

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
с помощью протоколов TCP и UDP

Работу

выполнил:

Сафонов С.В.

Группа: 43501/1

Преподаватель:

Алексюк А.О.

Санкт-Петербург
2018

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	8
3.3.3. Написание многопоточного TCP сервера на ОС Linux	9
3.3.4. Написание клиента на ОС Linux	14
3.4. Разработка UDP приложения по индивидуальному заданию	15
3.4.1. Индивидуальное задание	15
3.4.2. Разработка протокола взаимодействия	15
3.5. Дополнительное задание	16
3.5.1. Выполнение задания	16
4. Выводы	23

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 Linux

4. Разработка UDP приложения по индивидуальному заданию

- Индивидуальное задание
- Разработка протокола взаимодействия

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 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 TCP 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* connection_handler (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(  
4         ↪ cli_addr)));  
5     .....  
6 }
```

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_create(&tid, NULL, connection_handler, &newsockfd);  
5     pthread_detach(tid);  
6 }
```

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()`
- `sendto()` - отправка данных серверу
- `recvfrom()` - 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

В качестве индивидуального задания была выбрана система распределенных математических расчетов. Сервер и клиент реализованы на ОС Linux.

Серверное приложение реализует следующие функции:

- Прослушивание определенного порта
- Обработка запросов на подключение по этому порту от клиентов
- Поддержка одновременной работы нескольких клиентов с использованием механизма нитей и средств синхронизации доступа к разделяемым между нитями ресурсам
- Принудительное отключение конкретного клиента
- Хранение рассчитанных клиентами простых чисел, а также текущей нижней границы диапазона для нового запроса на расчет
- Выдача клиентам максимального рассчитанного простого числа
- Выдача клиентам последних N рассчитанных простых чисел
- Выдача клиентам необходимого диапазона расчета чисел
- * Сохранение состояния при выключении сервера

Клиентское приложение реализует следующие функции:

- Возможность параллельной работы нескольких клиентов с одного или нескольких IP-адресов
- Установление соединения с сервером (возможно, с регистрацией на сервере)
- Разрыв соединения
- Обработка ситуации отключения сервером
- Получение от сервера и вывод максимально рассчитанного простого числа
- Получение от сервера и вывод последних N рассчитанных простых чисел
- Получение от сервера диапазона расчета простых чисел (нижнюю грань выдает сервер, количество проверяемых чисел - клиент)
- Расчет простых чисел в требуемом диапазоне (имеет смысл проверять остатки от деления на все нечетные числа в пределах $\sqrt{N_{\max}}+1$)
- Передача серверу набора рассчитанных простых чисел

Сначала необходимо разработать собственный протокол взаимодействия клиента и сервера. Далее необходимо написать клиент-серверное приложение, 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 10 возможные команды, которые считываются из консоли, куда их вводит пользователь. Пользователь вводит команду следующего вида:

```
1 <command> <argument>
```

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 10 возможных, то выводится сообщение "Unknown type of request" 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 регистрации пользователя в базе сервера (REG <login>)

Клиент посылает посылку серверу с именем пользователя, которого нужно занести в папку пользователей. После этого сервер проверяет запрос на ошибки и на существование данного имени пользователя в базе. При наличии имени клиенту высылается ответ с сообщением, иначе сервер создаёт бинарный файл с именем соответствующего пользователя и файл сессии для работы с ним. Клиент получает результат запроса и логинится под данным именем для дальнейшей работы (большинство запросов не выполняются для пользователей со статусом 'new-user').

- Команда авторизации на сервере по имени пользователя (LOGIN <login>)

Алгоритм работы аналогичен работе запроса о регистрации пользователя, за исключением того, что сервер проверяет и высылает уведомление клиенту о несуществующем пользователе в ходе проверки своей базы.

- Команда удаления текущего пользователя из базы и его отключение сервером (DEL)

После получения запроса сервер проверяет наличие клиента, который послал запрос, в текущей сессии. В случае отсутствия высылается уведомление, иначе сервер удаляет бинарный файл с именем клиента из базы. После получения ответа клиент получает статус 'not-logged' и также, как и 'new-user', должен предварительно зарегистрироваться или авторизоваться для дальнейшей работы.

- Команда передачи максимального вычисленного простого числа (MAXPRIME)

Сервер просматривает все вычисленные простые числа с целью найти максимальное (реализуется простой поиск максимума в массиве). После этого найденный максимум высылается клиенту и выводится в консоль.

- Команда передачи последних вычисленных N простых чисел (PRIMES <N>)

Сервер просматривает все вычисленные простые числа с целью найти максимальное (реализуется простой поиск максимума в массиве) и находит его индекс. После этого в цикле он производит выборку последних N простых чисел, начиная с максимума в обратную сторону. Полученный массив чисел высылается клиенту и выводится в консоль.

- Команда передачи значения диапазона для вычислений (RANGE)

Сервер высылает клиенту значение диапазона, на котором будет выполняться вычисление простых чисел.

- Команда выполнения расчёта простых чисел клиентом (CALC)

В начале сервер высылает клиенту сообщение с рассчитанным диапазоном расчёта простых чисел, а также текущий диапазон и диапазон для расчёта. После получения посылки клиентом реализуется алгоритм вычисления и записи полученных значений в символьный массив. Полученный массив пересылается на сервер, где значения записываются в текущую базу значений. В конце производится запись всей текущей базы в файл, отвечающий за состояние сервера, из которого сервер вытаскивает данные при начале нового сеанса.

Для оптимизации работы и исключения появления ошибок сегментации было решено хранить в базе сервера 50 последних выполненных простых чисел.

