Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Технологии компьютерных сетей

Отчет по лабораторной работе
Изучение протоколов TCP и UDP

Работу выполнил:
Кузьмин Н.А.
Группа: 43501/1
Преподаватель:
Алексюк А.О.

Санкт-Петербург
2018

Содержание

1. Цель работы **3**

2. Теоретическая информация **3**
2.1. Создание сокета . . . 3
2.2. Организация соединения . . . 3
2.3. Передача и приём данных по протоколу TCP . . . 4
2.4. Передача и приём данных по протоколу UDP . . . 4
2.5. Завершение TCP-соединения . . . 4
2.6. Закрытие сокета . . . 4
2.7. Привязывание сокета . . . 5
2.8. Перевод TCP-сокета в состояние прослушивания . . . 5
2.9. Приём входящего TCP-соединения . . . 5

3. Ход работы **5**
3.1. Индвидуальное задание: платежная система . . . 5
3.1.1. Задание . . . 5
3.1.2. Описание реализации . . . 6
3.1.3. Протокол запроса клиент-сервер . . . 7
3.1.4. Протокол ответа сервер-клиент . . . 7
3.1.5. Организация хранения данных . . . 8
3.1.6. Организация одновременной работы нескольких клиентов . . . 8
3.2. Тестирование проекта . . . 8
3.2.1. Код проекта . . . 10
3.3. Socks5 прокси сервер . . . 11
3.3.1. Описание протокола . . . 11
3.3.2. Исходный код . . . 13

4. Выводы **18**

1. Цель работы

Изучение протоколов UDP и TCP и их использование на практике для программирования сокетов.

2. Теоретическая информация

С точки зрения архитектуры TCP/IP сокетом называется пара (IP-адрес, порт), однозначно идентифицирующая прикладное приложение в сети Internet.

С точки зрения операционной системы BSD-сокет (Berkley Software Distribution) или просто сокет — это выделенные операционной системой набор ресурсов, для организации сетевого взаимодействия. К таким ресур сам относятся, например, буфера для приёма/-посылки данных или очереди сообщений. Технология сокетов была разработана в университете Беркли и впервые появилась в операционной системе BSD-UNIX.

В операционной системе MS Windows имеется аналогичная библиотека сетевого взаимодействия WinSock, реализованная на основе библиотеки BSD-сокетов. Существует две версии библиотеки WinSock1 и WinSock2. В подавляющем большинстве случаев функции и типы библиотеки WinSock совпадают с функциями и типами BSD-сокетов. Об имеющихся отличиях будет сказано отдельно.

2.1. Создание сокета

Для создания сокета в библиотеках BSD-socket и WinSock имеется системный вызов socket:

int socket(int domain, int type, int protocol);

В случае успеха результат вызова функции – дескриптор созданного сокета, в случае ошибки (-1) в библиотеке BSD-socket и INVALID_SOCKET в библиотеке WinSock.

Параметр type определяет тип создаваемого сокета. Этот параметр может принимать значения:

- SOCK_STREAM – для организации надёжного канала связи с установлением соединений
- SOCK_DGRAM – для организации ненадёжного дейтаграммного канала связи
- SOCK_RAW – для организации низкоуровнего доступа на основе «сырых» сокетов

Параметр protocol – идентификатор используемого протокола. В большинстве случаев протокол однозначно определяется типом создаваемого сокета, и передаваемое значение этого параметра – 0.

2.2. Организация соединения

Для установления TCP-соединения используется вызов connect:

int connect(int s, const struct sockaddr* serv_addr, int addr_len);

Результатом выполнения функции является установление TCP-соединения с TCP-сервером. Функция возвращает значение 0 в случае успеха и -1 в случае ошибки.

Параметр s – дескриптор созданного сокета.

Параметр serv_addr – указатель на структуру, содержащую параметры удалённого узла.

Параметр addr_len – размер в байтах структуры, на которую указывает параметр serv_addr.

2.3. Передача и приём данных по протоколу TCP

Передача и приём данных в рамках установленного TCP-соединения осуществляется вызовами send и recv:

int send(int s, const void *msg, size_t len, int flags); int recv(int s, void *msg, size_t len, int flags);

Параметр s – дескриптор сокета, параметр msg – указатель на буфер, содержащий данные (вызов send), или указатель на буфер, предназначенный для приёма данных (вызов recv). Параметр len – длина буфера в байтах, параметр flags – опции посылки или приёма данных.

Возвращаемое значение – число успешно посланных или принятых байтов, в случае ошибки функция возвращает значение -1.

2.4. Передача и приём данных по протоколу UDP

В случае установленного адреса по умолчанию для протокола UDP (вызов connect) функции для передачи и приёма данных по протоколу UDP можно использовать вызовы send и recv.

