

Сети ЭВМ и телекоммуникации

С. А. Климов

26 декабря 2014 г.

Глава 1

Задание

Разработать приложение-клиент и приложение сервер электронной почты.

1.1 Функциональные требования

Серверное приложение должно реализовывать следующие функции:

1. Приём почтового сообщения от одного клиента для другого
2. Хранение электронной почты для клиентов
3. Посылка клиенту почтового сообщения по запросу с последующим удалением сообщения
4. Посылка клиенту сведений о состоянии почтового ящика
5. Обработка запроса на отключение клиента
6. Принудительное отключение клиента

Клиентское приложение должно реализовывать следующие функции:

1. Передача электронного письма на сервер для другого клиента
2. Проверка состояния своего почтового ящика
3. Получение конкретного письма с сервера
4. Разрыв соединения
5. Обработка ситуации отключения клиента сервером

1.2 Нефункциональные требования

Для сервера:

1. Прослушивание определенного порта
2. Поддержка одновременной работы нескольких почтовых клиентов через механизм нитей
3. Обработка запросов на подключение по этому порту от клиентов

Для клиента:

1. Установление соединения с сервером

1.3 Настройки приложений

Разработанное клиентское приложение должно предоставлять пользователю настройку IP-адреса или доменного имени удалённого сервера почты и номера порта, используемого сервером. Разработанное серверное приложение должно хранить почту для клиентов.

1.4 Накладываемые ограничения

Пакеты для обмена по протоколу TCP будут иметь размер 1024 байта. В пакетах под имя клиента выделяется до 20-ти символов, 1 символ команды, 3 разделяющих символа, остальные 1000 символов выделяется под отправляемое сообщение. Этой длины достаточно для отправки писем средней длины.

Одновременное количество подключенных к серверу клиентов как минимум 5.

Глава 2

Реализация для работы по протоколу TSP

2.1 Прикладной протокол

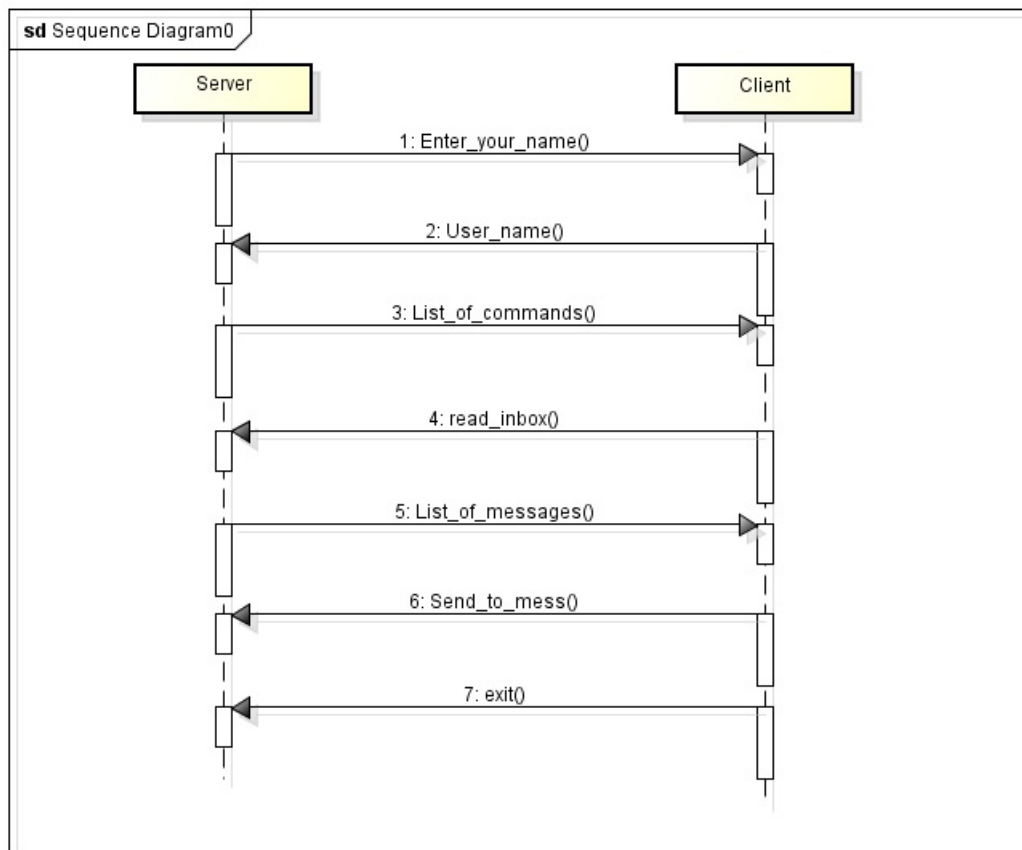
Первая команда ввода имя пользователя	username#
Команда написать kitty сообщение hello!	1#kitty#hello!#
Команда прочитать входящие	2#
Команда выхода	3#

2.2 Архитектура приложения

Взаимодействие сервера и клиента:

1. Сервер отправляет приветственное сообщение в котором клиенту предлагается ввести свой логин для идентификации его на сервере.
2. Клиент отправляет серверу свой логин.
3. Сервер отсылает клиенту список поддерживаемых команд(отослать письмо, прочитать входящие письма или выйти).
4. Клиент производит взаимодействие с сервером по средством ввода предложенных команд.
5. Клиент завершает работу с сервером.

Sequence диаграмма, демонстрирующая возможное взаимодействие клиента и сервера(после подключения клиента к серверу):



Почта для каждого клиента хранится на сервере в виде xml файла, имя которого совпадает с именем пользователя.

XML-Schema структуры файлов со входящими сообщениями:

```

1 <xs:schema attributeFormDefault="unqualified"
  elementFormDefault="qualified" xmlns:xs="http://www.w3.org
  /2001/XMLSchema">
2 <xs:element name="inbox">
3   <xs:sequence>
4     <xs:element name="mes" maxOccurs="unbounded"
      minOccurs="0">
5       <xs:simpleContent>
6         <xs:extension base="xs:string">
7           <xs:attribute type="xs:string" name="from"/>
8           <xs:attribute type="xs:string" name="text"/>
9         </xs:extension>
  
```

```
10         </xs:simpleContent>
11     </xs:element>
12 </xs:sequence>
13 </xs:element>
14 </xs:schema>
```

В данной архитектуре каждому клиенту выделяется свой поток для его обслуживания. Поток начинается при подключении очередного клиента к серверу. Далее он производит взаимодействие с клиентом, обрабатывая отправляемые им команды. Завершение потока происходит при отключении клиента от сервера.

2.3 Тестирование

2.3.1 Описание тестового стенда и методики тестирования

Тестирование проводилось на виртуальной машине Debian 4.7. Было запущено приложение сервера, затем - несколько приложений клиента. Таким образом сервер и клиент работали на одном компьютере.

2.3.2 Тестовый план и результаты тестирования

1. На первом клиенте был осуществлен вход под логином "serg" и было отправлено сообщение "hello!" клиенту "ali"
2. Был получен список входящих сообщений для пользователя "serg".
3. В другом терминале был запущен клиент с именем "ali" (терминал пользователя "serg" оставался активным).
4. Был получен список входящих сообщений, последним из них оказалось только что отправленное сообщение ("hello!") пользователем "serg".
5. Было отправлено сообщение "nice to meet you!" пользователю "serg" и осуществлен выход.
6. На терминале пользователя "serg" были прочитаны сообщения, последним из них оказалось только что отправленное сообщение ("nice to meet you!") пользователем "ali".

7. Был осуществлен выход пользователя "serg".

Ожидалось получить корректное взаимодействие сервера с клиентами, что и было получено: сервер корректно работает с несколькими клиентами. При вводе некорректных команд клиентское приложение сообщает об этом пользователю и продолжает работу. Серверное приложение так же выводит на консоль некорректные команды от пользователя.

Тестирование работы сервера с большим количеством клиентов:

1. Был запущен сервер.
2. Из различных терминалов было осуществлено подключение 5 клиентов.
3. Для каждого из клиентов была осуществлена отправка сообщения какому-либо пользователю.
4. В одном из клиентов была добавлена команда `sleep(10)`.
5. Остальные клиенты в это время продолжили взаимодействие с сервером.
6. Для каждого из клиентов был получен список писем.
7. Был осуществлен выход клиентами.

Ожидалось получить корректное взаимодействие сервера с данным количеством клиентов, это и было получено: клиенты корректно отправляют сообщения друг другу и осуществляют просмотр входящих сообщений.

Глава 3

Реализация для работы по протоколу UDP

3.1 Прикладной протокол

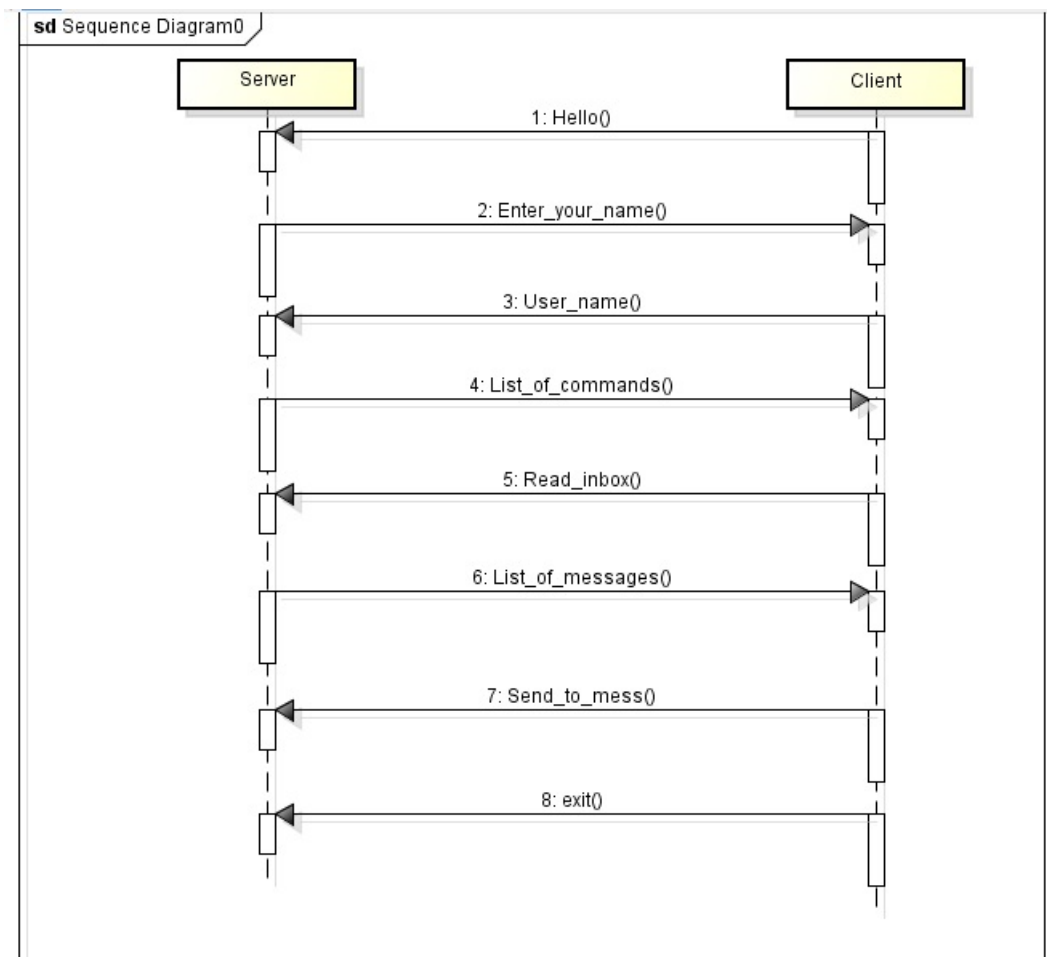
Прикладной протокол понес небольшие изменения по сравнению с UDP. Однако для пользователя взаимодействие с сервером осалось аналогичным.

3.2 Архитектура приложения

Взаимодействие сервера и клиента:

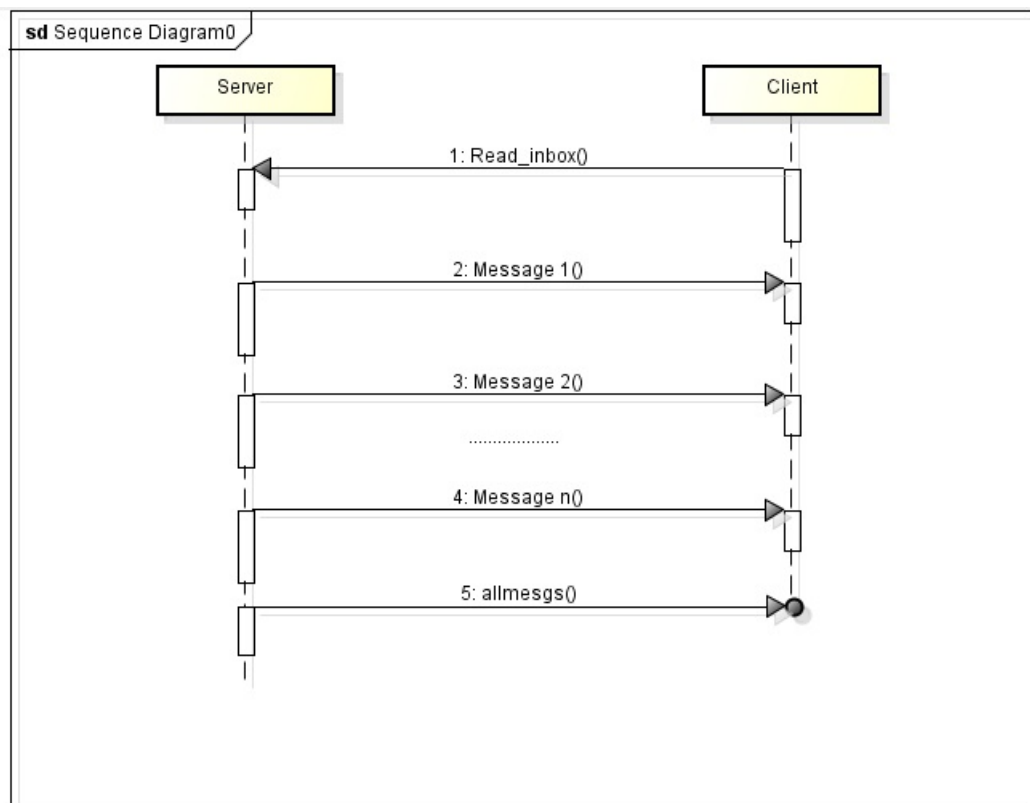
1. Клиент отправляет серверу приветственное сообщение для начала взаимодействия.
2. Сервер отвечает ему сообщением в котором клиенту предлагается ввести свой логин для идентификации его на сервере.
3. Клиент отправляет серверу свой логин.
4. Сервер отсылает клиенту список поддерживаемых команд(отослать письмо, прочитать входящие письма или выйти).
5. Клиент производит взаимодействие с сервером по средством ввода предложенных команд.
6. Клиент завершает работу с сервером.

Sequence диаграмма, демонстрирующая возможное взаимодействие клиента и сервера:



Изменение понес процесс отправки списка всех сообщений клиенту, если в TCP просто осуществляется потоковая передача, то в UDP мы принимаем пакеты, пока не придет пакет определенного за ранее содержания, сигнализирующий о конце отправки писем.

Sequence диаграмма, демонстрирующая взаимодействие при получении клиентом списка сообщений от сервера:



Почта также хранится в xml файлах, как и в TCP. XML-Schema структуры файлов со входящими сообщениями аналогична.

3.3 Тестирование

3.3.1 Описание тестового стенда и методики тестирования

Тестирование проводилось на виртуальной машине Debian 4.7. Было запущено приложение сервера, затем - несколько приложений клиента. Таким образом сервер и клиент работали на одном компьютере.