- Команда очистки данных с сервера (CLEAR)

Сервер обнуляет массив хранения простых чисел и текущий диапазон расчёта в текущей базе значений.

- Команда разрыва соединения клиентом (QUIT)

На стороне сервера удаляется сессия работы с текущим клиентом и закрывается поток и сокет. Клиент завершает свою работу.

- Команда вызова справки (HELP)

В консоль выводится описания всех возможных команд клиент-серверного приложения.

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.

Для отслеживания запросов и ответов, а также для определения команды были созданы соответствующие структуры:

- Структура команд хранит в себе один из десяти типов команды и аргумент (если он присутствует).

```
1 struct command {  
2     char* type;  
3     char* arg1;  
4 };  
5
```

- Структура запроса содержит поле типа структуры команд и токен сессии, в которой работает текущий клиент.

```

1  struct request {
2      struct command comm;
3      char* token;
4  };
5

```

- Структура реакции на запрос хранит тип возвращаемого ответа (ошибка, нормальная работа, удаление и текущий токен при авторизации) и содержимое ответа.

```

1  struct response {
2      char* type;
3      char* payload;
4  };
5

```

Для отслеживания клиентов при каждом появлении нового пользователя создаётся и присваивается новый поток, в котором выполняется обработка запросов от него до тех пор, пока клиент сам не захочет закончить сеанс.

Процедура приёма запроса заключается в принятии длины ответа и бинарного массива, в котором содержатся данные ответа. Большую часть тела процедуры занимает преобразование бинарного массива в соответствующую структуру.

```

1  int get_request(int sockfd, struct request* req)
2  {
3      int readRes; // Result of reading
4      char* buf; // Buffer for reading
5      int message_length; // Length of message without first sizeof(int) bytes
6
7      int arg_length;
8      int buf_pointer = 0;
9
10     // Get length of request
11     buf = (char*) malloc(sizeof(int));
12     readRes = read_socket(sockfd, buf, sizeof(int));
13
14     if (readRes != WORKING_SOCKET) {
15         return readRes;
16     }
17
18     message_length = *(int*)buf;
19     if (message_length <= 0 || message_length > 1024) {
20         return REQUEST_LENGTH_ERROR;
21     }
22
23     // Clear memory
24     free(buf);
25
26     // Read byte array of request
27     buf = (char*) malloc(message_length * sizeof(char));
28
29     readRes = read_socket(sockfd, buf, message_length);
30
31     if (readRes != WORKING_SOCKET) {
32         return readRes;
33     }

```

```

34
35 // Now convert byte array to request struct
36 // Get length of type
37 bcopy(&buf[buf_pointer], &arg_length, sizeof(int));
38 buf_pointer += sizeof(int);
39
40 // Get type of request
41 req->comm.type = (char*) malloc(arg_length * sizeof(char) + 1);
42 bzero(req->comm.type, arg_length * sizeof(char) + 1);
43 bcopy(&buf[buf_pointer], req->comm.type, arg_length * sizeof(char));
44 buf_pointer += arg_length * sizeof(char);
45
46 // Get length of arg1
47 bcopy(&buf[buf_pointer], &arg_length, sizeof(int));
48 buf_pointer += sizeof(int);
49
50 // Get arg1
51 if (arg_length > 0) {
52     req->comm.arg1 = (char*) malloc(arg_length * sizeof(char) + 1);
53     bzero(req->comm.arg1, arg_length * sizeof(char) + 1);
54     bcopy(&buf[buf_pointer], req->comm.arg1, arg_length * sizeof(char));
55     buf_pointer += arg_length;
56 } else {
57     req->comm.arg1 = 0;
58 }
59
60 // Get token
61 if (message_length - buf_pointer == 0) {
62     req->token = 0;
63 } else {
64     arg_length = message_length - buf_pointer;
65     req->token = (char*) malloc(arg_length * sizeof(char) + 1);
66     bzero(req->token, arg_length * sizeof(char) + 1);
67     bcopy(&buf[buf_pointer], req->token, arg_length * sizeof(char));
68 }
69
70 // Clear memory
71 free(buf);
72
73 return WORKING_SOCKET;
74 }

```

После определения типа запроса выполняется одна из 10 операций, каждая из которых заканчивается отправкой результата работы клиента. За отправку отвечает процедура `response_request`.

```

1 int response_request(int sockfd, char* type, char* payload)
2 {
3     int type_length; // Length of response type
4     int payload_length; // Length of payload type
5     int length; // Total length of response
6     char* buf; // Buffer for response
7
8     int res;
9     int buf_pointer = 0;
10
11     // Set type length
12     type_length = strlen(type);
13
14     // Set payload length

```

```

15  payload_length = strlen(payload);
16
17  // Count total length of response
18  length = sizeof(int) * 2 + (type_length + payload_length) * sizeof(char);
19
20  // Send length to client
21  res = send(sockfd, &length, sizeof(int), NULL);
22  if (res < 0) {
23      return WRITING_ERROR;
24  }
25
26  // Allocate memory
27  buf = (char*) malloc(length);
28  bzero(buf, length);
29
30  // Set response type length
31  bcopy(&type_length, &buf[buf_pointer], sizeof(int));
32  buf_pointer += sizeof(int);
33
34  // Set response type
35  bcopy(type, &buf[buf_pointer], type_length * sizeof(char));
36  buf_pointer += type_length * sizeof(char);
37
38  // Set response payload length
39  bcopy(&payload_length, &buf[buf_pointer], sizeof(int));
40  buf_pointer += sizeof(int);
41
42  // Set response payload
43  bcopy(payload, &buf[buf_pointer], payload_length * sizeof(char));
44  buf_pointer += payload_length * sizeof(char);
45
46  // Send response to client
47  res = send(sockfd, buf, length, NULL);
48  free(buf);
49
50  if (res < 0) {
51      return WRITING_ERROR;
52  }
53
54  return WORKING_SOCKET;
55 }