Если адрес и порт по умолчанию для протокола UDP не установлен, то параметры удалённой стороны необходимо указывать или получать при каждом вызове операций записи или чтения. Для протокола UDP имеется два аналогичных вызова sendto и recvfrom:

int sendto(int s, const void *buf, size_t len, int flags, struct sockaddr *to, int* tolen);

int recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, int* fromlen);

Параметры s, buf, len и flags имеют тот же смысл, что и в случае использования функций send и recv, параметры to и tolen – атрибуты адреса удалённого сокета при посылке данных, параметры from и fromlen – атрибуты структуры данных в которую помещаются параметры удалённого сокета при получении данных.

2.5. Завершение TCP-соединения

Завершение установленного TCP-соединения осуществляется в библиотеке BSD-socket с помощью вызова shutdown:

int shutdown(int s, int how);

Параметр s – дескриптор сокета, параметр how – определяет способ закрытия:

- SHUT_RD – запрещён приём данных
- SHUT_WR – запрещена передача данных
- SHUT_RDWR – запрещены и приём и передача данных

2.6. Закрытие сокета

По окончании работы следует закрыть сокет, для этого в библиотеке BSD-socket предусмотрен вызов close:

int close(int s);

Аналогичный вызов в библиотеке WinSock имеет название closesocket:

int closesocket(SOCKET s);

Параметр s – дескриптор сокета. Возвращаемое значение – 0, в случае успеха.

2.7. Привязывание сокета

Созданный сокет является объектом операционной системы, исполь- зующим её отдельные ресурсы. В то же время в большинстве случаев недостаточно просто выделить ресурсы операционной системы, а следует также связать эти ресурсы с конкретными сетевыми параметрами: сетевым адресом и номером порта. Особенно это важно для серверных сокетов, для которых такая связь – необходимое требование доступности разрабатываемого сетевого сервиса.

Организация привязки созданного вызовом socket() сокета к определённым IP-адресам и портам осуществляется с помощью функция bind:

int bind(int s, struct sockaddr *addr, socklen_t addrlen);

Параметр s – дескриптор сокета, параметр addr задаёт указатель на структуру, хранящую параметры адреса и порта, addrlen – размер структуры addr в байтахю

2.8. Перевод TCP-сокета в состояние прослушивания

Для перевода сокета в состояние прослушивания служит системный вызов listen:

int listen(int s, int backlog);

Параметр s – дескриптор сокета, параметр backlog – задаёт максимальную длину, до которой может расти очередь ожидающих соединений. В случае успеха возвращаемое значение – 0. При ошибке возвращается -1.

2.9. Приём входящего TCP-соединения

В случае, когда сокет находится в состоянии прослушивания (listen) необходимо отслеживать поступление входящих соединений. Для этого предусмотрен системный вызов accept:

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);

Параметр s – дескриптор прослушивающего сокета, параметр addr – указатель на структуру, содержащую параметры сокета, инициирующего соединение, addrlen – размер структуры addr в байтах.

Возвращаемое значение – дескриптор сокета, созданного для нового соединения. Большинство параметров нового сокета соответствуют параметрам слушающего сокета. Полученный сокет в дальнейшем может использоваться для передачи и приёма данных.

В случае если входящих соединений нет, то функция accept ожидает поступления запроса на входящее соединение.

3. Ход работы

3.1. Индвидуальное задание: платежная система

3.1.1. Задание

Разработать приложение–клиент и приложение–сервер платежной системы. Участники платежной системы имеют электронные кошельки. Электронный кошелек имеет уникальный номер. При регистрации пользователя в платежной системе на его счет зачисляется определенная сумма. Пользователя платежной системы могут осуществлять платежи друг другу через приложение–сервер.

Основные возможности. Серверное приложение должно реализовывать следующие функции:

1. Прослушивание определенного порта
2. Обработка запросов на подключение по этому порту от клиентов платежной системы
3. Поддержка одновременной работы нескольких клиентов платежной системы через механизм нитей
4. Прием запросов от клиента на регистрацию пользователя, передачу списка электронных кошельков пользователей платежной системы, осуществление платежей одного пользователя другому, проверка состояния счета кошелька
5. Осуществление добавления пользователя в платежную систему, хранение и изменение состояния электронных кошельков в зависимости от платежей пользователей
6. Передача запросов на платежи от одного пользователя другому, подтверждений платежей, номера нового кошелька при регистрации пользователя, списка электронных кошельков
7. Обработка запроса на отключение клиента
8. Принудительное отключение клиента

Клиентское приложение должно реализовывать следующие функции:

1. Установление соединения с сервером
2. Передача запросов на передачу списка электронных кошельков пользователей платежной системы, платежи одного пользователя другому, проверку состояния счета кошелька
3. Получение от сервера запросов на платеж от другого пользователя, результатов платежа
4. Разрыв соединения
5. Обработка ситуации отключения клиента сервером

3.1.2. Описание реализации

Было написано два приложения - клиент и сервер. Сервер принимает запросы на порту 5001 и создает новые потоки для работы с каждым отдельным клиентом. Формат команды для запуска сервера: ./server_linux.