3.3.2 Тестовый план и результаты тестирования

1. На первом клиенте был осуществлен вход под логином "serg" и было отправлено сообщение "hello!" клиенту "ali"
2. Был получен список входящих сообщений для пользователя "serg".

3. В другом терминале был запущен клиент с именем "ali" (терминал пользователя "serg" оставался активным).
4. Был получен список входящих сообщений, последним из них оказалось только что отправленное сообщение ("hello!") пользователем "serg".
5. Было отправлено сообщение "nice to meet you!" пользователю "serg" и осуществлен выход.
6. На терминале пользователя "serg" были прочитаны сообщения, последним из них оказалось только что отправленное сообщение ("nice to meet you!") пользователем "ali".
7. Был осуществлен выход пользователя "serg".

Ожидалось получить корректное взаимодействие сервера с клиентами, что и было получено: сервер корректно работает с несколькими клиентами. При вводе некорректных команд клиентское приложение сообщает об этом пользователю и продолжает работу. Серверное приложение так же выводит на консоль некорректные команды от пользователя.

Тестирование работы сервера с большим количеством клиентов:

1. Был запущен сервер.
2. Из различных терминалов было осуществлено подключение 5 клиентов.
3. Для каждого из клиентов была осуществлена отправка сообщения какому-либо пользователю.
4. В одном из клиентов была добавлена команда sleep(10).
5. Остальные клиенты в это время не смогли осуществить взаимодействие с клиентом, они ждали, пока закончится sleep.
6. Для каждого из клиентов был получен список писем.
7. Был осуществлен выход клиентами.

Ожидалось получить корректное взаимодействие сервера с данным количеством клиентов, это и было получено: клиенты корректно отправляют сообщения друг другу и осуществляют просмотр входящих сообщений.

Глава 4

Выводы

4.1 TCP

Протокол TCP является ориентированным на создание соединения, в нем производится так называемое "рукопожатие" для его установки. При установлении соединения становится возможной передача данных в обоих направлениях.

TCP является надежным протоколом, т.к. он управляет подтверждением, повторной передачей и тайм-аутом сообщений. Также TCP осуществляет контроль порядка доставки сообщений, т.е. если сообщения доставляются не друг за другом, а в случайном порядке, то они вначале кешируются, а затем упорядочиваются и только потом передаются приложению. Важной особенностью TCP является то, что данные считываются как единый поток байтов, нет никаких границ для отдельных сообщений. Очевидным достоинством TCP является его надежность. Однако, при реализации приложения с его использованием можно столкнуться со следующими трудностями: организация работы сервера с несколькими клиентами. Для взаимодействия с несколькими клиентами серверу необходимо создавать новый сокет для каждого из них и передавать управление в отдельную нить, организуемую для обслуживания клиента.

4.2 UDP

Протокол UDP является более простым протоколом, нежели TCP. Он не основан на установлении соединения, а просто использует механизм обмена данными для взаимодействия с клиентом.

Протокол UDP считается ненадежным, т.к. он не управляет

подтверждением, повторной передачей и тайм-аутом сообщений. Он не осуществляет контроль порядка передачи дэйтиграмм. Дэйтиграммы имеют определенные границы, т.е. они интерпретируются на приемной стороне однозначно, их целостность проверяется только после получения. Важной особенностью UDP является его простота - нет механизма "рукопожатия упорядочивания и отслеживания соединений. Хотя UDP является более простым, на разработчика ложатся дополнительные задачи при его использовании. Необходимо обеспечить контроль очередности доставки пакетов и повторную их отправку при потере пакета. При реализации взаимодействия с несколькими клиентами сервер UDP, в отличие от TCP, использует только один поток для их обслуживания, это необходимо учитывать при работе чтобы отправлять корректные данные тому или иному клиенту. В реализованном сервере используется хранение адреса и порта подключаемого клиента. Это не дает возможности параллельной работы со всеми клиентами, применением мьютексов также не удалось решить данную проблему.

Как показало тестирование, реализованные приложения как с использованием TCP, так и UDP работают корректно.

Глава 5

Приложения

5.1 Описание среды разработки

Linux Debian 7.6: Среда разработки - Eclipse.

Windows 8.1: Среда разработки - Visual Studio 2013.

5.2 Листинги

5.2.1 Сервер TCP Linux. Основной файл программы main.c

```
1
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7
8 #include <libxml/parser.h>
9 #include <libxml/tree.h>
10 #include <libxml/xmlwriter.h>
11 #include <libxml/xmlmemory.h>
12 #include <libxml/xpath.h>
13
14 #include <sys/stat.h>
15 #include <unistd.h>
16 #include <string.h>
17 #include <pthread.h>
18
19 #define MY_ENCODING "ISO-8859-1"
20 #define MY_PORT 5001
```

```

21
22 void get_args(char* inp_str, char* arg[]){//buffer and array
    to write
23     int i = 0, j;
24     const char s[2] = "#";
25     char *token;
26     token = strtok(inp_str, s);
27     arg[i] = token;
28     while(token != NULL){
29         i++;
30         token = strtok(NULL, s);
31         arg[i] = token;
32     }
33     i--;
34     arg[i] = NULL;
35 }
36
37 char *get_mes_from_client(int newsockfd, char *buffer){
38     int n;
39     bzero(buffer,256);
40     n = read(newsockfd, buffer, 255);
41     if (n < 0)
42     {
43         perror("ERROR_□reading_□from_□socket");
44         exit(1);
45     }
46     return buffer;
47 }
48
49 char* get_hello(int newsockfd, char *buffer, char* arg[]){
50     int i,j=0;
51     char user[10] = "no_□user";
52     if((arg[0]!=NULL)&&(arg[1]!=NULL)){
53         strcpy(user, arg[0]);
54         char command[7];
55         strcpy(command, arg[1]);
56         j = atoi(command);
57         printf("j=%d\n",j);
58         if (j==1){
59             do_send(user, arg[2], arg[3]);
60         }
61         else if (j==2){
62             do_read(newsockfd, buffer, user);
63         }
64         else if (j==3){
65             do_exit();
66         }
67         else {
68             puts("Wrong_□command!\n");

```

```

69     }
70 }
71 printf("user:_%s\n", user);
72 return user;
73 }
74
75 void do_send(char* from, char* to, char* mes){
76     printf("Send_command\n");
77     create_file(to, from, mes);
78 }
79
80 void do_read(int newsockfd, char *buffer, char* client){
81     printf("Read_command\n");
82     print_mesgs(buffer, newsockfd, client);
83 }
84
85 void do_exit(){
86     printf("Exit_command\n");
87     exit(1);
88 }
89
90 void do_serve(int newsockfd, char *buffer, char* arg[], char*
    user){
91     int i,j=0;
92     if(arg[0]!=NULL){
93         char command[7];
94         strcpy(command, arg[0]);
95         j = atoi(command);
96         printf("j=%d\n",j);
97         if (j==1){
98             do_send(user, arg[1], arg[2]);
99         }
100        else if (j==2){
101            do_read(newsockfd, buffer, user);
102        }
103        else if (j==3){
104            do_exit();
105        }
106        else {
107            puts("Wrong_command!\n");
108        }
109    }
110 }
111
112 void do_wait_mes(int newsockfd, char *buffer, char* user){
113     char* inp;
114     char* get_inp[4];
115     while(1){
116         inp=get_mes_from_client(newsockfd, buffer);

```



```

117     get_args(inp, get_inp);
118     do_serve(newsockfd, buffer, get_inp, user);
119 }
120 }
121
122 void add_mes(xmlNode* node, char* from, char* msg){
123     xmlNode* cur_node = NULL;
124     char buf[256];
125     bzero(buf, 256);
126     for (cur_node = node; cur_node; cur_node = cur_node->next)
127     {
128         if (cur_node->type == XML_ELEMENT_NODE) {
129             if ((!xmlStrcmp(cur_node->name, (const xmlChar *) "
inbox"))){
130                 xmlNodePtr nNode = xmlNewNode(0, (const xmlChar
*) "mes");
131                 xmlSetProp(nNode, (const xmlChar *) "from", (
const xmlChar *) from);
132                 xmlSetProp(nNode, (const xmlChar *) "text", (
const xmlChar *) msg);
133                 //xmlNodeSetContent(nNode, (xmlChar*)msg);
134                 xmlAddChild(cur_node, nNode);
135                 return;
136             }
137         }
138         add_mes(cur_node->children, from, msg);
139     }
140 }
141
142 void read_mes(xmlNode* node, char* mess, int newsockfd){
143     xmlNode *cur_node = NULL;
144     char buf[256];
145     bzero(buf, 256);
146     int n;
147     for (cur_node = node; cur_node; cur_node = cur_node->next
) {
148         if (cur_node->type == XML_ELEMENT_NODE) {
149             if ((!xmlStrcmp(cur_node->name, (const xmlChar *) "
mes"))){
150                 {
151                     strcpy(buf, "from:");
152                     strncat(buf, xmlGetProp(cur_node, "from"), strlen
(xmlGetProp(cur_node, "from")));
153                     strcat(buf, "message:");
154                     strncat(buf, xmlGetProp(cur_node, "text"), strlen
(xmlGetProp(cur_node, "text")));
155                     strcat(buf, "\n");
156                     n = write(newsockfd, buf, strlen(buf));
157                     //n=read(newsockfd, buf, 255);

```

```

158         }
159     }
160
161     read_mes(cur_node->children,mess,newsockfd);
162 }
163 }
164
165 void print_mesgs(char* buffer ,int newsockfd, char* user)
166 {
167
168     xmlDoc          *doc = NULL;
169     xmlNode          *root_element = NULL;
170     int n;
171     doc = xmlReadFile(user, NULL, 0);
172     if (doc == NULL)
173     {
174         printf("error: could not parse file %s\n",
175             user);
176         //strncat(buffer,"Error\n",6);
177         n = write(newsockfd,"Error\n",6);
178     }
179     else
180     {
181         root_element = xmlDocGetRootElement(doc);
182         n = write(newsockfd,"Messages:\n",11);
183         read_mes(root_element,buffer,newsockfd);
184         //n = write(newsockfd,"end",3);
185         xmlFreeDoc(doc);
186     }
187
188     xmlCleanupParser();
189
190     return;
191 }
192 }
193
194 void create_file(char *to, char *from, char *msg){
195     int rc;
196     xmlTextWriterPtr writer;
197     xmlDocPtr doc;
198     xmlNodePtr node, root;
199     xmlChar *tmp;
200     if(doc = xmlReadFile(to, NULL, 0)){
201         root = xmlDocGetRootElement(doc);
202         add_mes(root, from, msg);
203         xmlSaveFile(to, doc);
204         xmlFreeDoc(doc);
205     }else{

```

```

206     doc = xmlNewDoc(BAD_CAST XML_DEFAULT_VERSION);
207     node = xmlNewDocNode(doc, NULL, BAD_CAST "inbox", NULL)
        ;
208     xmlDocSetRootElement(doc, node);
209     add_mes(node, from, msg);
210     xmlSaveFile(to, doc);
211     xmlFreeDoc(doc);
212 }
213 xmlCleanupParser();
214 }
215 struct sockParams
216 {
217     int sockfd, newsockfd, port_number, client;
218     struct sockaddr_in serv_addr, cli_addr;
219 };
220
221 void startThread(void *in)
222 {
223     struct sockParams *sp = (struct sockParams *)in;
224     start_work(sp->newsockfd);
225 }
226
227 void start_work(int newsockfd){
228     char buffer[256];
229     int n;
230     bzero(buffer,256);
231     n = read( newsockfd,buffer,255 );
232     if (n < 0)
233     {
234         perror("ERROR reading from socket");
235         exit(1);
236     }
237     //int j;
238     char* get_inp[4];
239     get_args(buffer, get_inp);
240     char user_name[10]="nnm";
241     strcpy(user_name, get_hello(newsockfd, buffer,
        get_inp));
242     do_wait_mes(newsockfd, buffer, user_name);
243     //return 0;
244 }
245
246 int main( int argc, char *argv[] )
247 {
248     pthread_t thread[5], mainthread;
249     int i=0, j=0;
250     struct sockParams sp;
251
252     sp.sockfd = socket(AF_INET, SOCK_STREAM, 0);

```

```

253     if (sp.sockfd < 0)
254     {
255         perror("ERROR_␣opening_␣socket");
256         exit(1);
257     }
258     bzero((char *) &sp.serv_addr, sizeof(sp.serv_addr));
259     sp.port_number = MY_PORT;
260     //portno = 7771;
261     sp.serv_addr.sin_family = AF_INET;
262     sp.serv_addr.sin_addr.s_addr = INADDR_ANY;
263     sp.serv_addr.sin_port = htons(sp.port_number);
264
265     /* Now bind the host address using bind() call.*/
266     if (bind(sp.sockfd, (struct sockaddr *) &sp.serv_addr,
267             sizeof(sp.serv_addr)) < 0)
268     {
269         perror("ERROR_␣on_␣binding");
270         exit(1);
271     }
272
273     /* Now start listening for the clients, here process
274     will
275     * go in sleep mode and will wait for the incoming
276     connection
277     */
278     while(1){
279         listen(sp.sockfd,5);
280         sp.client = sizeof(sp.cli_addr);
281
282         //printf("cl: %s\n", sp.client);
283         /* Accept actual connection from the client */
284         sp.newsockfd = accept(sp.sockfd, (struct sockaddr *)&sp
285                               .cli_addr, &sp.client);
286         if (sp.newsockfd < 0)
287         {
288             perror("ERROR_␣on_␣accept");
289             exit(1);
290         }
291         pthread_create(&thread[i], NULL, startThread, (void*)&sp
292                       );
293         i++;
294     }
295
296     for(j=0;j<5;j++)
297         pthread_join(thread[j], NULL);
298     return 0;
299 }

```

5.2.2 Сервер TCP Linux. Файл сборки Makefile

```
1 #it's my makefile
2 all:
3     #gcc -I/home/user/Downloads/libxml2-2.7.8/include main.c -
4         Wall -g -O0 -L/home/user/Downloads/libxml2-2.7.8 -lxml2
5         -o server1
6     gcc -g -lpthread -o0 -Wall -o ./Debug/Server1 main.c -
7         lxml2 -I/home/user/Downloads/libxml2-2.7.8/include
8
9 clean:
10     rm ./Debug/Server1
11     rm *.o
```

5.2.3 Клиент TCP Linux. Основной файл программы main.c

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/socket.h>
4 #include <netinet/in.h>
5
6 #include <time.h>
7
8 void write_socket_start(char* buffer, int sockfd){
9     int n;
10    bzero(buffer,256);
11    fgets(buffer,255,stdin);
12    //puts(buffer);
13    /* Send message to the server */
14    n = write(sockfd,buffer,strlen(buffer));
15    if (n < 0)
16    {
17        perror("ERROR_writing_to_socket");
18        exit(1);
19    }
20 }
21
22 void read_response(char* buffer, int sockfd){
23     int n;
24     bzero(buffer,256);
25     n = read(sockfd,buffer,255);
26     if (n < 0)
27     {
28         perror("ERROR_reading_from_socket");
29         exit(1);
30     }
```

```

31     printf("%s",buffer);
32 }
33 //-----
34 void get_args(char* inp_str, char* arg[]){//buffer and array
    to write
35     int i = 0, j;
36     const char s[2] = "#";
37     char *token;
38     token = strtok(inp_str, s);
39     arg[i] = token;
40     while(token != NULL){
41         i++;
42         token = strtok(NULL, s);
43         arg[i] = token;
44     }
45     i--;
46     arg[i] = NULL;
47 }
48
49 char *get_mes_from_client(int newsockfd, char *buffer){
50     int n;
51     bzero(buffer,256);
52     n = read(newsockfd, buffer, 255);
53     if (n < 0)
54     {
55         perror("ERROR_□reading_□from_□socket");
56         exit(1);
57     }
58     return buffer;
59 }
60
61 void resp_send(char* from, char* to, char* mes){
62 }
63
64 void resp_read(int newsockfd, char *buffer, char* client){
65     print_mesgs(buffer, newsockfd, client);
66 }
67
68 void resp_exit(){
69     exit(1);
70 }
71
72 void print_mesgs(char* buffer ,int sockfd, char* user){
73     read_response(buffer, sockfd);
74     read_response(buffer, sockfd);
75 }
76
77 char* put_hello(int sockfd, char *buffer){
78     write_socket_start(buffer, sockfd);

```

