

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

Технологии компьютерных сетей

Отчет по лабораторным работам
Протоколы TCP и UDP

Работу
выполнила:
Шевченко А.С.
Группа: 43501/4
Преподаватель:
Алексюк А.О.

Санкт-Петербург
2018

Содержание

1. Цель работы	3
2. Программа работы	3
3. Теоретическая часть	3
3.1. Создание сокета	3
3.2. Установка соединения	4
3.3. Передача и прием данных	5
3.3.1. Протокол TCP	5
3.3.2. Протокол UDP	5
3.4. Привязывание сокета	6
3.5. Перевод TCP-сокета в состояние прослушивания	6
3.6. Приём входящего TCP-соединения	6
3.7. Завершение TCP-соединения	6
3.8. Закрытие сокета	7
3.9. Структура клиент-серверного приложения	7
3.9.1. Протокол TCP	7
3.9.2. Протокол UDP	8
4. Ход работы	9
4.1. Задание 1: клиент-серверное приложение на основе TCP сокетов - Linux . .	9
4.1.1. Листинг TCP-сервера	9
4.1.2. Листинг TCP-клиента	13
4.2. Задание 2: клиент-серверное приложение на основе TCP сокетов - Windows .	15
4.2.1. Листинг TCP-сервера	15
4.2.2. Листинг TCP-клиента	18
4.3. Задание 3: клиент-серверное приложение на основе UDP сокетов - Linux . .	21
4.3.1. Листинг UDP-сервера	21
4.3.2. Листинг UDP-клиента	23
4.4. Задание 4: Bug Tracker	25
4.4.1. Техническое задание	25
4.4.2. Основные возможности	25
4.4.3. Описание протокола	26
4.4.4. Сообщения об ошибках	27
4.4.5. Листинг программы	28
4.4.6. Демонстрация работы	29
4.5. Задание 5: протокол HTTP	32
4.5.1. Техническое задание	32
4.5.2. Основные возможности	32
4.5.3. Описание протокола	33
4.5.4. Листинг программы	34
4.5.5. Демонстрация работы	34
5. Список источников	37
6. Выводы	38

1. Цель работы

Целью работы является создание сетевых приложений на основе TCP и UDP сокетов.

2. Программа работы

Цикл работ состоит из следующих задач:

- Разработать клиент-серверное приложение с использованием TCP сокетов под ОС Linux;
- Разработать клиент-серверное приложение с использованием TCP сокетов под ОС Windows;
- Разработать клиент-серверное приложение с использованием UDP сокетов под ОС Linux;
- Разработать клиент-серверное приложение согласно индивидуальному заданию на основе TCP сокетов;
- Разработать клиент-серверное приложение по выбору: либо модифицировать предыдущее задание, заменив TCP сокет на UDP, либо реализовать другой протокол по выбору.

3. Теоретическая часть

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

С точки зрения операционной системы BSD-сокет (Berkley Software Distribution) или просто сокет — это выделенные операционной системой набор ресурсов, для организации сетевого взаимодействия. К таким ресурсам относятся, например, буфера для приёма/посылки данных или очереди сообщений.

В операционной системе MS Windows имеется аналогичная библиотека сетевого взаимодействия WinSock, реализованная на основе библиотеки BSD-сокетов. В подавляющем большинстве случаев функции и типы библиотеки WinSock совпадают с функциями и типами BSD-сокетов.

3.1. Создание сокета

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

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

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

Параметр `domain` указывает на домен, в пространстве которого создаётся данный сокет. Домен `AF_UNIX` используется для межпроцессного взаимодействия, домен `AF_INET` — для передачи с использованием стека протоколов TCP/IP. Параметр `type` определяет тип создаваемого сокета. Этот параметр может принимать значения:

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

Параметр `protocol` – идентификатор используемого протокола. В большинстве случаев протокол однозначно определяется типом создаваемого сокета, и передаваемое значение этого параметра – 0. Если же это не так (например, в случае SOCK_RAW), то необходимо явно задавать идентификатор протокола. Для его получения имеются системные вызовы: `getprotobyname` и `getprotobynumber`, которые разбирают файл `/etc/protocols` и получают идентификатор сетевого протокола:

```
struct protoent* getprotobyname(const char *name);
struct protoent* getprotobynumber(int proto);
```

Эти функции заполняют структуру `protoent`, поле `p_proto` которой следует использовать в качестве параметра `protocol` вызова `socket`:

```
struct protoent
{
char* p_name; // имя протокола из файла protocols
char** p_aliases; // список псевдонимов
int p_proto; // идентификатор протокола
}
```

3.2. Установка соединения

Для установления TCP-соединения используется вызов `connect`:

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

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

- Параметр `s` – дескриптор созданного сокета;
- Параметр `serv_addr` – указатель на структуру, содержащую параметры удалённого узла;
- Параметр `addr_len` – размер в байтах структуры, на которую указывает параметр `serv_addr`;

При программировании сокетов из домена AF_INET вместо структуры `sockaddr` используется приводимая к ней структура `sockaddr_in`, находящаяся в подключаемом файле `/usr/include/linux/in.h`:

```
struct sockaddr_in
{
sa_family_t sin_family; // Коммуникационный домен
unsigned short int sin_port; // Номер порта
struct in_addr sin_addr; // IP-адрес
...
};
```

Успех выполнения функции `connect` означает корректное установление логического канала связи и возможность начала передачи и приёма данных по протоколу TCP. В случае использования вызова `connect` для протокола UDP установления соединения не происходит, а адрес и порт из структуры `serv_addr` используется как адрес по умолчанию для последующих вызовов `send` и `recv`.

Для более простого заполнения параметров структуры `sockaddr_in` используются системные вызовы `htons` и `inet_addr`, осуществляющие замену порядка следования байт в номере порта и перевод IP-адреса из строкового вида в числовой соответственно, например:

```
serv_addr.sin_port = htons(3128);
10serv_addr.sin_addr.s_addr = inet_addr("192.168.1.1");
```

3.3. Передача и прием данных

3.3.1. Протокол TCP

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

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

Параметр `s` – дескриптор сокета, параметр `msg` – указатель на буфер, содержащий данные (вызов `send`), или указатель на буфер, предназначенный для приёма данных (вызов `recv`).

Параметр `len` – длина буфера в байтах, параметр `flags` – опции отправки или приёма данных.

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

3.3.2. Протокол UDP

В случае установленного адреса по умолчанию для протокола UDP (вызов `connect`) функции для передачи и приёма данных по протоколу UDP можно использовать вызовы `send` и `recv`. Если адрес и порт по умолчанию для протокола UDP не установлен, то параметры удалённой стороны необходимо указывать или получать при каждом вызове операций записи или чтения. Для протокола UDP имеется два аналогичных вызова `sendto` и `recvfrom`:

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

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

3.4. Привязывание сокета

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

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

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

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

3.5. Перевод TCP-сокета в состояние прослушивания

Для перевода сокета в состояние прослушивания служит системный вызов `listen`:

```
int listen(int s, int backlog);
```

Параметр `s` – дескриптор сокета, параметр `backlog` – задаёт максимальную длину, до которой может расти очередь ожидающих соединений.

В случае успеха возвращаемое значение – 0. При ошибке возвращается -1.

3.6. Приём входящего TCP-соединения

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

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

Параметр `s` – дескриптор прослушивающего сокета, параметр `addr` – указатель на структуру, содержащую параметры сокета, инициирующего соединение, `addrlen` – размер структуры `addr` в байтах. Возвращаемое значение – дескриптор сокета, созданного для нового соединения. Большинство параметров нового сокета соответствуют параметрам слушающего сокета. Полученный сокет в дальнейшем может использоваться для передачи и приёма данных. В случае если входящих соединений нет, то функция `accept` ожидает поступления запроса на входящее соединение.

3.7. Завершение TCP-соединения

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

```
int shutdown(int s, int how);
```

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

- `SHUT_RD` – запрещён приём данных;

- SHUT_WR – запрещена передача данных;
- SHUT_RDWR – запрещены и приём и передача данных.

В библиотеке WinSock семантика вызова несколько отличается:

```
int shutdown(SOCKET s, int how);
```

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

- SD_RECEIVE – запрещён приём данных. В случае наличия данных в очереди соединение разрывается;
- SD_SEND – запрещена передача данных;
- SD_BOTH – запрещены и приём и передача данных.

3.8. Закрытие сокета

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

```
int close(int s);
```

Аналогичный вызов в библиотеке WinSock имеет название `closesocket`:

```
int closesocket(SOCKET s);
```

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

3.9. Структура клиент-серверного приложения

3.9.1. Протокол TCP

Клиент протокола TCP создаёт экземпляр сокета, необходимый для взаимодействия с сервером, организует соединение, осуществляет обмен данными, в соответствии с протоколом прикладного уровня.

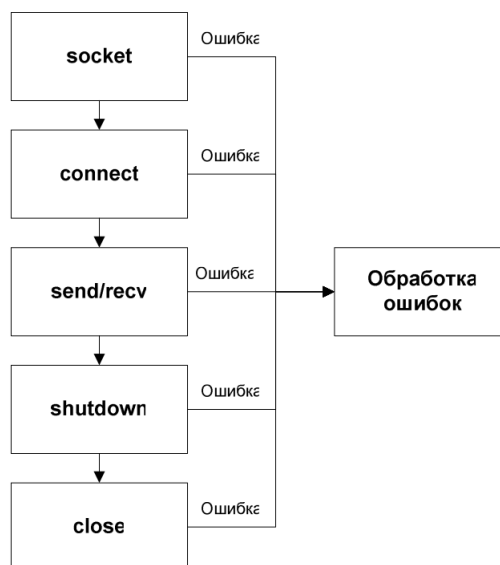


Рисунок 3.9.1. Структура TCP-клиента

Организация TCP-сервера отличается от TCP-клиента в первую очередь созданием слушающего сокета. Такой сокет находится в состоянии listen и предназначен только для приёма входящих соединений. В случае прихода запроса на соединение создаётся дополнительный сокет, который и занимается обменом данными с клиентом.