Клиент подключается к серверу, используя его ip-адрес и порт. Формат команды для запуска клиента: ./client_linux <ip-адрес> <номер порта>. Номер порта всегда равен 5001, он явно задан в коде сервера.

Обмен данными происходит следующим образом: сервер ожидает запроса от клиента. Клиент посылает серверу команду и необходимые аргументы. Сервер выполняет запрос и отправляет ответ клиенту. Клиент принимает ответ и выводит необходимые данные на консоль, после чего цикл повторяется. Все протоколы и команды описаны далее.

При регистрации клиент получает на счет 1000 у.е.

3.1.3. Протокол запроса клиент-сервер

Клиент посылает на сервер запрос следующего формата:

1. Длина запроса - 1 байт
2. Длина команды N - 1 байт
3. Команда - N байт
4. Длина первого аргумента M - 1 байт
5. Первый аргумент запроса - M байт
6. Длина второго аргумента K - 1 байт
7. Второй аргумент запроса - K байт
8. Токен - 50 байт.

Токен - это уникальная строка из 50 символов, идентифицирующая клиента на сервере. Токен выдается при регистрации или входе в аккаунт и хранится на клиенте до окончания сессии. Таким образом, к каждому запросу клиента прикрепляется токен, если имеется.

Команда передается как строка. Команды, доступные на клиенте:

- GET - получить состояние счета клиента. Доступно только после входа в аккаунт.
- REG <login> <password> - зарегистрировать нового клиента. После регистрации сервер автоматически залогинит пользователя в созданном аккаунте.
- LOGIN <login> <password> - войти в аккаунт.
- SEND <amount> <login> - переслать заданную сумму пользователю с указанным логином.
- DEL - удалить текущий аккаунт.
- QUIT - выйти из приложения.
- HELP - вывести справку по командам.

Если у клиента имеется токен, то он передается в каждом запросе.

3.1.4. Протокол ответа сервер-клиент

Сервер на запросы клиента посылает ответ следующего формата:

1. Длина ответа - 1 байт
2. Длина типа ответа N - 1 байт
3. Тип ответа - N байт
4. Длина тела ответа (payload) M - 1 байт
5. Тело ответа (payload) - M байт

Тип ответа передается как строка. Типы ответов сервера:

- OK - запрос успешно обработан, payload содержит запрашиваемую информацию, например, состояние счета клиента.
- ERR - произошла ошибка. Например, клиент не вошел систему и хочет выполнить действие, требующее авторизации. Payload содержит сообщение об ошибке
- TOKEN - запрос на вход систему успешно обработан, payload содержит токен для сессии клиента.
- DELETED - запрос на выход из системы успешно обработан, на сервере удален токен для сессии. Payload содержит сообщение об успешности операции.

3.1.5. Организация хранения данных

Данные хранятся в виде файлов в каталоге data, автоматически создаваемой при запуске сервера в той же папке. Каталог data содержит два подкаталога: clients и session.

В clients хранится информация о клиентах: их логин, пароль и состояние счета. Формат следующий: имя файла - логин пользователя. В файле в первой строке записано количество средств у пользователя, а во второй строке - пароль.

В session хранится информация о текущих сессиях. Формат следующий: имя файла - токен сессии. В файле записан логин пользователя, которому принадлежит этот токен.

При работе с клиентом данные о его счете получаются следующим образом: из запроса клиента считывается токен. Затем в подкаталоге session находится файл с именем, равым токену клиента. В этом файле хранится логин, по которому можно получить всю информацию о клиенте в подкаталоге clients. Если файл с именем-токеном не был найден, клиенту возвращается ошибка.

3.1.6. Организация одновременной работы нескольких клиентов

Для одновременной работы нескольких клиентов на сервере был реализован механизм нитей. Есть главный поток, он принимает команды пользователя. Список команд на сервере:

- q - выход из программы, выключение сервера, удаление всех текущих сессий.
- l - выключение/выключение логирования в консоль.
- list - выводит список текущих сессий и токенов для них.
- del <token> - принудительно завершить сессию по ее токену.

В главном потоке создается слушающий поток, который принимает новые соединения с помощью функции accept(). Для каждого нового клиента создается отдельный клиентский поток, благодаря чему все клиенты работают независимо друг от друга.

3.2. Тестирование проекта

Приведем примеры тестирования проекта.

Ручное тестирование. В первом терминале запускается сервер, во втором - клиент. Далее клиент подключается к серверу и выполняет необходимые запросы. В случае необходимости производится анализ логов сервера и клиента.

Подключимся к серверу.