```

79     char* arg[4];
80     get_args(buffer, arg);
81     int i,j=0;
82     char user[10] = "no_user";
83     if((arg[0]!=NULL)&&(arg[1]!=NULL)){
84         strcpy(user, arg[0]);
85         char command[7];
86         strcpy(command, arg[1]);
87         j = atoi(command);
88         if (j==1){
89             resp_send(user, arg[2], arg[3]);
90         }
91         else if (j==2){
92             resp_read(sockfd, buffer, user);
93         }
94         else if (j==3){
95             resp_exit();
96         }
97         else {
98             puts("Wrong_command!\n");
99         }
100     }
101     return user;
102 }
103
104 void keep_talking(int sockfd, char *buffer){
105     write_socket_start(buffer, sockfd);
106     char* arg[4];
107     get_args(buffer, arg);
108     int i,j=0;
109     char user[10] = "no_user";
110     if((arg[0]!=NULL)){
111         strcpy(user, arg[0]);
112         char command[7];
113         strcpy(command, arg[0]);
114         j = atoi(command);
115         if (j==1){
116             resp_send(user, arg[2], arg[3]);
117         }
118         else if (j==2){
119             resp_read(sockfd, buffer, user);
120         }
121         else if (j==3){
122             resp_exit();
123         }
124         else {
125             puts("Wrong_command!\n");
126         }
127     }

```

```

128 }
129
130 int main(int argc, char *argv[])
131 {
132     int sockfd, portno, n;
133     struct sockaddr_in serv_addr;
134     struct hostent *server;
135
136     char buffer[256];
137
138     if (argc < 3) {
139         fprintf(stderr, "usage %s hostname port\n", argv[0]);
140         exit(0);
141     }
142     portno = atoi(argv[2]);
143     /* Create a socket point */
144     sockfd = socket(AF_INET, SOCK_STREAM, 0);
145     if (sockfd < 0)
146     {
147         perror("ERROR opening socket");
148         exit(1);
149     }
150     server = gethostbyname(argv[1]);
151     if (server == NULL) {
152         fprintf(stderr, "ERROR, no such host\n");
153         exit(0);
154     }
155
156     bzero((char *) &serv_addr, sizeof(serv_addr));
157     serv_addr.sin_family = AF_INET;
158     serv_addr.sin_port = htons(portno);
159
160     /* Now connect to the server */
161     if (connect(sockfd, &serv_addr, sizeof(serv_addr)) < 0)
162     {
163         perror("ERROR connecting");
164         exit(1);
165     }
166     /* Now ask for a message from the user, this message
167     * will be read by server
168     */
169     put_hello(sockfd, buffer);
170
171     while(1)
172         keep_talking(sockfd, buffer);
173     return 0;
174 }

```


5.2.4 Клиент TCP Linux. Файл сборки Makefile

```
1 #it's my makefile
2 all:
3     #gcc -I/home/user/Downloads/libxml2-2.7.8/include main.c -
4         Wall -g -O0 -L/home/user/Downloads/libxml2-2.7.8 -lxml2
5         -o server1
6     gcc -g -O0 -Wall -o ./Debug/Client1 main.c -lxml2 -I/home
7         /user/Downloads/libxml2-2.7.8/include
8
9 clean:
10     rm ./Debug/Client1
11     rm *.o
```

5.2.5 Сервер UDP Linux. Основной файл программы main.c

```
1 #include<stdio.h> //printf
2 #include<string.h> //memset
3 #include<stdlib.h> //exit(0);
4 #include<arpa/inet.h>
5 #include<sys/socket.h>
6 #include <assert.h>
7
8 #include <libxml/parser.h>
9 #include <libxml/tree.h>
10 #include <libxml/xmlwriter.h>
11 #include <libxml/xmlmemory.h>
12 #include <libxml/xpath.h>
13
14 #define BUFLen 1024 //Max length of buffer
15 #define MY_PORT 5001 //The port on which to listen for
16     incoming data
17
18 void die(char *s)
19 {
20     perror(s);
21     exit(1);
22 }
23
24 char** splito_array(char* str){
25     //char str[] = "ls -l";
26     char ** res = NULL;
27     char * p = strtok (str, "#");
28     int n_spaces = 0, i;
29 }
```

```

30  /* split string and append tokens to 'res' */
31
32  while (p) {
33      res = realloc (res, sizeof (char*) * ++n_spaces);
34
35      if (res == NULL)
36          exit (-1); /* memory allocation failed */
37
38      res[n_spaces-1] = p;
39
40      p = strtok (NULL, "#");
41  }
42
43  /* realloc one extra element for the last NULL */
44
45  res = realloc (res, sizeof (char*) * (n_spaces+1));
46  res[n_spaces] = 0;
47
48  /* free the memory allocated */
49
50  free (res);
51  return res;
52 }
53
54 void get_args(char* inp_str, char* arg[]){//buffer and array
    to write
55     int i = 0, j;
56     const char s[2] = "#";
57     char *token;
58     token = strtok(inp_str, s);
59     arg[i] = token;
60     while(token != NULL){
61         i++;
62         token = strtok(NULL, s);
63         arg[i] = token;
64     }
65     i--;
66     arg[i] = NULL;
67 }
68
69 char** str_split(char* a_str, const char a_delim)
70 {
71     char** result      = 0;
72     size_t count       = 0;
73     char* tmp          = a_str;
74     char* last_comma   = 0;
75     char delim[2];
76     delim[0] = a_delim;
77     delim[1] = 0;

```

```

78
79  /* Count how many elements will be extracted. */
80  while (*tmp)
81  {
82      if (a_delim == *tmp)
83      {
84          count++;
85          last_comma = tmp;
86      }
87      tmp++;
88  }
89
90  /* Add space for trailing token. */
91  count += last_comma < (a_str + strlen(a_str) - 1);
92
93  /* Add space for terminating null string so caller
94     knows where the list of returned strings ends. */
95  count++;
96
97  result = malloc(sizeof(char*) * count);
98
99  if (result)
100  {
101      size_t idx = 0;
102      char* token = strtok(a_str, delim);
103
104      while (token)
105      {
106          assert(idx < count);
107          *(result + idx++) = strdup(token);
108          token = strtok(0, delim);
109      }
110      assert(idx == count - 1);
111      *(result + idx) = 0;
112  }
113
114  return result;
115 }
116
117 void do_send(char* from, char* to, char* mes){
118     printf("Send␣command\n");
119     create_file(to, from, mes);
120 }
121
122 void do_read(int newsockfd, char *buffer, char* client,
123             struct sockaddr_in si_other, int slen){
124     printf("Read␣command\n");
125     print_msgs(buffer, newsockfd, client, si_other, slen);

```

```

126
127 void read_mes(xmlNode* node, char* mess, int s, struct
    sockaddr_in si_other, int slen){
128     xmlNode *cur_node = NULL;
129     char buf[BUFLen];
130     bzero(buf, BUFLen);
131     int n;
132     for (cur_node = node; cur_node; cur_node = cur_node->next
        ) {
133         if (cur_node->type == XML_ELEMENT_NODE) {
134             if ((!xmlStrcmp(cur_node->name, (const xmlChar *) "
                mes"))))
135                 {
136                     strcpy(buf, "from: ");
137                     strncat(buf, xmlGetProp(cur_node, "from"), strlen
                        (xmlGetProp(cur_node, "from")));
138                     strcat(buf, "message: ");
139                     strncat(buf, xmlGetProp(cur_node, "text"), strlen
                        (xmlGetProp(cur_node, "text")));
140                     strcat(buf, "\n");
141
142                     if (sendto(s, buf, sizeof(buf),
143
                                0, (
                                    struct
                                    sockaddr
                                    *) &
                                    si_other
                                    , slen)
                                    == -1)
144                         {
145                             die
                                (
                                    "
                                sendto
                                ()
                                "
                                )
                                ;
146                         }
147
148                         //
                                receive
                                a
                                reply
                                and
                                print
                                it

```

149

```
//clear  
the  
  
buffer  
by  
filling  
  
null  
, it  
  
might  
  
have  
  
previously  
received
```

150

```
data  
//  
memset  
(buf  
, '\0',  
  
BUFLEN  
);
```

151

```
//try  
to  
receive  
  
some  
  
data  
,  
this  
is  
a  
blocking  
  
call
```

152

```
if (  
recvfrom  
(s,  
buf,  
  
sizeof  
(buf  
) ,
```

```

153 | 0,
    | (
    | struct
    | sockaddr
    | *)
    | &
    | si_other
    | ,
    | &
    | slen
    | )
    | ==
    | -1)
154 | {
155 | //
    | puts
    | ("
    | qq
    | ")
    | ;
156 | //
    | die
    | ("
    | recvfrom
    | ()
    | ")
    | ;
157 | }
158 |
159 | }
160 | }
161 |
162 | read_mes(cur_node->children,mess,s, si_other, slen);
163 | }
164 | }
165 |
166 | void print_mesgs(char* buffer ,int s, char* user, struct
    | sockaddr_in si_other, int slen)
167 | {
168 |

```

```

169         xmlDoc          *doc = NULL;
170         xmlNode          *root_element = NULL;
171         int n;
172         doc = xmlReadFile(user, NULL, 0);
173         if (doc == NULL)
174             {
175                 printf("error: could not parse file %s\n",
176                     user);
177             }
178         else
179             {
180                 root_element = xmlDocGetRootElement(doc);
181
182                 if (sendto(s, "Messages:\n", sizeof("
183                     Messages:\n"),
184                         0, (struct sockaddr *) &
185                             si_other, slen)==-1)
186                     {
187                         die("sendto()");
188                     }
189
190                     //receive a reply and
191                     //print it
192                     //clear the buffer by
193                     //filling null, it
194                     //might have
195                     //previously received
196                     //data
197                     //memset(buf, '\0',
198                     //    BUFLen);
199                     //try to receive some
200                     //data, this is a
201                     //blocking call
202                     if (recvfrom(s, "
203                         Messages:\n", sizeof(
204                             "Messages:\n"),
205                             0, (struct sockaddr
206                                 *) &si_other, &
207                                 slen) == -1)
208                         {
209                             //puts("qq");
210                             //die("recvfrom()")
211                             ;
212                         }
213
214                     //n = write(newsockfd, "Messages:\n", 11);

```

```

201         read_mes(root_element,buffer,s, si_other,
202                 slen);
203         //n = write(newsockfd,"end",3);
204         if (sendto(s, "allmsgs", sizeof("allmsgs"
205                 ),
206                 0 , (
207                     struct
208                         sockaddr
209                         *)
210                     &
211                     si_other
212                     ,
213                     slen
214                     )
215                     ==-1)
216         {
217             die
218             (
219                 "
220                 sendto
221                 ()
222                 "
223                 )
224             ;
225         }
226
227         //
228         receive
229         a
230
231         reply
232
233         and
234
235         print
236
237         it
238
239         //
240         clear
241
242         the
243
244         buffer
245
246         by

```


		<i>filling</i>
		<i>null</i>
		<i>,</i>
		<i>it</i>
		<i>might</i>
		<i>have</i>
		<i>previously</i>
		<i>received</i>
		<i>data</i>
212		<i>//</i>
		<i>memset</i>
		<i>(</i>
		<i>buf</i>
		<i>, '\0',</i>
		<i>BUFLN</i>
		<i>);</i>
213		
214	<i>if (recvfrom(s, "allmsgs", sizeof("</i>	
215	<i>allmsgs"),</i>	

```

216
217
218
219
220
221         xmlFreeDoc(doc);
222     }
223
224     xmlCleanupParser();
225
226     return;
227 }
228
229
230 void add_mes(xmlNode* node, char* from, char* msg){
231     xmlNode* cur_node = NULL;
232     char buf[BUFLLEN];
233     bzero(buf,BUFLLEN);
234     for (cur_node = node; cur_node; cur_node = cur_node->next)
235     {
236         if (cur_node->type == XML_ELEMENT_NODE) {
237             if ((!xmlStrcmp(cur_node->name,(const xmlChar *)"
238                 inbox"))){
239                 xmlNodePtr nNode = xmlNewNode(0,(const xmlChar
240                     *)"mes");
241                 xmlSetProp(nNode,(const xmlChar *)"from",(
242                     const xmlChar *)from);
243                 xmlSetProp(nNode,(const xmlChar *)"text",(
244                     const xmlChar *)msg);
245                 //xmlNodeSetContent(nNode, (xmlChar*)msg);
246                 xmlAddChild(cur_node,nNode);
247                 return;
248             }
249         }
250     }

```

```

246     add_mes(cur_node->children, from, msg);
247 }
248 }
249
250 void create_file(char *to, char *from, char *msg){
251     int rc;
252     xmlTextWriterPtr writer;
253     xmlDocPtr doc;
254     xmlNodePtr node, root;
255     xmlChar *tmp;
256     if(doc = xmlReadFile(to, NULL, 0)){
257         root = xmlDocGetRootElement(doc);
258         add_mes(root, from, msg);
259         xmlSaveFile(to, doc);
260         xmlFreeDoc(doc);
261     }else{
262         doc = xmlNewDoc(BAD_CAST XML_DEFAULT_VERSION);
263         node = xmlNewDocNode(doc, NULL, BAD_CAST "inbox", NULL)
                ;
264         xmlDocSetRootElement(doc, node);
265         add_mes(node, from, msg);
266         xmlSaveFile(to, doc);
267         xmlFreeDoc(doc);
268     }
269     xmlCleanupParser();
270 }
271
272 int get_cmd(char* buffer, int num){
273     int cmd;
274     char **temp;
275     temp=str_split(buffer, '#');
276
277     if(temp){
278         int i;
279         for (i = 0; *(temp + i); i++)
280         {
281             printf("inp␣:%s\n", *(temp + i));
282             if(i == num)
283                 cmd = atoi(*(temp + i));
284             free(*(temp + i));
285         }
286         free(temp);
287     }
288     return cmd;
289 }
290
291 char* get_cmd_ch(char* buffer, int num, char* res){
292     char* cmd;
293     char **temp;

```

```

294     char *teeemp;
295     temp=str_split(buffer, '#');
296
297         if(temp){
298             int i;
299             for (i = 0; *(temp + i); i++)
300                 {
301
302                     if(i == num)
303                         cmd = (*(temp + i));
304
305                     strcpy(res, (*(temp + i)));
306                     free(*(temp + i));
307                 }
308             free(temp);
309         }
310     return cmd;
311 }
312
313 int use_token(char* str, int num, char* res){
314     const char s[2] = "#";
315     char *token;
316     int i = 0;
317
318     token = strtok(str, s);
319     if(num==0){
320         res = token;
321         return 0;
322     }
323     while(token != NULL){
324         printf("i:%d, t:%s", i, token);
325         i++;
326         token = strtok(NULL, s);
327         if(num==i){
328             res = token;
329             return 0;
330         }
331     }
332     return 0;
333 }
334
335 void squeeze (char s[], int c) {
336     int i, j;
337
338     for (i = j = 0; s[i] != '\0'; i++)
339         if (s[i] != c)
340             s[j++] = s[i];
341     s[j] = '\0';
342 }

```

