

Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа машиностроения

Курсовая работа

Дисциплина: объектно-ориентированное программирование

Тема: конкурентный TCP сервер

Студент гр. 3331506/00401

Коваленко Д.А.

Преподаватель

Ананьевский М.С.

Санкт-Петербург

2023

Оглавление

Введение.....	3
Интерфейс сокетов.....	4
Заголовочные файлы.....	4
Socket()	4
Bind()	4
Listen()	5
Accept().....	5
Recv().....	5
Send()	5
Close()	5
Мультиплексирование ввода/вывода.....	5
Алгоритм работы программы	6
Вывод.....	7
Список литературы	8
Приложение	9
main.cpp	9
server.h	9
worker.h	10
worker.cpp.....	11
server.cpp	15

Введение

Для передачи данных через сеть используют сетевые приложения. Наиболее распространённой моделью сетевого приложения является клиент-сервер. Приложение, построенное на базе этой модели, состоит из процесса *сервера* и процесса *клиента*. Сервер – процесс, который управляет некоторым ресурсом и предоставляет некоторую услугу клиентам, манипулируя этим ресурсом. Клиент является инициатором сетевого взаимодействия. Сетевое взаимодействие, инициируемое клиентом, называется *транзакцией*.

Транзакция в модели клиент-сервер предусматривает выполнение четырёх шагов:

1. Клиент посылает запрос серверу.
2. Сервер получает запрос от клиента, интерпретирует его и выполняет соответствующие манипуляции со своим ресурсом.
3. Сервер посылает ответ клиенту, после чего ждёт следующий запрос.
4. Клиент получает ответ и обрабатывает его по своему усмотрению.

Сервер может обрабатывать запросы одновременно нескольких клиентов, для этого используются средства мультиплексирования ввода/вывода.

Взаимодействие в сети осуществляется в соответствии со стеком протоколов TCP/IP. Для адресации на сетевом уровне используется IP (Internet Protocol). Каждое устройство в сети имеет уникальный IP адрес в рамках этой сети. Для передачи информации на транспортном уровне используются протоколы TCP (Transmission Control Protocol) и UDP (User Datagram Protocol). Для адресации на транспортном уровне используется порт, который является уникальным для каждого процесса на хосте.

Для написания сетевых приложений используется *интерфейс сокетов*, который реализован в операционных системах. Впервые интерфейс сокетов появился на Unix системах. *Сокеты Беркли* – API, представляющий собой библиотеку для разработки приложений на C.

Большинство серверов работают на Unix системах, в связи с чем и было выбрано данной семейство операционных систем для реализация конкурентного TCP сервера. Исходный код сервера был написан на C++ с использованием стандартных библиотек.

Интерфейс сокетов

Заголовочные файлы

API для работы с сокетами на C++ в операционной системе Linux Ubuntu реализован в библиотеках, находящихся в заголовочных файлах:

- **<sys/socket.h>** - базовые функции сокетов BSD и структуры данных.
- **<netinet/in.h>** - семейства адресов/протоколов PF_INET и PF_INET6. Широко используются в сети Интернет, включают в себя IP-адреса, а также номера портов TCP и UDP.
- **<sys/un.h>** - семейство адресов PF_UNIX/PF_LOCAL. Используется для локального взаимодействия между программами, запущенными на одном компьютере. В компьютерных сетях не применяется.
- **<arpa/inet.h>** - функции для работы с числовыми IP-адресами.
- **<netdb.h>** - функции для преобразования протокольных имен и имен хостов в числовые адреса. Используются локальные данные аналогично DNS.

Socket()

Для создания сокета используется системный вызов **socket()**, который возвращает файловый дескриптор созданного сокета в случае успеха. В параметрах сокета указывается семейство протоколов для созданного сокета, тип сокета, а также протокол транспортного уровня.

Bind()

Чтобы связать сокет с конкретным IP-адресом и портом используется функция **bind()**, которая возвращает 0 в случае успеха. Для указания IP-адреса и

порта используется структура `sockaddr_in` (для IPv4). При передаче в эту структуру адреса и порта используются функции для преобразования порядка байт на хосте в порядок байт в сети. Данная функция используется только для создание серверного сокета.

Listen()

Прежде, чем начать принимать запросы от клиентов, серверный сокет следует перевести в слушающий режим с помощью функции **listen()**, которая возвращает 0 в случае успеха. В параметрах указывается количество запросов на соединение, которое может ожидать своей очереди обработки.

