

Санкт-Петербургский Политехнический Университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Сети и телекоммуникации

Отчет по лабораторной работе

Изучение сокетов и разработка собственных клиент-серверных приложений
на сырых сокетах

Работу

выполнил:

Беседин Д.С.

Группа: 43501/3

Преподаватель:

Алексюк А.О.

Санкт-Петербург
2017

Содержание

1. Цель работы	2
2. Программа работы	2
3. Ход выполнения работы	2
3.1. Простейшее TCP клиент-серверное приложение	2
3.1.1. Простейший эхо-сервер	2
3.1.2. Многопоточный TCP сервер	5
3.2. Простейший UDP клиент-сервер	6
3.3. Разработка TCP приложения по индивидуальному заданию	7
3.3.1. Индивидуальное задание	7
3.3.2. Разработка протокола взаимодействия	7
3.3.3. Написание многопоточного TCP сервера на ОС Linux	9
3.3.4. Написание клиента на ОС Windows	11
3.4. Разработка UDP приложения по индивидуальному заданию	11
3.4.1. Индивидуальное задание	11
3.4.2. Разработка протокола взаимодействия	12
3.4.3. Написание UDP сервера	12
3.4.4. Написание UDP клиента	13
3.5. Дополнительное задание	13
3.5.1. Описание задания	13
3.5.2. Выполнение задания	13
4. Выводы	15

1. Цель работы

Целью работы являются изучение основных функций для работы с сокетами со стороны клиента и сервера, а также разработать собственный протокол взаимодействия и создать клиент-серверное приложение, работающего согласно разработанному протоколу.

2. Программа работы

1. Простейшее TCP клиент-серверное приложение

- Простейший эхо-сервер
- Многопоточный TCP сервер

2. Простейшее UDP клиент-серверное приложение

- Простейший эхо-сервер

3. Разработка TCP приложения по индивидуальному заданию

- Индивидуальное задание
- Разработка протокола взаимодействия
- Написание многопоточного TCP сервера на ОС Linux
- Написание клиента на ОС Windows

4. Разработка UDP приложения по индивидуальному заданию

- Индивидуальное задание
- Разработка протокола взаимодействия
- Написание UDP сервера на ОС Windows
- Написание клиента на ОС Linux

5. Дополнительное задание

- Описание задания
- Выполнение задания

3. Ход выполнения работы

3.1. Простейшее TCP клиент-серверное приложение

3.1.1. Простейший эхо-сервер

Для разработки данного приложения необходимо изучить функции работы с сокетами на C.

Для создания сокета применяется функция:

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

Домен определяет пространство адресов, в котором располагается сокет, и множество протоколов, которые используются для передачи данных. Чаще других используются домены Unix и Internet, задаваемые константами AF_UNIX и AF_INET соответственно (префикс AF означает "address family" "семейство адресов"). При задании AF_UNIX для адресации используется файловая система Unix. В этом случае сокеты используются для межпроцессного взаимодействия на одном компьютере и не годятся для работы по сети. Константа AF_INET соответствует Internet-домену. Сокеты, размещенные в этом домене, могут использоваться для работы в любой IP-сети. Существуют и другие домены (AF_IPX для протоколов Novell, AF_INET6 для новой модификации протокола IP - IPv6 и т. д.).

Тип сокета определяет способ передачи данных по сети. Чаще других применяются:

- SOCK_STREAM. Передача потока данных с предварительной установкой соединения. Обеспечивается надежный канал передачи данных, при котором фрагменты отправленного блока не теряются, не переупорядочиваются и не дублируются. Поскольку этот тип сокетов является самым распространенным, до конца раздела мы будем говорить только о нём. Остальным типам будут посвящены отдельные разделы.
- SOCK_DGRAM. Передача данных в виде отдельных сообщений (датаграмм). Предварительная установка соединения не требуется. Обмен данными происходит быстрее, но является ненадежным: сообщения могут теряться в пути, дублироваться и переупорядочиваться. Допускается передача сообщения нескольким получателям (multicasting) и широковещательная передача (broadcasting).
- SOCK_RAW. Этот тип присваивается низкоуровневым (т. н. "сырым") сокетам. Их отличие от обычных сокетов состоит в том, что с их помощью программа может взять на себя формирование некоторых заголовков, добавляемых к сообщению.