```

343
344 int main(void)
345 {
346     struct sockaddr_in si_me, si_other;
347
348     int s, i, slen = sizeof(si_other) , recv_len;
349     char buf[BUFLen];
350
351     char* user_names[5];
352     char* user_addr[5];
353     //int user_port[5];
354     uint16_t user_port[5];
355
356     //create a UDP socket
357     if ((s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1)
358     {
359         die("socket");
360     }
361
362     // zero out the structure
363     memset((char *) &si_me, 0, sizeof(si_me));
364
365     si_me.sin_family = AF_INET;
366     si_me.sin_port = htons(MY_PORT);
367     si_me.sin_addr.s_addr = htonl(INADDR_ANY);
368
369     //bind socket to port
370     if( bind(s , (struct sockaddr*)&si_me, sizeof(si_me) ) ==
        -1)
371     {
372         die("bind");
373     }
374
375     //keep listening for data
376
377     socklen_t slen_o = sizeof(si_other);
378     int sa = sizeof("Commands:\u1#kitty#Hello#\u-\uSend\umessage\u'
        Hello'\u to\u user\u kitty;\n"
379         "\u2#\u-\u read\u ur\u inbox;\u3#\u-\u exit\n");
380     int k=0;
381
382     char* arg[4];
383     char* arg_tmp[4];
384     //char user_name[10]="nnm";
385     char* user_name;
386     //char* send_to;
387     //char* mess_to;
388     char buf_tmp[BUFLen];
389     //char buf_tmp2[BUFLen];

```

```

390 //char buf_tmp3[BUFLLEN];
391 char users[5][20];
392 char adrs[5][30];
393
394 char** teem;
395
396 int arr[2] = {0,1};
397
398 //number = 0;
399
400
401 char **qq;
402
403 while(1)
404 {
405     bzero(buf, sizeof(buf));
406     //printf("Waiting for data...");
407     fflush(stdout);
408
409     //try to receive some data, this is a blocking call
410     if ((recv_len = recvfrom(s, buf, BUFLLEN, 0, (struct
        sockaddr *) &si_other, &slen_o)) == -1)
411     {
412         die("recvfrom()");
413     }
414
415
416
417     //now reply the client with the same data
418     if (sendto(s, buf, recv_len, 0, (struct sockaddr*) &
        si_other, slen_o) == -1)
419     {
420         die("sendto()");
421     }
422
423     //if(nm < 5){
424     //user_addr[nm] =
425     //}
426
427
428     printf("Received␣packet␣from␣%s:%d\n", inet_ntoa(
        si_other.sin_addr), ntohs(si_other.sin_port));
429
430     strcpy(buf_tmp, buf);
431     int buu = 0;
432     buu = get_cmd(buf, 0);
433     //printf("b = %d", buu);
434
435     if(strcmp(buf, "Hello") == 0){

```

```

436         if (sendto(s, "Enter ur name ending with '#'\n",
437             sizeof("Enter ur name ending with '#'\n"),
438             0 , (struct sockaddr *) &si_other, slen_o)
439                 == -1)
440             {
441                 die("sendto()");
442             }
443
444             //receive a reply and print it
445             //clear the buffer by filling null, it
446             //might have previously received data
447             //memset(buf, '\0', BUFLen);
448             //try to receive some data, this is a
449             //blocking call
450             if (recvfrom(s, "Enter ur name ending
451                 with '#'\n", sizeof("Enter ur name
452                 ending with '#'\n"),
453                 0, (struct sockaddr *) &si_other, &
454                     slen_o) == -1)
455             {
456                 //puts("qq");
457                 //die("recvfrom()");
458             }
459             if ((recv_len = recvfrom(s, buf, BUFLen,
460                 0, (struct sockaddr *) &si_other, &
461                     slen_o)) == -1)
462             {
463                 die("recvfrom()");
464             }
465
466             //print details of the client/
467             //peer and the data received
468             //printf("Received packet from %s
469                 :%d\n", inet_ntoa(si_other.
470                     sin_addr), ntohs(si_other.
471                     sin_port));
472             //printf("Data: %s\n" , buf);
473
474             //now reply the client with the
475             //same data
476             if (sendto(s, buf, recv_len, 0, (
477                 struct sockaddr*) &si_other,
478                 slen_o) == -1)
479             {
480                 die("sendto()");
481             }

```

```

468         //bzero(user_name, sizeof(
469             user_name));
470
471         //teem = splito_array(buf);
472         //printf("us: %s", teem[0]);
473         //strcpy(user_name, teem[0]);
474         //get_cmd_ch(buf, 0, user_name);
475
476         strcpy(user_name, buf);
477         //strcpy(user_names[arr[0]], buf)
478         ;
479         user_name[strlen(user_name)-1]=0;
480         //user_names[arr[0]][strlen(
481             user_names[arr[0]])-1]=0;
482
483         strcpy(users[arr[0]], buf);
484
485         squeeze(users[arr[0]], '#');
486
487         strcpy(adrs[arr[0]], inet_ntoa(
488             si_other.sin_addr));
489
490         user_port[arr[0]] = ntohs(
491             si_other.sin_port);
492
493         arr[0]++;
494
495         printf("User:␣%s␣\n", user_name);
496         bzero(buf, sizeof(buf));
497     }
498     else if(strcmp(buf, "Start") == 0){
499         if (sendto(s, "Commands:␣1#kitty#Hello#␣-␣Send␣
500             message␣'Hello'␣to␣user␣kitty;\n"
501                 "2#␣-␣read␣ur␣inbox;␣3#␣-␣exit\n",
502                     sa, 0, (struct sockaddr *)&
503                         si_other, slen_o)==-1)
504         {
505             die("sendto()");
506         }
507
508         //receive a reply and print it
509         //clear the buffer by filling
510         null, it might have
511         previously received data
512         //memset(buf, '\0', BUFLen);

```



```

506                                     //try to receive some data, this
507                                     is a blocking call
508                                     if (recvfrom(s, "Commands:␣1#
509                                         kitty#Hello#␣-␣Send␣message␣'
510                                         Hello'␣to␣user␣kitty;\n"
511                                         "2#␣-␣read␣ur␣inbox;␣3#␣-␣exit\
512                                         n", sa, 0, (struct sockaddr
513                                             *) &si_other, &slen_o) ==
514                                             -1)
515                                     {
516                                         //puts("qq");
517                                         //die("recvfrom()");
518                                     }
519                                     bzero(buf, sizeof(buf));
520
521     }
522     else if(buu < 4){
523
524         puts("OBRABOTKA\n");
525
526         printf("uss:␣%s",users[0]);
527
528         qq = splito_array(buf_tmp);
529
530         if(buu!=0){
531             if (buu==1){
532                 //printf("name: %s, to: %s, mes: %s\n",
533                     user_name, qq[1], qq[2]);
534                 for(k=0;k<5;k++)
535                     if(user_port[k] == ntohs(si_other.sin_port)
536                         ){
537                         printf("k=%d␣p:␣%d,␣us:␣%s\n", k,
538                             user_port[k], users[k]);
539                         //do_send(user_names[k], qq[1], qq[2]);
540                         printf("from=%s␣to:␣%s,␣mes:␣%s\n",
541                             users[k], qq[1], qq[2]);
542                         do_send(users[k], qq[1], qq[2]);
543                     }
544
545                 //for(k=0;k<5;k++)
546                     //if(strcmp(user_port[k], ntohs(si_other.
547                         sin_port)) == 0)
548                         //strcpy(user_name, user_names[k]);
549
550                 //do_send(user_name, qq[1], qq[2]);
551
552                 //puts("com 1");
553             }
554             else if (buu==2){

```

```

544         for(k=0;k<5;k++)
545             if(user_port[k] == ntohs(si_other.sin_port)
546                 ){
547                 printf("Client_name:_%s\n", user_name);
548                 //do_read(s, buf, user_name, si_other,
549                     slen);
550                 do_read(s, buf, users[k], si_other, slen
551                     );
552             }
553             //puts("com 2");
554         }
555         else if (buu==3){
556             /*if(strcmp(user,"root") == 0){
557                 do_exit(user);
558                 exit(1);
559             }
560             else
561                 do_exit(user);*/
562             //puts("com 3");
563         }
564         else {
565             puts("Wrong_command!\n");
566         }
567     }
568     //printf("Data: %s\n" , buf);
569     bzero(buf, sizeof(buf));
570 }
571 bzero(buf, sizeof(buf));
572 }
573 close(s);
574 return 0;
575 }

```

5.2.6 Сервер UDP Linux. Файл сборки Makefile

```

1 #it's my makefile
2 all:
3     #gcc -I/home/user/Downloads/libxml2-2.7.8/include main.c -
4         Wall -g -O0 -L/home/user/Downloads/libxml2-2.7.8 -lxml2
5         -o server1
6     gcc -g -O0 -Wall -o ./Debug/Server3 main.c -lxml2 -I/home
7         /user/Downloads/libxml2-2.7.8/include
8
9 clean:

```

```
7   rm ./Debug/Server3
8   rm *.o
```

5.2.7 Клиент UDP Linux. Основной файл программы main.c

```
1  #include<stdio.h> //printf
2  #include<string.h> //memset
3  #include<stdlib.h> //exit(0);
4  #include<arpa/inet.h>
5  #include<sys/socket.h>
6  #include <unistd.h>
7  #include <assert.h>
8
9  #define SERVER "127.0.0.1"
10 #define BUFLen 512 //Max length of buffer
11 #define PORT 5001 //The port on which to send data
12
13 void die(char *s)
14 {
15     perror(s);
16     exit(1);
17 }
18
19 char** str_split(char* a_str, const char a_delim)
20 {
21     char** result    = 0;
22     size_t count     = 0;
23     char* tmp        = a_str;
24     char* last_comma = 0;
25     char delim[2];
26     delim[0] = a_delim;
27     delim[1] = 0;
28
29     /* Count how many elements will be extracted. */
30     while (*tmp)
31     {
32         if (a_delim == *tmp)
33         {
34             count++;
35             last_comma = tmp;
36         }
37         tmp++;
38     }
39
40     /* Add space for trailing token. */
41     count += last_comma < (a_str + strlen(a_str) - 1);
```

```

42
43     /* Add space for terminating null string so caller
44     knows where the list of returned strings ends. */
45     count++;
46
47     result = malloc(sizeof(char*) * count);
48
49     if (result)
50     {
51         size_t idx = 0;
52         char* token = strtok(a_str, delim);
53
54         while (token)
55         {
56             assert(idx < count);
57             *(result + idx++) = strdup(token);
58             token = strtok(0, delim);
59         }
60         assert(idx == count - 1);
61         *(result + idx) = 0;
62     }
63
64     return result;
65 }
66
67 char** splito_array(char* str){
68     //char    str[] = "ls -l";
69     char ** res = NULL;
70     char * p = strtok (str, "#");
71     int n_spaces = 0, i;
72
73
74     /* split string and append tokens to 'res' */
75
76     while (p) {
77         res = realloc (res, sizeof (char*) * ++n_spaces);
78
79         if (res == NULL)
80             exit (-1); /* memory allocation failed */
81
82         res[n_spaces-1] = p;
83
84         p = strtok (NULL, "#");
85     }
86
87     /* realloc one extra element for the last NULL */
88
89     res = realloc (res, sizeof (char*) * (n_spaces+1));
90     res[n_spaces] = 0;

```

```

91
92     /* print the result */
93
94     //for (i = 0; i < (n_spaces+1); ++i)
95         //printf ("res[%d] = %s\n", i, res[i]);
96
97     /* free the memory allocated */
98
99     free (res);
100     return res;
101 }
102
103 int get_cmd(char* buffer, int num){
104     int cmd;
105     char **temp;
106     char* arr1;
107     char* arr2;
108     temp=str_split(buffer, '#');
109
110     if(temp){
111         int i;
112         for (i = 0; *(temp + i); i++)
113         {
114             //printf("inp :%s\n", *(temp + i));
115             if(i == num)
116                 cmd = atoi(*(temp + i));
117             else if(i == 1)
118                 //memcpy(arr1, (*(temp + i)), strlen(*(
119                     temp + i))+1);
120                 //strcpy(arr1, (*(temp + i)));
121                 //else if(i == 2)
122                 //strcpy(arr2, (*(temp + i)));
123                 free(*(temp + i));
124             }
125             //printf("\n");
126             free(temp);
127
128             //printf("After: %s\n", arr1);
129             //printf("After: %s\n", arr2);
130             return cmd;
131 }
132
133 void get_cmd_ch(char* buffer, int num, char* res){
134     char* cmd;
135     char **temp;
136     temp=str_split(buffer, '#');
137
138     if(temp){
139         int i;

```

```

139         for (i = 0; *(temp + i); i++)
140         {
141             printf("inp_ch_:%s\n", *(temp + i));
142             //printf("i= %d inp :%s\n", i, teeemp);
143
144             //cmd = (*(temp + i));
145             //cmd = teeemp;
146             //res = teeemp;
147             //strcpy(res, cmd);
148             if(i == num)
149                 strcpy(res, *(temp + i));
150             free(*(temp + i));
151         }
152         //printf("\n");
153         free(temp);
154     }
155     //return cmd;
156 }
157
158 int use_token(char* str, int num, char* res){
159     const char s[2] = "#";
160     char *token;
161     int i = 0;
162
163     token = strtok(str, s);
164     if(num==0){
165         res = token;
166         return 0;
167     }
168     while(token != NULL){
169         //printf("i:%d, t:%s", i, token);
170         i++;
171         token = strtok(NULL, s);
172         if(num==i){
173             res = token;
174             return 0;
175         }
176     }
177     return 0;
178 }
179
180 void get_args(char* inp_str, char* arg[]){//buffer and array
    to write
181     int i = 0;
182     const char s[2] = "#";
183     char *token;
184     token = strtok(inp_str, s);
185     arg[i] = token;
186     while(token != NULL){

```

```

187         i++;
188         token = strtok(NULL, s);
189         arg[i] = token;
190     }
191     i--;
192     arg[i] = NULL;
193 }
194
195 int main(int argc, char *argv[])
196 {
197
198     int com,p;
199
200     //printf("serv: %s port: %s", argv[1], argv[2]);
201     struct sockaddr_in si_other;
202     int s, i, slen=sizeof(si_other);
203     char buf[BUFLEN];
204     char message[BUFLEN];
205
206     if ( (s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1)
207     {
208         die("socket");
209     }
210
211     memset((char *) &si_other, 0, sizeof(si_other));
212     si_other.sin_family = AF_INET;
213     si_other.sin_port = htons(PORT);
214
215     if (inet_aton(argv[1] , &si_other.sin_addr) == 0)
216     {
217         fprintf(stderr, "inet_aton() failed\n");
218         exit(1);
219     }
220     int recv_len;
221     /*
222     int recv_len;
223     if ((recv_len = recvfrom(s, buf, BUFLen, 0, (struct
224         sockaddr *) &si_other, &slen)) == -1)
225     {
226         die("recvfrom()");
227     }
228
229     //print details of the client/peer and the data
230     received
231     //printf("Received packet from %s:%d\n",
232         inet_ntoa(si_other.sin_addr), ntohs(si_other.
233         sin_port));
234     printf("From serv: %s\n" , buf);

```

```

232         //now reply the client with the same data
233         if (sendto(s, buf, recv_len, 0, (struct sockaddr
                *) &si_other, slen) == -1)
234         {
235             die("sendto()");
236         }
237         bzero(buf, sizeof(buf));
238     */
239
240     socklen_t slen_o = sizeof(si_other);
241
242     strcpy(message, "Hello");
243
244     if (sendto(s, message, strlen(message), 0, (struct
        sockaddr *) &si_other, slen_o) == -1)
245     {
246         die("sendto()");
247     }
248
249     //receive a reply and print it
250     //clear the buffer by filling null, it might
        have previously received data
251     memset(buf, '\0', BUFLen);
252     //try to receive some data, this is a
        blocking call
253     if (recvfrom(s, buf, BUFLen, 0, (struct
        sockaddr *) &si_other, &slen_o) == -1)
254     {
255         die("recvfrom()");
256     }
257
258     puts(buf);
259
260     if ((recv_len = recvfrom(s, buf, BUFLen, 0, (
        struct sockaddr *) &si_other, &slen_o)) ==
        -1)
261     {
262         die("recvfrom()");
263     }
264
265     //print details of the client/peer
        and the data received
266     //printf("Received packet from %s:%d\n",
        inet_ntoa(si_other.sin_addr),
        ntohs(si_other.sin_port));
267     printf("%s\n", buf); //from server
268
269     //now reply the client with the same
        data

```