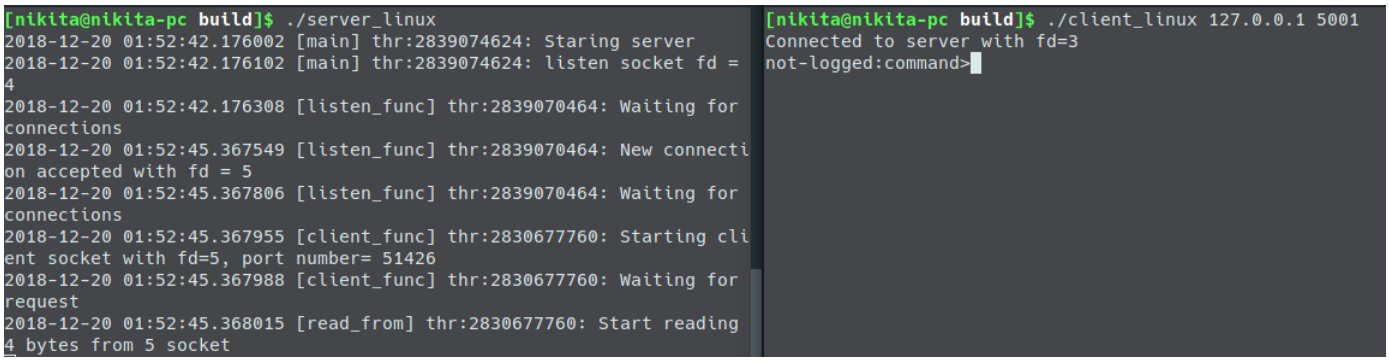

Рисунок 3.1. Подключение клиента к серверу

В дальнейшем логи сервера не будут приводиться на скриншотах. Теперь создадим два тестовых аккаунта test1 и test2 и проверим их счет.

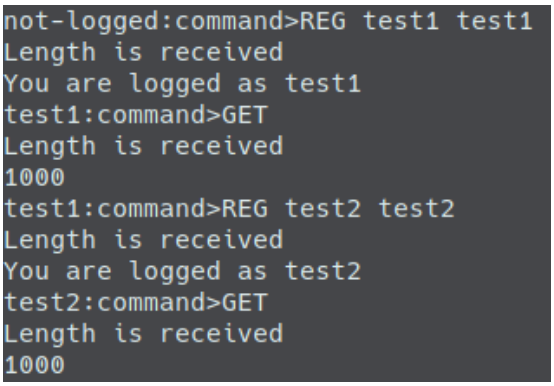

Рисунок 3.2. Создание аккаунтов test1 и test2

Совершим перевод денег от test2 к test1. Переведем сумму в 500 у.е.

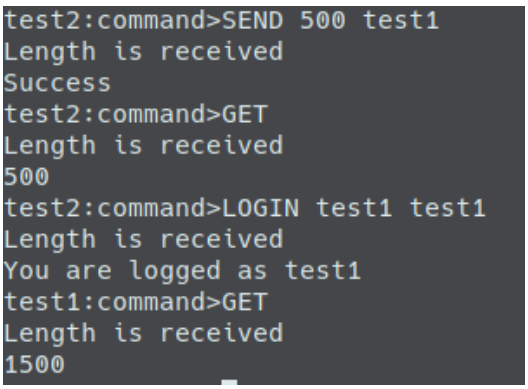

Рисунок 3.3. Платеж от одного пользователя к другому

Автоматизированные тесты. Для этого нужно сначала записать последовательность действий (или ввести вручную в файл userInput.txt), а затем запустить тестирование. Пример приведен ниже.

```
[nikita@nikita-pc client]$ ./NetworkTestBench -t 1 test
2018/12/20 02:00:32 Current test config is number 0 with {DelayTimeInMilliseconds:0 SegmentSize
:1460 DoReset:false ResetAfterNumOfSegments:0 NumOfClients:10}
Connected to server with fd=3
Connected to server with fd=3
not-logged:command>Length is received
ERROR: User already exists
not-logged:command>Length is received
Connected to server with fd=3
not-logged:command>Length is received
Connected to server with fd=3
Connected to server with fd=3
Connected to server with fd=3
Connected to server with fd=3
Connected to server with fd=3
You are logged as a
Connected to server with fd=3
Connected to server with fd=3
not-logged:command>Length is received
```

