Todo list

Не забыть вставить все исходники 109

Сети ЭВМ и телекоммуникации

К. Е. Назарова

24 декабря 2014 г.

Глава 1

Информационная система

1.1 Функциональные требования

1.1.1 Задание

Разработать распределённую информационную систему, состоящую из приложения-сервера и приложения-клиента. Информационная система является иерархическим хранилищем статей, каждая из которых состоит из названия, автора и текста статьи. Информационная система должна обеспечивать параллельный доступ к информации нескольким клиентам.

1.1.2 Основные возможности

Серверное приложение должно реализовывать следующие функции:

1. Прослушивание определенного порта
2. Обработка запросов на подключение по этому порту от клиентов
3. Поддержка одновременной работы нескольких клиентов через механизм нитей
4. Передача пользователю списка текущих разделов системы, списка статей
5. Переход в конкретный раздел системы по запросу клиента
6. Возврат на предыдущий уровень по запросу клиента
7. Передача пользователю конкретной статьи по названию

8. Передача пользователю всех статей текущего раздела, принадлежащих определенному автору
9. Приём от клиента новой статьи и сохранение в информационной системе
10. Обработка запроса на отключение клиента
11. Принудительное отключение клиента

Клиентское приложение должно реализовывать следующие функции:

1. Установление соединения с сервером
2. Получение и печать списка подразделов и статей раздела
3. Передача команды на переход в конкретный раздел
4. Передача команды на переход в раздел на уровень выше
5. Получение конкретной статьи из информационной системы
6. Получение статей конкретного автора
7. Посылка новой статьи в систему
8. Разрыв соединения
9. Обработка ситуации отключения клиента сервером

1.1.3 Настроки приложений

Разработанное клиентское приложение предоставляет пользователю возможность введения настройки IP-адреса или доменного имени, а также номера порта сервера информационной системы.

1.2 Нефункциональные требования

1.2.1 Требования к реализаии

Соединение начинает клиент. При подключении к порту и отправке начального сообщения, сервер передает клиенту список доступных разделов статей (содержимое корневой директории). Информационная система имеет иерархическую структуру, что позволяет киенту переходить в конкретный раздел системы и возвращаться на предыдущий уровень. В

ходе соединения клиент имеет возможность читать любые статьи, задавать поиск по автору статьи в конкретном разделе, а также добавлять статьи в раздел.

1.3 Накладываемые ограничения

1. Ограничения на длину пакета MSS (Maximum segment size) является параметром протокола TCP и определяет максимальный размер полезного блока данных в байтах для TCP пакета (сегмента). Таким образом этот параметр не учитывает длину заголовков TCP и IP. Для установления корректной TCP сессии с удалённым хостом должно соблюдаться следующее условие:

$$MSS + TCP + IP <= MTU \tag{1.1}$$

   Таким образом, максимальный размер MSS = MTU — размер заголовка IPv4 — размер заголовка TCP В данной реализации длина пакета составляет 1024 байта. Это объясняется там, что значение MSS обеих используемх операционных систем равно 1500 октетов:

```
root@debian:/home/ks/workspace/InformationSystem(Client-UDP)/Debug# ifconfig
eth0      Link encap:Ethernet  HWaddr 54:04:a6:3d:12:7a
          inet addr:172.16.51.83  Bcast:172.16.51.255  Mask:255.255.252.0
          inet6 addr: fe80::5604:a6ff:fe3d:127a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:163779872 errors:0 dropped:0 overruns:0 frame:0
          TX packets:133072 errors:0 dropped:0 overruns:0 carrier:304
          collisions:0 txqueuelen:1000
          RX bytes:2035296549 (1.8 GiB)  TX bytes:19892100 (18.9 MiB)
          Interrupt:52
```

Рис. 1.1: Значение MTU для ОС Unix

```
C:\Users\Kseniya\workspace\test_server\Debug>netsh interface ipv4 show subinterfaces

   MTU  Состояние определения носителя   Вх. байт  Исх. байт  Интерфейс
------  ---------------  ---------  ---------  -------------
4294967295                1          0    1076416  Loopback Pseudo-Interface 1
  1500                2  394143204    5022269  Беспроводная сеть 2
  1500                1  1132463768   20763140  Ethernet
  1500                5          0          0  Подключение по локальной сети* 15
```

Рис. 1.2: Значение MTU для ОС Windows 8

2. Ограничения на длину статьи На данном этапе максимальная длина статьи составляет 4096 байт. Такое значение выбрано для удобства реализации UDP-соединения, а также для простоты тестирования и отладки.

3. Ограничения на переход клиентом из родительской директории информационной системы на уровень выше.