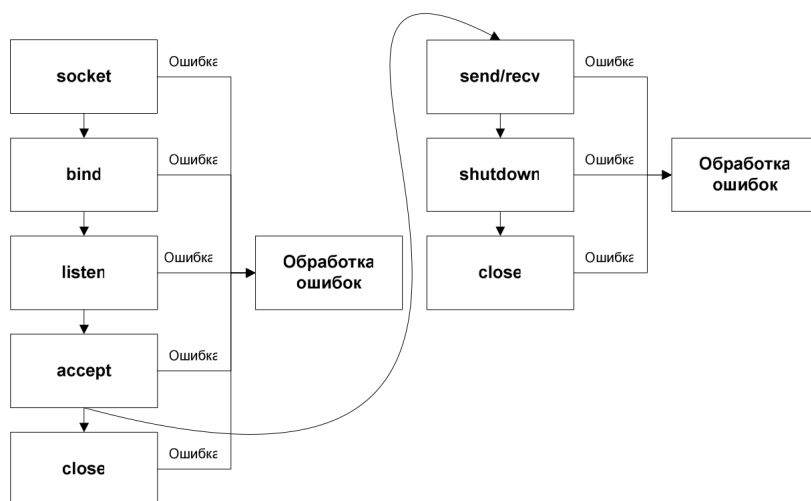


Рисунок 3.9.2. Структура TCP-сервера

3.9.2. Протокол UDP

Структура UDP-клиента ещё более простая, чем у TCP-клиента, так как нет необходимости создавать и разрывать соединение. Варианты организации UDP-клиента изображены ниже.

Наличие двух вариантов организации связано с возможностью в UDP-приложениях использовать вызов connect, устанавливающий значения по умолчанию для IP-адреса и порта сервера.

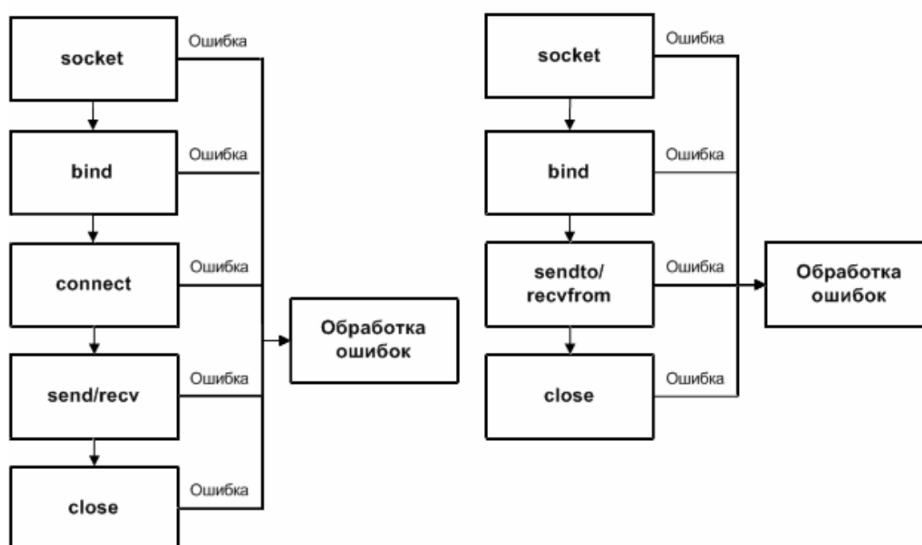


Рисунок 3.9.3. Структуры UDP-клиента

Ввиду того, что в протоколе UDP не устанавливается логический канал связи между клиентом и сервером, то для обмена данными между несколькими клиентами и сервером нет необходимости использовать со стороны сервера несколько сокетов. Для определения источника полученной дейтаграммы серверный сокет может использовать поля структуры `from` вызова `recvfrom`.

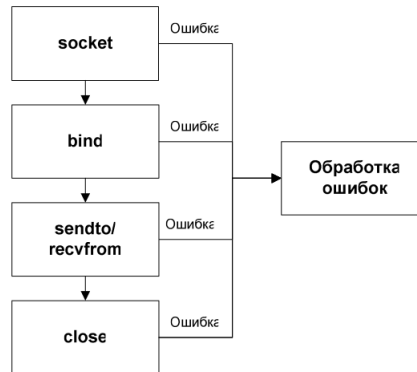


Рисунок 3.9.4. Структура UDP-сервера

4. Ход работы

В каждом клиент-серверном приложении, которое мы будем разрабатывать, обязательно должно быть соблюдено несколько принципов:

1. Приложение не должно завершаться после обслуживания одного клиента. Оно должно предоставлять возможность подключаться нескольким клиентам;
2. Приложение должно поддерживать многопоточность и обслуживать каждого из клиентов в отдельном потоке;
3. Приложение должно завершаться корректно, все потоки и сокеты должны быть закрыты;
4. Клиент и сервер должны гарантированно отправлять и принимать сообщения целиком, без потерь. В этом случае можно перед самим сообщением отправлять заголовок с его длиной либо использовать спецсимвол как признак конца сообщения.

4.1. Задание 1: клиент-серверное приложение на основе TCP сокетов - Linux

Примечание: задача многопоточности была решена путем создания для каждого клиента нового процесса системным вызовом функции `fork()`.

4.1.1. Листинг TCP-сервера

Листинг 1: TCP-сервер Linux

```

1 #include <stdio.h>
2 #include <stdlib.h>
3

```

```

4 #include <netdb.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7
8 #include <string.h>
9
10 void closeSocket (int socket);
11 void writeMessage(int newsock, char* buffer);
12 char* readMessage (int sock, int newsock);
13
14 int main(int argc, char *argv[]) {
15     int sockfd, newsockfd;
16     uint16_t portno;
17     unsigned int clilen;
18     struct sockaddr_in serv_addr, cli_addr;
19     char* buffer;
20
21     /* First call to socket() function */
22     sockfd = socket(AF_INET, SOCK_STREAM, 0);
23
24     if (sockfd < 0) {
25         perror("ERROR_opening_socket");
26         exit(1);
27     }
28
29     /* Initialize socket structure */
30     bzero((char *) &serv_addr, sizeof(serv_addr));
31     portno = 5001;
32
33     serv_addr.sin_family = AF_INET;
34     serv_addr.sin_addr.s_addr = INADDR_ANY;
35     serv_addr.sin_port = htons(portno);
36
37     if(setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &(int) {1}, sizeof(int)) <
↪ 0) {
38         perror("ERROR_on_setsockopt");
39         closeSocket(sockfd);
40
41         exit(1);
42     }
43
44     /* Now bind the host address using bind() call.*/
45     if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
46         perror("ERROR_on_binding");
47         closeSocket(sockfd);
48
49         exit(1);
50     }
51
52     /* Now start listening for the clients, here process will
53        * go in sleep mode and will wait for the incoming connection
54        */
55     listen(sockfd, 5);
56     clilen = sizeof(cli_addr);
57
58     if(fork() > 0){
59         while(getchar() != 'q'){
60             }
61         closeSocket(sockfd);
62     }

```

```

63
64     while (1) {
65
66         /* Accept actual connection from the client */
67         newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
68
69         if (newsockfd < 0) {
70             perror("ERROR_on_accept");
71             closeSocket(sockfd);
72
73             exit(1);
74         }
75
76         switch(fork()) {
77
78             case -1:
79                 perror("ERROR_on_fork");
80                 break;
81
82             case 0:
83                 close(sockfd);
84
85                 /* Get the message from client */
86                 buffer = readMessage(sockfd, newsockfd);
87                 printf("Here_is_the_message:_%s\n", buffer);
88                 free(buffer);
89
90                 /* Write a response to the client */
91                 writeMessage(newsockfd, "I_GOT_YOUR_MESSAGE");
92                 closeSocket(newsockfd);
93
94                 exit(0);
95
96             default:
97                 close(newsockfd);
98         }
99     }
100
101     closeSocket(newsockfd);
102     closeSocket(sockfd);
103
104     return 0;
105 }
106
107 void closeSocket (int sock) {
108
109     shutdown(sock, SHUT_RDWR);
110     close(sock);
111 }
112
113
114 char* readMessage (int sock, int newsock) {
115
116     char* buffer = (char*) calloc(256, sizeof(char));
117     char* bufForLen = (char*) calloc(4, sizeof(char));
118     uint32_t messlen;
119
120     /* Read length of message from the client */
121     ssize_t n = read(newsock, bufForLen, 4);
122