```

270         if (sendto(s, buf, recv_len, 0, (
271             struct sockaddr*) &si_other,
272             slen_o) == -1)
273         {
274             die("sendto()");
275         }
276         bzero(buf, sizeof(buf));
277     bzero(message, sizeof(message));
278     //printf("Enter message : ");
279     gets(message);
280     //send the message
281
282     if (sendto(s, message, strlen(message), 0, (
283         struct sockaddr *) &si_other, slen_o)==-1)
284     {
285         die("sendto()");
286     }
287     //receive a reply and print it
288     //clear the buffer by filling null, it might have
289     //previously received data
290     memset(buf, '\0', BUFLen);
291     //try to receive some data, this is a blocking
292     //call
293     if (recvfrom(s, buf, BUFLen, 0, (struct sockaddr
294         *) &si_other, &slen_o) == -1)
295     {
296         die("recvfrom()");
297     }
298     //puts(buf);
299
300     strcpy(message, "Start");
301
302     if (sendto(s, message, strlen(message), 0, (struct
303         sockaddr *) &si_other, slen_o)==-1)
304     {
305         die("sendto()");
306     }
307     //receive a reply and print it
308     //clear the buffer by filling null, it might have
309     //previously received data
310     memset(buf, '\0', BUFLen);
311     //try to receive some data, this is a blocking
312     //call

```

```

310         if (recvfrom(s, buf, BUFLen, 0, (struct sockaddr
311             *) &si_other, &slen_o) == -1)
312         {
313             die("recvfrom()");
314         }
315         puts(buf);
316
317         if ((recv_len = recvfrom(s, buf, BUFLen, 0, (
318             struct sockaddr *) &si_other, &slen_o)) == -1)
319         {
320             die("recvfrom()");
321         }
322         //print details of the client/peer and
323         //the data received
324         //printf("Received packet from %s:%d\n",
325             inet_ntoa(si_other.sin_addr), ntohs(
326             si_other.sin_port));
327         printf("%s\n" , buf);
328
329         //now reply the client with the same data
330         if (sendto(s, buf, recv_len, 0, (struct
331             sockaddr*) &si_other, slen_o) == -1)
332         {
333             die("sendto()");
334         }
335         bzero(buf, sizeof(buf));
336         char* arg[4];
337         char** arg_spli;
338         char** qq;
339         char* arg1;
340         char* arg2;
341
342         while(1)
343         {
344             //printf("Enter message : ");
345             gets(message);
346
347             //send the message
348
349             if (sendto(s, message, strlen(message) , 0 , (struct
350                 sockaddr *) &si_other, slen_o)==-1)
351             {
352                 die("sendto()");
353             }
354
355             //receive a reply and print it

```

```

352         //clear the buffer by filling null, it might have
           previously received data
353     memset(buf, '\0', BUFLen);
354     //try to receive some data, this is a blocking call
355     if (recvfrom(s, buf, BUFLen, 0, (struct sockaddr *) &
           si_other, &slen_o) == -1)
356     {
357         die("recvfrom()");
358     }
359
360     //puts(buf);
361
362     /*
363     qq=str_split(buf, '#');
364
365     if(qq){
366         int i;
367         for (i = 0; *(qq + i); i++)
368         {
369             //printf("inp :%s\n", *(qq + i));
370
371             if(i == 0)
372                 com = atoi(*(qq + i));
373             free(*(qq + i));
374         }
375         //printf("\n");
376         free(qq);
377     }*/
378
379     int spli;
380     arg_spli = splito_array(buf);
381     //for(spli=0;spli<sizeof(arg_spli);spli++)
382     //printf("aa: %s\n", arg_spli[spli]);
383     com = get_cmd(buf, 0);
384     //get_cmd_ch(buf, 1, arg1);
385     //get_cmd_ch(buf, 2, arg2);
386     //use_token(buf, 1, arg1);
387     //use_token(buf, 2, arg2);
388     //printf("test arg1: %s\n", arg1);
389     //printf("test arg2: %s\n", arg2);
390     //printf("com: %d", com);
391
392     //get_args(buf, arg);
393     //for(p = 0; p<4;p++)
394     //printf("i = %d inp: %s\n", i, arg[p]);
395     //puts(arg[0]);
396     //if(arg[0]!=NULL){
397     //    com = atoi(arg[0]);
398     //}

```

```

399     if(com == 1){
400         //puts("CL: coma 1");
401     }
402     else if(com == 2){
403         while(strcmp(buf, "allmsgs") != 0){
404             if ((recv_len = recvfrom(s, buf, BUFLen, 0, (struct
                sockaddr *) &si_other, &slen_o)) == -1)
405                 {
406                     die("recvfrom()");
407                 }
408
409                 //print details of the client/
                peer and the data received
410             //printf("Received packet from %
                s:%d\n", inet_ntoa(si_other.
                sin_addr), ntohs(si_other.
                sin_port));
411             printf("%s\n", buf);
412
413             //now reply the client with the
                same data
414             if (sendto(s, buf, recv_len, 0,
                (struct sockaddr*) &si_other,
                slen_o) == -1)
415                 {
416                     die("sendto()");
417                 }
418
419         }
420
421         /*if ((recv_len = recvfrom(s, buf,
                BUFLen, 0, (struct sockaddr *)
                &si_other, &slen_o)) == -1)
422             {
423                 die("
                recvfrom
                ()");
424             }
425
426             //print
                details of
                the client/
                peer and
                the data
                received
                //printf("
                Received
                packet from
                %s:%d\n",
                inet_ntoa(

```

427 428 429 430 431 432 433 434 435 436 437 438 439	<pre> si_other. sin_addr), ntohs(si_other. sin_port)); printf("%s\n" , buf); //now reply the client with the same data if (sendto(s, buf, recv_len, 0, (struct sockaddr*) &si_other, slen_o) == -1) { die(" sendto (")"); }*/ /*if ((recv_len = recvfrom(s, buf, BUFLen, 0, (struct sockaddr *) & si_other, & slen_o)) == -1) </pre>
---	--

440

441

442
443

444

445

446

447

```
448                                     //puts(buf);
449                                     bzero(buf, sizeof(buf));
450
451                                     //puts("CL: coma 2");
452
453     }
454     else if(com == 3){
455         puts("Bye!\n");
456         exit(1);
457     }
458     else
459         puts("Wrong input\n");
460
461     /*if(strcmp(message, "Start") == 0){
462         //
463         if ((recv_len = recvfrom(s, buf, BUFLen, 0, (
464             struct sockaddr *) &si_other, &slen_o)) == -1)
465             {
466                 die("recvfrom()");
467             }
```



```

467
468         //print details of the client/peer and
         the data received
469         //printf("Received packet from %s:%d\n",
         inet_ntoa(si_other.sin_addr), ntohs(
         si_other.sin_port));
470         printf("From serv: %s\n" , buf);
471
472         //now reply the client with the same data
473         if (sendto(s, buf, recv_len, 0, (struct
         sockaddr*) &si_other, slen_o) == -1)
474         {
475             die("sendto()");
476         }
477         bzero(buf, sizeof(buf));
478     }*/
479     bzero(message, sizeof(message));
480 }
481
482     close(s);
483     return 0;
484 }

```

5.2.8 Клиент UDP Linux. Файл сборки Makefile

```

1 #it's my makefile
2 all:
3     #gcc -I/home/user/Downloads/libxml2-2.7.8/include main.c -
        Wall -g -O0 -L/home/user/Downloads/libxml2-2.7.8 -lxml2
        -o server1
4     #gcc -g -o0 -Wall -o ./Debug/Client1 main.c -lxml2 -I/
        home/user/Downloads/libxml2-2.7.8/include
5     gcc -g -o0 -Wall -o ./Client3 main.c -lxml2 -I/home/user/
        Downloads/libxml2-2.7.8/include
6
7 clean:
8     rm ./Client3
9     rm *.o

```

5.2.9 Сервер TCP Windiws. Основной файл программы main.c

```

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

```

```

4 #include <sys/types.h>
5 #include <assert.h>
6 #include <winsock2.h>
7 #include <ws2tcpip.h>
8 #include <string.h>
9
10 #include <libxml\parser.h>
11 #include <libxml\tree.h>
12 #include <libxml\xmlmemory.h>
13 #include <libxml\xpath.h>
14
15
16 #pragma comment (lib, "Ws2_32.lib")
17 #pragma comment (lib, "Mswsock.lib")
18 #pragma comment (lib, "AdvApi32.lib")
19 #pragma comment (lib, "libxml2.lib")
20
21 #define MY_ENCODING "ISO-8859-1"
22 #define MY_PORT 5001
23 #define MAXUSERS 5
24
25 DWORD dwThreadId[MAXUSERS];
26
27 #define bzero(b,len) (memset((b), '\0', (len)), (void) 0)
28
29 /*void add_mes(xmlNode* node, char* from, char* msg){
30     xmlNode* cur_node = NULL;
31     char buf[256];
32     bzero(buf, 256);
33     for (cur_node = node; cur_node; cur_node = cur_node->next)
34     {
35         if (cur_node->type == XML_ELEMENT_NODE) {
36             if ((!xmlStrcmp(cur_node->name, (const xmlChar *)"
inbox"))){
37                 xmlNodePtr nNode = xmlNewNode(0, (const xmlChar
*)"mes");
38                 xmlSetProp(nNode, (const xmlChar *)"from", (const
xmlChar *)from);
39                 xmlSetProp(nNode, (const xmlChar *)"text", (const
xmlChar *)msg);
40                 //xmlNodeSetContent(nNode, (xmlChar*)msg);
41                 xmlAddChild(cur_node, nNode);
42                 return;
43             }
44         }
45         add_mes(cur_node->children, from, msg);
46     }
47 }
48

```

```

49 void create_file(const char *to, char *from, char *msg){
50     //int rc;
51     //xmlTextWriterPtr writer;
52     xmlDocPtr doc;
53     xmlNodePtr node, root;
54     //const char *UserFilename = "users.xml"
55     //xmlChar *tmp;
56     if (doc = xmlReadFile(to, NULL, 0)){
57         root = xmlDocGetRootElement(doc);
58         add_mes(root, from, msg);
59         xmlSaveFile(to, doc);
60         xmlFreeDoc(doc);
61     }
62     else{
63         doc = xmlNewDoc(BAD_CAST XML_DEFAULT_VERSION);
64         node = xmlNewDocNode(doc, NULL, BAD_CAST "inbox", NULL)
65         ;
66         xmlDocSetRootElement(doc, node);
67         add_mes(node, from, msg);
68         xmlSaveFile(to, doc);
69         xmlFreeDoc(doc);
70     }
71     xmlCleanupParser();
72 }*/
73
74 /*void main(){
75     //create_file("name", "serg", "hello");
76 }*/
77
78 void print_msgs(char* buffer, int newsockfd, char* user);
79 void create_file(char *to, char *from, char *msg);
80 void start_work(int newsockfd);
81
82 /*#include <stdlib.h>
83 #include <stdio.h>
84 #include <sys/types.h>
85 #include <sys/socket.h>
86 #include <netinet/in.h>
87
88 #include <libxml/parser.h>
89 #include <libxml/tree.h>
90 //#include <libxml/xmlwriter.h>
91 #include <libxml/xmlmemory.h>
92 #include <libxml/xpath.h>
93
94 #include <sys/stat.h>
95 #include <unistd.h>
96 #include <string.h>

```

```

97 #include <pthread.h>
98
99 #define MY_ENCODING "ISO-8859-1"
100 #define MY_PORT 5001
101 */
102 void get_args(char* inp_str, char* arg[]){//buffer and array
103     to write
104     int i = 0, j;
105     const char s[2] = "#";
106     char *token;
107     token = strtok(inp_str, s);
108     arg[i] = token;
109     while (token != NULL){
110         i++;
111         token = strtok(NULL, s);
112         arg[i] = token;
113     }
114     i--;
115     arg[i] = NULL;
116 }
117
118 char *get_mes_from_client(int newsockfd, char *buffer){
119     int n;
120     bzero(buffer, 256);
121     n = read(newsockfd, buffer, 255);
122     if (n < 0)
123     {
124         perror("ERROR_␣reading_␣from_␣socket");
125         exit(1);
126     }
127     return buffer;
128 }
129
130 char* get_hello(int newsockfd, char *buffer, char* arg[]){
131     int i, j = 0;
132     char user[10] = "no_␣user";
133     if (arg[0] != NULL)
134         strcpy(user, arg[0]);
135     else
136         puts("No_␣username!");
137     /*if((arg[0]!=NULL)&&(arg[1]!=NULL)){
138     strcpy(user, arg[0]);
139     char command[7];
140
141     strcpy(command, arg[1]);
142     printf("com=%s\n", arg[1]);
143
144     j = atoi(command);
145     printf("j=%d\n", j);

```

```

145     if (j==1){
146         do_send(user, arg[2], arg[3]);
147     }
148     else if (j==2){
149         do_read(newsockfd, buffer, user);
150     }
151     else if (j==3){
152         do_exit();
153     }
154     else {
155         puts("Wrong command!\n");
156     }
157 }*/
158 printf("user:_%s\n", user);
159 return user;
160 }
161
162 void do_send(char* from, char* to, char* mes){
163     printf("Send_command\n");
164     create_file(to, from, mes);
165 }
166
167 void do_read(int newsockfd, char *buffer, char* client){
168     printf("Read_command\n");
169     print_mesgs(buffer, newsockfd, client);
170 }
171
172 void do_exit(char* user){
173     printf("Exit_command_from_user:_%s\n", user);
174     //exit(1);
175 }
176
177 void do_serve(int newsockfd, char *buffer, char* arg[], char*
    user){
178     int i, j = 0;
179     if (arg[0] != NULL){
180         char command[7];
181         strcpy(command, arg[0]);
182         j = atoi(command);
183         printf("j=%d\n", j);
184         if (j == 1){
185             do_send(user, arg[1], arg[2]);
186         }
187         else if (j == 2){
188             do_read(newsockfd, buffer, user);
189             //printf("us: %d\n", strcmp(user, "root"));
190         }
191         else if (j == 3){
192             if (strcmp(user, "root") == 0){

```

```

193         do_exit(user);
194         exit(1);
195     }
196     else
197         do_exit(user);
198 }
199 else {
200     puts("Wrong command!\n");
201 }
202 }
203 }
204
205 void do_wait_mes(int newsockfd, char *buffer, char* user){
206     char* inp;
207     char* get_inp[4];
208     while (1){
209         inp = get_mes_from_client(newsockfd, buffer);
210         get_args(inp, get_inp);
211         do_serve(newsockfd, buffer, get_inp, user);
212     }
213 }
214
215 void add_mes(xmlNode* node, char* from, char* msg){
216     xmlNode* cur_node = NULL;
217     char buf[256];
218     bzero(buf, 256);
219     for (cur_node = node; cur_node; cur_node = cur_node->next)
220     {
221         if (cur_node->type == XML_ELEMENT_NODE) {
222             if ((!xmlStrcmp(cur_node->name, (const xmlChar *) "
inbox"))){
223                 xmlNodePtr nNode = xmlNewNode(0, (const xmlChar
*) "mes");
224                 xmlSetProp(nNode, (const xmlChar *) "from", (const
xmlChar *) from);
225                 xmlSetProp(nNode, (const xmlChar *) "text", (const
xmlChar *) msg);
226                 //xmlNodeSetContent(nNode, (xmlChar*)msg);
227                 xmlAddChild(cur_node, nNode);
228                 return;
229             }
230         }
231         add_mes(cur_node->children, from, msg);
232     }
233 }
234
235 void read_mes(xmlNode* node, char* mess, int newsockfd){
236     xmlNode *cur_node = NULL;
237     char buf[256];

```