4. Количество файлов в разделе. Их максимальное значение равно 100. Такое значение взято для удобного поиска по автору статьи.

5. Разрыв соединения, при котором теряются введенные клиентом запросы и данные, и при очередом подключении он снова оказывается в корневой директории, а статья, которую он создавал, не сохранилась.

6. Сервер и клиент не оповещают друг друга о потере связи.

7. Не до конца реализовано параллельное соединение клиентов в UDP. На данном этапе при подключении кажды следующий клиент должен дождаться завершения работы предыдущего.

Глава 2

Реализация для работы по протоколу TCP

2.1 Прикладной протокол

Соединение начинается с задания ip-адреса и номера порта. Для этого используются ключи -p:{№порта} -i:{ip-адрес}. По умолчанию используется порт №5001 и соединение с localhost (127.0.0.1).

Пользовательские команды	
Команда	Назначение
:start	Оповещение сервера о начале работы
open [файл | директория | . | ..]	Позволяет открывать файлы и перемещаться медлу разделами
find [автор]	Команда поиска статей по имени автора в текущем разделе
add[Заголовок][Автор][Содержимое][:end]	Команда для добавления новой статьи в текущую директорию
:end	Оповещение о конце ввода новой статьи
:exit	Оповещение о разрыве соединения клиентом

Все команды вводятся в текстовом формате. При этом накладываются следующие ограничения на длину параметров команд:

Команда	Параметр	Формат
open	файл	char [1024]
	директория	char [1024]
	родительская директория	..
	текущая директория	.
find	автор	char [128]
add	заголовок	char [128]
	автор	char [128]
	содержимое	char [4096]

2.2 Архитектура приложения

При начальном подключении протокол имеет возмжность введения настройки ip-адреса или доменного имени и номера порта сервера информационной системы.

Сама информационная система расположена на сервере в каталоге "./Information System/"и состоит из каталогово и txt-файлов. При получении сообщения от клиента к серверу о соединении, первый получает содержимое корневой директории ИС и ее полный путь. Сама информационная система имеет иерархическую струтуру, что позволяет клиенту свободно перемещаться между разделами как на уровень ниже, так и на уровень выше (но не глубже корневой директории).

Протокол подразумевает наличие всего шести команд, каждая из которых обрабатывается особым образом.

Для получения содержимого (командой *open*) любых разделов и статей сервер использует системные вызовы stat, dirent, и т.п. Это позволяет получить содержимое необходимого каталога и распознать типы файлов, находящихся в нем. В зависимости от результатов выполнения этих вызовов, сервер возвращает соотвтствующий ответ на запрос клиента.

Для поиска статей текущего раздела по имени автора (команда *find*) для каждого файла создается структура со следующими полями:

```
1 typedef struct {
2     char filename[BUF_SIZE];    // Полный путь к файлу;
3     char title[BUF_SIZE];       // Заголовок статьи
4     char author[BUF_SIZE];      // Автор статьи
5 } Article;
```

Такая организация позволяет осуществлять простой поиск по заголовку или автору статьи (в данной реализации только по имени автора). При этом алгоритм поиска устроен так, что результат не зависит от ре-

гистра букв в введенном клиентом имени, и полного совпадения имени автора какой-либо статьи с введенным именем, т.е. поиск будет удачным, если имя автора хотя бы одной статьи раздела содержит набор введенных в том же порядке символов. Чтобы поиск по заданному параметру осуществляся корректно, статьи информационной системы должны храниться в следующем формате:

```
1 ЗАГОЛОВОК
2 АВТОР
3 .....
4 СОДЕРЖИМОЕ
5 .....
```

Почти каждый запрос клиента/сервера сопровождается ответным сообщением-отчетом, который позволяет зафиксировать ошибки на обеих сторонах. Так, например, добавление статьи клиентом (команда *add*) в информационную базу осложено тем, что одновременно несколько клиентов могут добавлять в один и тот же раздел файл с одинаковым именем. Для решения этой проблемы посылаются дополнительные сообщения-отчеты сразу после ввода заголовка, и после ввода всего содержимого. Если в первом случае клиент сразу получит сообщение об ошибке (статья уже существует), то во втором - клиент сначала записывает данные, а при осуществлении записи выводится сообщение об ошибке, и введенная информация теряется.