```

Для описания реакций на всевозможные ошибки, возникающие в ходе работы приложения, была создана отдельная процедура-обработчик ошибок. В связи с этим клиент может точно отобразить тип ошибки в рабочей консоли.

```

1  int handle_errors(int sockfd, int error)
2  {
3      switch (error) {
4          case OK:
5              return 0;
6          case USER_ALREADY_EXISTS:
7              send_response(sockfd, RESPONSE_ERROR, "User_already_exists");
8              return 1;
9          case USER_NOT_FOUND:
10             send_response(sockfd, RESPONSE_ERROR, "User_not_found");
11             return 1;
12          case OPEN_FILE_ERROR:
13             send_response(sockfd, RESPONSE_ERROR, "Can't_open_file");
14             return 1;

```

```

15  case READ_FILE_ERROR:
16      send_response(sockfd , RESPONSE_ERROR, "Can't_read_from_file");
17      return 1;
18  case WRITE_FILE_ERROR:
19      send_response(sockfd , RESPONSE_ERROR, "Can't_write_to_file");
20      return 1;
21  case CLOSE_FILE_ERROR:
22      send_response(sockfd , RESPONSE_ERROR, "Can't_close_file");
23      return 1;
24  case BOUND_IS_NOT_MULTIPLE:
25      send_response(sockfd , RESPONSE_ERROR, "Bound_must_be_multiple_to_range_or_
↪ equal_0");
26      return 1;
27  case RANGE_IS_ALREADY_USED:
28      send_response(sockfd , RESPONSE_ERROR, "Bound_is_already_used");
29      return 1;
30  case NEGATIVE_BOUND:
31      send_response(sockfd , RESPONSE_ERROR, "Bound_can't_be_negative");
32      return 1;
33  case INCORRECT_BOUNDS:
34      send_response(sockfd , RESPONSE_ERROR, "Upper_bound_must_be_larger_than_
↪ lower_bound");
35      return 1;
36  case INCORRECT_COUNT:
37      send_response(sockfd , RESPONSE_ERROR, "Count_must_be_larger_than_0_and_
↪ less_than_array_of_primes_size");
38      return 1;
39  case OTHER_ERROR:
40      send_response(sockfd , RESPONSE_ERROR, "Internal_error");
41      return 1;
42  case READING_ERROR:
43      close_socket(sockfd , "ERROR_while_reading_from_socket");
44      pthread_exit(0);
45  case WRITING_ERROR:
46      close_socket(sockfd , "ERROR_while_writing_to_socket");
47      pthread_exit(0);
48  case READING_IS_NOT_FINISHED:
49      close_socket(sockfd , "Client_closed_connection");
50      pthread_exit(0);
51  case REQUEST_LENGTH_ERROR:
52      close_socket(sockfd , "ERROR_in_request_length");
53      pthread_exit(0);
54  default:break;
55 }
56 return 0;
57 }

```

В дополнение ко всему, отдельными процедурами были реализованы проверка корректности полученного вместе с запросом аргумента, который зависит от типа команды и крайних значений, и проверка достоверности токена сессии, в которой работает текущий клиент.

```

1  // Function for checking arguments
2  int check_arguments(int sockfd , struct request req)
3  {
4      // Check if arguments are NULL
5
6      if (req.comm.arg1 == NULL) {
7          send_response(sockfd , RESPONSE_ERROR, "Illegal_argument");

```

```

8     return 1;
9 }
10 return 0;
11 }
12
13 // Function for checking token
14 int check_token(int sockfd, struct request req)
15 {
16     // Check if token is NULL
17     if (req.token == NULL) {
18         send_response(sockfd, RESPONSE_ERROR, "You_are_not_logged");
19         return 1;
20     }
21     return 0;
22 }

```

3.3.4. Написание клиента на ОС Linux

При соединении клиента с сервером в начале определяется длина запроса с целью проверки целостности послыки, а затем сам запрос, после чего вызывается функцию разделения.

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

```

1 int parse_request(char* str, struct request* req)
2 {
3     char* str_token; // First found token in the string str
4
5     // Get type of request
6     str_token = strtok(str, "\n");
7     if (str_token == NULL)
8         return INVALID_REQUEST;
9     req->comm.type = str_token;
10
11     // Get arg1
12     str_token = strtok(NULL, "\n");
13     if (str_token == NULL) {
14         req->comm.arg1 = 0;
15         return WORKING_SOCKET;
16     }
17     req->comm.arg1 = str_token;
18
19     return WORKING_SOCKET;
20 }

```

После записи токена сессии в структуру запроса, выполняется операция отправки запроса серверу, по структуре схожая с процедурой отправки ответа клиенту.

Процедура приёма ответа аналогична по структуре процедуре приёма запроса сервером.