Accept()

Чтобы принять запрос на соединение от клиента, используется функция **accept()**, которая возвращает файловый дескриптор клиентского сокета в случае успеха.

Recv()

Чтобы принять данные от клиента используется функция **recv()**, которая считывает данные из клиентского файлового дескриптора.

Send()

Чтобы отправить данные клиенту используется функция **send()**, которая записывает данные в клиентский файловый дескриптор.

Close()

Чтобы завершить работу с файловым дескриптором, необходимо его закрыть с помощью системного вызова **close()**.

Мультиплексирование ввода/вывода

При работе сервера с более, чем одним клиентом возникает проблема порядка обработки запросов: пока сервер ожидает действие от одного клиента, он не может обработать запрос от другого. Для решения данной проблемы

используют мультиплексирование ввода/вывода. На Unix система реализованы 3 основные функции мультиплексирования ввода: **select()**, **poll()**, **epoll()**. Ознакомиться с работой данных системных вызовов можно в соответствующей литературе. Особенности работы **select()** и **poll()** ограничивают масштабируемость сервера, в частности не позволяют обслужить более 10 000 одновременных соединений (C10K Problem). Для решения данной проблемы существует системный вызов **epoll()**, который в дальнейшем был использован при написании сервера.

Алгоритм работы программы

Сервер реализован на C++, в качестве клиентского приложения использована утилита telnet. Клиент может запросить текущую дату и время, сервер отправляет в качестве ответа дату и время в формате YYYY-MM-DD HH:MM:SS.

В программе реализован класс Server, содержит файловый дескриптор сокета, IP-адрес, порт, а также экземпляр класса Worker. Класс Worker реализует обработку клиентских запросов при помощи системного вызова epoll(). Класс Server и класс Worker в терминах ООП связаны композицией.

Алгоритм работы программы:

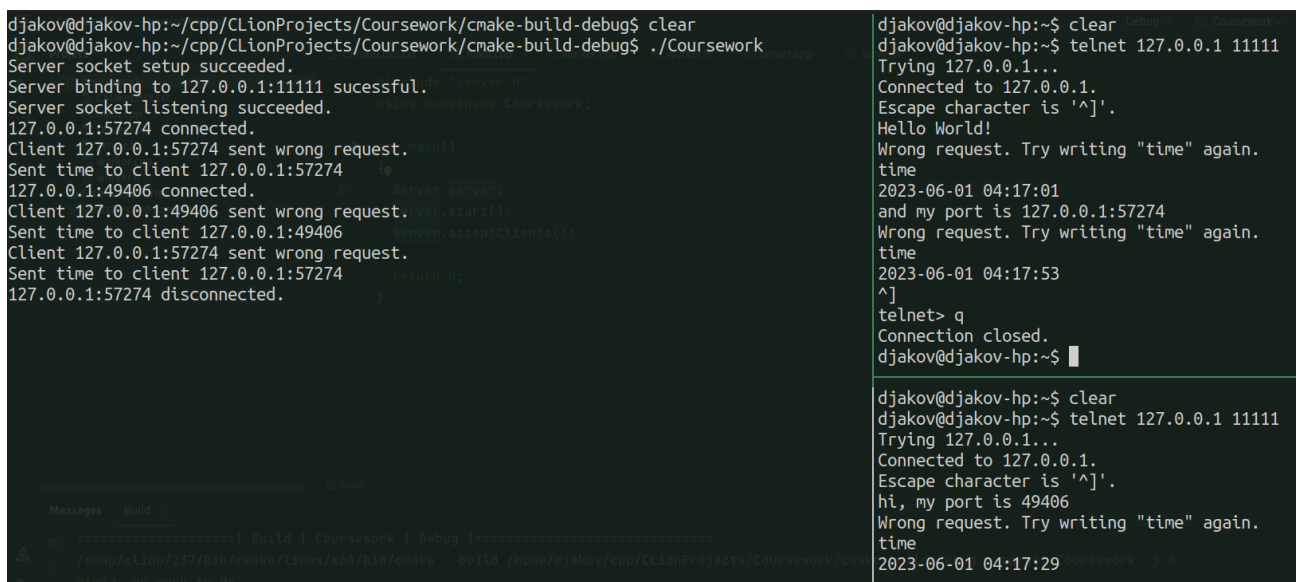
1. Создание неблокирующего серверного сокета.
2. Описание адреса этого сокета.
3. Связывание адреса и серверного сокета.
4. Перевод серверного сокета в слушающий режим.
5. Добавление серверного сокета в отслеживание при помощи epoll()
6. Вызов epoll() для получения файловых дескрипторов, ожидающих обработки события.
7. Если файловый дескриптор, ожидающий обработки события, принадлежит серверу, то происходит попытка принять новое соединение, иначе обрабатывает событие клиентского сокета.