В реализации TCP используется многопоточность. Поток создается при подсоединении нового клиента, и заканчвает свое выполнение при получении команды *:exit* от клиента. Sequence-диаграммы, определяющие фозможные сценарии, отображены на Рис. 2.1- 2.5

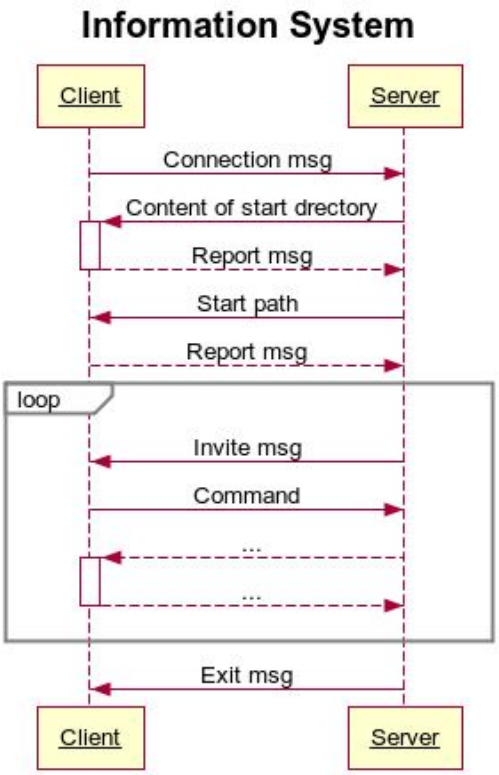

Рис. 2.1: Основная sequence-диаграмма

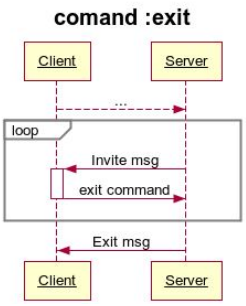

Рис. 2.2: Итерация с вызовом команды *:exit*

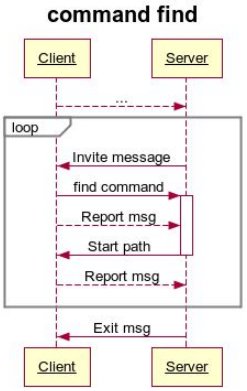

Рис. 2.3: Итерация с удачным вызовом команды *find*

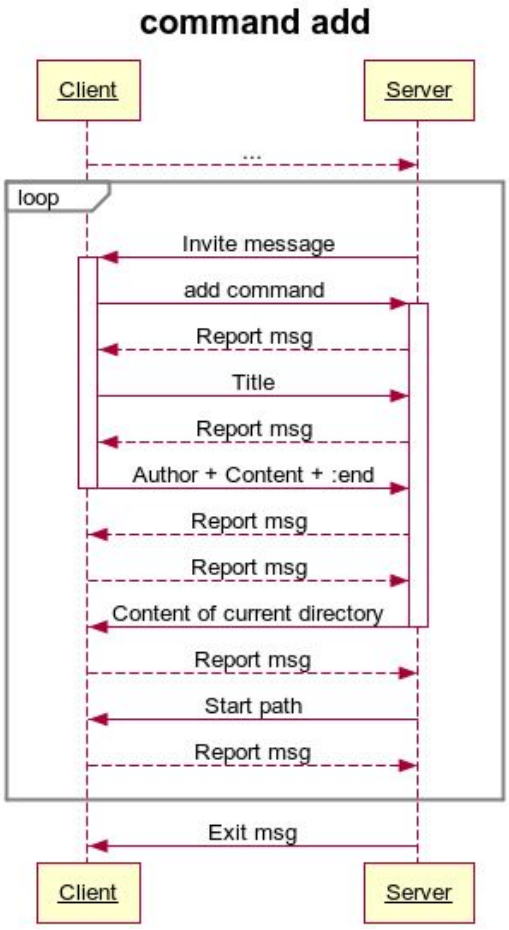

Рис. 2.4: Итерация с удачным вызовом команды *add*

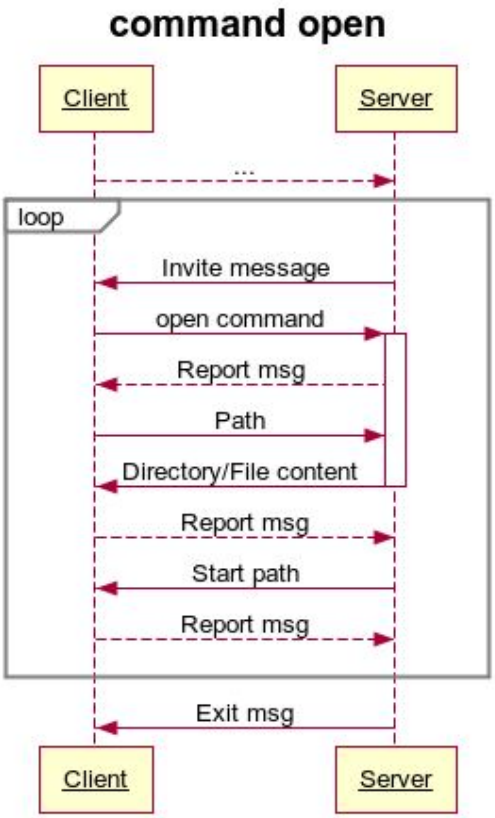

Рис. 2.5: Итерация с удачным вызовом команды *open*

2.3 Тестирование

2.3.1 Описание тестового стенда

Работа протокола тестируется в двух ОС: 1) Linux debian 3.2.0-4-686-pae #1 SMP Debian 3.2.63-2+deb7u2 i686 GNU/Linux
2) Windows 8.1