Последние операции, которые выполняет клиент - это вызов обработчиков, зависящих от типа полученного ответа от сервера или выбора клиентом команды `QUIT`.

```

1 // Handle response
2 if (strcmp(resp.type, RESPONSE_TOKEN) == 0) {
3     strcpy(token, resp.payload);
4     strcpy(login, req.comm.arg1);
5     printf("You_are_logged_as_%s\n", login);

```

```

6  }
7
8  if (strcmp(resp.type, RESPONSE_DELETED) == 0) {
9      bzero(token, 50);
10     printf("Session_is_over\n");
11     strcpy(login, "not-logged");
12 }
13
14 if (strcmp(resp.type, RESPONSE_OK) == 0) {
15     printf("%s\n", resp.payload);
16 }
17
18 if (strcmp(resp.type, RESPONSE_ERROR) == 0) {
19     printf("ERROR: %s\n", resp.payload);
20 }
21
22 if (strcmp(req.comm.type, "QUIT") == 0) {
23     break;
24 }

```

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 структур запроса и ответа введено поле, отвечающее за номер сообщения.

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

1. Для первого запроса записывается номер, равный 1 и который увеличивается при каждом новом запросе
2. После обработки запроса от клиента сервер записывает номер ответа, равный номеру запроса
3. Клиент получает ответ от сервера и первоначально сравнивает номера запроса и ответа
4. В случае несовпадения происходит обработка ошибки о потере и перемешивании пакетов

Для отправки номера сообщения в метод `send_request` добавлен вызов функции `sendto`.

```

1 res = sendto(sockfd, &index, sizeof(int), 0, (struct sockaddr *) serv_addr,
2   ↪ sizeof(*serv_addr));
3 if (res == -1) {
4     close_socket(sockfd, "ERROR_write_length_to_socket");
5     return -1;
6 }
7 req->index = index;
8 }

```


В функции приёма ответа от сервера добавлена проверка на соответствие номеров запроса и ответа.

```
1 buf = (char*) malloc(sizeof(int));
2 res = read_socket(sockfd, buf, sizeof(int), serv_addr);
3 // Throw errors
4 if (res != WORKING_SOCKET) {
5     return res;
6 }
7
8 num = *(int*) buf;
9 if (num != req->index) {
10     return LOST_OR_WRONG_PACKET;
11 }
12 resp->index = num;
```

3.5. Дополнительное задание

Попробовать проанализировать код с помощью статического и динамического анализатора. Когда закончите индивидуальные задания, проверьте исходный код своих программ с помощью clang-tidy и cppcheck. При запуске утилит включайте все доступные проверки. После этого проверьте с помощью valgrind свои программы на предмет утечек памяти и неправильного использования многопоточности. Опишите все найденные ошибки в отчете, а также укажите, как их можно исправить.

3.5.1. Выполнение задания

- Статические анализаторы кода

Статический анализ кода - это процесс выявления ошибок и недочетов в исходном коде программ. Он заключается в совместном внимательном чтении исходного кода и высказывании рекомендаций по его улучшению. В процессе чтения кода выявляются ошибки или участки кода, которые могут стать ошибочными в будущем. Главное преимущество статического анализ состоит в возможности существенной снижении стоимости устранения дефектов в программе.

Задачи, решаемые программами статического анализа кода можно разделить на 3 категории:

1. Выявление ошибок в программах
2. Рекомендации по оформлению кода
3. Подсчет метрик

Преимущества статических анализаторов кода:

- Полное покрытие кода. Статические анализаторы проверяют даже те фрагменты кода, которые получают управление крайне редко. Такие участки кода, как правило, не удастся протестировать другими методами.
- Статический анализ не зависит от используемого компилятора и среды, в которой будет выполняться скомпилированная программа. Это позволяет находить скрытые ошибки, которые могут проявить себя только через несколько лет.

- Можно легко и быстро обнаруживать опечатки и последствия использования Copy-Paste. Как правило, нахождение этих ошибок другими способами является крайне неэффективной тратой времени и усилий.

В качестве статических анализаторов были выбраны утилиты clang-tidy и cppcheck в связи с тем, что приложение было написано на языке C. Проверка была проведена для варианта индивидуального задания для протокола TCP.

Проверка сервера с помощью clang-tidy:

```

1  /home/mrsandman/CLionProjects/server_linux/main.c:47:27: warning:
   ↳ statement should be inside braces [google-readability-braces-around-
   ↳ statements]
2  if (strlen(buff) != 0) retrieve_data(&data, buff);
3  ^
4  {
5  /home/mrsandman/CLionProjects/server_linux/main.c:48:9: warning:
   ↳ statement should be inside braces [hicpp-braces-around-statements]
6  else clear_data(&data);
7  ^
8  {
9  /home/mrsandman/CLionProjects/server_linux/main.c:58:25: warning: prefer
   ↳ accept4() to accept() because accept4() allows SOCK_CLOEXEC [android-
   ↳ cloexec-accept]
10 while ((newsockfd = accept(sockfd, (struct sockaddr*)&cli_addr, &clilen))
   ↳ ) {
11 ~~~~~~
12 accept4(sockfd, (struct sockaddr*)&cli_addr, &clilen, SOCK_CLOEXEC)
13 /home/mrsandman/CLionProjects/server_linux/main.c:92:40: warning:
   ↳ statement should be inside braces [google-readability-braces-around-
   ↳ statements]
14 if (handle_errors(sockfd, res))
15 ^
16 {
17 /home/mrsandman/CLionProjects/server_linux/main.c:126:38: warning:
   ↳ statement should be inside braces [readability-braces-around-
   ↳ statements]
18 if (check_arguments(sockfd, req))
19 ^
20 {
21 /home/mrsandman/CLionProjects/server_linux/main.c:145:36: warning:
   ↳ statement should be inside braces [google-readability-braces-around-
   ↳ statements]
22 if (handle_errors(sockfd, res))
23 ^
24 {
25 /home/mrsandman/CLionProjects/server_linux/main.c:151:36: warning:
   ↳ statement should be inside braces [google-readability-braces-around-
   ↳ statements]
26 if (handle_errors(sockfd, res))
27 ^
28 {
29 /home/mrsandman/CLionProjects/server_linux/main.c:162:38: warning:
   ↳ statement should be inside braces [google-readability-braces-around-
   ↳ statements]
30 if (check_arguments(sockfd, req))
31 ^
32 {