Для реализации SOCK_STREAM используется протокол TCP, для реализации SOCK_DGRAM - протокол UDP,

Прежде чем передавать данные через сокет, его необходимо связать с адресом в выбранном домене с помощью функции:

```
1 int bind(int sockfd, struct sockaddr *addr, int addrlen);
```

В качестве первого параметра передается дескриптор сокета, который мы хотим привязать к заданному адресу. Второй параметр, addr, содержит указатель на структуру с адресом, а третий - длину этой структуры. Структура sockaddr имеет следующий вид:

```
1 struct sockaddr {
2     unsigned short    sa_family;    // Семейство адресов, AF_xxx
3     char              sa_data[14];  // 14 байтов для хранения адреса
4 };
```

Работать с этой структурой напрямую не очень удобно, поэтому будем использовать вместо sockaddr одну из альтернативных структур вида sockaddr_XX (XX - суффикс, обозначающий домен: "un" Unix, "in" Internet и т. д.). При передаче в функцию bind указатель на эту структуру приводится к указателю на sockaddr. Рассмотрим для примера структуру sockaddr_in:

```
1 struct sockaddr_in {
2     short int          sin_family;   // Семейство адресов
3     unsigned short int sin_port;     // Номер порта
4     struct in_addr     sin_addr;     // IP-адрес
5     unsigned char      sin_zero[8];  // "Дополнение" до размера структуры sockaddr
6 };
```

На следующем шаге создается очередь запросов на соединение. При этом сокет переводится в режим ожидания запросов со стороны клиентов:

```
1 int listen(int sockfd, int backlog);
```

Первый параметр - дескриптор сокета, а второй задает размер очереди запросов.

Когда сервер готов обслужить очередной запрос, он использует функцию `accept`:

```
1 int accept(int sockfd, void *addr, int *addrlen);
```

Функция `accept` создает для общения с клиентом новый сокет и возвращает его дескриптор. Параметр `sockfd` задает слушающий сокет. После вызова он остается в слушающем состоянии и может принимать другие соединения. В структуру, на которую ссылается `addr`, записывается адрес сокета клиента, который установил соединение с сервером.

После того как соединение установлено, можно начинать обмен данными. Для этого используются функции `send` и `recv` в ОС Windows и `read` и `write` в ОС Linux:

```
1 int send(int sockfd, const void *msg, int len, int flags);  
2 int recv(int sockfd, const void *msg, int len, int flags);
```

Здесь `sockfd` - это, как всегда, дескриптор сокета, через который мы отправляем данные, `msg` - указатель на буфер с данными для `send` и буфер для приема данных для `recv`, `len` - длина буфера в байтах (сколько будет передано/считано), а `flags` - набор битовых флагов, управляющих работой функции.

```
1 int write(int sockfd, const void *msg, int len);  
2 int read(int sockfd, const void *msg, int len);
```

Функции для ОС Linux отличаются лишь отсутствием флагов.

Для закрытия сокета используются функции `shutdown` и `close`:

```
1 int shutdown(int sockfd, int how);  
2 int close(int fd);
```

С помощью `shutdown` можно закрыть сокет для передачи в каком-то направлении с помощью параметра `how`: 0 - запретить чтение, 1 - запретить запись, 2 - запретить и то и то. `Close()` освобождает связанные с сокетом системные ресурсы.

С стороны клиента также установить соединение с сокетом, который будет открыт командой `accept()` сервера. Для этого используется функция `connect`:

```
1 int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

Здесь `sockfd` - сокет, который будет использоваться для обмена данными с сервером, `serv_addr` содержит указатель на структуру с адресом сервера, а `addrlen` - длину этой структуры.

Порядок вызовов функций для работы с сокетами на стороне эхо-сервера:

- `socket()` - создает сокет
- `bind()` - привязка созданного сокета к заданным IP-адресам и портам
- `listen()` - переводит сокет в состояние прослушивания
- `accept()` - принимает поступающие запросы на подключение и возвращает сокет для нового соединения
- `recv()` - чтение данных от клиента из сокета, возвращенного на предыдущем шаге
- `send()` - отправка только что принятых данных клиенту

- shutdown() - разрыв соединения с клиентом
- close() - для закрытия клиентского и слушающего сокетов

Порядок вызовов функций для работы с сокетами на стороне клиента:

- socket() - создает сокет
- connect() - установка соединения для сокета, который будет связан с серверным сокетом, порожденным вызовом accept()
- send() - отправка данных серверу
- recv() - чтение тех же данных от сервера
- shutdown() - разрыв соединения с клиентом
- close() - для закрытия клиентского и слушающего сокетов

3.1.2. Многопоточный ТСП сервер

Для организации работы сервера с множеством клиентов необходимо сделать следующее:

- Вынести общение с клиентом (send и recv) в отдельную функцию для того, чтобы была возможность вызывать ее в новом потоке.
- Организовать работу функций listen() и accept() в цикле. listen() должен работать с первым слушающий сокетом, а accept каждый раз будет создавать новый сокет для общения с клиентом.
- Открывать новый поток, вызывая функцию общения с клиентом. В функцию общения с клиентом необходимо передавать новый сокет, дескриптор которого возвращает accept.

Написанная функция общения с клиентом:

```
1 void* readAndWrite (void* temp);
```

Функция принимает в качестве аргумента ссылку на дескриптор сокета. Возвращаемый результат void* и тип аргумента void * необходимы для вызова функции в новом потоке. Преобразование аргумента в целочисленный дескриптор происходит следующим образом:

```
1 int sock = *((int *) temp);
```

Вызов функций listen() и accept() выполнен в цикле:

```
1 while(1){
2     listen(sockfd, 5);
3     newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &(sizeof(
    ↪ cli_addr)));
4     .....
5 }
```

После открытия нового сокета создается поток. В качестве аргументов ему передается функция работы с клиентом как стартовая функция работы потока, а также дескриптор сокета, чтобы его получила функция общения с клиентом. Создание такого потока на примере сервера на ОС Linux:

```

1 while (1) {
2     ....
3     pthread_t tid;
4     pthread_attr_t attr;
5     pthread_attr_init(&attr);
6     pthread_create(&tid, &attr, readAndWrite, &newsockfd);
7     pthread_detach(tid);
8 }

```

Использовались POSIX потоки.

3.2. Простейший UDP клиент-сервер

Для организации обмена через UDP и обмена с помощью датаграмм необходимо внести следующие изменения в функцию создания сокета, изменить функции чтения и записи, а также изменить порядок вызовов функций.

При создании сокетов как на стороне сервера, так и на стороне клиента, необходимо вторым аргументом (тип данных) передать константу `SOCK_DGRAM`, для организации передачи датаграммами без подтверждения соединения.

Также при передаче через UDP изменятся функции чтения и передачи сообщений:

```

1 ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *
  ↪ src_addr, socklen_t *addrlen);
2 ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct
  ↪ sockaddr *dest_addr, socklen_t addrlen);

```

В качестве аргументов передаются дескриптор сокета, буфер для чтения/записи, флаги. структура с информацией о передающей/принимающей стороне, длина этой структуры. Возвращаемое значение - число реально принятых/переданных символов.

Изменится и состав вызова функций для работы с сокетами на клиенте и сервере. Порядок действий сервера:

- `socket()` - создает сокет
- `bind()` - привязка созданного сокета к заданным IP-адресам и портам
- `recvfrom()` - чтение данных от клиента из сокета, возвращенного на предыдущем шаге
- `sendto()` - отправка только что принятых данных клиенту
- `shutdown()` - разрыв соединения с клиентом
- `close()` - для закрытия клиентского и слушающего сокетов

Для организации обмена по UDP не происходит прослушивание сокетом на подключение и подтверждения соединения с созданием нового сокета.

Порядок действий клиента:

- `socket()` - создает сокет
- `connect()` - установка соединения для сокета, который будет связан с серверным сокетом, порожденным вызовом `accept()`
- `sendfrom()` - отправка данных серверу
- `recvto()` - чтение тех же данных от сервера

- shutdown() - разрыв соединения с клиентом
- close() - для закрытия клиентского и слушающего сокетов

Действия клиента не меняются, однако возможно не использовать подтверждение соединения с помощью connect()

3.3. Разработка ТСП приложения по индивидуальному заданию

3.3.1. Индивидуальное задание

Необходимо разработать приложение для обмена файлами, основанное на протоколе FTP. Реализуемые функции:

- ls - получение содержимого текущего каталога
- cd - смена текущей директории
- pull - передача файла от сервера клиенту
- push - передача файла от клиента на сервер

Сначала необходимо разработать собственный протокол взаимодействия клиента и сервера. Далее необходимо написать клиент-серверное приложение (Сервер на ОС Linux, клиент на ОС Windows), работающее согласно разработанному протоколу и реализующее заданную функциональность.

3.3.2. Разработка протокола взаимодействия

После подключения клиента к серверу, второй начинает ожидать присланной ему команды от клиента. Соответственно клиент имеет 4 возможные команды, которые считываются из консоли, куда их вводит пользователь. Пользователь вводит команду следующего вида:

1 <command> <argument>

В качестве команд применяются строки "ls "cd, "pull "push". Далее, после пробела, следует аргумент. Аргументом может служить путь к директории от текущего положения клиента (для команд ls и cd), имя файла (для pull и push). После ввода команды в консоль, приложение клиента запускает процедуру распарсивания полученной строки для получения аргумента и определения, какая команда была введена. Если введенная команда не совпадает ни с одной из 4-х возможных, то выводится сообщение "Unkown command и приложение ждет ввода новой команды. Если было обнаружено совпадение с одной из возможных команд, то вызывается функция общения с сервером. Далее представлено взаимодействие клиента и сервера друг с другом при вводе различных команд.

- Команда получения содержимого текущего каталога ls

Клиент составляет команду для отправки ее на сервер. Для этого в пустую строку в 256 зарезервированных символов заносится сама команда с пробелом после ("ls "), затем к ней приписывается полный путь к текущему каталогу, в котором находится клиент на сервере, затем без пробелов приписывается введенный пользователем аргумент. Далее серверу отправляется сообщение длиной в 3 символа, в котором в строчном виде записана длина полученной команды. Сервер, принимая это сообщение, переводит полученную

строку и целочисленной значение и ждет приема сообщения указанной длины. Клиент посылает команду серверу и начинает ждать ответного сообщения длиной в 1 символ.