2.3.2 Тестовый план и результаты тестирования

План тестирования:

Linux	Linux
Windows	Windows
Windows(Server)	Linux(Client)
Windows(Client)	Linux(Server)

1. Все возможные команды с разными параметрами()
   - Удачные вызовы
     - :start
     - open (директория(текущая, дочерняя, родительская), файл)
     - add
     - :end
     - find
     - :exit
   - Неудачные вызовы
     - Те же команды с неверными парамерами
     - Несуществующие команды
2. Ошибки переполнения при вводе команд и их параметров
3. Возможность параллельного соединения клиентов и их конкуренция при сохранении новой статьи

Сервер запускается на Windows системе. К нему по TCP-соединению параллельно подключаются 3 клиента, 1 из которых расположен на Windows ОС.

Во время активной работы клиентов с сервером, последний фиксирует все приходящие и уходящие сообщения и их размер.

```
...
SEND  [56 bytes]: current path '/home/ks/workspace/
   InformationSystem/Information System/'
RECV  [4 bytes]: report message  '000'
SEND  [4 bytes]: invitation message '>   '
RECV  [4 bytes]: command 'open'
SEND  [3 bytes]: report message '000'
RECV  [58 bytes]: path to file message '/home/ks/workspace/
   InformationSystem/Information System/..'
SEND  [100 bytes]: directory content '----------------------|
   Articles|new.txt|Fairy Tales|first.txt|1.txt|Poems
   |..|----------------------|'
RECV  [4 bytes]: report message  '000'
SEND  [56 bytes]: current path '/home/ks/workspace/
   InformationSystem/Information System/'
RECV  [4 bytes]: report message  '000'
SEND  [4 bytes]: invitation message '>   '
RECV  [4 bytes]: command 'open'
SEND  [3 bytes]: report message '000'
RECV  [61 bytes]: path to file message '/home/ks/workspace/
   InformationSystem/Information System/Poems'
SEND  [71 bytes]: directory content '----------------------|
   Life|Time|Love|Nature|..|----------------------|'
RECV  [4 bytes]: report message  '000'
SEND  [62 bytes]: current path '/home/ks/workspace/
   InformationSystem/Information System/Poems/'
RECV  [4 bytes]: report message  '000'
SEND  [4 bytes]: invitation message '>   '
RECV  [4 bytes]: command 'open'
SEND  [3 bytes]: report message '000'
....
```

Подключение первого клиента:

1. Два раза подряд ошибочный ввод команды :start
2. На трейтий команда вызвана удачно, как результат, сервер посылает сдержимое корневой директории нформациооной системы
3. Появляется приглашение на ввод команды

```
ks@debian:~/workspace/InformationSystem_ClientTCP/Debug$ ./
   InformationSystem_ClientTCP
:sdw
:feueueueueueuue
:start
----------------------
```

```
Articles
new.txt
Fairy Tales
first.txt
1.txt
Poems
..
----------------------
>
```

1. Клиент вызывает команду *open*, чтобы открыть и прочесть файл
2. Сервер пересылает содержимое файла клиенту
3. Клиент пытается зайти в родительскую директорию "корневого"каталога. Доступ запрещен.
4. Приглашение на ввод команды

```
>   open 1.txt
1
me

Once on December...
>   open ..
----------------------
Articles
new.txt
Fairy Tales
first.txt
1.txt
Poems
..
----------------------
>
```

1. Клиент переходит в раздел Poems/Love
2. Клиент вводит команду поиска статей по имени автора *find*. Вводит часть имени: *shak*
3. Клиент отправляет команду *open*, чтобы открыть найденный файл, и получает его содержимое
4. Клиент запрашивает содержимое текущего раздела

```
>   open Poems
----------------------
Life
Time
Love
Nature
..
----------------------
>   open Love
----------------------
Love is my Sin.txt
Love.txt
..
----------------------
>   find shak
Search results for author: shak
----------------------
William Shakespeare
:    Love is my Sin.txt
----------------------
>   open Love is my Sin.txt
Love is my Sin
William Shakespeare

CXLII.

Love is my sin and thy dear virtue hate,
Hate of my sin, grounded on sinful loving:
O, but with mine compare thou thine own state,
And thou shalt find it merits not reproving;
Or, if it do, not from those lips of thine,
That have profaned their scarlet ornaments
And seal'd false bonds of love as oft as mine,
Robb'd others' beds' revenues of their rents.
Be it lawful I love thee, as thou lovest those
Whom thine eyes woo as mine importune thee:
Root pity in thy heart, that when it grows
Thy pity may deserve to pitied be.
If thou dost seek to have what thou dost hide,
By self-example mayst thou be denied!
>   open .
----------------------
Love is my Sin.txt
Love.txt
..
```