```

238     bzero(buf, 256);
239     int n;
240     for (cur_node = node; cur_node; cur_node = cur_node->next)
241     {
242         if (cur_node->type == XML_ELEMENT_NODE) {
243             if ((!xmlStrcmp(cur_node->name, (const xmlChar *)"
244                 mes"))))
245             {
246                 strcpy(buf, "from:");
247                 strncat(buf, xmlGetProp(cur_node, "from"), strlen
248                     (xmlGetProp(cur_node, "from")));
249                 strcat(buf, "message:");
250                 strncat(buf, xmlGetProp(cur_node, "text"), strlen
251                     (xmlGetProp(cur_node, "text")));
252                 strcat(buf, "\n");
253                 n = write(newsockfd, buf, strlen(buf));
254                 //n=read(newsockfd, buf, 255);
255             }
256         }
257     }
258     read_mes(cur_node->children, mess, newsockfd);
259 }
260
261 void print_mesgs(char* buffer, int newsockfd, char* user)
262 {
263     xmlDoc          *doc = NULL;
264     xmlNode          *root_element = NULL;
265     int n;
266     doc = xmlReadFile(user, NULL, 0);
267     if (doc == NULL)
268     {
269         printf("error: could not parse file %s\n", user);
270         //strncat(buffer, "Error\n", 6);
271         n = write(newsockfd, "Error\n", 6);
272     }
273     else
274     {
275         root_element = xmlDocGetRootElement(doc);
276         n = write(newsockfd, "Messages:\n", 11);
277         read_mes(root_element, buffer, newsockfd);
278         //n = write(newsockfd, "end", 3);
279         xmlFreeDoc(doc);
280     }
281     xmlCleanupParser();
282 }

```

```

283     //return;
284
285 }
286
287 void create_file(char *to, char *from, char *msg){
288     int rc;
289     //xmlTextWriterPtr writer;
290     xmlDocPtr doc;
291     xmlNodePtr node, root;
292     xmlChar *tmp;
293     if (doc = xmlReadFile(to, NULL, 0)){
294         root = xmlDocGetRootElement(doc);
295         add_mes(root, from, msg);
296         xmlSaveFile(to, doc);
297         xmlFreeDoc(doc);
298     }
299     else{
300         doc = xmlNewDoc(BAD_CAST XML_DEFAULT_VERSION);
301         node = xmlNewDocNode(doc, NULL, BAD_CAST "inbox", NULL)
302         ;
303         xmlDocSetRootElement(doc, node);
304         add_mes(node, from, msg);
305         xmlSaveFile(to, doc);
306         xmlFreeDoc(doc);
307     }
308     xmlCleanupParser();
309 }
310 struct sockParams
311 {
312     int sockfd, newsockfd, port_number, client;
313     struct sockaddr_in serv_addr, cli_addr;
314 };
315 struct userThread
316 {
317     char login[MAXUSERS][100];
318     int count;
319 };
320
321 DWORD WINAPI startThread(LPVOID in)
322 {
323     struct sockParams *sp = (struct sockParams *)in;
324     start_work(sp->newsockfd);
325     return 0;
326 }
327
328 void start_work(int newsockfd){
329     char buffer[256];
330     int n;

```



```

331     n = write(newsockfd, "Please, write ur name ending with
        '#'\n", 38);
332
333     //bzero(buffer, 256);
334     bzero(buffer, 256);
335     n = read(newsockfd, buffer, 255);
336     if (n < 0)
337     {
338         perror("ERROR reading from socket");
339         exit(1);
340     }
341     //int j;
342     char* get_inp[4];
343     get_args(buffer, get_inp);
344     char user_name[10] = "nnm";
345     strcpy(user_name, get_hello(newsockfd, buffer, get_inp));
346     n = write(newsockfd, "Commands: 1#kitty#Hello#-Send
        message 'Hello' to user kitty;\n"
        "2#-read ur inbox; 3#-exit\n", 93);
347     do_wait_mes(newsockfd, buffer, user_name);
348     //return 0;
349 }
350
351
352 struct userThread usersthr;
353
354 DWORD WINAPI workMainTh(LPVOID lpParam)
355 {
356     char buffer[10];
357     int i;
358     while (1)
359     {
360         fgets(buffer, 10 - 1, stdin);
361         for (i = 0; i < MAXUSERS; i++)
362         {
363             if (strcmp(buffer, usersthr.login[i]))
364                 ExitThread(dwThreadId[i]);
365         }
366     }
367     return 0;
368 }
369
370 int main(int argc, char *argv[])
371 {
372     HANDLE thread[MAXUSERS], mainthread;
373     //pthread_t thread[5], mainthread;
374     int i = 0, j = 0;
375     struct sockParams sp;
376
377     WSADATA wsaData;

```

```

378 WSASStartup(MAKEWORD(2, 2), &wsaData);
379 usersthr.count = 0;
380
381 sp.sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
382 if (sp.sockfd < 0)
383 {
384     perror("ERROR opening socket");
385     exit(1);
386 }
387 bzero((char *)&sp.serv_addr, sizeof(sp.serv_addr));
388 sp.port_number = MY_PORT;
389 //portno = 7771;
390 sp.serv_addr.sin_family = AF_INET;
391 sp.serv_addr.sin_addr.s_addr = INADDR_ANY;
392 sp.serv_addr.sin_port = htons(sp.port_number);
393
394 /* Now bind the host address using bind() call.*/
395 if (bind(sp.sockfd, (struct sockaddr *)&sp.serv_addr,
396     sizeof(sp.serv_addr)) < 0)
397 {
398     perror("ERROR on binding");
399     exit(1);
400 }
401
402 /* Now start listening for the clients, here process will
403 * go in sleep mode and will wait for the incoming
404 connection
405 */
406 while (1){
407     listen(sp.sockfd, 5);
408     sp.client = sizeof(sp.cli_addr);
409
410     //printf("cl: %s\n", sp.client);
411     /* Accept actual connection from the client */
412     sp.newsockfd = accept(sp.sockfd, (struct sockaddr *)&sp
413         .cli_addr, &sp.client);
414
415     sp.port_number = i;
416
417     /*if (sp.newsockfd < 0)
418     {
419         perror("ERROR on accept");
420         exit(1);
421     }
422     pthread_create(&thread[i], NULL, startThread, (void*)&
423         sp);
424     i++;
425 }

```

```

424     for (j = 0; j<5; j++)
425         pthread_join(thread[j], NULL);*/
426
427     thread[i] = CreateThread(NULL, 0, startThread, (LPVOID)
        &sp, 0, &dwThreadId[i]);
428     i++;
429 }
430 return 0;
431 }

```

5.2.10 Клиент TCP Windiws. Основной файл программы main.c

```

1  // #include <stdio.h>
2  // #include <sys/types.h>
3  // #include <sys/socket.h>
4  // #include <netinet/in.h>
5
6  #define _CRT_SECURE_NO_WARNINGS
7  #define _WINSOCK_DEPRECATED_NO_WARNINGS
8  #include <time.h>
9
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <winsock2.h>
13 #include <ws2tcpip.h>
14 #include <string.h>
15 #include <assert.h>
16
17 #pragma comment (lib, "Ws2_32.lib")
18 #pragma comment (lib, "Mswsock.lib")
19 #pragma comment (lib, "AdvApi32.lib")
20
21 #define bzero(b,len) (memset((b), '\0', (len)), (void) 0)
22 #define bcopy(b1,b2,len) (memmove((b2), (b1), (len)), (void)
    0)
23
24 void print_mesgs(char* buffer, int sockfd, char* user);
25
26 void write_socket_start(char* buffer, int sockfd){
27     int n;
28     bzero(buffer, 256);
29     fgets(buffer, 255, stdin);
30     //puts(buffer);
31     /* Send message to the server */
32     n = write(sockfd, buffer, strlen(buffer));
33     if (n < 0)

```

```

34     {
35         perror("ERROR_writing_to_socket");
36         exit(1);
37     }
38 }
39
40 void read_response(char* buffer, int sockfd){
41     int n;
42     bzero(buffer, 256);
43     n = read(sockfd, buffer, 255);
44     if (n < 0)
45     {
46         perror("ERROR_reading_from_socket");
47         exit(1);
48     }
49     printf("%s", buffer);
50 }
51 //-----
52 void get_args(char* inp_str, char* arg[]){//buffer and array
53     to write
54     int i = 0, j;
55     const char s[2] = "#";
56     char *token;
57     token = strtok(inp_str, s);
58     arg[i] = token;
59     while (token != NULL){
60         i++;
61         token = strtok(NULL, s);
62         arg[i] = token;
63     }
64     i--;
65     arg[i] = NULL;
66 }
67 char *get_mes_from_client(int newsockfd, char *buffer){
68     int n;
69     bzero(buffer, 256);
70     n = read(newsockfd, buffer, 255);
71     if (n < 0)
72     {
73         perror("ERROR_reading_from_socket");
74         exit(1);
75     }
76     return buffer;
77 }
78
79 void resp_send(char* from, char* to, char* mes){
80 }
81

```

```

82 void resp_read(int newsockfd, char *buffer, char* client){
83     print_mesgs(buffer, newsockfd, client);
84 }
85
86 void resp_exit(){
87     exit(1);
88 }
89
90 void print_mesgs(char* buffer, int sockfd, char* user){
91     read_response(buffer, sockfd);
92     read_response(buffer, sockfd);
93 }
94
95 char* put_hello(int sockfd, char *buffer){
96     write_socket_start(buffer, sockfd);
97     char* arg[4];
98     get_args(buffer, arg);
99     int i, j = 0;
100    char user[10] = "no_user";
101    if ((arg[0] != NULL) && (arg[1] != NULL)){
102        strcpy(user, arg[0]);
103        char command[7];
104        strcpy(command, arg[1]);
105        j = atoi(command);
106        if (j == 1){
107            resp_send(user, arg[2], arg[3]);
108        }
109        else if (j == 2){
110            resp_read(sockfd, buffer, user);
111        }
112        else if (j == 3){
113            resp_exit();
114        }
115        else {
116            puts("Wrong command!\n");
117        }
118    }
119    return user;
120 }
121
122 void keep_talking(int sockfd, char *buffer){
123     write_socket_start(buffer, sockfd);
124     char* arg[4];
125     get_args(buffer, arg);
126     int i, j = 0;
127     char user[10] = "no_user";
128     if ((arg[0] != NULL)){
129         strcpy(user, arg[0]);
130         char command[7];

```

```

131     strcpy(command, arg[0]);
132     j = atoi(command);
133     if (j == 1){
134         resp_send(user, arg[2], arg[3]);
135     }
136     else if (j == 2){
137         resp_read(sockfd, buffer, user);
138     }
139     else if (j == 3){
140         resp_exit();
141     }
142     else {
143         puts("Wrong command!\n");
144     }
145 }
146 }
147
148 int main(int argc, char *argv[])
149 {
150     SOCKET sockfd;
151     int portno, n;
152     struct sockaddr_in serv_addr;
153     struct hostent *server;
154
155     char buffer[256];
156
157     WSADATA wsaData;
158
159     WSASStartup(MAKEWORD(2, 2), &wsaData);
160
161     if (argc < 3) {
162         fprintf(stderr, "usage %s hostname port\n", argv[0]);
163         exit(0);
164     }
165     portno = atoi(argv[2]);
166     /* Create a socket point */
167     sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
168     if (sockfd == INVALID_SOCKET)
169     {
170         perror("ERROR opening socket");
171         exit(1);
172     }
173     server = gethostbyname(argv[1]);
174     if (server == NULL) {
175         fprintf(stderr, "ERROR, no such host\n");
176         exit(0);
177     }
178
179     bzero((char *)&serv_addr, sizeof(serv_addr));

```

```

180     serv_addr.sin_family = AF_INET;
181
182     bcopy((char *)server->h_addr,
183         (char *)&serv_addr.sin_addr.s_addr,
184         server->h_length);
185
186     serv_addr.sin_port = htons(portno);
187
188     /* Now connect to the server */
189     //if (connect(sockfd, (sockaddr *)&serv_addr, sizeof(
190         serv_addr))>0)
191     if(connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(
192         serv_addr)) > 0)
193     {
194         perror("ERROR_␣connecting");
195         exit(1);
196     }
197     /* Now ask for a message from the user, this message
198     * will be read by server
199     */
200     read_response(buffer, sockfd);
201     put_hello(sockfd, buffer);
202     read_response(buffer, sockfd);
203     while (1)
204         keep_talking(sockfd, buffer);
205     return 0;
206 }

```

5.2.11 Сервер UDP Windiws. Основной файл программы main.c

```

1  /*#include<stdio.h> //printf
2  #include<string.h> //memset
3  #include<stdlib.h> //exit(0);
4  #include<arpa/inet.h>
5  #include<sys/socket.h>
6  #include <assert.h>
7
8  #include <libxml/parser.h>
9  #include <libxml/tree.h>
10 #include <libxml/xmlwriter.h>
11 #include <libxml/xmlmemory.h>
12 #include <libxml/xpath.h>*/
13
14 #define _WINSOCK_DEPRECATED_NO_WARNINGS
15 #define _CRT_SECURE_NO_DEPRECATE

```

```

16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <winsock2.h>
19 #include <ws2tcpip.h>
20 #include <string.h>
21 #include <assert.h>
22 #include <libxml/parser.h>
23 #include <libxml/tree.h>
24 #include <stdint.h>
25
26 #define BUFLen 1024 //Max length of buffer
27 #define MY_PORT 5001 //The port on which to listen for
    incoming data
28
29 #define bzero(b,len) (memset((b), '\0', (len)), (void) 0)
30
31 #pragma comment (lib, "Ws2_32.lib")
32 #pragma comment (lib, "Mswsock.lib")
33 #pragma comment (lib, "AdvApi32.lib")
34 #pragma comment (lib, "libxml2.lib")
35
36 void print_msgs(char* buffer, int s, char* user, struct
    sockaddr_in si_other, int slen);
37 void create_file(char *to, char *from, char *msg);
38
39 void die(char *s)
40 {
41     perror(s);
42     exit(1);
43 }
44
45 char** splito_array(char* str){
46     //char    str[] = "ls -l";
47     char ** res = NULL;
48     char * p = strtok(str, "#");
49     int n_spaces = 0, i;
50
51     /* split string and append tokens to 'res' */
52
53     while (p) {
54         res = realloc(res, sizeof(char*) * ++n_spaces);
55
56         if (res == NULL)
57             exit(-1); /* memory allocation failed */
58
59         res[n_spaces - 1] = p;
60
61         p = strtok(NULL, "#");
62

```



```

63     }
64
65     /* realloc one extra element for the last NULL */
66
67     res = realloc(res, sizeof(char*) * (n_spaces + 1));
68     res[n_spaces] = 0;
69
70     /* free the memory allocated */
71
72     free(res);
73     return res;
74 }
75
76 void get_args(char* inp_str, char* arg[]){//buffer and array
    to write
77     int i = 0, j;
78     const char s[2] = "#";
79     char *token;
80     token = strtok(inp_str, s);
81     arg[i] = token;
82     while (token != NULL){
83         i++;
84         token = strtok(NULL, s);
85         arg[i] = token;
86     }
87     i--;
88     arg[i] = NULL;
89 }
90
91 char** str_split(char* a_str, const char a_delim)
92 {
93     char** result = 0;
94     size_t count = 0;
95     char* tmp = a_str;
96     char* last_comma = 0;
97     char delim[2];
98     delim[0] = a_delim;
99     delim[1] = 0;
100
101     /* Count how many elements will be extracted. */
102     while (*tmp)
103     {
104         if (a_delim == *tmp)
105         {
106             count++;
107             last_comma = tmp;
108         }
109         tmp++;
110     }

```