После приема самой команды, на стороне сервера также запускается распарсивание, в ходе которого сервер отделяет аргумент от команды, и вызывает функцию для обработки команды `ls`. Сервер делает попытку открыть директорию по указанному в аргументе пути, записывая в строку для ответа клиенту "n" при неудачной попытке открыть каталог и "y" при успешной. Далее сервер посылает клиенту это сообщение длиной в 1 символ.

Клиент принимает от сервера подтверждение ("y") или отказ ("n"). В случае отказа функция завершается и приложение снова ждет ввода команды от пользователя. В случае подтверждения клиент начинает в цикле принимать сообщения по 256 символов. Условием окончания приема становится принятая строка `"_end_of_ls"`.

Сервер же, после отправки подтверждения, начинает в цикле получать по одному имени файлов и директорий открытого каталога, записывать их в строку и отправлять клиенту по 256 символов за одно сообщение. После перебора всего содержимого сервер отправляет строку `"_end_of_ls"` и переходит в режим принятия следующей команды.

- Команда смены текущей директории каталога `cd`

Клиент составляет полную команду для отправки серверу по тому же алгоритму, что и в команде `ls` (в качестве команды с пробелом используется строка `"cd "`). Далее клиент отправляет серверу сообщение в 3 символа, в котором содержится длина получившейся команды, а затем и саму команду. После клиент переходит в режим ожидания подтверждения: ожидает прием сообщения длиной в один символ.

Сервер, после получения сообщения с длиной команды, принимает команду указанной длины и обрабатывает ее. Сервер делает попытку открыть директорию, указанную в аргументе, и составляет сообщение-ответ: "n" при неудачной попытке открытия и "y" при успешном открытии. Это сообщение отправляется клиенту. Сервер завершает обработку команды и переходит в режим ожидания следующей команды.

Клиент, после приема сообщения проверяет его: если принят символ "y" то текущая директория меняется согласно указанному пользователем аргументу (либо приписывается новый каталог, либо (если аргумент `".."`) стирается последний каталог).

- Команда взятия файла с сервера `pull`

Клиент делает попытку создать файл с указанным именем и открыть его. В случае неудачи выводится ошибка и клиент не совершает никакого обмена с сервером. Если же файл был успешно создан, то клиент составляет команду по алгоритму выше, но в качестве команды использует строку `"pull "`. Также отправляется длина полученной полной команды, а затем и сама команда. Клиент переходит в состояние ожидания подтверждения или отказа.

Сервер после принятия команды делает попытку открыть указанный в аргументе файл, и составляет сообщение-ответ: "n" при неудаче и "y" при успешном открытии файла. Данное сообщение отправляется клиенту.

Сервер после отправки подтверждения начинает читать открытый файл, максимум по 256 символов за одну итерацию цикла. Затем отправляется сообщение длиной в 3 символа, которое содержит кол-во считанных из файла символов. Далее отправляется сообщение, содержащее информацию, которую считали из файла. После окончания файла сервер посылает строку `"_end_of_file"`.

После получения подтверждения, клиент начинает в цикле прием сообщений от сервера. На каждой итерации цикла клиент принимает 2 сообщения: первое длиной в 3 символа

содержит длину информационной посылки, второе - информационная посылка. После приема информационного сообщения оно записывается в файл. Условием окончания цикла является прием строки "_end_of_file" в качестве информационного сообщения.

- Команда записи файла на сервер push

Данная команда выполняется аналогично команде pull, но клиент и сервер меняются местами в основном цикле приема-передачи: клиент в цикле читает файл, отправляет длину считанных данных и отправляет данные, а сервер осуществляет прием и запись в файл, пока не получит строку "_end_of_file".

3.3.3. Написание многопоточного TCP сервера на ОС Linux

Основная логика создания сокетов, привязка их к адресам, выделения новых потоков для общения с клиентами осталась такой же, как и в простейшем многопоточном эхо-сервере.

Изменилась функция общения с клиентом: теперь она содержит прием сразу двух сообщений подряд: длину команды и саму команду, после чего вызывает функцию парса, в которую передает принятую команду.

Функция parse разделяет принятое сообщение на команду и аргумент, проверяет команду и вызывает функцию обработки команды:

```
1 if (!(strcmp(command, "cd", 2))) direxist(sock, arg);
2 if (!(strcmp(command, "ls", 2))) ls(sock, arg);
3 if (!(strcmp(command, "pull", 4))) pull(sock, arg);
4 if (!(strcmp(command, "push", 4))) push(sock, arg);
```

Функции обработки действуют по описанным в протоколе алгоритмам:

- dirExist открывает каталог и отправляет сообщение с подтверждением или отказом.
- ls открывает каталог, отправляет подтверждение, и отправляет содержимое в сообщениях по 256 байт.
- pull открывает файл на чтение, отправляет подтверждение, читает файл по 256 байт и отправляет считанные данные, пока не прочтет весь файл, перед каждым сообщением отправляя размер текущей посылки. После окончания файла посылает сообщение-метку "_end_of_file".
- push создает файл для записи, отправляет подтверждение и начинает принимать сообщения о длине информационной посылки и саму информационную посылку, пока не будет принята метка об окончании файла. Каждое информационное сообщение записывается в созданный файл.

Для получения перевода целочисленного значения длины посылки в char[] была написана функция:

```
1 char * makeStrFromInt (int len) {
2     char buf[3];
3     char * p = buf;
4     int temp;
5     p[2] = ((int)(len % 10) + '0');
6     temp = len / 10;
7     p[1] = ((int)(temp % 10) + '0');
8     p[0] = ((int)(temp / 10) + '0');
9     return p;
10 }
```

Также необходимо было учитывать, что в командах ls, pull и push используются метки для указания о завершении выполнения команды, а значит может существовать название файла или строка в файле, которые совпадут с этой меткой. Для этого была сделана функция экранирования:

```
1 void pastShield (char * sendbuff) {
2     for (int i = strlen(sendbuff, 256) - 1; i >= 0; i--)
3         sendbuff[i + 1] = sendbuff[i];
4     sendbuff[0] = '/';
5 }
```

Если нам необходимо отправить информационную строку, которая совпадает с меткой, то вызывается эта функция.

Соответственно если было принято сообщение с "экраном" значит нам необходимо этот экран убрать:

```
1 void deleteShield (char * str) {
2     for (int i = 0; i < strlen(str, 256) - 1; i++)
3         str[i] = str[i + 1];
4 }
```

Для выполнения требования о возможности сервера отключать клиентов, а также отключаться полностью было добавлено следующее:

1. Массив сокетов
2. Массив флагов
3. Функция приема и обработки команд с серверной консоли
4. Вызов этой функции в отдельном потоке
5. Проверка флагов на отключение клиента в цикле работе с клиентом

Массив сокетов - массив целых чисел, где хранятся дескрипторы сокетов всех подключенных клиентов.

Массив флагов - флаги для определения "свободных" ячеек массива сокетов, "занятых" и "посланных на отключение".

Функция приема сообщений с консоли сервера:

```
1 void* servConsole(void* temp);
```

В качестве аргумента ей передается слушающий сокет сервера, т.к. отключение сервера предполагает закрытие этого сокета. В этой функции осуществляется считывание строк с консоли, определяется, написана ли строка "exit" или "closeXXX". В случае exit и closeall осуществляется установка флагов всех клиентов в режим закрытия (exit также закрывает слушающий сокет). Если после "close" указан номер клиента, то функция установит флаг этого клиента в состояние закрытия.

Проверка флагов осуществляется каждым потоком перед принятием очередной команды. Таким образом, закрытие сокета клиента произойдет только после окончания текущего обмена.

Данная функция вызывается в отдельном потоке перед входом в цикл ожидания подключения новых клиентов.

3.3.4. Написание клиента на ОС Windows

Сетевая часть клиента также не отличается от простейшего TCP клиента. Также изменилась функция общения с сервером: она лишь получает введенную в консоль строку и запускает для нее процедуру parse.

Процедура парса также отделяет команду от аргумента и проверяет, какая команда была введена, вызывая соответствующую функцию (если введенная команда не совпадает ни с одной из возможных, то функция возвращает константу UNKNOWN_COMMAND, что приведет к выводу соответствующего сообщения в консоль).

Функции обработки введенных команд действуют по описанным в протоколе алгоритмам:

- `dirChange` (вызывается при команде `cd`) составляет команду, отправляет ее длину, отправляет команду и ждет подтверждения. В случае успешного открытия каталога на стороне сервера меняет текущий рабочий каталог.
- `ls` составляет команду, отправляет ее длину и саму команду, после чего ждет подтверждения. В случае успешного открытия каталога на стороне сервера начинает прием посылок по 256 байт, выводя их в консоль, пока не будет принята посылка об окончании содержимого каталога.
- `pull` создает файл для записи, составляет команду, отправляет ее длину и саму команду, после чего ждет подтверждения. После приема подтверждения начинает принимать сообщения о длине информационной посылки и саму информационную посылку, пока не будет принята метка об окончании файла. Каждое информационное сообщение записывается в созданный файл.
- `push` открывает файл на чтение, составляет команду, отправляет ее длину и саму команду, после чего ждет подтверждения. При приеме подтверждения читает файл по 256 байт и отправляет считанные данные, пока не прочтет весь файл, перед каждым сообщением отправляя размер текущей посылки. После окончания файла посылает сообщение-метку `"_end_of_file"`.

Также как и в сервере здесь необходимы функции экранирования (`pastShield` и `deleteShield`) и перевода целого числа в `char[]`.

Т.к. клиент в каждой функции обработки запроса должен составлять команду, то это действие было вынесено в отдельную функцию:

```
1 int makePost(char * post, char * com, char * arg) {  
2     memset(post, 0, 256);  
3     strncat(post, com, strlen(com));  
4     strncat(post, addr, strlen(addr));  
5     strncat(post, arg, 256 - strlen(com) - strlen(arg));  
6     return 1;  
7 }
```

Принимая указатели на строку для записи итоговой команды, строки, содержащие команду и аргумент, эта функция записывает в строку для итоговой команды саму команду с пробелом, затем полный путь к текущей директории, затем аргумент.

3.4. Разработка UDP приложения по индивидуальному заданию

3.4.1. Индивидуальное задание

Функциональные требования к приложению остались теми же, как и у TCP приложения. Отличие от TCP приложения: необходимо добавить контроль номера принятых и

посланных пакетов, чтобы определить случаи потери или перемешивания пакетов.

3.4.2. Разработка протокола взаимодействия

Основные особенности протокола в формате и длине пакетов остались без изменений, однако для каждого отправленного пакета первые 2 байта резервируются для номера пакета. Таким образом, максимальная длина команд и данных для отправки становится 254 байта вместо 256.

3.4.3. Написание UDP сервера

UDP сервер был написан на основе TCP сервера, описанного выше. Отличия от TCP сервера:

1. Вызывать функцию работы с клиентами в основном потоке, т.к. нет необходимости в создании новых сокетов для клиентов. Все запросы от клиентов будут помещаться в очередь и обрабатываться последовательно.
2. Вставлять в отправляемые пакеты их номер в текущей транзакции
3. Проверять номер принятых пакетов
4. Передавать в функции обработки команд данные о клиенте, с которым идет обмен сообщениями.

Для работы с номерами пакетов были написаны 2 функции: для вставки номера пакета в сообщение и для проверки номера принятого пакета.

```
1 void pastPacketNumber(char * post, int number, int len) {
2     for (int i = len - 1; i > 1; i--) {
3         post[i] = post[i-2];
4     }
5     post[0] = (int)(number / 10) + '0';
6     post[1] = number % 10 + '0';
7 }
```

Функция принимает ссылку на сообщение, номер, который необходимо вставить в сообщение и длину сообщения. Функция "сдвигает" сообщение на 2 символа и в освободившиеся 2 байта вставляет номер пакета.

```
1 int checkRec(char * mes, int counter) {
2     char temp[2];
3     temp[0] = mes[0];
4     temp[1] = mes[1];
5     for (int i = 0; i < strlen(mes); i++) {
6         mes[i] = mes[i+2];
7     }
8     int co = atoi(temp);
9     if (co == counter + 1) return 0;
10    printf("Ошибка_приема_-_неверный_порядок_пакетов\n");
11    return 1;
12 }
```

Функция проверки номера принятого пакета принимает ссылку на буфер с принятым сообщением и номер пакета, уменьшенный на единицу. Осуществляется проверка номера принятого пакета и номер стирается из сообщения.

Также была изменена функция обработки команды на стороне сервера о завершении сервера. Данная функция просто осуществляет считывание команды с консоли и закрывает сокет, если была написана необходимая команда - "close".

3.4.4. Написание UDP клиента

UDP клиент также написан на основе TCP клиента. Однако также как и для сервера необходимо было добавить функции для вставки номера пакета в сообщения и проверки номера пакета в принятом сообщении.

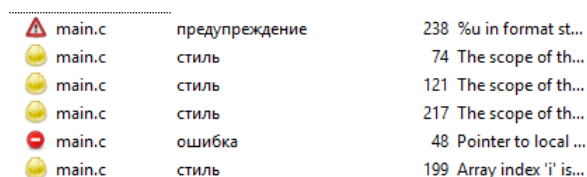
3.5. Дополнительное задание

3.5.1. Описание задания

Проанализировать код с помощью статического и динамического анализатора. Проверить код с помощью `srpcheck`, описать полученные ошибки и предупреждения, предложить вариант исправления. Также проанализировать работу программ на предмет утечки памяти с помощью утилиты `valgrind`.

3.5.2. Выполнение задания

Проверим исходный код TCP клиента по индивидуальному заданию с помощью `srpcheck`. В ходе проверки программа выдала 6 предупреждений:



	main.c	предупреждение	238 %u in format st...
	main.c	стиль	74 The scope of th...
	main.c	стиль	121 The scope of th...
	main.c	стиль	217 The scope of th...
	main.c	ошибка	48 Pointer to local ...
	main.c	стиль	199 Array index 'i' is...

Рисунок 3.1. Предупреждения от `srpcheck` для кода TCP клиента

Результат содержит 4 стилистических предупреждения.

3 из них указывают на то, что переменная, объявляемая в начале функции, инициализируется и используется лишь после проверки условия. Т.е. утилита говорит о том, что область видимости этой переменной можно уменьшить, объявляя ее уже после проверки условия.

Последнее стилистическое предупреждение говорит о том, что в цикле `for` в качестве переменной для перебора используется переменная, которая была объявлена и использована до вызова `for`. Данной предупреждение исправить нельзя: в функции `parse` есть необходимость перебирать в цикле строку не с первого символа, а с того, на котором закончит предыдущий цикл, т.е. использовать ту же самую переменную.

Также присутствует одно предупреждение: в строке, выводимой в консоль, требуется указать переменную типа `unsigned int`, а на деле ей передается просто `int` переменная. Исправить легко: переделать выводимую строку так, чтобы она принимала `int`, или объявить переменную как `unsigned int`.

Последнее предупреждение `srpcheck` относится к ошибкам. Оно указывает на функцию получения строки в 3 символа из числа. Дело в том, что данная функция возвращает указатель на массив `char`, который создается на стеке. После выхода из функции данные в памяти могут быть стерты, что приведет к неверному возвращаемому значению. Решить это можно выделением памяти с помощью `malloc` или же передавая в функцию указатель, по которому будет записываться требуемое значение. Для исправления был выбран второй способ, т.к. первый требует обязательного освобождения выделенной памяти каждый раз после вызова функции.

Проверим также UDP сервер:

main.c	стиль	148 The scope of th...
main.c	ошибка	42 Pointer to local ...
main.c	предупреждение	156 Dangerous usa...
main.c	стиль	67 Array index 'i' is...

Рисунок 3.2. Предупреждения от crrccheck для кода UDP сервера

Ошибка и 2 стилистических предупреждения совпадают с указанными выше, а вот предупреждение новое: оно отсылается к функции `strncat`, точнее к ее третьему аргументу. Предполагается, что третьим аргументом будет задано максимальное число байт для записи в строку, однако в данном месте кода в качестве третьего аргумента указана длина буфера, в который производится запись. Суть предупреждения в том, что если буфер был не пустой, то при записи максимального числа байт, которая начнется не с начала буфера, можно переписать участки памяти, располагающиеся за буфером. В данной программе это не приводит к ошибке, потому что данная запись производится в пустую строку и всегда начинается с нуля. Однако в общем случае можно в качестве третьего аргумента передавать максимальную длину массива, уменьшенную на число уже записанных в него символов (его фактическую длину).

Далее осуществим динамическую проверку на предмет утечек памяти с помощью утилиты `valgrind`. Для начала проверим многопоточный TCP сервер (подключимся клиентом к серверу и попробуем сделать каждую возможную операцию):