1. Клиент1 вводит команду *add* чтобы создать статью с заголовкам "3"

2. В это время подключается Клиент2 и пытается создать статью с таким же заголовком

3. Клиент2 закончил ввод содержимого быстрее, в результате Клиент1 после ввода своего содержимого получает ошибку, его введенные данные теряются

4. Клиент1 обижается и уходит, послав команду *:exit*

```
>   add
3
Input author: me
name's read: 3 [1 bytes]
author's read: me [1 bytes]
Put content:
[0 of 3840]  I'm fine! And you?
:end
!Such file already exist

----------------------
3.txt
Articles
new.txt
Fairy Tales
first.txt
1.txt
Poems
..
----------------------
>   open 3.txt
3
he

Hello!
How are you!

>   :exit
Bye-bye!!!
```

```
>   add 3
Input author: he
name's read: 3 [1 bytes]
author's read: he [1 bytes]
Put content:
[0 of 3840]  Hello!
How are you!
:end

```

```
----------------------
3.txt
Articles
new.txt
Fairy Tales
first.txt
1.txt
Poems
..
----------------------
>   open 3.txt
3
he

Hello!
How are you!
```

Глава 3

Реализация для работы по протоколу UDP

3.1 Прикладной протокол и ахитектура

Реализация UDP протокола отличается от TCP только использованием функций recv/recvfrom и send/sendto. А также тем, что в данном случае не удалось реализовать многопоточность. Были попытки установления параллельного подключения клиентов с использованием мьютексов. Однако эи попытки не увенчались успехом. В данной реализации используется последовательное подключение клиентов: Каждый новый ждет завершения работы предыдущего клиента.

3.2 Тестирование

3.2.1 Описание тестового стенда

Работа протокола тестируется в двух ОС: 1) Linux debian 3.2.0-4-686-pae #1 SMP Debian 3.2.63-2+deb7u2 i686 GNU/Linux
2) Windows 8.1

3.2.2 Тестовый план и результаты тестирования

1. Ввод несуществующих команд
2. удачный и неудачный поиск
3. Тест на переполнение при вводе новой статьи

Сервер запускается на UNIX ОС. К нему по UDP-соединению последовательно подключаются 2 клиента: один из ОС Windows, другой - из Unix.

1. Клиент1 подсоединяется к порту 5001 и серверу с ip-адресом 172.16.51.83 командой *:start*
2. Неудачный поиск командой *find*
3. Удачный поиск по имени автора

```
C:\Users\Kseniya\workspace\client_udp\Debug>client_udp.exe -p
    :5001 -i:172.16.51.83
:start
----------------------
Literature
second.txt
first.txt
1.txt
..
----------------------
>   open 1.txt
title:  STORY
author: PUSHKIN
Hello, everybody!
LALAL
>   find pusjkin
There are no articles of pusjkin
>   find pushkin
Search results for author: pushkin
----------------------
author: PUSHKIN
:    1.txt
----------------------
>
```

1. Клиент1 переходит в раздел Literature
2. Вводит команду *add*, чтобы добавить статью в систему
3. Вызов переполнения при вставке большого куска текста
4. Проверка возможности чтения добавленной статьи
5. Попытка открыть несуществующий файл/раздел

```
>   open Literature
----------------------
story.txt
Rapunzel.txt
..
Clever Hans.txt
----------------------
>   add Something
Input author: Hans Chrustian Andersen
name's read: Something [9 bytes]
author's read: Hans Chrustian Andersen [9 bytes]
Put content:
[0 of 3840]  MEAN to be somebody, and do something useful in the world,"␣said␣the␣eldest␣of␣five␣bro
thers.␣"I don't care how humble my position is, so that I can only do some good, which will be somet
hing. I intend to be a brickmaker; bricks are always wanted, and I shall be really doing something."
"Your 'something' is not enough for me,"␣said␣the␣second␣brother;␣"what you talk of doing is nothing
 at all, it is journeyman's work, or might even be done by a machine. No! I should prefer to be a bu
ilder at once, there is something real in that. A man gains a position, he becomes a citizen, has hi
s own sign, his own house of call for his workmen: so I shall be a builder. If all goes well, in tim
e I shall become a master, and have my own journeymen, and my wife will be treated as a master's wif
.....
ich formed a dyke on the sea-coast, a poor woman named Margaret wished to build herself a house, so
all the imperfect bricks were given to her, and a few whole ones with them; for the eldest brother w
as a kind-hearted man, although he never achieved anything higher than making bricks. The poor woman
 built herself a little house-it was small and narrow, and the window was quite crooked, the door to
o low, and the straw roof might have been better thatched. But still it was a shelter, and from with
[3840 of 3840]  !Type less or :end
[3840 of 3840]  !Type less or :end
[3840 of 3840]  !Type less or :end
[3840 of 3840]  !Type less or :end
[3840 of 3840]  !Type less or :end
[3840 of 3840]  !Type less or :end
[3840 of 3840]  !Type less or :end
[3840 of 3840]  :end

