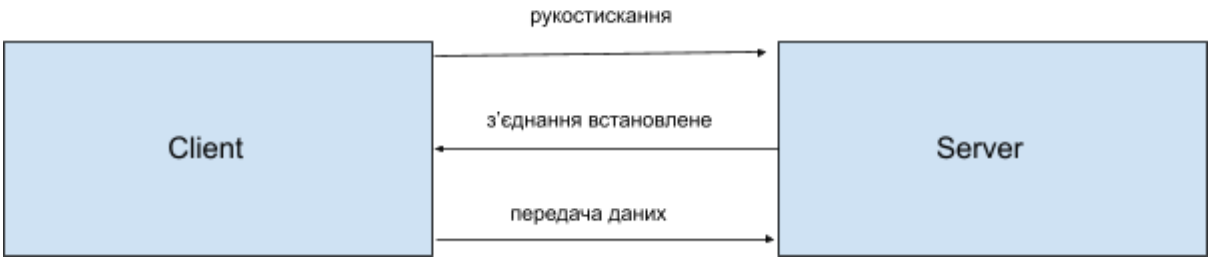
На кожному з вищенаведених рівнів існують свої протоколи(деякі концептуальні домовленості у розробці ПО) за допомогою яких вирішують задачі в межах певного рівня і дозволяють підвищити рівень абстракції, починаючи від ,грубо кажучи, дротів і роутерів та закінчуючи HTTP, протоколом на якому оснований весь сучасний інтерфейс веб-сторінок.

Початок роботи

Під час виконання даної задачі, оскільки мова йде про передавання байтів між пристроями – я буду користуватись протоколом транспортного рівня TCP.

Чому саме TCP?

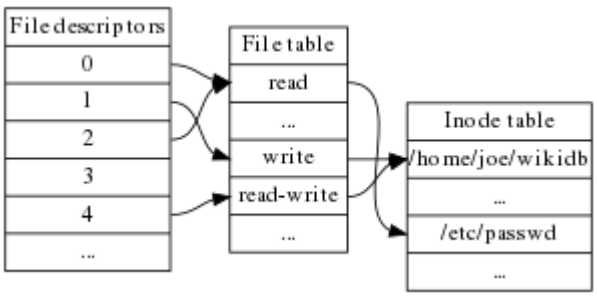
На відміну від іншого поширеного протоколу UDP, TCP – надійний протокол, це значить, що він зберігає порядок надісланих пакетів байтів та гарантує їх доставку – це досягається принципом “рукостискання”, який реалізований у протоколі.



.

Сокет

– деякий програмний інтерфейс для обміну інформацією між процесами (процеси можуть належати як одному комп’ютеру так і різним).



Що таке файловий дескриптор(Linux)?

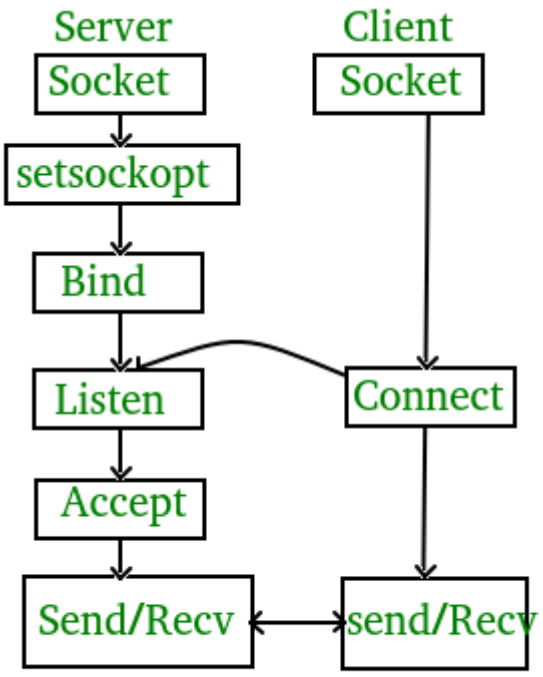
У комп’ютерному програмуванні, файловий дескриптор це абстрактний показник для доступу до файлу. Термін зазвичай використовується в операційних системах POSIX.[Wikipedia]