8. Если клиент соединился, то добавляем его файловый дескриптор в отслеживание `epoll()`, а также добавляем его в список подключённых клиентов.
9. Если происходит обработка клиентского сокета, и клиент отключился, то удаляем его из списка подключённых клиентов.
10. Если клиент пытается отправить сообщение, то принимаем его и проверяем на наличие запроса даты и времени.
11. Если клиент запросил текущую дату и время, то обрабатываем его запрос.
12. Если клиент произвёл любой другой запрос, то отправляем сообщение с просьбой отправить корректный запрос.

Исходный код полученный программы представлен в приложении.

Вывод

В результате выполнения курсовой работы был создан сервер, обеспечивающий одновременную работу с множеством клиентов. Пример работы сервера представлен на рисунке 1.



```
djakov@djakov-hp:~/cpp/CLionProjects/Coursework/cmake-build-debug$ clear
djakov@djakov-hp:~/cpp/CLionProjects/Coursework/cmake-build-debug$ ./Coursework
Server socket setup succeeded.
Server binding to 127.0.0.1:11111 successful.
Server socket listening succeeded.
127.0.0.1:57274 connected.
Client 127.0.0.1:57274 sent wrong request.
Sent time to client 127.0.0.1:57274
127.0.0.1:49406 connected.
Client 127.0.0.1:49406 sent wrong request.
Sent time to client 127.0.0.1:49406
Client 127.0.0.1:57274 sent wrong request.
Sent time to client 127.0.0.1:57274
127.0.0.1:57274 disconnected.

djakov@djakov-hp:~/cpp/CLionProjects/Coursework/cmake-build-debug$ clear
djakov@djakov-hp:~/cpp/CLionProjects/Coursework/cmake-build-debug$ ./Coursework
Server socket setup succeeded.
Server binding to 127.0.0.1:11111 successful.
Server socket listening succeeded.
127.0.0.1:57274 connected.
Client 127.0.0.1:57274 sent wrong request.
Sent time to client 127.0.0.1:57274
127.0.0.1:49406 connected.
Client 127.0.0.1:49406 sent wrong request.
Sent time to client 127.0.0.1:49406
Client 127.0.0.1:57274 sent wrong request.
Sent time to client 127.0.0.1:57274
127.0.0.1:57274 disconnected.
```

```
djakov@djakov-hp:~$ clear
djakov@djakov-hp:~$ telnet 127.0.0.1 11111
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Hello World!
Wrong request. Try writing "time" again.
time
2023-06-01 04:17:01
and my port is 127.0.0.1:57274
Wrong request. Try writing "time" again.
time
2023-06-01 04:17:53
^]
telnet> q
Connection closed.
djakov@djakov-hp:~$

djakov@djakov-hp:~$ clear
djakov@djakov-hp:~$ telnet 127.0.0.1 11111
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
hi, my port is 49406
Wrong request. Try writing "time" again.
time
2023-06-01 04:17:29
```

Рисунок 1

Список литературы

1. Beej's Guide to Network Programming.
2. Роберт Лав. Linux. Системное программирование, 2014
3. Randal E. Bryant, David R. O'Hallaron. Computer Systems. A Programmer's Perspective. Third Edition, 2016
4. У.Р. Стивенс, Б. Феннер, Э.М. Рудофф. UNIX. Разработка сетевых приложений. 3-е изд. – СПб.:Питер, 2007

Приложение

main.cpp

```
#include "server.h"
using namespace Coursework;

int main()
{
    Server server;
    server.start();
    server.acceptClients();

    return 0;
}
```

server.h

```
#ifndef COURSEWORK_SERVER_H
#define COURSEWORK_SERVER_H

#include <sys/socket.h> // socket(), bind(), listen(),
accept()
#include <netinet/in.h> // sockaddr_in
#include <arpa/inet.h> // inet_aton()
#include <csignal> // close()
#include <cstdint>
#include <string>
#include "worker.h"

namespace Coursework
{
    class Server
    {
    public:
        explicit Server(uint16_t port = 11111, const
std::string& ip = ""); // create server socket and bind
it
        ~Server() noexcept; // close socket
        void start(uint16_t connections = SOMAXCONN) const;
// start listening
        void acceptClients(); // start handling clients
requests
    };
}
```

```

        inline int32_t getDescriptor() const { return
descriptor; }
        inline uint16_t getPort() const { return port; }
    private:
        Worker worker;
        uint16_t port;
        std::string ip;
        int32_t descriptor;

};
}