```

```
----------------------
story.txt
Rapunzel.txt
Something.txt
..
Clever Hans.txt
----------------------
>   open Something.txt
Something
Hans Chrustian Andersen

MEAN to be somebody, and do something useful in the world,"␣
    said␣the␣eldest␣of␣five␣brothers.␣"I don
't care how humble my position is, so that I can only do some
     good, which will be something. I inten
d to be a brickmaker; bricks are always wanted, and I shall
    be really doing something."

"Your 'something' is not enough for me,"␣said␣the␣second␣
    brother;␣"what you talk of doing is nothing
 at all, it is journeyman's work, or might even be done by a
     machine. No! I should prefer to be a bu
ilder at once, there is something real in that. A man gains a
     position, he becomes a citizen, has hi
s own sign, his own house of call for his workmen: so I shall
     be a builder. If all goes well, in tim
e I shall become a master, and have my own journeymen, and my
     wife will be treated as a master's wif
e. This is what I call something."

"I call it all nothing,"␣said␣the␣third;␣"not in reality any
    position. There are many in a town far
above a master builder in position. You may be an upright man
    , but even as a master you will only be
...
 always be men like these five brothers. And what became of
     them? Were they each nothing or somethin
g? You shall hear; it is quite a history.

The eldest brother, he who fabricated bricks, soon discovered
     that each brick, when finished, brough
t him in a small coin, if only a copper one; an
>   open ,'
!No such file or directory
```

Глава 4

Выводы

В результате выполнения данных заданий можно сделать вывод, что для предложенного протокола гораздо удобнее испоьзовать TCP-соединение. Это сязано с тем, что TCP является надеждным соединение, в отличие от UDP, а так как работа информационной системы связана с передачей больших текстовых сообщений, то выбор TCP будет наиболее предпочтителен. Еще одним приемуществом использования TCP является возможность реализации многопоточности, чего не удалось достичь при использовании UDP.

Приложения

Описание среды разработки

Версии ОС, компиляторов, утилит, и проч., которые использовались в процессе разработки