```

```

33 /home/mrsandman/CLionProjects/server_linux/main.c:174:36: warning:
    ↳ statement should be inside braces [readability-braces-around-
    ↳ statements]
34 if (handle_errors(sockfd, res))
35 ^
36 {
37 /home/mrsandman/CLionProjects/server_linux/main.c:180:36: warning:
    ↳ statement should be inside braces [hicpp-braces-around-statements]
38 if (handle_errors(sockfd, res))
39 ^
40 {
41 /home/mrsandman/CLionProjects/server_linux/main.c:190:34: warning:
    ↳ statement should be inside braces [google-readability-braces-around-
    ↳ statements]
42 if (check_token(sockfd, req))
43 ^
44 {
45 /home/mrsandman/CLionProjects/server_linux/main.c:196:36: warning:
    ↳ statement should be inside braces [google-readability-braces-around-
    ↳ statements]
46 if (handle_errors(sockfd, res))
47 ^
48 {
49 /home/mrsandman/CLionProjects/server_linux/main.c:199:36: warning:
    ↳ statement should be inside braces [google-readability-braces-around-
    ↳ statements]
50 if (handle_errors(sockfd, res))
51 ^
52 {
53 /home/mrsandman/CLionProjects/server_linux/main.c:204:36: warning:
    ↳ statement should be inside braces [google-readability-braces-around-
    ↳ statements]
54 if (handle_errors(sockfd, res))
55 ^
56 {
57 /home/mrsandman/CLionProjects/server_linux/main.c:214:34: warning:
    ↳ statement should be inside braces [google-readability-braces-around-
    ↳ statements]
58 if (check_token(sockfd, req))
59 ^
60 {
61 /home/mrsandman/CLionProjects/server_linux/main.c:231:34: warning:
    ↳ statement should be inside braces [google-readability-braces-around-
    ↳ statements]
62 if (check_token(sockfd, req))
63 ^
64 {
65 /home/mrsandman/CLionProjects/server_linux/main.c:235:38: warning:
    ↳ statement should be inside braces [google-readability-braces-around-
    ↳ statements]
66 if (check_arguments(sockfd, req))
67 ^
68 {
69 /home/mrsandman/CLionProjects/server_linux/main.c:243:36: warning:
    ↳ statement should be inside braces [google-readability-braces-around-
    ↳ statements]
70 if (handle_errors(sockfd, res))
71 ^
72 {
73 /home/mrsandman/CLionProjects/server_linux/main.c:253:34: warning:

```

```

    ↪ statement should be inside braces [google-readability-braces-around-
    ↪ statements]
74 if (check_token(sockfd, req))
75 ^
76 {
77 /home/mrsandman/CLionProjects/server_linux/main.c:268:34: warning:
    ↪ statement should be inside braces [google-readability-braces-around-
    ↪ statements]
78 if (check_token(sockfd, req))
79 ^
80 {
81 /home/mrsandman/CLionProjects/server_linux/main.c:273:43: warning:
    ↪ incompatible pointer to integer conversion passing 'void*' to
    ↪ parameter of type 'int' [clang-diagnostic-int-conversion]
82 int res = send(sockfd, buff, BUFSIZE, NULL);
83 ^
84 /usr/lib/llvm-6.0/bin/../lib/clang/6.0.0/include/stddef.h:105:16: note:
    ↪ expanded from macro 'NULL'
85 # define NULL ((void*)0)
86 ^
87 /usr/include/x86_64-linux-gnu/sys/socket.h:138:67: note: passing argument
    ↪ to parameter '__flags' here
88 extern ssize_t send (int __fd, const void *__buf, size_t __n, int __flags
    ↪ );
89 ^
90 /home/mrsandman/CLionProjects/server_linux/main.c:284:40: warning:
    ↪ incompatible pointer to integer conversion passing 'void*' to
    ↪ parameter of type 'int' [clang-diagnostic-int-conversion]
91 res = send(sockfd, calc_range, 20, NULL);
92 ^
93 /usr/lib/llvm-6.0/bin/../lib/clang/6.0.0/include/stddef.h:105:16: note:
    ↪ expanded from macro 'NULL'
94 # define NULL ((void*)0)
95 ^
96 /usr/include/x86_64-linux-gnu/sys/socket.h:138:67: note: passing argument
    ↪ to parameter '__flags' here
97 extern ssize_t send (int __fd, const void *__buf, size_t __n, int __flags
    ↪ );
98 ^
99 /home/mrsandman/CLionProjects/server_linux/main.c:294:35: warning:
    ↪ incompatible pointer to integer conversion passing 'void*' to
    ↪ parameter of type 'int' [clang-diagnostic-int-conversion]
100 res = send(sockfd, range, 20, NULL);
101 ^
102 /usr/lib/llvm-6.0/bin/../lib/clang/6.0.0/include/stddef.h:105:16: note:
    ↪ expanded from macro 'NULL'
103 # define NULL ((void*)0)
104 ^
105 /usr/include/x86_64-linux-gnu/sys/socket.h:138:67: note: passing argument
    ↪ to parameter '__flags' here
106 extern ssize_t send (int __fd, const void *__buf, size_t __n, int __flags
    ↪ );
107 ^
108 /home/mrsandman/CLionProjects/server_linux/main.c:319:27: warning:
    ↪ statement should be inside braces [hicpp-braces-around-statements]
109 if (prime == NULL) break;
110 ^
111 {
112 /home/mrsandman/CLionProjects/server_linux/main.c:335:34: warning:
    ↪ statement should be inside braces [google-readability-braces-around-