```

```

#endif // COURSEWORK_SERVER_H

```

worker.h

```

#ifndef COURSEWORK_WORKER_H
#define COURSEWORK_WORKER_H

```

```

#include <cstdint>
#include <sys/epoll.h>
#include <string>
#include <vector>
#include <memory>
#include <unordered_map>

```

```

namespace Coursework {
    const uint16_t kMaxEvents{32};

    class Worker {
    public:
        explicit Worker();
        ~Worker();
        void setServerEvent(int32_t serverfd);
        void addToEpoll(int32_t descriptor) const;
        int16_t waitForEvents();
        void eventsHandling(int16_t numEvents);
        static void setClientNonblock(int32_t clientfd);
    private:
        int32_t epollfd;
        epoll_event serverEvent;
        std::vector<epoll_event> events;
        std::unordered_map<int32_t, std::string> clients;
    };
}

```

```
}
```

```
#endif //COURSEWORK_WORKER_H
```

worker.cpp

```
#include "worker.h"
```

```
#include <iostream>
```

```
#include <sstream>
```

```
#include <cstring>
```

```
#include <chrono>
```

```
#include <ctime>
```

```
#include <iomanip>
```

```
#include <sys/socket.h>
```

```
#include <sys/fcntl.h>
```

```
#include <netinet/in.h>
```

```
#include <arpa/inet.h>
```

```
#include <csignal>
```

```
Coursework::Worker::Worker() : serverEvent{}
```

```
{
```

```
    try
```

```
    {
```

```
        epollfd = epoll_create1(0);
```

```
        if (epollfd < 0)
```

```
            throw std::string{"Epoll creation failed.\n"};
```

```
    }
```

```
    catch (std::string& exception)
```

```
    {
```

```
        std::cerr << exception;
```

```
        exit(-1);
```

```
    }
```

```
    serverEvent.events = EPOLLIN;
```

```
    events.resize(kMaxEvents);
```

```
}
```

```
Coursework::Worker::~~Worker()
```

```
{
```

```
    for (auto& client : clients)
```

```
        close(client.first);
```

```
}
```

```

void Coursework::Worker::setServerEvent(int32_t serverfd)
{
    serverEvent.data.fd = serverfd;
    addToEpoll(serverfd);
}

void Coursework::Worker::addToEpoll(int32_t descriptor)
const
{
    epoll_event event{};
    event.data.fd = descriptor;
    event.events = EPOLLIN;
    try
    {
        bool isAddToEpollFailed = epoll_ctl(epollfd,
EPOLL_CTL_ADD, descriptor, &event);
        if (isAddToEpollFailed) throw std::string{"Addition
to epoll error.\n"};
    }
    catch (std::string& exception)
    {
        std::cerr << exception;
        std::cout << errno << "\n";
    }
}

int16_t Coursework::Worker::waitForEvents()
{
    try
    {
        auto numOfEvents =
static_cast<int16_t>(epoll_wait(epollfd, &events[0],
kMaxEvents, -1));
        if (numOfEvents < 0) throw std::string{"Waiting for
events error.\n"};
        return numOfEvents;
    }
    catch (std::string& exception)
    {
        std::cerr << exception;
        return {};
    }
}

```

```

void Coursework::Worker::eventsHandling(int16_t
numOfEvents)
{
    for (int16_t i = 0; i < numOfEvents; i++)
    {
        epoll_event& e = events[i];

        if (e.data.fd == serverEvent.data.fd)    // branch for
server event handling
        {
            sockaddr_in clientAddress{};
            socklen_t sizeofClientAddress =
sizeof(clientAddress);

            int32_t clientfd = accept(serverEvent.data.fd,
                                     reinterpret_cast<sockaddr
*>(&clientAddress), &sizeofClientAddress);

            setClientNonblock(clientfd);

            addToEpoll(clientfd);

            std::stringstream str;
            str << inet_ntoa(clientAddress.sin_addr) << ":" <<
ntohs(clientAddress.sin_port);
            clients[clientfd] = str.str();
            std::cout << clients[clientfd] << " connected.\n";
        }
        else
        {
            static const uint16_t length{1024};
            static std::vector<char> buffer(length, 0);

            auto result = static_cast<int32_t>(recv(e.data.fd,
buffer.data(), length, MSG_NOSIGNAL));

            if (result == 0 && errno != EAGAIN)
            {
                close(e.data.fd);
                std::stringstream str;
                str << clients[e.data.fd] << " disconnected.\n";
                std::cout << str.str();
                clients.erase(e.data.fd);
            }
        }
    }
}