```

111
112 /* Add space for trailing token. */
113 count += last_comma < (a_str + strlen(a_str) - 1);
114
115 /* Add space for terminating null string so caller
116 knows where the list of returned strings ends. */
117 count++;
118
119 result = malloc(sizeof(char*) * count);
120
121 if (result)
122 {
123     size_t idx = 0;
124     char* token = strtok(a_str, delim);
125
126     while (token)
127     {
128         assert(idx < count);
129         *(result + idx++) = _strdup(token);
130         token = strtok(0, delim);
131     }
132     assert(idx == count - 1);
133     *(result + idx) = 0;
134 }
135
136 return result;
137 }
138
139 void do_send(char* from, char* to, char* mes){
140     printf("Send␣command\n");
141     create_file(to, from, mes);
142 }
143
144 void do_read(int newsockfd, char *buffer, char* client,
145             struct sockaddr_in si_other, int slen){
146     printf("Read␣command\n");
147     print_msgs(buffer, newsockfd, client, si_other, slen);
148 }
149 void read_mes(xmlNode* node, char* mess, int s, struct
150             sockaddr_in si_other, int slen){
151     xmlNode *cur_node = NULL;
152     char buf[BUFLen];
153     bzero(buf, BUFLen);
154     int n;
155     for (cur_node = node; cur_node; cur_node = cur_node->next)
156     {
157         if (cur_node->type == XML_ELEMENT_NODE) {

```

```

156         if ((!xmlStrcmp(cur_node->name, (const xmlChar *)"
157             mes"))
158         {
159             strcpy(buf, "from:");
160             strncat(buf, xmlGetProp(cur_node, "from"), strlen
161                 (xmlGetProp(cur_node, "from")));
162             strcat(buf, "message:");
163             strncat(buf, xmlGetProp(cur_node, "text"), strlen
164                 (xmlGetProp(cur_node, "text")));
165             strcat(buf, "\n");
166
167             if (sendto(s, buf, sizeof(buf),
168                 0, (struct sockaddr *) &si_other, slen) == -1)
169             {
170                 die("sendto()");
171             }
172
173             //receive a reply and print it
174             //clear the buffer by filling null, it might have
175             //previously received data
176             //memset(buf, '\0', BUFLen);
177             //try to receive some data, this is a blocking
178             //call
179             if (recvfrom(s, buf, sizeof(buf),
180                 0, (struct sockaddr *) &si_other, &slenn) ==
181                 -1)
182             {
183                 //puts("qq");
184                 //die("recvfrom()");
185             }
186         }
187     }
188
189     read_mes(cur_node->children, mess, s, si_other, slen);
190 }
191
192 void print_msgs(char* buffer, int s, char* user, struct
193     sockaddr_in si_other, int slen)
194 {
195     xmlDoc          *doc = NULL;
196     xmlNode          *root_element = NULL;
197     int n;
198     doc = xmlReadFile(user, NULL, 0);
199     if (doc == NULL)
200     {
201         printf("error: could not parse file %s\n", user);
202     }

```

```

198     }
199     else
200     {
201
202         root_element = xmlDocGetRootElement(doc);
203
204         if (sendto(s, "Messages:\n", sizeof("Messages:\n"),
205             0, (struct sockaddr *) &si_other, slen) == -1)
206         {
207             die("sendto()");
208         }
209
210         //receive a reply and print it
211         //clear the buffer by filling null, it might have
212         previously received data
213         //memset(buf, '\0', BUFLen);
214         //try to receive some data, this is a blocking call
215         if (recvfrom(s, "Messages:\n", sizeof("Messages:\n"),
216             0, (struct sockaddr *) &si_other, &slen) == -1)
217         {
218             //puts("qq");
219             //die("recvfrom()");
220         }
221
222         //n = write(newsockfd, "Messages:\n", 11);
223         read_mes(root_element, buffer, s, si_other, slen);
224         //n = write(newsockfd, "end", 3);
225
226         if (sendto(s, "allmesgs", sizeof("allmesgs"),
227             0, (struct sockaddr *) &si_other, slen) == -1)
228         {
229             die("sendto()");
230         }
231
232         //receive a reply and print it
233         //clear the buffer by filling null, it might have
234         previously received data
235         //memset(buf, '\0', BUFLen);
236
237         if (recvfrom(s, "allmesgs", sizeof("allmesgs"),
238             0, (struct sockaddr *) &si_other, &slen) == -1)
239         {
240             //puts("qq");
241             //die("recvfrom()");
242         }
243
244         xmlFreeDoc(doc);

```

```

245
246     xmlCleanupParser();
247
248     return;
249
250 }
251
252 void add_mes(xmlNode* node, char* from, char* msg){
253     xmlNode* cur_node = NULL;
254     char buf[BUFLen];
255     bzero(buf, BUFLen);
256     for (cur_node = node; cur_node; cur_node = cur_node->next)
257     {
258         if (cur_node->type == XML_ELEMENT_NODE) {
259             if ((!xmlStrcmp(cur_node->name, (const xmlChar *)"
260                 inbox"))){
261                 xmlNodePtr nNode = xmlNewNode(0, (const xmlChar
262                     *)"mes");
263                 xmlSetProp(nNode, (const xmlChar *)"from", (const
264                     xmlChar *)from);
265                 xmlSetProp(nNode, (const xmlChar *)"text", (const
266                     xmlChar *)msg);
267                 //xmlNodeSetContent(nNode, (xmlChar*)msg);
268                 xmlAddChild(cur_node, nNode);
269                 return;
270             }
271         }
272         add_mes(cur_node->children, from, msg);
273     }
274 }
275
276 void create_file(char *to, char *from, char *msg){
277     int rc;
278     //xmlTextWriterPtr writer;
279     xmlDocPtr doc;
280     xmlNodePtr node, root;
281     xmlChar *tmp;
282     if (doc = xmlReadFile(to, NULL, 0)){
283         root = xmlDocGetRootElement(doc);
284         add_mes(root, from, msg);
285         xmlSaveFile(to, doc);
286         xmlFreeDoc(doc);
287     }
288     else{
289         doc = xmlNewDoc(BAD_CAST XML_DEFAULT_VERSION);
290         node = xmlNewDocNode(doc, NULL, BAD_CAST "inbox", NULL)
291             ;
292         xmlDocSetRootElement(doc, node);
293         add_mes(node, from, msg);

```

```

289     xmlSaveFile(to, doc);
290     xmlFreeDoc(doc);
291 }
292 xmlCleanupParser();
293 }
294
295 int get_cmd(char* buffer, int num){
296     int cmd;
297     char **temp;
298     temp = str_split(buffer, '#');
299
300     if (temp){
301         int i;
302         for (i = 0; *(temp + i); i++)
303         {
304             printf("inp_:%s\n", *(temp + i));
305             if (i == num)
306                 cmd = atoi(*(temp + i));
307             free(*(temp + i));
308         }
309         free(temp);
310     }
311     return cmd;
312 }
313
314 /*char* get_cmd_ch(char* buffer, int num, char* res){
315     char* comd;
316     char **temp;
317     char *teeemp;
318     temp = str_split(buffer, '#');
319
320     if (temp){
321         int i;
322         for (i = 0; *(temp + i); i++)
323         {
324
325             if (i == num)
326                 comd = (*(temp + i));
327
328             strcpy(res, (*(temp + i)));
329             free(*(temp + i));
330         }
331         free(temp);
332     }
333     return comd;
334 }*/
335
336 int use_token(char* str, int num, char* res){
337     const char s[2] = "#";

```

```

338     char *token;
339     int i = 0;
340
341     token = strtok(str, s);
342     if (num == 0){
343         res = token;
344         return 0;
345     }
346     while (token != NULL){
347         printf("i:%d, t:%s", i, token);
348         i++;
349         token = strtok(NULL, s);
350         if (num == i){
351             res = token;
352             return 0;
353         }
354     }
355     return 0;
356 }
357
358 void squeeze(char s[], int c) {
359     int i, j;
360
361     for (i = j = 0; s[i] != '\0'; i++)
362         if (s[i] != c)
363             s[j++] = s[i];
364     s[j] = '\0';
365 }
366
367 int main(void)
368 {
369     struct sockaddr_in si_me, si_other;
370
371     int s, i, slen = sizeof(si_other), recv_len;
372     char buf[BUFLen];
373
374     char* user_names[5];
375     char* user_addr[5];
376     //int user_port[5];
377     uint16_t user_port[5];
378
379     //create a UDP socket
380     if ((s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1)
381     {
382         die("socket");
383     }
384
385     // zero out the structure
386     memset((char *)&si_me, 0, sizeof(si_me));

```

```

387
388     si_me.sin_family = AF_INET;
389     si_me.sin_port = htons(MY_PORT);
390     si_me.sin_addr.s_addr = htonl(INADDR_ANY);
391
392     //bind socket to port
393     if (bind(s, (struct sockaddr*)&si_me, sizeof(si_me)) ==
        -1)
394     {
395         die("bind");
396     }
397
398     //keep listening for data
399
400     socklen_t slen_o = sizeof(si_other);
401     int sa = sizeof("Commands: 1#kitty#Hello#  Send_message,
        Hello' to user kitty;\n"
402         "2# read ur inbox; 3#  exit\n");
403     int k = 0;
404
405     char* arg[4];
406     char* arg_tmp[4];
407     //char user_name[10]="nnm";
408
409     //char* user_name;
410
411     //char* send_to;
412     //char* mess_to;
413     char buf_tmp[BUFLen];
414     //char buf_tmp2[BUFLen];
415     //char buf_tmp3[BUFLen];
416     char users[5][20];
417     char adrs[5][30];
418
419     char** teem;
420
421     int arr[2] = { 0, 1 };
422
423     //number = 0;
424
425
426     char **qq;
427
428     while (1)
429     {
430         bzero(buf, sizeof(buf));
431         //printf("Waiting for data...");
432         fflush(stdout);
433

```



```

434 //try to receive some data, this is a blocking call
435 if ((recv_len = recvfrom(s, buf, BUFLen, 0, (struct
      sockaddr *) &si_other, &slen_o)) == -1)
436 {
437     die("recvfrom()");
438 }
439
440
441
442 //now reply the client with the same data
443 if (sendto(s, buf, recv_len, 0, (struct sockaddr*) &
      si_other, slen_o) == -1)
444 {
445     die("sendto()");
446 }
447
448 //if(nm < 5){
449 //user_addr[nm] =
450 //}
451
452
453 printf("Received packet from %s:%d\n", inet_ntoa(
      si_other.sin_addr), ntohs(si_other.sin_port));
454
455 strcpy(buf_tmp, buf);
456 int buu = 0;
457 buu = get_cmd(buf, 0);
458 //printf("b = %d", buu);
459 //char* user_name;
460 char user_name[20];
461 if (strcmp(buf, "Hello") == 0){
462     if (sendto(s, "Enter ur name ending with '#'\n",
        sizeof("Enter ur name ending with '#'\n"),
463         0, (struct sockaddr *) &si_other, slen_o) == -1)
464     {
465         die("sendto()");
466     }
467
468     //receive a reply and print it
469     //clear the buffer by filling null, it might have
        previously received data
470     //memset(buf, '\0', BUFLen);
471     //try to receive some data, this is a blocking call
472     if (recvfrom(s, "Enter ur name ending with '#'\n",
        sizeof("Enter ur name ending with '#'\n"),
473         0, (struct sockaddr *) &si_other, &slen_o) == -1)
474     {
475         //puts("qq");
476         //die("recvfrom()");

```

```

477     }
478     if ((recv_len = recvfrom(s, buf, BUFLen, 0, (struct
        sockaddr *) &si_other, &slen_o)) == -1)
479     {
480         die("recvfrom()");
481     }
482
483     //print details of the client/peer and the data
        received
484     //printf("Received packet from %s:%d\n", inet_ntoa(
        si_other.sin_addr), ntohs(si_other.sin_port));
485     //printf("Data: %s\n" , buf);
486
487     //now reply the client with the same data
488     if (sendto(s, buf, recv_len, 0, (struct sockaddr*) &
        si_other, slen_o) == -1)
489     {
490         die("sendto()");
491     }
492
493
494     //bzero(user_name, sizeof(user_name));
495
496     //teem = splito_array(buf);
497     //printf("us: %s", teem[0]);
498     //strcpy(user_name, teem[0]);
499     //get_cmd_ch(buf, 0, user_name);
500
501     strcpy(user_name, buf);
502     //strcpy(user_names[arr[0]], buf);
503     user_name[strlen(user_name) - 1] = 0;
504     //user_names[arr[0]][strlen(user_names[arr[0]])
        -1]=0;
505
506     strcpy(users[arr[0]], buf);
507
508
509     squeeze(users[arr[0]], '#');
510
511     strcpy(adrs[arr[0]], inet_ntoa(si_other.sin_addr));
512
513     user_port[arr[0]] = ntohs(si_other.sin_port);
514
515     arr[0]++;
516
517
518
519     printf("User:␣%s\n", user_name);
520     bzero(buf, sizeof(buf));

```

```

521     }
522     else if (strcmp(buf, "Start") == 0){
523         if (sendto(s, "Commands:␣1#kitty#Hello#␣␣Send␣
524             message␣'Hello'␣to␣user␣kitty;\n"
525             "2#␣␣read␣ur␣inbox;␣3#␣␣exit\n", sa, 0, (struct
526                 sockaddr *) &si_other, slen_o) == -1)
527         {
528             die("sendto()");
529         }
530         //receive a reply and print it
531         //clear the buffer by filling null, it might have
532         //previously received data
533         //memset(buf, '\0', BUFLen);
534         //try to receive some data, this is a blocking call
535         if (recvfrom(s, "Commands:␣1#kitty#Hello#␣␣Send␣
536             message␣'Hello'␣to␣user␣kitty;\n"
537             "2#␣␣read␣ur␣inbox;␣3#␣␣exit\n", sa, 0, (struct
538                 sockaddr *) &si_other, &slen_o) == -1)
539         {
540             //puts("qq");
541             //die("recvfrom()");
542         }
543         bzero(buf, sizeof(buf));
544     }
545     else if (buu < 4){
546         puts("OBRABOTKA\n");
547         printf("usss:␣%s", users[0]);
548         qq = splito_array(buf_tmp);
549         if (buu != 0){
550             if (buu == 1){
551                 //printf("name: %s, to: %s, mes: %s\n",
552                     user_name, qq[1], qq[2]);
553                 for (k = 0; k < 5; k++)
554                     if (user_port[k] == ntohs(si_other.sin_port
555                         )){
556                         printf("k=%d␣p:␣%d,␣us:␣%s\n", k,
557                             user_port[k], users[k]);
558                         //do_send(user_names[k], qq[1], qq[2]);
559                         printf("from=%s␣to:␣%s,␣mes:␣%s\n",
560                             users[k], qq[1], qq[2]);
561                         do_send(users[k], qq[1], qq[2]);
562                     }

```

```

561         //for(k=0;k<5;k++)
562         //if(strcmp(user_port[k], ntohs(si_other.
           sin_port)) == 0)
563         //strcpy(user_name, user_names[k]);
564
565         //do_send(user_name, qq[1], qq[2]);
566
567         //puts("com 1");
568     }
569     else if (buu == 2){
570         for (k = 0; k<5; k++)
571             if (user_port[k] == ntohs(si_other.sin_port
           )){
572                 printf("Client_name: %s\n", user_name);
573                 //do_read(s, buf, user_name, si_other,
           slen);
574                 do_read(s, buf, users[k], si_other, slen
           );
575             }
576         //puts("com 2");
577     }
578     else if (buu == 3){
579         /*if(strcmp(user,"root") == 0){
580             do_exit(user);
581             exit(1);
582         }
583         else
584             do_exit(user);*/
585         //puts("com 3");
586     }
587     else {
588         puts("Wrong command!\n");
589     }
590 }
591
592
593     //printf("Data: %s\n" , buf);
594     bzero(buf, sizeof(buf));
595 }
596     bzero(buf, sizeof(buf));
597 }
598
599     close(s);
600     return 0;
601 }

```

5.2.12 Клиент UDP Windiws. Основной файл программы main.c