Рисунок 3.4. Автоматизированный тест

В данном случае был запущен тест без задержек, с десятью параллельными подключениями к серверу, размер пакета 1460 байт. Часть логов сервера приведена ниже.

```
2018-12-20 02:00:32.601610 [free_mem] thr:2709497600: Start free request memory
2018-12-20 02:00:32.601614 [free_mem] thr:2390738688: Start free request memory
2018-12-20 02:00:32.601618 [free_mem] thr:2709497600: Free request memory is done
2018-12-20 02:00:32.601622 [free_mem] thr:2390738688: Free request memory is done
2018-12-20 02:00:32.601627 [client_func] thr:2709497600: Waiting for request
2018-12-20 02:00:32.601631 [client_func] thr:2390738688: Waiting for request
2018-12-20 02:00:32.601635 [read_from] thr:2709497600: Start reading 4 bytes from 8 socket
2018-12-20 02:00:32.601639 [read_from] thr:2390738688: Start reading 4 bytes from 13 socket
2018-12-20 02:00:32.604292 [read_from] thr:2726283008: Read 4 bytes, left 0 bytes
2018-12-20 02:00:32.604317 [read_from] thr:2717890304: Read 4 bytes, left 0 bytes
2018-12-20 02:00:32.604324 [read_from] thr:2399131392: Read 4 bytes, left 0 bytes
2018-12-20 02:00:32.604346 [read_from] thr:2415916800: Read 4 bytes, left 0 bytes
2018-12-20 02:00:32.604358 [read_from] thr:2701104896: Read 4 bytes, left 0 bytes
2018-12-20 02:00:32.604367 [read_from] thr:2726283008: OK. Sockfd = 5, read_length = 0
2018-12-20 02:00:32.604373 [read_from] thr:2701104896: OK. Sockfd = 9, read_length = 0
2018-12-20 02:00:32.604383 [read_from] thr:2717890304: OK. Sockfd = 7, read_length = 0
2018-12-20 02:00:32.604399 [get_request] thr:2726283008: Request length = 66
2018-12-20 02:00:32.604387 [get_request] thr:2701104896: Request length = 16
2018-12-20 02:00:32.604414 [read_from] thr:2415916800: OK. Sockfd = 10, read_length = 0
2018-12-20 02:00:32.604451 [get_request] thr:2701104896: REQUEST LENGTH BUFER BYTES:
2018-12-20 02:00:32.604458 [get_request] thr:2415916800: Request length = 16
2018-12-20 02:00:32.604462 [get_request] thr:2701104896: BUF: 10
2018-12-20 02:00:32.604425 [read_from] thr:2399131392: OK. Sockfd = 12, read_length = 0
2018-12-20 02:00:32.604475 [get_request] thr:2701104896: BUF: 0
2018-12-20 02:00:32.604481 [get_request] thr:2399131392: Request length = 16
```

Рисунок 3.5. Автоматизированный тест

Формат лога следующий:
дата время [вызывающая функция] thr:идентификатор потока: сообщение
По идентификаторам потоков (поле thr) можно увидеть, что происходит параллельное чтение запросов от нескольких клиентов.

3.2.1. Код проекта

Код проекта доступен по ссылке https://github.com/K1ta/NetworksLab2018/tree/individual_task/i

3.3. Socks5 прокси сервер

В качестве дополнительного проекта был реализован прокси сервер, работающий по протоколу Socks5. В качестве языка была использована Java8. Проект реализован с использованием неблокирующих сокетов из Java Nio и может обеспечивать работу нескольких клиентов сразу.

3.3.1. Описание протокола

SOCKS 5 расширяет модель SOCKS 4, добавляя к ней поддержку UDP, обеспечение универсальных схем строгой аутентификации и расширяет методы адресации, добавляя поддержку доменных имен и адресов IPv6. Начальная установка связи теперь состоит из следующего:

- Клиент подключается, и посылает приветствие, которое включает перечень поддерживаемых методов аутентификации.
- Сервер выбирает из них один (или посылает ответ о неудаче запроса, если ни один из предложенных методов не приемлем).
- В зависимости от выбранного метода, между клиентом и сервером может пройти некоторое количество сообщений.
- Клиент посылает запрос на соединение, аналогично SOCKS 4.
- Сервер отвечает, аналогично SOCKS 4.

Методы аутентификации пронумерованы следующим образом. В задании был реализован только метод, не требующий аутентификации.

0x00	Аутентификация не требуется
0x01	GSSAPI
0x02	Имя пользователя / пароль
0x03-0x7F	Зарезервировано IANA
0x80-0xFE	Зарезервировано для методов частного использования

Начальное приветствие от клиента.

Размер	Описание
1 байт	Номер версии SOCKS (должен быть 0x05 для этой версии)
1 байт	Количество поддерживаемых методов аутентификации
n байт	Номера методов аутентификации, переменная длина, 1 байт для каждого поддерживаемого метода

Сервер сообщает о своём выборе.

Размер	Описание
1 байт	Номер версии SOCKS (должен быть 0x05 для этой версии)
1 байт	Выбранный метод аутентификации или 0xFF, если не было предложено приемлемого метода

Последующая идентификация зависит от выбранного метода. В задании из команд была реализована только установка TCP/IP соединения. Поддерживаются только адреса IPv4. Запрос клиента:

Размер	Описание
1 байт	Номер версии SOCKS (должен быть 0x05 для этой версии)
1 байт	Код команды: • 0x01 = установка TCP/IP соединения • 0x02 = назначение TCP/IP порта (binding) • 0x03 = ассоциирование UDP-порта
1 байт	Зарезервированный байт, должен быть 0x00
1 байт	Тип адреса: • 0x01 = адрес IPv4 • 0x03 = имя домена • 0x04 = адрес IPv6
Зависит от типа адреса	Назначение адреса: • 4 байта для адреса IPv4 • Первый байт — длина имени, затем следует имя домена без завершающего нуля на конце • 16 байт для адреса IPv6
2 байта	Номер порта, в порядке от старшего к младшему (big-endian)