```

```

    ↪ statements]
113 if (check_token(sockfd , req))
114 ^
115 {
116 /home/mrsandman/CLionProjects/server_linux/main.c:347:34: warning:
    ↪ statement should be inside braces [google-readability-braces-around-
    ↪ statements]
117 if (check_token(sockfd , req))
118 ^
119 {
120 /home/mrsandman/CLionProjects/server_linux/main.c:352:36: warning:
    ↪ statement should be inside braces [google-readability-braces-around-
    ↪ statements]
121 if (handle_errors(sockfd , res))

```

Как видно из анализа, большая часть ошибок являются стилистическими и связаны с тем, что при использовании условного оператора даже с одной операцией стоит ограничивать его фигурными скобками. Также присутствуют ошибки о приведении типов, исправление которых не всегда является хорошей идеей, так как это может привести к некорректной работе выполнения функции. Особое место занимает предупреждение о том, чтобы вместо вызова `assert` использовать `assert4()` за счёт модификатора `SOCK_CLOEXEC`, который устанавливает 'close-on-exec' флаг для файлового дескриптора для многопоточных программ.

Проверка клиента с помощью clang-tidy:

```

1 /home/mrsandman/CLionProjects/client_linux/main.c:40:13: warning: Call to
    ↪ function 'strcpy' is insecure as it does not provide bounding of the
    ↪ memory buffer. Replace unbounded copy functions with analogous
    ↪ functions that support length arguments such as 'strncpy'. CWE-119 [
    ↪ clang-analyzer-security.insecureAPI strcpy]
2 strcpy(login , "new-user");
3 ^
4 /home/mrsandman/CLionProjects/client_linux/main.c:40:13: note: Call to
    ↪ function 'strcpy' is insecure as it does not provide bounding of the
    ↪ memory buffer. Replace unbounded copy functions with analogous
    ↪ functions that support length arguments such as 'strncpy'. CWE-119
5 /home/mrsandman/CLionProjects/client_linux/main.c:86:13: warning: Call to
    ↪ function 'strcat' is insecure as it does not provide bounding of the
    ↪ memory buffer. Replace unbounded copy functions with analogous
    ↪ functions that support length arguments such as 'strlcat'. CWE-119 [
    ↪ clang-analyzer-security.insecureAPI strcpy]
6 strcat(mes_buf , "\n");
7 ^
8 /home/mrsandman/CLionProjects/client_linux/main.c:86:13: note: Call to
    ↪ function 'strcat' is insecure as it does not provide bounding of the
    ↪ memory buffer. Replace unbounded copy functions with analogous
    ↪ functions that support length arguments such as 'strlcat'. CWE-119
9 /home/mrsandman/CLionProjects/client_linux/main.c:87:20: warning: format
    ↪ string is not a string literal (potentially insecure) [clang-
    ↪ diagnostic-format-security]
10 printf(mes_buf);
11 ~~~~~~
12 "%s",
13 /home/mrsandman/CLionProjects/client_linux/main.c:87:20: note: treat the
    ↪ string as an argument to avoid this
14 /home/mrsandman/CLionProjects/client_linux/main.c:116:49: warning:
    ↪ incompatible pointer to integer conversion passing 'void*' to