```

```

123     if (n < 0) {
124         perror("ERROR_lenght_of_message");
125         closeSocket(sock);
126         closeSocket(newsock);
127
128         exit(1);
129     }
130
131     messlen = atol(bufForLen);
132     free(bufForLen);
133
134     /* Read message from the client */
135     for(unsigned int i = 0; i < messlen; i += n) {
136
137         n = read(newsock, buffer + i, 255);
138
139         if (n < 0) {
140             perror("ERROR_of_message");
141             closeSocket(sock);
142             closeSocket(newsock);
143
144             exit(1);
145         }
146     }
147
148     return buffer;
149 }
150
151
152 void writeMessage(int newsock, char* buffer) {
153
154     uint32_t messlen;
155     char* bufForLen = (char*) calloc(4, sizeof(char));
156
157     /* Send length of message to the client */
158     messlen = strlen(buffer);
159     sprintf(bufForLen, "%04d", messlen);
160     int n = write(newsock, bufForLen, strlen(bufForLen));
161
162     if (n < 0) {
163         perror("ERROR_writing_to_socket_length_of_message");
164         closeSocket(newsock);
165
166         exit(1);
167     }
168     free(bufForLen);
169
170     /* Send message to the client */
171     n = write(newsock, buffer, messlen);
172
173     if (n < 0) {
174         perror("ERROR_writing_to_socket_message");
175         closeSocket(newsock);
176
177         exit(1);
178     }
179 }

```

4.1.2. Листинг TCP-клиента

Листинг 2: TCP-клиент Linux

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include <netdb.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7
8 #include <string.h>
9
10 void closeSocket (int socket);
11 void writeMessage (int sock, char* buffer);
12 char* readMessage(int sock);
13
14 int main(int argc, char *argv[]) {
15     int sockfd;
16     uint16_t portno;
17     struct sockaddr_in serv_addr;
18     struct hostent *server;
19
20     char* buffer = (char*)calloc(256, sizeof(char));
21
22
23     if (argc < 3) {
24         fprintf(stderr, "usage_%s_hostname_port\n", argv[0]);
25         exit(0);
26     }
27
28     portno = (uint16_t) atoi(argv[2]);
29
30     /* Create a socket point */
31     sockfd = socket(AF_INET, SOCK_STREAM, 0);
32
33     if (sockfd < 0) {
34         perror("ERROR_opening_socket");
35         exit(1);
36     }
37
38     server = gethostbyname(argv[1]);
39
40     if (server == NULL) {
41         fprintf(stderr, "ERROR_no_such_host\n");
42         closeSocket(sockfd);
43
44         exit(0);
45     }
46
47     bzero((char *) &serv_addr, sizeof(serv_addr));
48     serv_addr.sin_family = AF_INET;
49     bcopy(server->h_addr, (char *) &serv_addr.sin_addr.s_addr, (size_t) server->
↪ h_length);
50     serv_addr.sin_port = htons(portno);
51
52     /* Now connect to the server */
53     if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
↪ {
54         perror("ERROR_connecting");
55         closeSocket(sockfd);
```

```

56
57     exit(1);
58 }
59
60 /* Now ask for a message from the user, this message
61  * will be read by server*/
62 printf("Please_enter_the_message:_");
63 fgets(buffer, 255, stdin);
64 writeMessage(sockfd, buffer);
65 free(buffer);
66
67 /* Now read server response */
68 buffer = readMessage(sockfd);
69 free(buffer);
70
71 closeSocket(sockfd);
72
73 return 0;
74 }
75
76 void closeSocket (int sock) {
77
78     shutdown(sock, SHUT_RDWR);
79     close(sock);
80 }
81 }
82
83 void writeMessage (int sock, char* buffer) {
84
85     uint32_t messlen;
86     char* bufForLen = (char*) calloc(4, sizeof(char));
87
88     /* Send length of message to the server */
89     messlen = strlen(buffer);
90     sprintf(bufForLen, "%04d", messlen);
91     int n = write(sock, bufForLen, strlen(bufForLen));
92
93     if (n < 0) {
94         perror("ERROR_writing_to_socket_length_of_message");
95         closeSocket(sock);
96
97         exit(1);
98     }
99     free(bufForLen);
100
101     /* Send message to the server */
102     n = write(sock, buffer, messlen);
103
104     if (n < 0) {
105         perror("ERROR_writing_to_socket_message");
106         closeSocket(sock);
107
108         exit(1);
109     }
110 }
111
112 char* readMessage(int sock) {
113
114     char* buffer = (char*) calloc(256, sizeof(char));
115     char* bufForLen = (char*) calloc(4, sizeof(char));

```

```

116     uint32_t messlen;
117
118     /*Read lenght of message from the server*/
119     ssize_t n = read(sock, bufForLen, 4);
120
121     if (n < 0) {
122         perror("ERROR_lenght_of_message");
123         closeSocket(sock);
124
125         exit(1);
126     }
127
128     messlen = atol(bufForLen);
129     free(bufForLen);
130
131     /*Read message from the server*/
132     for(unsigned int i = 0; i < messlen; i += n) {
133
134         n = read(sock, buffer + i, 255);
135
136         if (n < 0) {
137             perror("ERROR_of_message");
138             closeSocket(sock);
139
140             exit(1);
141         }
142     }
143
144     printf("%s\n", buffer);
145
146     return buffer;
147 }

```

4.2. Задание 2: клиент-серверное приложение на основе TCP сокетов - Windows

Примечание: многопоточность обеспечивается за счет создания нового потока функцией CreateThread().

4.2.1. Листинг TCP-сервера

Листинг 3: TCP-сервер Windows

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <winsock2.h>
4 #include <unistd.h>
5 #include <string.h>
6 #include <stdint.h>
7
8 #define SHUT_RDWR 2
9
10 volatile BOOL flag = TRUE;
11
12 DWORD WINAPI waitFunc();
13 DWORD WINAPI commFunc(LPVOID temp);
14 void closeSocket(SOCKET sock);
15 char* readMessage(SOCKET sock, SOCKET newsock);

```

```

16 void writeMessage (SOCKET newsock, char* buffer);
17
18 int main(int argc, char *argv[]) {
19     /* Initialize library wsock32.dll */
20     WSADATA WsaData;
21     int WsaError;
22
23     WsaError = WSASStartup(0x0101, &WsaData);
24
25     if (WsaError != 0) {
26         perror("ERROR_on_WSASStartup");
27         exit(1);
28     }
29
30     /* Create thread for waiting of request */
31     CreateThread(NULL, 0, waitFunc, NULL, 0, NULL);
32     while(getchar() != 'q') {
33     }
34
35     return 0;
36 }
37
38 /* Function to create socket and wait for requests */
39 DWORD WINAPI waitFunc() {
40     SOCKET sockfd;
41     uint64_t portno;
42     struct sockaddr_in serv_addr;
43     BOOL temp;
44
45
46     sockfd = socket(AF_INET, SOCK_STREAM, 0);
47
48     if (sockfd == INVALID_SOCKET) {
49         perror("ERROR_opening_socket");
50         exit(1);
51     }
52
53     /* Initialize socket structure */
54     memset((char *) &serv_addr, 0, sizeof(serv_addr));
55     portno = 5001;
56
57     serv_addr.sin_family = AF_INET;
58     serv_addr.sin_addr.s_addr = INADDR_ANY;
59     serv_addr.sin_port = htons(portno);
60
61     temp = TRUE;
62
63     if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (char *) &temp, sizeof(BOOL)
64     ↪ ) < 0) {
65         perror("ERROR_on_setsockopt");
66         closeSocket(sockfd);
67         exit(1);
68     }
69
70     /* Now bind the host address using bind() call. */
71     if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
72         perror("ERROR_on_binding");
73         closeSocket(sockfd);
74
75         exit(1);

```



```

75     }
76
77     /* Now start listening for the clients*/
78     listen(sockfd, 5);
79     while(1) {
80         if (flag == TRUE) {
81             flag = FALSE;
82             CreateThread(NULL, 0, commFunc, &sockfd, 0, NULL);
83         }
84     }
85 }
86
87 /*Function for handling request*/
88 DWORD WINAPI commFunc(LPVOID temp) {
89     int clilen;
90     struct sockaddr_in cli_addr;
91     int sockfd;
92
93     char* buffer = (char*)calloc(256, sizeof(char));
94
95     clilen = sizeof(cli_addr);
96     sockfd = *(int *) temp;
97
98     /* Accept actual connection from the client */
99     SOCKET newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
100    flag = TRUE;
101    if (newsockfd == INVALID_SOCKET) {
102        perror("ERROR_on_accept");
103        closeSocket(sockfd);
104
105        exit(1);
106    }
107    /*Get the message from client*/
108    buffer = readMessage(sockfd, newsockfd);
109    printf("Here_is_the_message:_%s\n", buffer);
110    free(buffer);
111
112    /*Write a response to the client*/
113    writeMessage(newsockfd, "I_GOT_YOUR_MESSAGE");
114
115    closesocket(newsockfd);
116    return 0;
117 }
118
119 void closeSocket(SOCKET sock) {
120     shutdown(sock, SHUT_RDWR);
121     closesocket(sock);
122 }
123
124 char* readMessage (SOCKET sock, SOCKET newsock) {
125     int n;
126     uint32_t messlen;
127     char* buffer = (char*)calloc(256, sizeof(char));
128     char* bufForLen = (char*)calloc(4, sizeof(char));
129
130     n = recv(newsock, bufForLen, 4, 0);
131
132     if (n < 0) {
133         perror("ERROR_lenght_of_message");
134         closeSocket(sock);