Ответ сервера:

Размер	Описание
1 байт	Номер версии SOCKS (0x05 для этой версии)
1 байт	Код ответа: • 0x00 = запрос предоставлен • 0x01 = ошибка SOCKS-сервера • 0x02 = соединение запрещено набором правил • 0x03 = сеть недоступна • 0x04 = хост недоступен • 0x05 = отказ в соединении • 0x06 = истечение TTL • 0x07 = команда не поддерживается / ошибка протокола • 0x08 = тип адреса не поддерживается
1 байт	Байт зарезервирован, должен быть 0x00
1 байт	Тип последующего адреса: • 0x01 = адрес IPv4 • 0x03 = имя домена • 0x04 = адрес IPv6
Зависит от типа адреса	Назначение адреса: • 4 байта для адреса IPv4 • Первый байт — длина имени, затем следует имя домена без завершающего нуля на конце • 16 байт для адреса IPv6
2 байта	Номер порта, в порядке от старшего к младшему (big-endian)

3.3.2. Исходный код

Листинг 1: Socks5Server.java

```
1  package socks5;
2
3  import java.io.IOException;
4  import java.net.InetAddress;
5  import java.net.InetSocketAddress;
6  import java.nio.channels.SelectionKey;
7  import java.nio.channels.Selector;
8  import java.nio.channels.ServerSocketChannel;
9  import java.nio.channels.SocketChannel;
10 import java.util.Arrays;
11 import java.util.Iterator;
12 import java.util.Set;
13 import java.util.StringTokenizer;
14 import java.util.logging.Logger;
15
16 public class Socks5Server {
17     private static int port = 1080;
18     private static Logger logger = Logger.getLogger(Socks5Server.class.getName()
   ↪ );
19
20     public static void main(String[] args) {
21         try {
22             logger.info("Server starting..");
23             Selector selector = Selector.open();
24             ServerSocketChannel serverSocketChannel = ServerSocketChannel.open()
   ↪ ;
25             logger.info("ServerSocketChannel is opened");
```

```
26              serverSocketChannel.bind(new InetSocketAddress(port));
27              logger.info("ServerSocketChannel is bind");
28              serverSocketChannel.configureBlocking(false);
29              logger.info("ServerSocketChannel is configured non-blocking");
30              serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
31              while (true) {
32                  selector.select();
33                  Set<SelectionKey> selectedKeys = selector.selectedKeys();
34                  Iterator<SelectionKey> it = selectedKeys.iterator();
35                  while (it.hasNext()) {
36                      SelectionKey key = it.next();
37                      if (key.isAcceptable()) {
38                          accept(key);
39                      } else if (key.isConnectable()) {
40                          connect(key);
41                      } else if (key.isReadable()) {
42                          read(key);
43                      } else if (key.isWritable()) {
44                          write(key);
45                      }
46                      it.remove();
47                  }
48              }
49          } catch (IOException e) {
50              e.printStackTrace();
51          }
52      }
53
54      /*
55      принять запрос на подключение от клиента
56       */
57      private static void accept(SelectionKey key) throws IOException {
58          //принимаем запрос
59          SocketChannel newChannel = ((ServerSocketChannel) key.channel()).accept
    ↪ ();
60          //делаем неблокирующим
61          newChannel.configureBlocking(false);
62          //регестрируем в селекторе
63          newChannel.register(key.selector(), SelectionKey.OP_READ);
64      }
65
66      /*
67      соединение с другим сервером завершено
68       */
69      private static void connect(SelectionKey key) throws IOException {
70          SocketChannel channel = (SocketChannel) key.channel();
71          Attachment attachment = (Attachment) key.attachment();
72          //завершаем соединение
73          channel.finishConnect();
74
75          //набираем сообщение, которое нужно отправить клиенту
76          //нам нужен адрес, по которому мы будем связываться с удаленным сервером
77          StringBuilder sb = new StringBuilder(channel.getLocalAddress().toString
    ↪ ());
78          sb = sb.replace(0, 1, "");
79          StringTokenizer st = new StringTokenizer(sb.toString(), ":");
80          InetAddress ip = InetAddress.getByName(st.nextToken());
81          int intPort = Integer.parseInt(st.nextToken());
82          byte[] bytesIp = ip.getAddress();
```

```
83          byte[] bytesPort = new byte[]{(byte) ((intPort & 0xFF00) >> 8), (byte) (
    ↪ intPort & 0xFF)};
84
85          Attachment peerAttachment = (Attachment) attachment.peer.attachment();
86          peerAttachment.writeBuffer.clear();
87          //сообщаем, что все ок
88          peerAttachment.writeBuffer.put(new byte[]{0x05, 0x00, 0x00, 0x01});
89          //добавляем в сообщение адрес и порт
90          peerAttachment.writeBuffer.put(bytesIp);
91          peerAttachment.writeBuffer.put(bytesPort);
92
93          //добавляем интерес на запись клиенту
94          attachment.peer.interestOps(SelectionKey.OP_WRITE);
95          System.out.println("Connected");
96      }
97
98      /*
99      прочитать запрос от клиента
100      */
101     private static void read(SelectionKey key) throws IOException {
102         SocketChannel clientChannel = (SocketChannel) key.channel();
103         Attachment clientAttachment = (Attachment) key.attachment();
104         if (clientAttachment == null) {
105             //если у клиента нет attachment, то значит, он у нас в первый раз
106             //нужно проверить запрос на соответствие шаблону и отправить номер версии и
    тип аутентификации
107             clientAttachment = new Attachment();
108             //читаем запрос от клиента
109             int code = clientChannel.read(clientAttachment.readBuffer);
110             if (code < 1) {
111                 //если -1 - то разрыв соединения, 0 - нет места в буфере
112                 close(key);
113                 return;
114             }
115             //печатаем прочитанное в консоль
116             System.out.println("Read1 from " + clientChannel.getRemoteAddress()
    ↪ + ": " + Arrays.toString(clientAttachment.readBuffer.array()));
117             //проверяем сообщение на соответствие приветствию