```

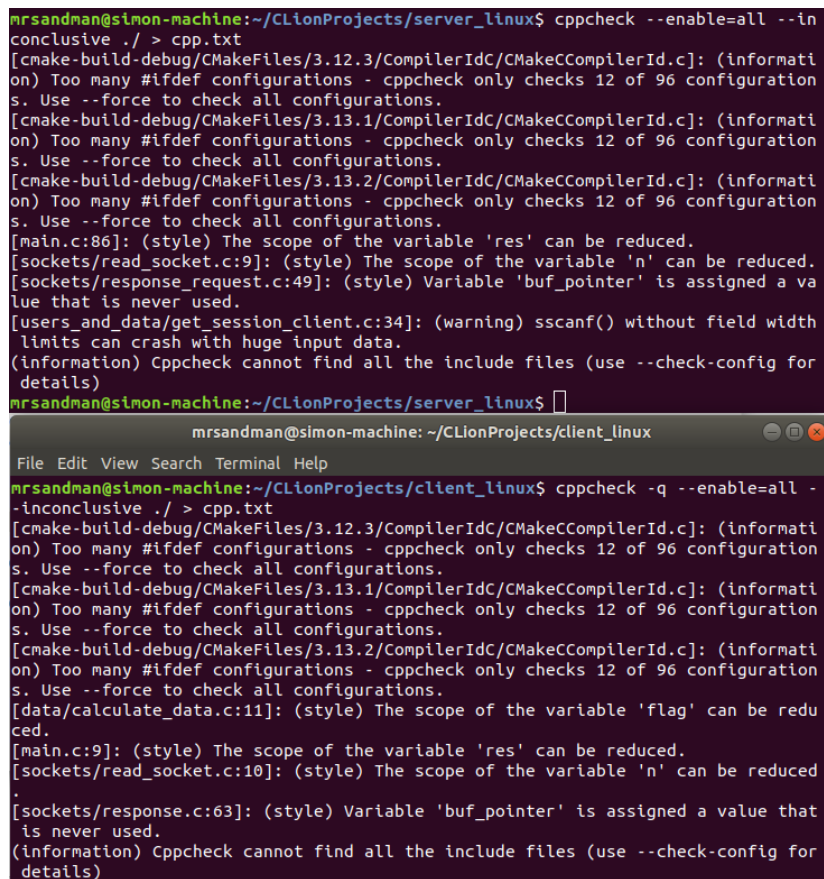
```

    ↪ parameter of type 'int' [clang-diagnostic-int-conversion]
15 res = send(sockfd, send_data, 5000, NULL);
16 ^
17 /usr/lib/llvm-6.0/bin/./lib/clang/6.0.0/include/stddef.h:105:16: note:
    ↪ expanded from macro 'NULL'
18 # define NULL ((void*)0)
19 ^
20 /usr/include/x86_64-linux-gnu/sys/socket.h:138:67: note: passing argument
    ↪ to parameter '__flags' here
21 extern ssize_t send (int __fd, const void *__buf, size_t __n, int __flags);
22 ^
23 /home/mrsandman/CLionProjects/client_linux/main.c:133:13: warning: Call to
    ↪ function 'strcpy' is insecure as it does not provide bounding of the
    ↪ memory buffer. Replace unbounded copy functions with analogous
    ↪ functions that support length arguments such as 'strncpy'. CWE-119 [
    ↪ clang-analyzer-security.insecureAPI.strcpy]
24 strcpy(token, resp.payload);
25 ^
26 /home/mrsandman/CLionProjects/client_linux/main.c:133:13: note: Call to
    ↪ function 'strcpy' is insecure as it does not provide bounding of the
    ↪ memory buffer. Replace unbounded copy functions with analogous
    ↪ functions that support length arguments such as 'strncpy'. CWE-119
27 /home/mrsandman/CLionProjects/client_linux/main.c:134:13: warning: Call to
    ↪ function 'strcpy' is insecure as it does not provide bounding of the
    ↪ memory buffer. Replace unbounded copy functions with analogous
    ↪ functions that support length arguments such as 'strncpy'. CWE-119 [
    ↪ clang-analyzer-security.insecureAPI.strcpy]
28 strcpy(login, req.comm.arg1);
29 ^
30 /home/mrsandman/CLionProjects/client_linux/main.c:134:13: note: Call to
    ↪ function 'strcpy' is insecure as it does not provide bounding of the
    ↪ memory buffer. Replace unbounded copy functions with analogous
    ↪ functions that support length arguments such as 'strncpy'. CWE-119
31 /home/mrsandman/CLionProjects/client_linux/main.c:141:13: warning: Call to
    ↪ function 'strcpy' is insecure as it does not provide bounding of the
    ↪ memory buffer. Replace unbounded copy functions with analogous
    ↪ functions that support length arguments such as 'strncpy'. CWE-119 [
    ↪ clang-analyzer-security.insecureAPI.strcpy]
32 strcpy(login, "not-logged");
33 ^
34 /home/mrsandman/CLionProjects/client_linux/main.c:141:13: note: Call to
    ↪ function 'strcpy' is insecure as it does not provide bounding of the
    ↪ memory buffer. Replace unbounded copy functions with analogous
    ↪ functions that support length arguments such as 'strncpy'. CWE-119

```

В случае клиента большинство ошибок связана с применением небезопасные функций. Опасность этих функций связана с тем, что необходимо определять длину буфера, который указывается в аргументах, а также убеждаться, что он заканчивается нулевым символом. Наиболее распространёнными 'опасными' функциями считаются strcpy и memset, но также сюда относят strcat, разновидности print и другие.

Теперь проведём анализ с помощью cppcheck.



```
mrsandman@simon-machine:~/CLionProjects/server_linux$ cppcheck --enable=all --inconclusive ./ > cpp.txt
[cmake-build-debug/CMakeFiles/3.12.3/CompilerIdC/CMakelCCompilerId.c]: (information) Too many #ifdef configurations - cppcheck only checks 12 of 96 configurations. Use --force to check all configurations.
[cmake-build-debug/CMakeFiles/3.13.1/CompilerIdC/CMakelCCompilerId.c]: (information) Too many #ifdef configurations - cppcheck only checks 12 of 96 configurations. Use --force to check all configurations.
[cmake-build-debug/CMakeFiles/3.13.2/CompilerIdC/CMakelCCompilerId.c]: (information) Too many #ifdef configurations - cppcheck only checks 12 of 96 configurations. Use --force to check all configurations.
[main.c:86]: (style) The scope of the variable 'res' can be reduced.
[sockets/read_socket.c:9]: (style) The scope of the variable 'n' can be reduced.
[sockets/response_request.c:49]: (style) Variable 'buf_pointer' is assigned a value that is never used.
[users_and_data/get_session_client.c:34]: (warning) sscanf() without field width limits can crash with huge input data.
(information) Cppcheck cannot find all the include files (use --check-config for details)
mrsandman@simon-machine:~/CLionProjects/server_linux$

mrsandman@simon-machine:~/CLionProjects/client_linux$ cppcheck -q --enable=all --inconclusive ./ > cpp.txt
[cmake-build-debug/CMakeFiles/3.12.3/CompilerIdC/CMakelCCompilerId.c]: (information) Too many #ifdef configurations - cppcheck only checks 12 of 96 configurations. Use --force to check all configurations.
[cmake-build-debug/CMakeFiles/3.13.1/CompilerIdC/CMakelCCompilerId.c]: (information) Too many #ifdef configurations - cppcheck only checks 12 of 96 configurations. Use --force to check all configurations.
[cmake-build-debug/CMakeFiles/3.13.2/CompilerIdC/CMakelCCompilerId.c]: (information) Too many #ifdef configurations - cppcheck only checks 12 of 96 configurations. Use --force to check all configurations.
[data/calculate_data.c:11]: (style) The scope of the variable 'flag' can be reduced.
[main.c:9]: (style) The scope of the variable 'res' can be reduced.
[sockets/read_socket.c:10]: (style) The scope of the variable 'n' can be reduced.
[sockets/response.c:63]: (style) Variable 'buf_pointer' is assigned a value that is never used.
(information) Cppcheck cannot find all the include files (use --check-config for details)
```