```
1  /*#include<stdio.h> //printf
2  #include<string.h> //memset
3  #include<stdlib.h> //exit(0);
4  #include<arpa/inet.h>
5  #include<sys/socket.h>
6  #include <unistd.h>
7  #include <assert.h>*/
8
9  #define _WINSOCK_DEPRECATED_NO_WARNINGS
10 #define _CRT_SECURE_NO_DEPRECATED
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <winsock2.h>
14 #include <ws2tcpip.h>
15 #include <string.h>
16 #include <assert.h>
17 #include <libxml/parser.h>
18 #include <libxml/tree.h>
19 #include <stdint.h>
20
21
22
23 #define bzero(b,len) (memset((b), '\0', (len)), (void) 0)
24
25 #pragma comment (lib, "Ws2_32.lib")
26 #pragma comment (lib, "Mswsock.lib")
27 #pragma comment (lib, "AdvApi32.lib")
28 #pragma comment (lib, "libxml2.lib")
29
30 #define SERVER "127.0.0.1"
31 #define BUFLen 512 //Max length of buffer
32 #define PORT 5001 //The port on which to send data
33
34 void die(char *s)
35 {
36     perror(s);
37     exit(1);
38 }
39
40 char** str_split(char* a_str, const char a_delim)
41 {
42     char** result = 0;
43     size_t count = 0;
44     char* tmp = a_str;
45     char* last_comma = 0;
```

```

46     char delim[2];
47     delim[0] = a_delim;
48     delim[1] = 0;
49
50     /* Count how many elements will be extracted. */
51     while (*tmp)
52     {
53         if (a_delim == *tmp)
54         {
55             count++;
56             last_comma = tmp;
57         }
58         tmp++;
59     }
60
61     /* Add space for trailing token. */
62     count += last_comma < (a_str + strlen(a_str) - 1);
63
64     /* Add space for terminating null string so caller
65     knows where the list of returned strings ends. */
66     count++;
67
68     result = malloc(sizeof(char*) * count);
69
70     if (result)
71     {
72         size_t idx = 0;
73         char* token = strtok(a_str, delim);
74
75         while (token)
76         {
77             assert(idx < count);
78             *(result + idx++) = _strdup(token);
79             token = strtok(0, delim);
80         }
81         assert(idx == count - 1);
82         *(result + idx) = 0;
83     }
84
85     return result;
86 }
87
88 char** splito_array(char* str){
89     //char    str[] = "ls -l";
90     char ** res = NULL;
91     char * p = strtok(str, "#");
92     int n_spaces = 0, i;
93
94

```

```

95     /* split string and append tokens to 'res' */
96
97     while (p) {
98         res = realloc(res, sizeof(char*) * ++n_spaces);
99
100        if (res == NULL)
101            exit(-1); /* memory allocation failed */
102
103        res[n_spaces - 1] = p;
104
105        p = strtok(NULL, "#");
106    }
107
108    /* realloc one extra element for the last NULL */
109
110    res = realloc(res, sizeof(char*) * (n_spaces + 1));
111    res[n_spaces] = 0;
112
113    /* print the result */
114
115    //for (i = 0; i < (n_spaces+1); ++i)
116    //printf ("res[%d] = %s\n", i, res[i]);
117
118    /* free the memory allocated */
119
120    free(res);
121    return res;
122 }
123
124 int get_cmd(char* buffer, int num){
125     int cmd;
126     char **temp;
127     char* arr1;
128     char* arr2;
129     temp = str_split(buffer, '#');
130
131     if (temp){
132         int i;
133         for (i = 0; *(temp + i); i++)
134         {
135             //printf("inp :%s\n", *(temp + i));
136             if (i == num)
137                 cmd = atoi(*(temp + i));
138             else if (i == 1)
139                 //memcpy(arr1, (*(temp + i)), strlen(*(temp + i))
140                 +1);
141                 //strcpy(arr1, (*(temp + i)));
142                 //else if(i == 2)
143                 //strcpy(arr2, (*(temp + i)));

```

```

143         free(*(temp + i));
144     }
145     //printf("\n");
146     free(temp);
147 }
148 //printf("After: %s\n", arr1);
149 //printf("After: %s\n", arr2);
150 return cmd;
151 }
152
153 void get_cmd_ch(char* buffer, int num, char* res){
154     char* cmd;
155     char **temp;
156     temp = str_split(buffer, '#');
157
158     if (temp){
159         int i;
160         for (i = 0; *(temp + i); i++)
161         {
162             printf("inp_ch_:%s\n", *(temp + i));
163             //printf("i= %d inp :%s\n", i, teeemp);
164
165             //cmd = (*(temp + i));
166             //cmd = teeemp;
167             //res = teeemp;
168             //strcpy(res, cmd);
169             if (i == num)
170                 strcpy(res, *(temp + i));
171             free(*(temp + i));
172         }
173         //printf("\n");
174         free(temp);
175     }
176     //return cmd;
177 }
178
179 int use_token(char* str, int num, char* res){
180     const char s[2] = "#";
181     char *token;
182     int i = 0;
183
184     token = strtok(str, s);
185     if (num == 0){
186         res = token;
187         return 0;
188     }
189     while (token != NULL){
190         //printf("i:%d, t:%s", i, token);
191         i++;

```



```

192     token = strtok(NULL, s);
193     if (num == i){
194         res = token;
195         return 0;
196     }
197 }
198 return 0;
199 }
200
201 void get_args(char* inp_str, char* arg[]){//buffer and array
    to write
202     int i = 0;
203     const char s[2] = "#";
204     char *token;
205     token = strtok(inp_str, s);
206     arg[i] = token;
207     while (token != NULL){
208         i++;
209         token = strtok(NULL, s);
210         arg[i] = token;
211     }
212     i--;
213     arg[i] = NULL;
214 }
215
216 int main(int argc, char *argv[])
217 {
218
219     int com, p;
220
221     //printf("serv: %s port: %s", argv[1], argv[2]);
222     struct sockaddr_in si_other;
223     int s, i, slen = sizeof(si_other);
224     char buf[BUFLen];
225     char message[BUFLen];
226
227     if ((s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) == -1)
228     {
229         die("socket");
230     }
231
232     memset((char *)&si_other, 0, sizeof(si_other));
233     si_other.sin_family = AF_INET;
234     si_other.sin_port = htons(PORT);
235
236     //if (inet_aton(argv[1], &si_other.sin_addr) == 0)
237     if (inet_pton(AF_INET, argv[1], &si_other.sin_addr) == 0)
238     {
239         fprintf(stderr, "inet_aton() failed\n");

```

```

240     exit(1);
241 }
242 int recv_len;
243 /*
244  int recv_len;
245  if ((recv_len = recvfrom(s, buf, BUFLen, 0, (struct
      sockaddr *) &si_other, &slen)) == -1)
246  {
247      die("recvfrom()");
248  }
249
250  //print details of the client/peer and the data received
251  //printf("Received packet from %s:%d\n", inet_ntoa(
      si_other.sin_addr), ntohs(si_other.sin_port));
252  printf("From serv: %s\n" , buf);
253
254  //now reply the client with the same data
255  if (sendto(s, buf, recv_len, 0, (struct sockaddr*) &
      si_other, slen) == -1)
256  {
257      die("sendto()");
258  }
259  bzero(buf, sizeof(buf));
260  */
261
262  socklen_t slen_o = sizeof(si_other);
263
264  strcpy(message, "Hello");
265
266  if (sendto(s, message, strlen(message), 0, (struct
      sockaddr *) &si_other, slen_o) == -1)
267  {
268      die("sendto()");
269  }
270
271  //receive a reply and print it
272  //clear the buffer by filling null, it might have
      previously received data
273  memset(buf, '\0', BUFLen);
274  //try to receive some data, this is a blocking call
275  if (recvfrom(s, buf, BUFLen, 0, (struct sockaddr *) &
      si_other, &slen_o) == -1)
276  {
277      die("recvfrom()");
278  }
279
280  puts(buf);
281

```

```

282     if ((recv_len = recvfrom(s, buf, BUFLen, 0, (struct
283         sockaddr *) &si_other, &slen_o)) == -1)
284     {
285         die("recvfrom()");
286     }
287     //print details of the client/peer and the data received
288     //printf("Received packet from %s:%d\n", inet_ntoa(
289         si_other.sin_addr), ntohs(si_other.sin_port));
290     printf("%s\n", buf); //from server
291     //now reply the client with the same data
292     if (sendto(s, buf, recv_len, 0, (struct sockaddr*) &
293         si_other, slen_o) == -1)
294     {
295         die("sendto()");
296     }
297     bzero(buf, sizeof(buf));
298     bzero(message, sizeof(message));
299     //printf("Enter message : ");
300     gets(message);
301     //send the message
302     if (sendto(s, message, strlen(message), 0, (struct
303         sockaddr *) &si_other, slen_o) == -1)
304     {
305         die("sendto()");
306     }
307     //receive a reply and print it
308     //clear the buffer by filling null, it might have
309     //previously received data
310     memset(buf, '\0', BUFLen);
311     //try to receive some data, this is a blocking call
312     if (recvfrom(s, buf, BUFLen, 0, (struct sockaddr *) &
313         si_other, &slen_o) == -1)
314     {
315         die("recvfrom()");
316     }
317     //puts(buf);
318     strcpy(message, "Start");
319     if (sendto(s, message, strlen(message), 0, (struct
320         sockaddr *) &si_other, slen_o) == -1)

```

```

324 {
325     die("sendto()");
326 }
327
328 //receive a reply and print it
329 //clear the buffer by filling null, it might have
    previously received data
330 memset(buf, '\0', BUFLen);
331 //try to receive some data, this is a blocking call
332 if (recvfrom(s, buf, BUFLen, 0, (struct sockaddr *) &
    si_other, &slen_o) == -1)
333 {
334     die("recvfrom()");
335 }
336
337 puts(buf);
338
339 if ((recv_len = recvfrom(s, buf, BUFLen, 0, (struct
    sockaddr *) &si_other, &slen_o)) == -1)
340 {
341     die("recvfrom()");
342 }
343
344 //print details of the client/peer and the data received
345 //printf("Received packet from %s:%d\n", inet_ntoa(
    si_other.sin_addr), ntohs(si_other.sin_port));
346 printf("%s\n", buf);
347
348 //now reply the client with the same data
349 if (sendto(s, buf, recv_len, 0, (struct sockaddr*) &
    si_other, slen_o) == -1)
350 {
351     die("sendto()");
352 }
353 bzero(buf, sizeof(buf));
354 char* arg[4];
355 char** arg_spli;
356 char** qq;
357 char* arg1;
358 char* arg2;
359
360
361 while (1)
362 {
363     //printf("Enter message : ");
364     gets(message);
365
366     //send the message
367

```

```

368     if (sendto(s, message, strlen(message), 0, (struct
369         sockaddr *) &si_other, slen_o) == -1)
370     {
371         die("sendto()");
372     }
373     //receive a reply and print it
374     //clear the buffer by filling null, it might have
        previously received data
375     memset(buf, '\0', BUFLen);
376     //try to receive some data, this is a blocking call
377     if (recvfrom(s, buf, BUFLen, 0, (struct sockaddr *) &
        si_other, &slen_o) == -1)
378     {
379         die("recvfrom()");
380     }
381
382     //puts(buf);
383
384     /*
385     qq=str_split(buf, '#');
386
387     if(qq){
388     int i;
389     for (i = 0; *(qq + i); i++)
390     {
391         //printf("inp :%s\n", *(qq + i));
392
393         if(i == 0)
394             com = atoi(*(qq + i));
395         free(*(qq + i));
396     }
397     //printf("\n");
398     free(qq);
399     }*/
400
401     int spli;
402     arg_spli = splito_array(buf);
403     //for(spli=0;spli<sizeof(arg_spli);spli++)
404     //printf("aa: %s\n", arg_spli[spli]);
405     com = get_cmd(buf, 0);
406     //get_cmd_ch(buf, 1, arg1);
407     //get_cmd_ch(buf, 2, arg2);
408     //use_token(buf, 1, arg1);
409     //use_token(buf, 2, arg2);
410     //printf("test arg1: %s\n", arg1);
411     //printf("test arg2: %s\n", arg2);
412     //printf("com: %d", com);
413

```

```

414 //get_args(buf, arg);
415 //for(p = 0; p<4;p++)
416 //printf("i = %d inp: %s\n", i, arg[p]);
417 //puts(arg[0]);
418 //if(arg[0]!=NULL){
419 // com = atoi(arg[0]);
420 //}
421 if (com == 1){
422     //puts("CL: coma 1");
423 }
424 else if (com == 2){
425     while (strcmp(buf, "allmesgs") != 0){
426         if ((recv_len = recvfrom(s, buf, BUFLen, 0, (
427             struct sockaddr *) &si_other, &slen_o)) == -1)
428         {
429             die("recvfrom()");
430         }
431         //print details of the client/peer and the data
432         //received
433         //printf("Received packet from %s:%d\n",
434             inet_ntoa(si_other.sin_addr), ntohs(si_other.
435             sin_port));
436         printf("%s\n", buf);
437         //now reply the client with the same data
438         if (sendto(s, buf, recv_len, 0, (struct sockaddr
439             *) &si_other, slen_o) == -1)
440         {
441             die("sendto()");
442         }
443     }
444     /*if ((recv_len = recvfrom(s, buf, BUFLen, 0, (
445         struct sockaddr *) &si_other, &slen_o)) == -1)
446     {
447         die("recvfrom()");
448     }
449     //print details of the client/peer and the data
450     //received
451     //printf("Received packet from %s:%d\n", inet_ntoa(
452         si_other.sin_addr), ntohs(si_other.sin_port));
453     printf("%s\n", buf);
454     //now reply the client with the same data
455     if (sendto(s, buf, recv_len, 0, (struct sockaddr*) &
456         si_other, slen_o) == -1)
457     {

```

```

454     die("sendto()");
455 }*/
456 /*if ((recv_len = recvfrom(s, buf, BUFLen, 0, (
      struct sockaddr *) &si_other, &slen_o)) == -1)
457 {
458     die("recvfrom()");
459 }
460
461 //print details of the client/peer and the data
      received
462 //printf("Received packet from %s:%d\n", inet_ntoa(
      si_other.sin_addr), ntohs(si_other.sin_port));
463 printf("%s\n" , buf);
464
465 //now reply the client with the same data
466 if (sendto(s, buf, recv_len, 0, (struct sockaddr*) &
      si_other, slen_o) == -1)
467 {
468     die("sendto()");
469 }*/
470 //puts(buf);
471 bzero(buf, sizeof(buf));
472
473 //puts("CL: coma 2");
474
475 }
476 else if (com == 3){
477     puts("Bye!\n");
478     exit(1);
479 }
480 else
481     puts("Wrong input\n");
482
483 /*if(strcmp(message, "Start") == 0){
484 //
485 if ((recv_len = recvfrom(s, buf, BUFLen, 0, (struct
      sockaddr *) &si_other, &slen_o)) == -1)
486 {
487     die("recvfrom()");
488 }
489
490 //print details of the client/peer and the data
      received
491 //printf("Received packet from %s:%d\n", inet_ntoa(
      si_other.sin_addr), ntohs(si_other.sin_port));
492 printf("From serv: %s\n" , buf);
493
494 //now reply the client with the same data

```

```
495         if (sendto(s, buf, recu_len, 0, (struct sockaddr*) &  
496             si_other, slen_o) == -1)  
497             die("sendto()");  
498     }  
499     bzero(buf, sizeof(buf));  
500     }*/  
501     bzero(message, sizeof(message));  
502 }  
503  
504 close(s);  
505 return 0;  
506 }
```