118             if (clientAttachment.readBuffer.get(0) == 0x05 && clientAttachment.
    ↪ readBuffer.get(1) == 0x01) {
119                 //отвечаем версией протокола и типом аутентификации
120                 clientAttachment.writeBuffer.put(((byte) 0x05));
121                 clientAttachment.writeBuffer.put(((byte) 0x00));
122                 key.interestOps(SelectionKey.OP_WRITE);
123             } else {
124                 throw new IllegalStateException("Bad Request");
125             }
126         } else if (clientAttachment.peer == null) {
127             //если у клиента есть attachment, но нет peer, то это запрос с адресом для
    подключения
128             //нужно прочитать адрес и порт и создать соединение с полученным адресом
129             //также нужно зарегестрировать это соединение, создать SelectionKey и настроить
    peer клиента и нового key
130             clientAttachment.readBuffer.clear();
131             //читаем запрос от клиента
132             int code = clientChannel.read(clientAttachment.readBuffer);
133             if (code < 1) {
134                 //если -1 - то разрыв соединения, 0 - нет места в буфере
135                 close(key);
136                 return;
137             }
```

```
138             //печатаем прочитанное в консоль
139             System.out.println("Read2 from " + clientChannel.getRemoteAddress()
    ↪ + ": " + Arrays.toString(clientAttachment.readBuffer.array()));
140             //проверяем сообщение на соответствие
141             byte[] rb = clientAttachment.readBuffer.array();
142             if (!(rb[0] == 0x05 && rb[1] == 0x01 && rb[2] == 0x00 && rb[3] == 0
    ↪ x01)) {
143                 System.out.println("array: " + Arrays.toString(rb));
144                 throw new IllegalStateException("Bad Request");
145             }
146             //читаем адрес и порт
147             byte[] addr = new byte[]{rb[4], rb[5], rb[6], rb[7]};
148             int p = ((rb[8] & 0xFF) << 8) | (rb[9] & 0xFF);
149             //создаем новый сокет
150             SocketChannel newSocketChannel = SocketChannel.open();
151             //делаем его неблокирующим
152             newSocketChannel.configureBlocking(false);
153             //коннектимся по прочитанному от клиента адресу к серверу телеграма
154             newSocketChannel.connect(new InetSocketAddress(InetAddress.
    ↪ getByAddress(addr), p));
155             //регестрируем сокет в селекторе
156             SelectionKey newSelectionKey = newSocketChannel.register(key.
    ↪ selector(), SelectionKey.OP_CONNECT);
157             //теперь настраиваем peer клиента на новый key и peer newSelectionKey на клиента
158             //writeBuffer клиента - это readBuffer peer'а и наоборот
159             Attachment newAttachment = new Attachment();
160             newAttachment.peer = key;
161             newAttachment.writeBuffer = clientAttachment.readBuffer;
162             newAttachment.readBuffer = clientAttachment.writeBuffer;
163             clientAttachment.peer = newSelectionKey;
164             newSelectionKey.attach(newAttachment);
165         } else {
166             //если у клиента уже все есть, то обмениваемся сообщениями
167             //читаем из attachment клиента readBuffer
168             //и передаем его в attachment пира в writeBuffer
169             clientAttachment.readBuffer.clear();
170             //читаем запрос от клиента
171             int code = clientChannel.read(clientAttachment.readBuffer);
172             if (code < 1) {
173                 //если -1 - то разрыв соединения, 0 - нет места в буфере
174                 close(key);
175                 return;
176             }
177             //печатаем прочитанное в консоль
178             System.out.println("Read3 from " + clientChannel.getRemoteAddress()
    ↪ + ": " + Arrays.toString(clientAttachment.readBuffer.array()));
179             key.interestOps(key.interestOps() ^ SelectionKey.OP_READ);
180             //ставим интерес прочитать пиру клиента
181             clientAttachment.peer.interestOps(SelectionKey.OP_WRITE);
182         }
183         key.attach(clientAttachment);
184     }
185
186     /*
187     отправить ответ клиенту
188      */
189     private static void write(SelectionKey key) throws IOException {
190         SocketChannel clientChannel = (SocketChannel) key.channel();
191         Attachment clientAttachment = (Attachment) key.attachment();
192         //этот метод отправляет серверу или клиенту данные из буфера writeBuffer
```

```
193             System.out.println("Write to " + clientChannel.getRemoteAddress() + ": "
    ↪  + Arrays.toString(clientAttachment.writeBuffer.array()));
194             clientAttachment.writeBuffer.flip();
195             if (clientChannel.write(clientAttachment.writeBuffer) == -1) {
196                 //если -1 - то разрыв соединения, 0 - нет места в буфере
197                 close(key);
198                 return;
199             }
200             if (clientAttachment.writeBuffer.remaining() == 0) {
201                 if (clientAttachment.peer == null) {
202                     //если пира еще нет, то готовимся принимать ответ
203                     key.interestOps(SelectionKey.OP_READ);
204                 } else {
205                     //если всё записано, чистим буфер
206                     clientAttachment.writeBuffer.clear();
207                     //добавялем пиру интерес на чтение
208                     clientAttachment.peer.interestOps(SelectionKey.OP_READ);
209                     //убираем интерес на запись
210                     key.interestOps(SelectionKey.OP_READ);
211                 }
212             }
213         }
214 
215         private static void close(SelectionKey key) throws IOException {
216             key.cancel();
217             key.channel().close();
218             Attachment attachment = ((Attachment) key.attachment());
219             if (attachment != null && attachment.peer != null) {
220                 Attachment peerKey = ((Attachment) attachment.peer.attachment());
221                 if (peerKey != null && peerKey.peer != null) {
222                     peerKey.peer.cancel();
223                 }
224                 attachment.peer.cancel();
225             }
226 
227         }
228 }
```