Листинги

UNIX

TCP Server

main.c

```
1  /*
2   * main.c
3   *
4   *  Created on: Nov 7, 2014
5   *      Author: user
6   */
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <sys/types.h>
11 #include <sys/stat.h>
12 #include <dirent.h>
13 #include <string.h>
14 #include <fcntl.h>
15 #include <sys/socket.h>
16 #include <netinet/in.h>
17 #include <libgen.h>
18 #include <pthread.h>
19 #include <errno.h>
20 #include <stdbool.h>
21 #include "article.h"
22
```

```
23
24 #define DEFAULT_PORT 5001
25 #define SIZE_CMD 5
26 #define SIZE_BUF 1024
27 #define SIZE_CONTENT 4096
28 #define SIZE_STR 128
29 #define MAX_CONNECT 3
30 #define SUCCESS "000"
31 #define UNSUCCESS "111"
32 #define START_PATH "/home/ks/workspace/InformationSystem/
       Information System/"
33
34 pthread_t t1;
35 int port = DEFAULT_PORT;
36 bool interface = 0;
37 char szAddress[SIZE_STR];
38
39 int send_content(int sock, char *dir_name);
40 int open_file(int sock, char *path);
41 void sendPath_recvReport(int sock, char *path);
42 void *pthread_handler(void *sock);
43 void send_input_error(int sock);
44 void send_report(int sock, char *status);
45 void recv_report(int sock);
46 void validateArgs(int argc, char **argv);
47 void usage();
48
49 int main(int argc, char **argv)
50 {
51    const int on = 1;
52    int sock, newsock, client;
53    struct sockaddr_in server, client_addr;
54    validateArgs(argc, argv);
55
56    if (interface)
57    {
58       server.sin_addr.s_addr = inet_addr(szAddress);
59       if (server.sin_addr.s_addr == INADDR_NONE)
60          usage();
61    }
62    else
63       server.sin_addr.s_addr = htonl(INADDR_ANY);
64    server.sin_family = AF_INET;
65    server.sin_port = htons(port);
66    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
67    {
68       perror("Socket␣is␣not␣created");
69       exit(1);
70    }
```

```
71      setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)
            );
72      if (bind(sock, (struct sockaddr *)&server, sizeof(server))
             < 0)
73      {
74          perror("Socket␣is␣not␣bound");
75          exit(1);
76      }
77
78      if (listen(sock, 5) < 0)
79      {
80          perror("Listening␣error");
81          exit(1);
82      }
83
84      int i = 0;
85      while(1)
86      {
87          client = sizeof(client_addr);
88          if((newsock = accept(sock, (struct sockaddr *)&
                client_addr, (socklen_t *)&client)) < 0)
89          {
90              perror("Accepting␣error");
91              exit(1);
92          }
93          pthread_create(&t1, NULL, pthread_handler, (void *)
                newsock);
94          i++;
95      }
96
97      for(; i > 0; i--)
98          pthread_join(t1, NULL);
99      close(sock);
100     return 0;
101 }
102
103 void *pthread_handler(void *newsock)
104 {
105     int sock = (int) newsock, msg_size;
106     const char *invite_msg = ">␣␣␣";
107     const char *exit_msg = "Bye-bye!!!|";
108     char path[SIZE_BUF];
109     char name[SIZE_STR];
110     char buffer[SIZE_BUF];
111     char author[SIZE_STR];
112     char content[SIZE_CONTENT];
113
114     while(strcmp(buffer, ":start"))
115     {
```

```
116         bzero(buffer, sizeof(buffer));
117         if ((msg_size = recv(sock, buffer, sizeof(buffer), 0))
                < 0)
118         {
119             perror("RECV start message error");
120             exit(1);
121         }
122     }
123     printf("RECV [%d bytes]: start message '%s'\n", msg_size,
            buffer);
124     send_content(sock, START_PATH);
125     strcpy(path, START_PATH);
126
127     while(1)
128     {
129         if ((msg_size = send(sock, invite_msg, strlen(
                invite_msg), 0)) < 0)
130         {
131             perror("SEND invitation message error");
132             exit(1);
133         }
134         printf("SEND [%d bytes]: invitation message '%s'\n",
                msg_size, invite_msg);
135         bzero(buffer, sizeof(buffer));
136         if ((msg_size = recv(sock, buffer, sizeof(buffer), 0))
                < 0)
137         {
138             perror("RECV command error");
139             exit(1);
140         }
141         printf("RECV [%d bytes]: command '%s'\n", msg_size,
                buffer);
142
143         if (!strcmp(buffer, ":exit"))
144         {
145             if ((msg_size = send(sock, exit_msg, strlen(exit_msg
                    ), 0)) < 0)
146             {
147                 perror("SEND directory content error");
148                 exit(1);
149             }
150             printf("SEND [%d bytes]: directory content '%s'\n",
                     msg_size, exit_msg);
151             break;
152         }
153         if (strcmp(buffer, "find") && strcmp(buffer, "open") &&
                 strcmp(buffer, "add"))
154         {
155             send_input_error(sock);
```

```
156             send_content(sock, path);
157             continue;
158         }
159         send_report(sock, SUCCESS);
160 
161         if (!strcmp(buffer, "open"))
162         {
163             bzero(path, sizeof(path));
164             if ((msg_size = recv(sock, path, sizeof(path), 0)) <
                     0)
165             {
166                 perror("RECV path to file error");
167                 exit(1);
168             }
169             printf("RECV  [%d bytes]: path to file message '%s'\
                    n", msg_size, path);
170             open_file(sock, path);
171         }
172 
173         if (!strcmp(buffer, "find"))
174         {
175             bzero(author, sizeof(author));
176             if ((msg_size = recv(sock, author, sizeof(author),
                    0)) < 0)