```
user@user-VirtualBox:~/networkslab/tcp_template/tcp_template/ftp_tcp_serv/cmake-  
build-debug$ valgrind ./ftp_tcp_serv  
==21300== Memcheck, a memory error detector  
==21300== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.  
==21300== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info  
==21300== Command: ./ftp_tcp_serv  
==21300==  
^C==21300==  
==21300== Process terminating with default action of signal 2 (SIGINT)  
==21300== at 0x4E4A76D: ??? (syscall-template.S:84)  
==21300== by 0x4022E7: main (main.c:293)  
==21300==  
==21300== HEAP SUMMARY:  
==21300== in use at exit: 544 bytes in 2 blocks  
==21300== total heap usage: 9 allocs, 7 frees, 137,480 bytes allocated  
==21300==  
==21300== LEAK SUMMARY:  
==21300== definitely lost: 0 bytes in 0 blocks  
==21300== indirectly lost: 0 bytes in 0 blocks  
==21300== possibly lost: 544 bytes in 2 blocks  
==21300== still reachable: 0 bytes in 0 blocks  
==21300== suppressed: 0 bytes in 0 blocks  
==21300== Rerun with --leak-check=full to see details of leaked memory  
==21300==  
==21300== For counts of detected and suppressed errors, rerun with: -v  
==21300== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Рисунок 3.3. Предупреждения от valgrind для кода TCP сервера

Утилита показала отсутствие ошибок, но посоветовала запустить полную проверку с помощью ключа `-leak-check=full`. Итоги такой проверки:

```

user@user-VirtualBox:~/networkslab/tcp_template/tcp_template/ftp_tcp_serv/cmake-
build-debug$ valgrind --leak-check=full ./ftp_tcp_serv
==21314== Memcheck, a memory error detector
==21314== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==21314== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==21314== Command: ./ftp_tcp_serv
==21314==
^C==21314==
==21314== Process terminating with default action of signal 2 (SIGINT)
==21314==   at 0x4E4A76D: ??? (syscall-template.S:84)
==21314==   by 0x4022E7: main (main.c:293)
==21314==
==21314== HEAP SUMMARY:
==21314==   in use at exit: 544 bytes in 2 blocks
==21314==   total heap usage: 7 allocs, 5 frees, 132,832 bytes allocated
==21314==
==21314== 272 bytes in 1 blocks are possibly lost in loss record 1 of 2
==21314==   at 0x4C2FB55: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64
-linux.so)
==21314==   by 0x40138A4: allocate_dtv (dl-tls.c:322)
==21314==   by 0x40138A4: _dl_allocate_tls (dl-tls.c:539)
==21314==   by 0x4E4226E: allocate_stack (allocatestack.c:588)
==21314==   by 0x4E4226E: pthread_create@@GLIBC_2.2.5 (pthread_create.c:539)
==21314==   by 0x40229E: main (main.c:284)
==21314==
==21314== 272 bytes in 1 blocks are possibly lost in loss record 2 of 2
==21314==   at 0x4C2FB55: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64
-linux.so)
==21314==   by 0x40138A4: allocate_dtv (dl-tls.c:322)
==21314==   by 0x40138A4: _dl_allocate_tls (dl-tls.c:539)
==21314==   by 0x4E4226E: allocate_stack (allocatestack.c:588)
==21314==   by 0x4E4226E: pthread_create@@GLIBC_2.2.5 (pthread_create.c:539)
==21314==   by 0x402368: main (main.c:306)
==21314==
==21314== LEAK SUMMARY:
==21314==   definitely lost: 0 bytes in 0 blocks
==21314==   indirectly lost: 0 bytes in 0 blocks
==21314==   possibly lost: 544 bytes in 2 blocks
==21314==   still reachable: 0 bytes in 0 blocks
==21314==   suppressed: 0 bytes in 0 blocks
==21314==
==21314== For counts of detected and suppressed errors, rerun with: -v
==21314== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)