Листинг 2: Attachment.java

```
1  package socks5;
2  
3  import java.nio.ByteBuffer;
4  import java.nio.channels.SelectionKey;
5  
6  public class Attachment {
7  
8      SelectionKey peer;
9  
10     ByteBuffer writeBuffer;
11     ByteBuffer readBuffer;
12 
13     public Attachment() {
14         writeBuffer = ByteBuffer.allocate(8*1024);
15         readBuffer = ByteBuffer.allocate(8*1024);
16     }
17 
18 }
```

4. Выводы

В ходе выполнения работ были изучены и использованы на практике протоколы TCP и UDP. Также были рассмотрены особенности программирования сокетов на ОС Linux и Windows.

Для реализации первого индивидуального задания - платежной системы - был выбран протокол TCP, так как в этой работе нам нужно обеспечить целостность данных, что с TCP сделать значительно проще, чем с UDP.

Для организации работы нескольких клиентов был реализован механизм на основе потоков: слушающий поток принимает входящие соединения и создает отдельный поток для каждого клиента.

Для хранения данных использовались обычные файлы, разделенные на две группы: данные о текущих сессиях (data/session) и данные о клиентах (data/clients). Данные хранятся в незашифрованном виде, что является потенциальной уязвимостью, однако это уже выходит за рамки задания.

Для логирования была написана собственная библиотека, поддерживающая форматированный вывод. Использование похоже на функцию printf, но написанная библиотека выводит данные на консоль и в файл server.log. Также выводится дополнительная информация: время, в которое было совершено логирование, имя функции и идентификатор потока, вызвавшего функцию логирования.

Тестирование производилось вручную с запуском сервера и нескольких клиентов, после чего анализировались логи сервера. В ходе проверки была использована утилита gdb для тестирования в консоли, и wireshark для просмотра пакетов, передаваемых между приложениями. Также были использованы автоматические тесты с использованием ПО, предоставленного преподавателем.

В качестве второго индивидуального задания был реализован прокси сервер на протоколе Socks5. Это сетевой протокол, который позволяет пересылать пакеты от клиента к серверу через прокси-сервер прозрачно (незаметно для них) и таким образом использовать сервисы за межсетевыми экранами (фаерволами).

Для реализации использовалась Java 8 с неблокирующими сокетами. Для тестирования был использован сервис AWS EC2. С помощью ssh был настроен удаленный сервер, после чего на нем был запущен и протестирован прокси сервер.