По суті файловий дескриптор в UNIX-системах це якесь ціле число (int) яке дорівнює позиції у масиві вказівників, які вказують на певний файл у пам’яті. Тобто коли ми відкриваємо будь-який файл ОС створює деякий файл у пам’яті де містяться тимчасові змінні про відкритий нами файл. У UNIX-подібних системах “усе” є файлами і сокети також, отже процес запису/читання сокету можна ототожнити з процесом запису/читання файлового дескриптора.

Де знайти цю інформацію і навіть більше?

Оскільки Linux це ОС з відкритим source code(дякую, Торвальдсе), то всю інформацію, яка потрібна для розробки якогось програмного забезпечення під цю ОС з легкістю можна знайти там, де можна “знайти будь-що у Linux” – у консолі. Використовуючи команду **man**(manual)**,** користувач зможе побачити детальну інформацію щодо усіх функцій та інтерфейсів, які згадуються далі.

Створення базових TCP/IP сервера/клієнта



План наступних дій(і роботи програми) вичерпно описує наступна схема:

Сервер

1. Створюємо сокет – **create().**
2. Зв’язуємо сокет до адреси сервера – **bind()**.
3. Блокуємо виконання коду і встановлюємо режим очікування нового клієнта – **listen().**
4. Коли клієнт знайшовся процес передається у функцію – **accept().**
5. Передаємо/читаємо дані **read()/write()**.
6. Повертаємось до пункту 3.

---

7. Закриваємо з’єднання – **close().**

Для реалізації вищенаведених дій нам знадобляться такі хедерфайли:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
```

Потім нам варто визначити IP та порт:

```
#define SERVER_PORT 5555
#define SERVER_NAME "127.0.0.1"
```

Далі у main() визначимо деякі змінні які будуть потрібні потім:

```
int server_sock, client_sock;                   //номери файлових дескрипторів
struct sockaddr_in server_addr, client_addr;
socklen_t addr_size;
char buffer[1024];                              //буфер для обміну даними
int n;
```

Структура **sockaddr_in** складається з типу протоколу(sa_family_t) (AF_INET зазвичай), номеру порту (in_port_t) та IP адреси (sin_addr).

(ПУНКТ 1) Далі створюємо сокет і отримуємо номер дескриптора:

```
server_sock = socket(AF_INET, SOCK_STREAM, 0);
```

варто перевірити server_sock на невід’ємність впевнившись, шо сокет дійсно утворився успішно (номер дескриптора за нормального створення сокета буде 3, бо 0 – це дескриптор зарезервований під обслуговування стандартного вводу, 1 - виводу, а 2 – для виводу помилок)

Наступним кроком буде заповнення структури sockaddr_in, але спочатку варто виділити під неї пам’ять:

```
memset(&server_addr, '\0', sizeof(server_addr));
```

```
server_addr.sin_family = AF_INET;
server_addr.sin_port = SERVER_PORT;
server_addr.sin_addr.s_addr = inet_addr(SERVER_NAME);
```

(ПУНКТ 2)

```
n = bind(server_sock, (struct sockaddr*)&server_addr, sizeof(server_addr));
```

(ПУНКТ 3)

варто зазначити, що другим параметром являється розмір черги з клієнтів на підключення

```
listen(server_sock, 5);
```

(ПУНКТ 4)

```
client_sock = accept(server_sock, (struct sockaddr*)&client_addr, &addr_size);
```

Далі за допомогою функцій наведених у пункті №5 реалізовуємо бізнес-логіку.

(ПУНКТ 7)