Рисунок 3.1. Анализ клиента и сервера с помощью cррсcheck

По полученным данным видно, что анализатор обратил внимание на использование переменных на слишком широком диапазоне, а также на возможность падения программы при большом объёме входных данных из-за использования `sscanf` без задания длины поля. Решением проблемы падения программы можно решить применением безопасной функции вместо `sscanf`.

- Динамические анализаторы кода

Динамический анализ кода - это способ анализа программы непосредственно при ее выполнении. Процесс динамического анализа можно разбить на несколько этапов - подготовка исходных данных, проведение тестового запуска программы и сбор необходимых параметров, анализ полученных данных.

С помощью динамического тестирования могут быть получены следующие метрики:

- Используемые ресурсы - время выполнения программы в целом или ее отдельных модулей, количество внешних запросов, количество используемой оперативной памяти и других ресурсов
- Цикломатическая сложность, степень покрытия кода тестами и другие метрики программы
- Программные ошибки - деление на ноль, разыменованное нулевого указателя, утечки памяти, "состояние гонки"
- Наличие в программе уязвимостей.

Достоинства динамического анализа кода:

- В большинстве реализаций появление ложных срабатываний исключено, так как обнаружение ошибки происходит в момент ее возникновения в программе; таким образом, обнаруженная ошибка является не предсказанием, сделанным на основе анализа модели программы, а констатацией факта ее возникновения
- Зачастую не требуется исходный код; это позволяет протестировать программы с закрытым кодом

В качестве динамического анализатора была выбрана valgrind. Проверка была проведена для варианта индивидуального задания для протокола ТСР.

```
mrsandman@simon-machine:~/CLionProjects/server_linux/cmake-build-debug$ valgrind
--leak-check=full --leak-resolution=high --undef-value-errors=yes ./server_linu
x
==10989== Memcheck, a memory error detector
==10989== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==10989== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==10989== Command: ./server_linux
==10989==
^C==10989==
==10989== Process terminating with default action of signal 2 (SIGINT)
==10989==   at 0x4E4D6A1: accept (accept.c:26)
==10989==   by 0x109238: main (main.c:58)
==10989==
==10989== HEAP SUMMARY:
==10989==   in use at exit: 0 bytes in 0 blocks
==10989==   total heap usage: 4 allocs, 4 frees, 74 bytes allocated
==10989==
==10989== All heap blocks were freed -- no leaks are possible
==10989==
==10989== For counts of detected and suppressed errors, rerun with: -v
==10989== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Рисунок 3.2. Анализ сервера с помощью valgrind

```
mrsandman@simon-machine:~/CLionProjects/client_linux/cmake-build-debug$ valgrind
--leak-check=full --leak-resolution=high --undef-value-errors=yes ./client_linu
x
==11031== Memcheck, a memory error detector
==11031== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==11031== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==11031== Command: ./client_linux
==11031==
usage ./client_linux hostname port
==11031==
==11031== HEAP SUMMARY:
==11031==   in use at exit: 0 bytes in 0 blocks
==11031==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==11031==
==11031== All heap blocks were freed -- no leaks are possible
==11031==
==11031== For counts of detected and suppressed errors, rerun with: -v
==11031== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Рисунок 3.3. Анализ клиента с помощью valgrind

По результатам анализа можно утверждать, что в ходе работы работы утечки памяти не возникают и отсутствуют неправильные использования многопоточности. Следовательно, дальнейших исправлений и улучшений не требуется.

4. Выводы

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

TCP сервер должен был создавать отдельный сокет для каждого нового клиента, а также обмениваться сообщениями в отдельных потоках для каждого клиента. Также в TCP предусматривается возможность любого клиента по 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 десяти 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 1024 байт.

Также было выполнено дополнительное задание, подразумевающее запоминание состояния сервера при завершении работы. Для этого был создан текстовый файл, из которого при каждом запуске сервера выбираются все данные. Данные представлены в следующем формате: <текущий диапазон>;<диапазон для расчёта>;<список вычисленных простых чисел>. После каждого выполнения запроса CALC в файл записываются обновлённые данные.

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 добавления контроля принятых пакетов на UDP приложении пришлось для каждого запроса и ответа на запрос запоминать его номер. В ходе обработки запроса ответу присваивается номер, который при передаче клиенту сравнивается с номером запроса. По несовпадающим номерам можно утверждать, что произошла потеря или перемешивание пакетов.

В ходе разработки приложения я столкнулся с проблемой постоянной отладки клиента и сервера, так как в языке C несовпадающие размеры буферов в связке recv/send могут привести к ошибке сегментации. Также возникла проблема с сохранением вычисленных данных в файл из-за очистки неопределённой части памяти. В связи с этим вместо работы с переменными типа FILE были использованы стандартные для ОС UNIX функции работы с файлами.