```

```

        else if (result > 0)
        {
            std::stringstream str;
            buffer[4] = '\0';
            if (strcmp(buffer.data(), "time") == 0)
            {
                const std::time_t t_c =
std::chrono::system_clock::to_time_t(std::chrono::system_
clock::now());
                str << std::put_time(std::localtime(&t_c), "%F
%T\n");
                std::cout << "Sent time to client " <<
clients[e.data.fd] << "\n";
            }
            else
            {
                std::cout << "Client " << clients[e.data.fd] <<
" sent wrong request.\n";
                str << "Wrong request. Try writing \"time\"
again.\n";
            }
            send(e.data.fd, str.str().data(),
str.str().length(), MSG_NOSIGNAL);
        }
    }
}

void Coursework::Worker::setClientNonblock(int32_t
clientfd)
{
    int16_t flags;
#ifdef O_NONBLOCK
    flags = static_cast<int16_t>(fcntl(clientfd, F_GETFL,
0));
    if (flags == -1)
        flags = 0;
    fcntl(clientfd, F_SETFL, flags | O_NONBLOCK);
#else
    flags = 1;
    ioctl(clientfd, FIONBIO, &flags);
#endif
}

```

server.cpp

```
#include "server.h"
#include <iostream>
#include <arpa/inet.h>
#include <sys/socket.h>

Coursework::Server::Server(uint16_t port, const
std::string& ip)
    : port(port), ip(ip)
{
    try
    {
        descriptor = socket(AF_INET, SOCK_STREAM |
SOCK_NONBLOCK, IPPROTO_TCP); // set NONBLOCK flag
        if (descriptor == -1)
            throw std::string{"Server socket establishment
error.\n"};
        else
            std::cout << "Server socket setup succeeded.\n";
    }
    catch (std::string& exception)
    {
        std::cerr << exception;
        exit(-1);
    }

    sockaddr_in serverAddr{};
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(port);
    if (ip.empty())
        serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    // 0.0.0.0, accept any connections
    else
        inet_aton(ip.data(), &serverAddr.sin_addr); //
convert ip to binary form (network byte order)

    setsockopt(descriptor, SOL_SOCKET, SO_REUSEADDR, //
reuse addres
                reinterpret_cast<socklen_t*>(1),
sizeof(int));
    setsockopt(descriptor, SOL_SOCKET, SO_REUSEPORT,
                reinterpret_cast<socklen_t*>(1),
sizeof(int)); // reuse port
```

```

    try
    {
        bool isBindFailed = bind(descriptor,
reinterpret_cast<sockaddr*>(&serverAddr),
sizeof(serverAddr));

        if (isBindFailed) throw std::string{"Server socket
binding error. Server socket closed. Exit...\n"};
        else
            std::cout << "Server binding to " <<
                (ip.empty() ? "127.0.0.1" : ip) << ":" <<
port << " sucessful.\n";
    }
    catch (std::string& exception)
    {
        std::cerr << exception;
        exit(-1);
    }
    worker.setServerEvent(descriptor);
}

```

```

Coursework::Server::~~Server() noexcept
{
    bool isCloseFailed = close(descriptor);
    if (isCloseFailed)
        std::cerr << "Server socket closing error.\n";
    else
        std::cout << "Server socket closing succeeded.\n";
}

```

```

void Coursework::Server::start(uint16_t connections)
const
{
    try
    {
        bool isListenFailed = listen(descriptor, connections);
        if (isListenFailed) throw std::string{"Server socket
listening error. Exit...\n"};
        else
            std::cout << "Server socket listening succeeded.\n";
    }
    catch (std::string& exception)
    {
        std::cerr << exception;
        exit(-1);
    }
}

```



```
    }  
}  
  
void Coursework::Server::acceptClients()  
{  
    while (true)  
    {  
        int16_t events = worker.waitForEvents();  
        worker.eventsHandling(events);  
    }  
}
```