```

Рисунок 3.4. Предупреждения от valgrind для кода ТСП сервера

Видим, что может происходить потеря 544 байт информации. Valgrind указывает на строки, где может происходить потеря: оба сообщения показывают на функцию создания нового потока `pthread_create`.

Дело в том, что при создании нового потока происходит выделение памяти для него, копирование контекста, создание необходимых структур. После его завершения память должна очищаться, однако после вызова `pthread_detach` поток не закрывается сразу, а переходит в режим "зомби". Таким образом, пока поток находится в этом состоянии, за ним все еще зарезервирована память. Через какое-то время система сама уберет поток из таблицы процессов и освободит память.

4. Выводы

В ходе выполнения работы были разработаны 2 клиент-серверных приложения для обмена файлами: на ТСП и UDP сокетах. Приложения работали согласно заданию, реализуют разработанный протокол, а также отлавливая ошибки при работе с сетью: ошибки при создании сокетов, соединении, приеме и передаче информации.

Особенности задания для приложений: обеспечить перемещение по каталогам сервера, просмотр содержимого каталога, обмен файлами с сервером (возможность как брать файлы с сервера, так и загружать файлы на сервер). ТСП сервер должен создавать отдельный сокет для каждого нового клиента, а также обмениваться сообщениями в отдельных потоках для каждого клиента. Также в ТСП предусматривается возможность отключать всех клиентов, и любого клиента в отдельности. UDP приложение не предусматривает общение с клиентами в отдельных потоках, а также производит все через один сокет, но необходимо

реализовать контроль пакетов: выявлять случаи потери или перемешивания пакетов.

При разработке протокола было принято решение о поддержании сервером 4-х команд для необходимых функций. Для удобства формирования и обработки команд все они были приведены к одному формату: команда + пробел + аргумент. Также для удобства было введено ограничение на длину команды в 256 байт.

При разработке приложения были написаны отдельные функции, вызываемые для каждой команды, для реализации исполнения этой команды как на стороне сервера, так и на стороне клиента. Это, а также общий формат для всех команд, позволило использовать и на стороне сервера и на стороне клиента одну функцию для обработки сообщения и определения введенной команды. Далее обмен производится в вызываемых при этом функциях. Однозначное соответствие функций клиента и сервера позволило облегчить как написание приложения, так и его отладку.

Для выполнения требования отключения клиентов (всех или одного) на TCP сервер были добавлены массивы для хранения сокетов (которые необходимо закрывать по требованию сервера) и для хранения флагов (флаги указывают, занят ли сокет клиентом, свободен или требуется закрытие этого сокета). Такая реализация позволила отключать всех клиентов лишь после завершения обработки текущей команды (проверка флагов проводилась между обработками принятых команд), т.е. сохранить целостность данных, не отключая клиента прямо во время отправки ему файла.

Для добавления контроля принятых пакетов на UDP приложении пришлось не только зарезервировать 2 первых байта всех посылок для записи в них номера пакета, но и учесть, что в таком случае максимальная длина команд (а также кусков файлов для передачи за 1 сообщение) стала равна 254. Также пришлось учесть и изменить функции работы со строками, чтобы не происходило "затирание" информации, которая лежит за пределами массива символов и может являться другой важной переменной программы.

В ходе разработки приложения я столкнулся с проблемой записи информации в файл и ее корректного сохранения там. Команды записи в файл должны записывать ровно столько байт, сколько было указано (если указать меньше байт, чем нужно, произойдет потеря информации, а если больше - запись не произойдет, пока не будет получено достаточное кол-во байт). Для решения этой проблемы в обмен непосредственно файлами были включены отправки длины посылаемых информационных строк. На приемной стороне, соответственно, происходил и прием только нужного кол-ва символов, и их запись в файл. Также решение отправлять длину отправки перед самой отсылкой позволило не отправлять по сети множество нуль-символов в тех случаях, когда сообщение имеет меньшую длину.