```
close(client_socket);
```

Клієнт

План:

1. Створюємо сокет.
2. З'єднуємо з сервером **connect().**
3. Бізнес-логіка.
4. **close()**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
int main(){
  char *ip = "127.0.0.1";
  int port = 5566;
  int sock;
  struct sockaddr_in addr;
  socklen_t addr_size;
  char buffer[1024];
  int n;
  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock < 0){
    perror("[-]Socket error");
```

```
    exit(1);
  }
  printf("[+]TCP server socket created.\n");
  memset(&addr, '\0', sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_port = port;
  addr.sin_addr.s_addr = inet_addr(ip);
  connect(sock, (struct sockaddr*)&addr, sizeof(addr));
  printf("Connected to the server.\n");

/// бізнес-логіка

close(sock);
return 0;
}
```

Підтримка багатьох клієнтів

Дуже часто під час розробки клієнт серверних застосунків виникає потреба у підтримці більш ніж одного клієнта одним сервером. Вище ми навчились створювати базовий сервер який підтримуватиме лише одного клієнта – зараз я наведу кілька шаблонів рішень для розробки мультиклієнтних систем.

O(n) системних викликів

Перше, що спадає на думку, коли мова заходить про те, щоб підтримувати більше одного клієнта це створити список номерів файлових дескрипторів(кожен номер – це якийсь клієнт) і в безкінечному циклі кожен раз їх опитувати. командою **read()** на наявність нових повідомлень. Це рішення цілком природне і його реалізація досить тривіальна – нам не потрібно розбиратись з якимись додатковими технологіями, а просто розв’язати задачу “в лоб”. Але є один значний мінус цього методу: наші клієнти не завжди всі будуть відправляти повідомлення на сервер, ба більше – відправляти може лише один з великої кількості клієнтів, і для того, щоб знайти його нам доведеться опитати усіх клієнтів(у найгіршому випадку), що займе n системних викликів, а це дуже ресурсоємко. **Can we do better?**

select()

Щоб зменшити кількість системних викликів з n до одного програмісти розробили функцію **select()**(API цієї функції можна отримати за допомогою команди *man select*). Ця функція стара, як світ IT і з’явилась у 80-х роках минулого століття, а це значить. що її підтримують усі існуючі сучасні (і не дуже) платформи. Отже ми маємо готове

рішення на всіх платформах тоді чому в змісті цього реферату після цього пункту є ще деякі, що відносяться до цієї теми?
select все ж має певні вади:

- дослідивши API цієї функції ви побачите, що під час ініціалізації select вимагає максимальний номер дескриптора серед тих які використовуються(щоб не опитувати максимально доступне їх число) – не критично, але неприємно
- для роботи з select ми матимемо заповнити структури fd_sets, але селект працює так, що ці структури не можна буде перевикористати і нам потрібно буде їх кожен раз перевизначати
- для того, щоб зрозуміти який дескриптор з fd_sets здійснив якусь подію нам треба буде їх усі вручну опитати за допомогою FD_ISSET на що піде досить багато ресурсів
- select не підтримує більше 1024 дескрипторів(це можна обійти, перевизначивши FD_SETSIZE перед підключенням хедерфайла sys/select.h)
- вибір подій за якими ми можемо спостерігати досить обмежений
- є певні проблеми з багатопоточністю.

Не дивлячись на вищенаведені недоліки, select крім своєї кросплатформеності має ще один козир – select (якщо дозволяє залізо) може працювати з таймаутами в одну наносекунду, що робить його незамінними під час моделювання певного роду систем.

poll()

Це дійсно хороший метод опитування сокетів в якому позбулись більшості недоліків, якими володів select. Він досить добре підходить для створення сучасних мережевих сервісів. Тут немає ліміта в 1024 дескриптора, не модифікується структура pollfd(детальніше див. man poll), але не можна забувати обнуляти revents, poll так як і select – кроссплатформенний.

!!!У структури pollfd є поля events та revents, які виконують зовсім різні задачі і варто бути обережним, щоб одруківка в одну букву не коштувала вам довгих годин пошуку бага(примітка з власного досвіду автора).!!!

Інше

У цій статті я не розглядав можливості виділення під окремого клієнта окремого процесу (**fork)** чи потоку – ці рішення зазвичай погано підходять, адже не є легко маштабованими, складними в розробці та в пошуці помилок.