```

```

135         closeSocket(newsock);
136
137         exit(1);
138     }
139
140     messlen = atol(bufForLen);
141     free(bufForLen);
142
143     for(unsigned int i = 0; i < messlen; i += n) {
144         n = recv(newsock, buffer + i, 255, 0);
145         if (n < 0) {
146             perror("ERROR_of_message");
147             closeSocket(sock);
148             closeSocket(newsock);
149
150             exit(1);
151         }
152     }
153     return buffer;
154 }
155
156 void writeMessage (SOCKET newsock, char* buffer) {
157     int n;
158     uint32_t messlen;
159     char* bufForLen = (char*) calloc(4, sizeof(char));
160
161     messlen = strlen(buffer);
162     sprintf(bufForLen, "%04d", messlen);
163     n = send(newsock, bufForLen, strlen(bufForLen), 0);
164
165     if (n < 0) {
166         perror("ERROR_writing_to_socket_length_of_message");
167         closeSocket(newsock);
168
169         exit(1);
170     }
171     free(bufForLen);
172
173     n = send(newsock, buffer, messlen, 0);
174
175     if (n < 0) {
176         perror("ERROR_writing_to_socket");
177         closeSocket(newsock);
178
179         exit(1);
180     }
181 }

```

4.2.2. Листинг TCP-клиента

Листинг 4: TCP-клиент Windows

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <winsock2.h>
4 #include <windows.h>
5 #include <unistd.h>
6 #include <stdint.h>
7

```

```

8 #include <string.h>
9 #define SHUT_RDWR 2
10
11 void closeSocket(SOCKET sock);
12 void writeMessage(SOCKET sock, char* buffer);
13 char* readMessage(SOCKET sock);
14
15 int main(int argc, char *argv[]) {
16     SOCKET sockfd;
17     uint16_t portno;
18     struct sockaddr_in serv_addr;
19     struct hostent *server;
20
21     char* buffer = (char*)calloc(256, sizeof(char));
22
23     /* Initialize library wsock32.dll */
24     WSADATA WsaData;
25     int WsaError;
26     WsaError = WSASStartup(0x0101, &WsaData);
27
28     if (WsaError != 0) {
29         perror("ERROR_on_WSASStartup");
30         exit(1);
31     }
32
33     if (argc < 3) {
34         fprintf(stderr, "usage_%s_hostname_port\n", argv[0]);
35         exit(0);
36     }
37
38     portno = (int) atoi(argv[2]);
39
40     /* Create a socket point */
41     sockfd = socket(AF_INET, SOCK_STREAM, 0);
42
43     if (sockfd == INVALID_SOCKET) {
44         perror("ERROR_opening_socket");
45         exit(1);
46     }
47
48     server = gethostbyname(argv[1]);
49
50     if (server == NULL) {
51         fprintf(stderr, "ERROR, _no_such_host\n");
52         closeSocket(sockfd);
53
54         exit(0);
55     }
56
57     /* Initialize socket structure */
58     memset((char *)&serv_addr, 0, sizeof(serv_addr));
59     serv_addr.sin_family = AF_INET;
60     memmove((char *)&serv_addr.sin_addr.s_addr, server->h_addr, (size_t)server->
    ↪ h_length);
61     serv_addr.sin_port = htons(portno);
62
63     /* Now connect to the server */
64     if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) ==
    ↪ SOCKET_ERROR) {
65         perror("ERROR_connecting");

```

```

66         closeSocket(sockfd);
67
68         exit(1);
69     }
70
71     /* Now ask for a message from the user, this message will be read by server
72     → */
73     printf("Please_enter_the_message:_");
74     fgets(buffer, 255, stdin);
75
76     /* Send message to the server */
77     writeMessage(sockfd, buffer);
78     free(buffer);
79
80     /* Now read server response */
81     buffer = readMessage(sockfd);
82
83     closeSocket(sockfd);
84     free(buffer);
85
86     return 0;
87 }
88
89 void closeSocket(SOCKET sock) {
90     shutdown(sock, SHUT_RDWR);
91     closesocket(sock);
92 }
93
94 void writeMessage (SOCKET sock, char* buffer) {
95     uint32_t messlen;
96     char* bufForLen = (char*) calloc(4, sizeof(char));
97     int n;
98
99     /* Send length of message to the server */
100    messlen = strlen(buffer);
101    sprintf(bufForLen, "%04d", messlen);
102    n = send(sock, bufForLen, strlen(bufForLen), 0);
103
104    if (n < 0) {
105        perror("ERROR_writing_to_socket_length_of_message");
106        closeSocket(sock);
107
108        exit(1);
109    }
110    free(bufForLen);
111
112    /* Send message to the server */
113    n = send (sock, buffer, messlen, 0);
114
115    if (n < 0) {
116        perror("ERROR_writing_to_socket_message");
117        closeSocket(sock);
118
119        exit(1);
120    }
121 }
122
123 char* readMessage(SOCKET sock) {
124

```

```

125     char* buffer = (char*)calloc(256, sizeof(char));
126     char* bufForLen = (char*)calloc(4, sizeof(char));
127     uint32_t messlen;
128     int n;
129
130     /*Read lenght of message from the server*/
131     n = recv(sock, bufForLen, 4, 0);
132
133     if (n < 0) {
134         perror("ERROR_lenght_of_message");
135         closeSocket(sock);
136
137         exit(1);
138     }
139
140     messlen = atol(bufForLen);
141     free(bufForLen);
142
143     /*Read message from the server*/
144     for(unsigned int i = 0; i < messlen; i += n) {
145
146         n = recv(sock, buffer + i, 255, 0);
147
148         if (n < 0) {
149             perror("ERROR_of_message");
150             closeSocket(sock);
151
152             exit(1);
153         }
154     }
155
156     printf("%s\n", buffer);
157
158     return buffer;
159 }

```

4.3. Задание 3: клиент-серверное приложение на основе UDP сокетов - Linux

4.3.1. Листинг UDP-сервера

Листинг 5: UDP-сервер Linux

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #include <netdb.h>
5  #include <netinet/in.h>
6  #include <unistd.h>
7
8  #include <string.h>
9
10 void closeSocket (int sock);
11
12 int main(int argc, char *argv[]) {
13     int sockfd, n;
14     uint16_t portno;
15     unsigned int clilen;
16     struct sockaddr_in serv_addr, cli_addr;

```

```

17 char* servResp = "I_GOT_YOUR_MESSAGE";
18
19 char* buffer = (char*)calloc(256, sizeof(char));
20
21 /* First call to socket() function */
22 sockfd = socket(AF_INET, SOCK_DGRAM, 0);
23
24 if (sockfd < 0) {
25     perror("ERROR_opening_socket");
26     exit(1);
27 }
28
29 /* Initialize socket structure */
30 bzero((char *) &serv_addr, sizeof(serv_addr));
31 portno = 5001;
32 serv_addr.sin_family = AF_INET;
33 serv_addr.sin_addr.s_addr = INADDR_ANY;
34 serv_addr.sin_port = htons(portno);
35
36 if(setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &(int) {1}, sizeof(int)) <
↪ 0) {
37     perror("ERROR_on_setsockopt");
38     closeSocket(sockfd);
39
40     exit(1);
41 }
42
43 /* Now bind the host address using bind() call.*/
44 if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
45     perror("ERROR_on_binding");
46     closeSocket(sockfd);
47
48     exit(1);
49 }
50
51 /* Now start listening for the clients, here process will
52 * go in sleep mode and will wait for the incoming connection*/
53 clilen = sizeof(cli_addr);
54
55 if(fork() > 0){
56     while(getchar() != 'q'){
57     }
58     closeSocket(sockfd);
59 }
60
61 while(1) {
62
63     bzero(buffer, 256);
64     n = recvfrom(sockfd, buffer, 256, 0, (struct sockaddr *) &cli_addr, &
↪ clilen);
65
66     if (n < 0) {
67         perror("ERROR_reading_from_socket");
68         closeSocket(sockfd);
69
70         exit(1);
71     }
72
73     printf("Here_the_message:_%s\n", buffer);
74     bzero(buffer, 256);

```

```

75
76     n = sendto(sockfd, servResp, strlen(servResp), 0, (struct sockaddr *) &
↪ cli_addr, cli_len);
77
78     if (n < 0) {
79         perror("ERROR_sending_to_socket");
80         closeSocket(sockfd);
81
82         exit(1);
83     }
84 }
85
86 closeSocket(sockfd);
87 free(buffer);
88
89 return 0;
90 }
91
92 void closeSocket (int sock) {
93
94     shutdown(sock, SHUT_RDWR);
95     close(sock);
96
97 }

```

4.3.2. Листинг UDP-клиента

Листинг 6: UDP-клиент Linux

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include <netdb.h>
5 #include <netinet/in.h>
6 #include <unistd.h>
7 #include <arpa/inet.h>
8
9 #include <string.h>
10
11 void closeSocket (int sock);
12
13 int main(int argc, char *argv[]) {
14     int sockfd;
15     ssize_t n;
16     uint16_t portno;
17     unsigned int servlen;
18     struct sockaddr_in serv_addr;
19
20     char* buffer = (char*)calloc(256, sizeof(char));
21     bzero(buffer, 256);
22
23     if (argc < 3) {
24         fprintf(stderr, "usage_%s_hostname_port\n", argv[0]);
25         exit(0);
26     }
27
28     portno = (uint16_t) atoi(argv[2]);
29
30     /* Create a socket point */

```

```

31     sockfd = socket(AF_INET, SOCK_DGRAM, 0);
32
33     if (sockfd < 0) {
34         perror("ERROR_opening_socket");
35         exit(1);
36     }
37
38     bzero((char *) &serv_addr, sizeof(serv_addr));
39     serv_addr.sin_family = AF_INET;
40     serv_addr.sin_port = htons(portno);
41
42     /* Now ask for a message from the user, this message
43      * will be read by server*/
44     printf("Please_enter_the_message:_");
45     fgets(buffer, 255, stdin);
46
47     servlen = sizeof(serv_addr);
48
49     n = sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr *) &serv_addr
50 ↪ , servlen);
51
52     if (n < 0) {
53         perror("ERROR_writing_to_socket_message");
54         closeSocket(sockfd);
55
56         exit(1);
57     }
58
59     bzero(buffer, 256);
60
61     /* Now read server response */
62     n = recvfrom(sockfd, buffer, 256, 0, (struct sockaddr *) &serv_addr, &
63 ↪ servlen);
64
65     if (n < 0) {
66         perror("ERROR_reading_from_socket");
67         closeSocket(sockfd);
68
69         exit(1);
70     }
71
72     printf("%s\n", buffer);
73
74     closeSocket(sockfd);
75     free(buffer);
76
77     return 0;
78 }
79
80 void closeSocket (int sock) {
81     shutdown(sock, SHUT_RDWR);
82     close(sock);
83 }

```


4.4. Задание 4: Bug Tracker

4.4.1. Техническое задание

Разработать клиент-серверную систему регистрации, учета и управления ошибками в программных проектах. Система должна состоять из сервера, хранящего репозиторий ошибок и рабочих клиентских мест, позволяющих программистам и тестерам управлять ошибками, найденными в программных проектах.

4.4.2. Основные возможности

Для реализации поставленной выше задачи было принято решение использовать TCP протокол, который обеспечивает более надежную передачу данных между клиентом и сервером.

Серверное приложение должно реализует следующие функции:

1. Прослушивание определенного порта;
2. Обработка запросов на подключение по этому порту от клиентов;
3. Поддержка одновременной работы нескольких клиентов через механизм нитей;
4. Регистрация подключившегося клиента в качестве тестера или разработчика;
5. Выдача тестеру списка исправленных ошибок;
6. Выдача тестеру списка активных (неисправленных) ошибок;
7. Прием от тестера новой ошибки с указанием разработчика, проекта, идентификатора ошибки, текста ошибки;
8. Прием от тестера команды о подтверждении или отклонении исправления ошибки;
9. Выдача разработчику списка найденных в его проектах ошибок: идентификаторов и текстов;
10. Прием команды разработчика об исправлении ошибки;
11. Обработка запроса на отключение клиента;
12. Принудительное отключение клиента.

Клиентское приложение реализует следующие функции:

1. Установление соединения с сервером;
2. Посылка регистрационных данных клиента (как тестера или как разработчика);
3. Для тестера: получение списка активных ошибок;
4. Для тестера: получение списка исправленных ошибок;
5. Для тестера: добавление новой ошибки;
6. Для тестера: посылка команды подтверждения или отклонения исправления активной ошибки;

7. Для разработчика: получение списка активных ошибок;
8. Для разработчика: посылка команды исправления активной ошибки;
9. Разрыв соединения;
10. Обработка ситуации отключения клиента сервером.

4.4.3. Описание протокола

Согласно заданию, клиенты делятся на тестировщиков и разработчиков. Сервис предоставляет возможность авторизоваться в системе и осуществлять редактирование данных о том или ином репозитории. При этом считаем, что для того, чтобы зарегистрироваться, необходимо заранее поговорить с администратором приложения Bug Tracker, и тот запишет ваш логин и идентификатор должности (1 - тестировщик, 0 - разработчик) в специальный файл logins.

Для того, чтобы привести приложение в рабочее состояние, необходимо сперва запустить сервер, и после этого к нему уже могут подключаться клиенты.

Как только клиент подключается, ему от сервера приходит приглашение:

Enter your login

Важный момент взаимодействия клиента и сервера: когда сообщение отправляется, то сперва формируется посылка с его длиной, а затем посылается само сообщение. Это необходимо для обеспечения гарантированной доставки информации клиенту или серверу.

Далее клиент может ввести свой логин, при этом, чтобы авторизация прошла успешно, он должен быть заранее зарегистрирован в файле logins. В противном случае сервер попросит ввести логин еще раз, рассчитывая на то, что вы все таки авторизируетесь как существующий пользователь:

Please, try again

Enter your login

Важно: сервер сам говорит клиенту, какую операцию ему нужно совершить - чтение или запись. Если клиент принял от сервера код 1 - от него ожидаются отправка каких-то данных, если 0 - клиент читает то, что посылает ему сервер.

После того, как пользователь успешно авторизовался, сервер самостоятельно интерпретирует его как тестировщика или разработчика. Если подключился разработчик:

Your login is okay

You are a developer

Если подключился тестировщик:

Your login is okay

You are a tester

После того, как сервер определил, кем является пользователь, он отправляет ему меню с действиями, которые ему доступны. Для тестировщика это:

1. List of fixed bugs
2. List of active bugs
3. Report a new bug
4. Confirm or reject a bugfix
5. Exit

Для разработчика:

1. List of bugs found in your project
2. Mark bug as fixed
3. Exit

Далее в зависимости от того, какой пункт меню выбрал клиент, сервер предоставляет ему соответствующую возможность.

Взаимодействие клиента и сервера может продолжаться до тех пор, пока:

- Клиент не выберет пункт меню Exit;
- Сервер не отключит клиента по каким либо причинам;
- Сервер не завершит свою работу;
- Пул клиентов не переполнится и новые клиенты не начнут подключаться вместо старых.

Согласно заданию, сервер имеет возможность принудительно отключать клиентов.

Если нажать клавишу 'l', сервер выведет список всех подключенных на данный момент клиентов:

Now working on the server:

ID	login	IP	port
0	hello@me	16777343	1 55507

Далее можно использовать команду 'd <id клиента>' для того, чтобы отключить клиента с id равным <id клиента>.

Также работу сервера можно полностью завершить, нажав клавишу 'q'.

Необходимо отметить, что вся информация о репозиториях разработчиков тоже хранится в файле. Каждый раз, когда сервер запускается, он достает все данные из файла и помещает в структуру. После того, как сервер получает команду завершения, он все отредактированные данные из структуры выгружает обратно в файл.

4.4.4. Сообщения об ошибках

Существует множество случаев, в которых взаимодействие клиента и сервера может быть нарушено. В данном приложении обрабатываются следующие:

- Пользователь вводит неверный логин.

Ответ сервера:

```
Please, try again
Enter your login
```

Состояние приложения: продолжает работу.

- Пользователь выбирает несуществующий пункт меню.

Ответ сервера:

```
Error choosing menu
```

Состояние приложения: продолжает работу.

- Тетсировщик, меняя статус ошибки, вместо корректного идентификатора указывает буквы.

Ответ сервера:

Only digits are allowed! Please, try again

Состояние приложения: продолжает работу.

- Тетсировщик, меняя статус ошибки, вместо указывает несуществующий идентификатор ошибки.

Ответ сервера:

There is no such bug with given id! Please, try again

Состояние приложения: продолжает работу.

- Тетсировщик, меняя статус ошибки, указывает несуществующий номер статуса.

Ответ сервера:

Only 1 and 2 are allowed! Please, try again

Состояние приложения: продолжает работу.

- Разработчик, меняя статус ошибки на 'fixed', вместо корректного идентификатора указывает буквы.

Ответ сервера:

Only digits are allowed! Please, try again

Состояние приложения: продолжает работу.

- Разработчик, меняя статус ошибки на 'fixed', вместо указывает несуществующий идентификатор ошибки.

Ответ сервера:

There is no such bug with given id! Please, try again

Состояние приложения: продолжает работу.

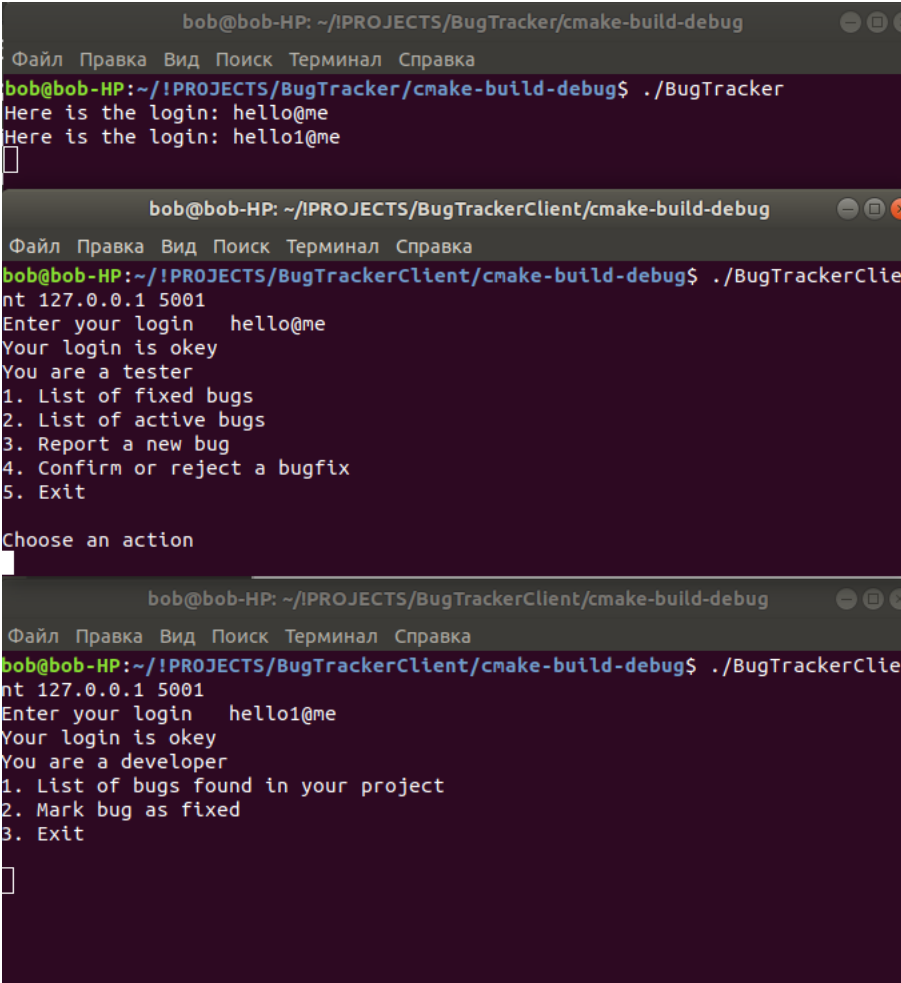
4.4.5. Листинг программы

Язык программирования: си.

Исходный код проекта находится здесь:

4.4.6. Демонстрация работы

Реализуем простой сценарий: к серверу подключается два клиента - тестировщик и разработчик.



The image displays three terminal windows stacked vertically, illustrating the connection of two clients to a server. The top window shows the server's output: 'Here is the login: hello@me' and 'Here is the login: hello1@me'. The middle window shows the 'BugTrackerClient' running as a tester, logging in with 'hello@me' and seeing a menu of actions. The bottom window shows the same client running as a developer, logging in with 'hello1@me' and seeing a different menu of actions.

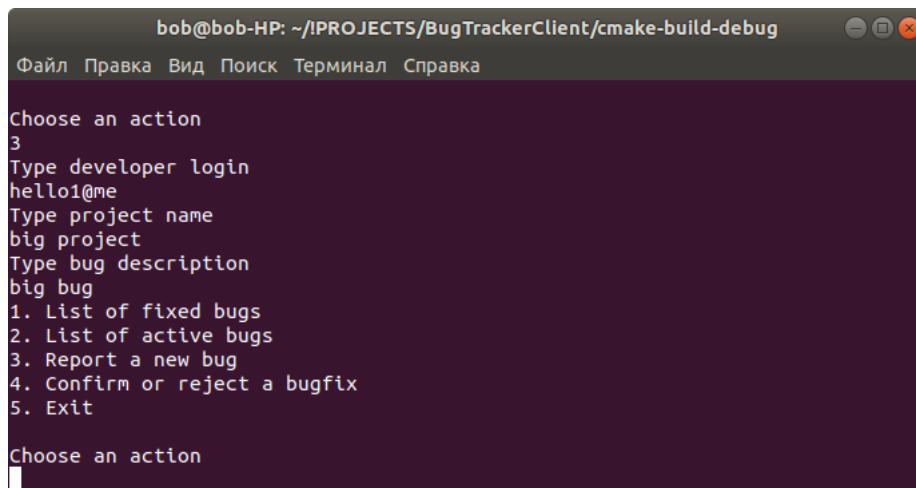
```
bob@bob-HP: ~/!PROJECTS/BugTracker/cmake-build-debug
Файл Правка Вид Поиск Терминал Справка
bob@bob-HP:~/!PROJECTS/BugTracker/cmake-build-debug$ ./BugTracker
Here is the login: hello@me
Here is the login: hello1@me
█

bob@bob-HP: ~/!PROJECTS/BugTrackerClient/cmake-build-debug
Файл Правка Вид Поиск Терминал Справка
bob@bob-HP:~/!PROJECTS/BugTrackerClient/cmake-build-debug$ ./BugTrackerClient
nt 127.0.0.1 5001
Enter your login  hello@me
Your login is okay
You are a tester
1. List of fixed bugs
2. List of active bugs
3. Report a new bug
4. Confirm or reject a bugfix
5. Exit
Choose an action

bob@bob-HP: ~/!PROJECTS/BugTrackerClient/cmake-build-debug
Файл Правка Вид Поиск Терминал Справка
bob@bob-HP:~/!PROJECTS/BugTrackerClient/cmake-build-debug$ ./BugTrackerClient
nt 127.0.0.1 5001
Enter your login  hello1@me
Your login is okay
You are a developer
1. List of bugs found in your project
2. Mark bug as fixed
3. Exit
█
```

Рисунок 4.4.1. Подключение клиентов к серверу

Далее тестировщик решает сообщить, что разработчик под логином hello1@me допустил баг в некотором проекте:

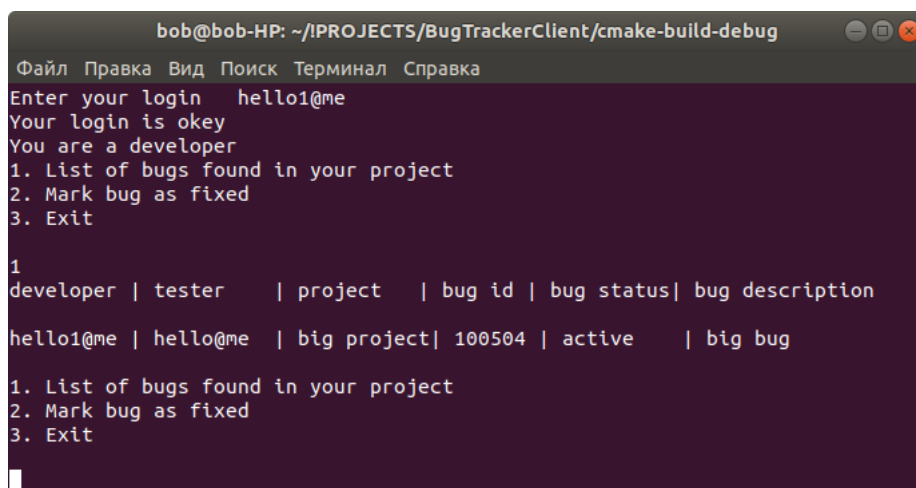


```
bob@bob-HP: ~/IPROJECTS/BugTrackerClient/cmake-build-debug
Файл  Правка  Вид  Поиск  Терминал  Справка

Choose an action
3
Type developer login
hello1@me
Type project name
big project
Type bug description
big bug
1. List of fixed bugs
2. List of active bugs
3. Report a new bug
4. Confirm or reject a bugfix
5. Exit
Choose an action
```

Рисунок 4.4.2. Тестировщик сообщает об ошибке

Теперь разработчик решил посмотреть список обнаруженных в его проектах ошибок:



```
bob@bob-HP: ~/IPROJECTS/BugTrackerClient/cmake-build-debug
Файл  Правка  Вид  Поиск  Терминал  Справка

Enter your login  hello1@me
Your login is okay
You are a developer
1. List of bugs found in your project
2. Mark bug as fixed
3. Exit

1
developer | tester   | project   | bug id | bug status| bug description
hello1@me | hello@me  | big project| 100504 | active   | big bug

1. List of bugs found in your project
2. Mark bug as fixed
3. Exit
```

Рисунок 4.4.3. Разработчик смотрит список активных багов

В списках разработчика hello1@me появилась только что добавленная ошибка. Далее программист ее исправляет и меняет ее статус:

```
bob@bob-HP: ~/PROJECTS/BugTrackerClient/cmake-build-debug
Файл Правка Вид Поиск Терминал Справка
developer | tester | project | bug id | bug status| bug description
hello1@me | hello@me | big project| 100504 | active | big bug

1. List of bugs found in your project
2. Mark bug as fixed
3. Exit

2
Enter bug id
100504
Operation succeed
1. List of bugs found in your project
2. Mark bug as fixed
3. Exit
```

Рисунок 4.4.4. Разработчик помечает баг как исправленный

Теперь тестировщик хочет узнать, занялся ли решением обозначенной им проблемы разработчик:

```
bob@bob-HP: ~/PROJECTS/BugTrackerClient/cmake-build-debug
Файл Правка Вид Поиск Терминал Справка

Choose an action
1
developer | tester | project | bug id | bug status| bug description
hello1@me | hello@me | big project| 100504 | fixed | big bug

1. List of fixed bugs
2. List of active bugs
3. Report a new bug
4. Confirm or reject a bugfix
5. Exit

Choose an action
```

Рисунок 4.4.5. Тестировщик получает список исправленных багов

Теперь проблема решена, тестировщик удовлетворен и решает закрыть ошибку:

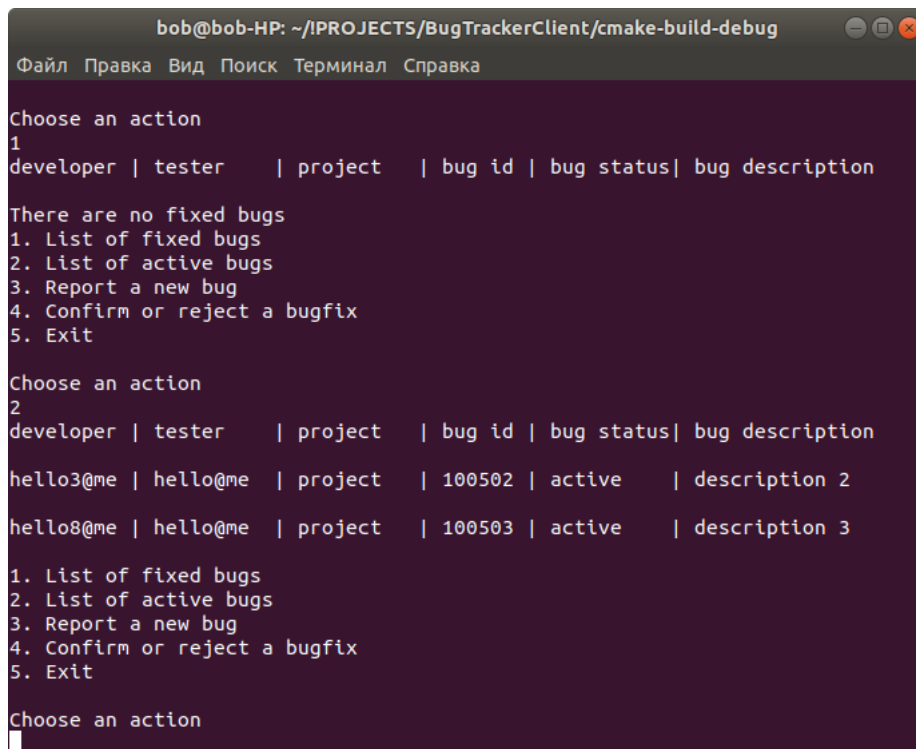
```
bob@bob-HP: ~/PROJECTS/BugTrackerClient/cmake-build-debug
Файл Правка Вид Поиск Терминал Справка

Choose an action
4
Enter bug id
100504
Enter action. 1 - close, 2 - reopen
1
Operation succeed
1. List of fixed bugs
2. List of active bugs
3. Report a new bug
4. Confirm or reject a bugfix
5. Exit

Choose an action
```

Рисунок 4.4.6. Тестировщик закрывает исправленный баг

Теперь в списке исправленных ошибок, ошибка разработчика hello1@me не отображается, ровно как и в списке активных:



```
bob@bob-HP: ~/PROJECTS/BugTrackerClient/cmake-build-debug
Файл  Правка  Вид  Поиск  Терминал  Справка

Choose an action
1
developer | tester   | project | bug id | bug status| bug description

There are no fixed bugs
1. List of fixed bugs
2. List of active bugs
3. Report a new bug
4. Confirm or reject a bugfix
5. Exit

Choose an action
2
developer | tester   | project | bug id | bug status| bug description

hello3@me | hello@me | project | 100502 | active   | description 2
hello8@me | hello@me | project | 100503 | active   | description 3

1. List of fixed bugs
2. List of active bugs
3. Report a new bug
4. Confirm or reject a bugfix
5. Exit

Choose an action
```

Рисунок 4.4.7. Тестировщик убеждается, что ошибка закрыта

4.5. Задание 5: протокол HTTP

4.5.1. Техническое задание

Разработать приложение для операционных систем семейства Windows или Linux, обеспечивающее базовые функции сервера протокола HTTP (Web-сервера).

4.5.2. Основные возможности

Приложение реализует следующие функции:

1. Обработка подключения клиента;
2. Разбор строки URL;
3. Выдача клиенту запрошенного ресурса;
4. Обеспечение параллельной загрузки клиенту страниц и медиаэлементов;
5. Обеспечение параллельной работы нескольких клиентов;
6. Отображение параметров, передаваемых вместе с методом POST;
7. Формирование необходимых заголовков протокола HTTP;
8. Протоколирование соединения клиента с сервером.

Реализуемые методы:

- GET – для передачи Web-страниц и медиа-элементов;
- HEAD – для передачи заголовков Web-страниц и медиа-элементов;
- POST – для получения от клиента параметров Web-форм.

Клиентом является любой браузер.

4.5.3. Описание протокола

HTTP - широко распространенный протокол обмена данными, изначально предназначенный для передачи гипертекстовых документов.

Предполагает использование клиент-серверной структуры передачи данных. Клиентское приложение формирует запрос и отправляет его на сервер, после чего серверное программное обеспечение обрабатывает запрос и отправляет ответ.

Для того, чтобы сформировать HTTP-запрос, необходимо составить стартовую строку, а также задать по крайней мере один заголовок - это заголовок Host, который является обязательным:

Метод URI HTTP/Версия

Пример:

GET / HTTP/1.1

- Метод - определяет операцию, которую необходимо осуществить с выбранным ресурсом;
- URI - идентификатор ресурса - путь до конкретного ресурса, над которым необходимо осуществить операцию;
- Версия - версия стандарта HTTP.

Итак, для того, чтобы обратиться к web-странице по ее адресу, необходимо составить запрос:

GET / HTTP/1.1
Host: resource.com

Сервер отправит ответ в следующем формате:

HTTP/Версия Код_состояния Пояснение

- Версия - версия стандарта HTTP.
- Код состояния - три цифры, которые определяют результат совершения запроса;
- Пояснение - текстовое пояснение к коду ответа.

После стартовой строки следуют заголовки, а затем тело ответа, например:

```
HTTP/1.1 200 OK
Server: nginx/1.2.1
Date: Sat, 08 Mar 2018 12:33:23 GMT
Content-type: application/octet-stream
Content-Length: 5
Last-Modified: Sat, 08 Mar 2018 12:30:23 GMT
Connection: keep-alive
Accept-Ranges: bytes
```

body

4.5.4. Листинг программы

Язык программирования: Java.

Исходный код проекта находится здесь:

4.5.5. Демонстрация работы

На сервере в данный момент есть четыре HTML страницы:

- index.html
- bird.html
- HTTP.html
- formExamble.html

Для того, чтобы получить один из ресурсов, пользователь должен точно знать его имя вместе с расширением html. Если пользователь не указал имя, то по умолчанию сервер воспримет это как запрос страницы index.html:

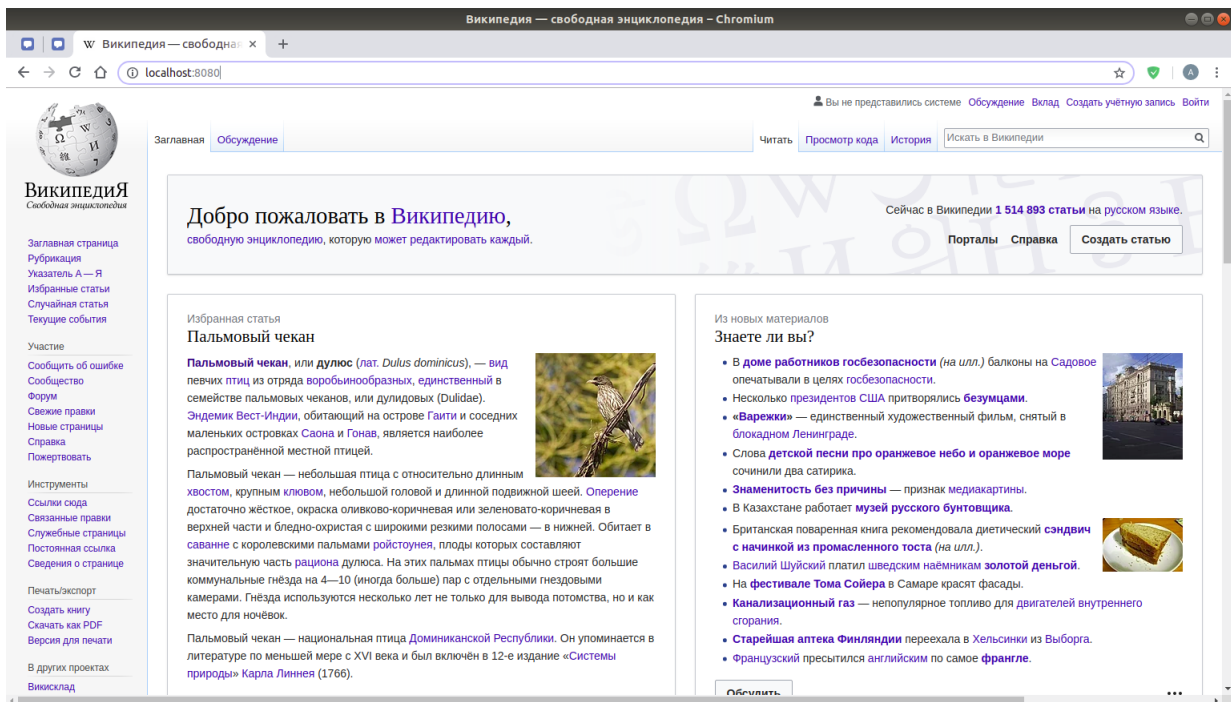


Рисунок 4.5.1. Пользователь получает страничку index.html

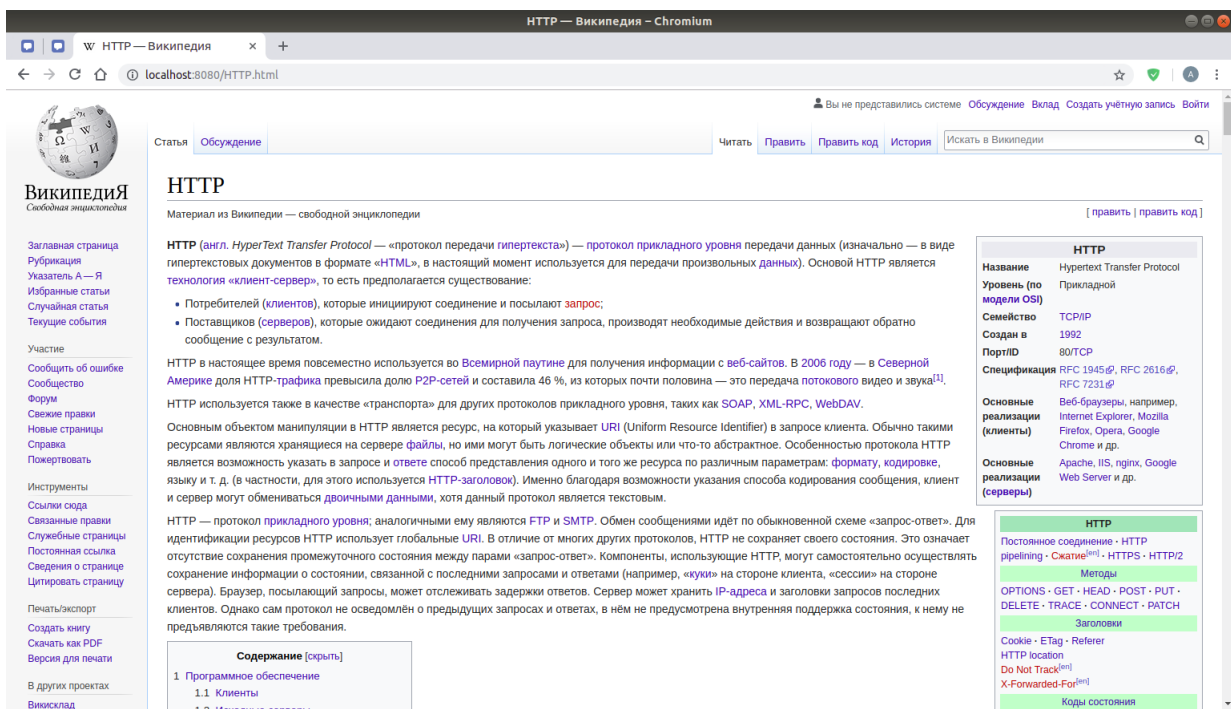


Рисунок 4.5.2. Пользователь получает страничку HTTP.html

Запросы клиента в проекте логируются, но также их можно посмотреть и в самом браузере:

```

GET
URI:
/bird.html
Parameters:
null
Headers:
Host    localhost8080
Connection    keep-alive
Pragma    no-cache
Cache-Control    no-cache
Upgrade-Insecure-Requests    1
User-Agent    Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Ubuntu Chromium/71.0.3578.80 Chrome/71.0.3578.80 Safari/537.36
Accept    text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding    gzip, deflate, br
Accept-Language    ru-RU,ru;q=0.9,en-US;q=0.8,en;q=0.7

```

Рисунок 4.5.3. Запрос клиента в среде разработки

Если пользователь неверно укажет имя страницы, появится сервер выдаст код состояния 404:

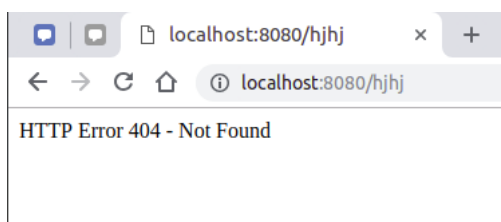


Рисунок 4.5.4. Пример ответа на некорректный запрос

Важно отметить, что для получение такой странички одного GET запроса недостаточно. Ведь страница наполнения медиа-элементами и гиперссылками, и браузер сам формирует оставшиеся запросы согласно html-коду страницы, которую мы указали.

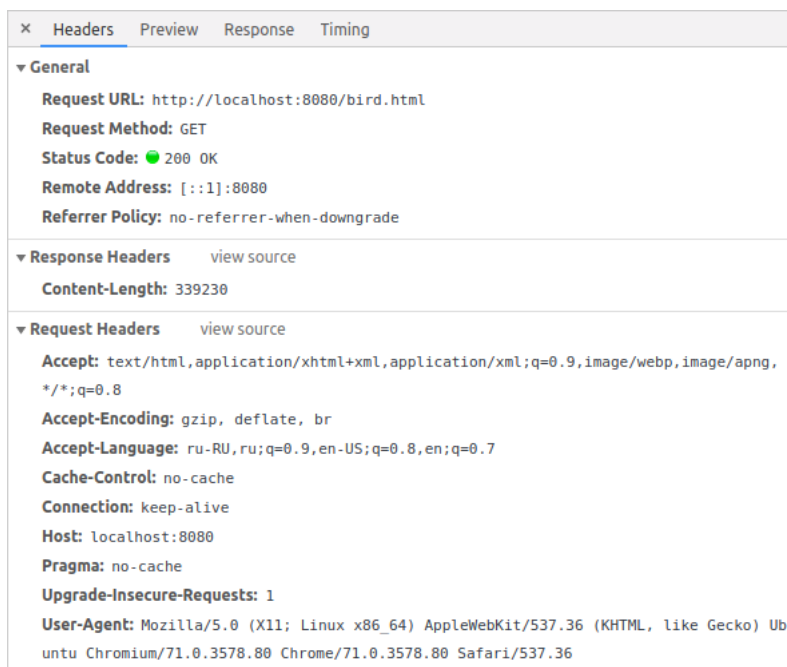


Рисунок 4.5.5. Запрос клиента в браузере

Приведенный выше пример демонстрирует обработку GET-запроса.

Для отправки POST-запроса есть тестовая форма для заполнения formExample.html. Для того, чтобы отправить запрос данного метода, пользователь должен ввести логин 'admin' и пароль 'password' в форму:

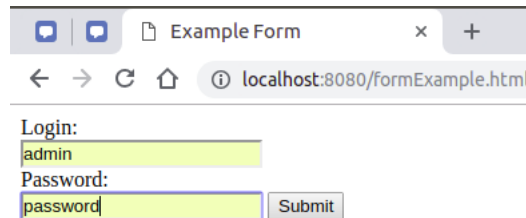


Рисунок 4.5.6. Пользователь заполняет форму

Если данные корректны, сервер ответит:

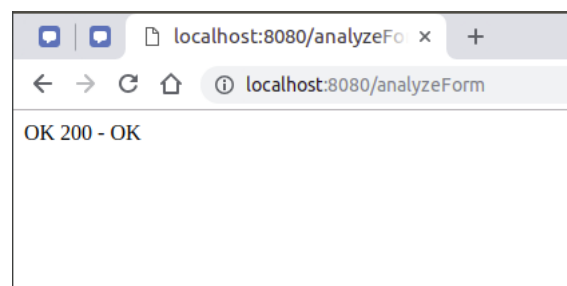


Рисунок 4.5.7. Ответ сервера на POST-запрос

Для того, чтобы продемонстрировать обработку HEAD запроса, можно воспользоваться программой cURL. В терминале необходимо ввести следующую строку:

```
curl -I -X HEAD http://localhost:8080/index.html
```

В ответе сервера мы должны получить только стартовую строку и заголовки:

```
bob@bob-HP:~$ curl -I -X HEAD http://localhost:8080/  
HTTP/1.1 200 OK  
Content-Length: 0
```

Рисунок 4.5.8. Обработка HEAD запроса

В ответе на данный запрос есть только один заголовок - Content-Length.

5. Список источников

1. В.М. Ицыксон. Методическое пособие «Технологии компьютерных сетей. Программирование сетевых приложений».
2. В.М. Ицыксон. Конспект лекций по курсу «Технологии компьютерных сетей».
3. Простым языком об HTTP [Электронный ресурс] <https://habr.com/post/215117/>
4. Программирование сокетов в Linux [Электронный ресурс] <https://rdsn.org/article/unix/sockets.xml>
5. Design and Implementation of an HTTP Server [Электронный ресурс] <https://users.cs.jmu.edu/bernstdh/web/common/lectures/>

6. Выводы

Таким образом, в процессе выполнения работы я получила базовый опыт создания сетевых приложений на основе TCP и UDP сокетов.

Мне удалось решить задачи многопоточности, обслуживания нескольких клиентов, корректного завершения работы и гарантированной передачи сообщения для клиент-серверных проектов на ОС Windows и Linux.

Знания, полученные по ходу выполнения первых трех заданий, я применила во время разработки первого индивидуального проекта - сетевого приложения мониторинга ошибок Bug Tracker.

За основу было взято клиент-серверное приложение на TCP сокетах для Linux, однако пришлось исправить один существенный момент - многопоточность в первой реализации сервера обеспечивалась за счет создания новых процессов, что для индивидуального задания оказалось не совсем удобным. Поэтому программу пришлось модифицировать, организовав обработку клиентов в новых нитях (потоках), а не в процессах.

В целом, проект первого индивидуального задания оказался довольно интересным, и я хотела бы даже продолжить его разработку и в дальнейшем перейти с консольного интерфейса на более приятный графический.

В качестве второго индивидуального задания я выбрала задачу реализации http протокола. Это один из самых популярных протоколов обмена данными между клиентом и сервером. Ежедневно мы десятки раз его используем, поэтому я посчитала важным углубиться в особенности его работы.

В процессе изучения протокола, я активно пользовалась материалами из источника [5]. Мне так же хотелось бы продолжить работу над этим проектом, потому как в связи с короткими сроками сдачи я недостаточно подробно разобралась.