177             {
178                 perror("RECV author to find error");
179                 exit(1);
180             }
181             printf("RECV  [%d bytes]: author to find '%s'\n",
                    msg_size, author);
182 
183             find_for_author(sock, path, author);
184             sendPath_recvReport(sock, path);
185         }
186 
187         if (!strcmp(buffer, "add"))
188         {
189             bzero(name, sizeof(name));
190             if ((msg_size = recv(sock, name, sizeof(name), 0)) <
                     0)
191             {
192                 perror("RECV name error");
193                 exit(1);
194             }
195             printf("RECV  [%d bytes]: name '%s'\n", msg_size,
                    name);
196             char *dir = strdup(path);
197             strcat(path, name);
198             strcat(path, ".txt");
```

```
199             if (check_file_existence(path) < 0)
200             {
201                 send_report(sock, UNSUCCESS);
202                 recv_report(sock);
203             }
204             else
205             {
206                 send_report(sock, SUCCESS);
207                 bzero(author, sizeof(author));
208                 if ((msg_size = recv(sock, author, sizeof(author)
                        , 0)) < 0)
209                 {
210                     perror("RECV␣author␣error");
211                     exit(1);
212                 }
213                 printf("RECV␣␣[%d␣bytes]:␣author␣'%s'\n",
                        msg_size, author);
214                 send_report(sock, SUCCESS);
215                 bzero(content, sizeof(content));
216                 if ((msg_size = recv(sock, content, sizeof(
                        content), 0)) < 0)
217                 {
218                     perror("RECV␣content␣of␣file␣error");
219                     exit(1);
220                 }
221                 printf("RECV␣␣[%d␣bytes]:␣content␣of␣file␣'%s'\n"
                        , msg_size, content);
222                 strcat(name, "\n");
223                 strcat(author, "\n\n");
224                 if (add_article(path, name, author, content) < 0)
225                     send_report(sock, UNSUCCESS);
226                 else
227                     send_report(sock, SUCCESS);
228                 recv_report(sock);
229             }
230             send_content(sock, dir);
231             dirname(path);
232             if (path[strlen(path) - 1] != '/')
233                 strcat(path, "/");
234         }
235     }
236     pthread_exit(0);
237     return 0;
238 }
239
240 void validateArgs(int argc, char **argv)
241 {
242     int i;
243
```

```
244     for(i = 1; i < argc; i++)
245     {
246        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
247        {
248           switch (tolower(argv[i][1]))
249           {
250           case 'p':
251              port = atoi(&argv[i][3]);
252              break;
253           case 'i':
254              interface = 1;
255              if (strlen(argv[i]) > 3)
256                 strcpy(szAddress, &argv[i][3]);
257              break;
258           default:
259              usage();
260              break;
261           }
262        }
263     }
264  }
265
266  void usage()
267  {
268     printf("usage: server [-p:x] [-i:IP]\n\n");
269     printf(" -p:x Port number to listen on\n");
270     printf(" -i:str Interface to listen on\n");
271  }
272
273  void send_input_error(int sock)
274  {
275     send_report(sock, UNSUCCESS);
276     recv_report(sock);
277  }
278
279  int send_content(int sock, char *dir_name)
280  {
281     char buffer[SIZE_BUF];
282     char *filename;
283     const char *delimiter = "----------------------|";
284     int msg_size;
285     DIR *dir = opendir(dir_name);
286
287     bzero(buffer,sizeof(buffer));
288     if(dir)
289     {
290        struct dirent *ent;
291        strcat(buffer, delimiter);
292        while((ent = readdir(dir)) != NULL)
```

```
293         {
294             filename = ent->d_name;
295             if (strcmp(filename, ".")==0)
296                 continue;
297             strcat(filename, "|");
298             strcat(buffer, filename);
299         }
300         closedir(dir);
301         strcat(buffer, delimiter);
302         if ((msg_size = send(sock, buffer, strlen(buffer), 0))
                < 0)
303         {
304             perror("SEND␣directory␣content␣error");
305             exit(1);
306         }
307         printf("SEND␣␣[%d␣bytes]:␣directory␣content␣'%s'\n",
                msg_size, buffer);
308         bzero(filename,sizeof(filename));
309         strcpy(filename, dir_name);
310         if (!strcmp(basename(filename), "."))
311             dirname(dir_name);
312         else if (!strcmp(basename(filename), ".."))
313             dirname(dirname(dir_name));
314     }
315     else
316     {
317         const char *err_msg = "!No␣such␣file␣or␣directory|";
318         if ((msg_size = send(sock, err_msg, strlen(err_msg), 0)
                ) < 0)
319         {
320             perror("SEND␣no␣file␣or␣directory␣error");
321             exit(1);
322         }
323         printf("SEND␣␣[%d␣bytes]:␣no␣file␣or␣directory␣message␣
                '%s'\n", msg_size, buffer);
324         dirname(dir_name);
325     }
326
327     sendPath_recvReport(sock, dir_name);
328     return 0;
329 }
330
331 int open_file(int sock, char *path)
332 {
333     FILE *fp;
334     int msg_size;
335     struct stat about_file;
336
337     char buffer[SIZE_STR], text[SIZE_CONTENT];
```

```
338     bzero(text, sizeof(text));
339     bzero(buffer, sizeof(buffer));
340 
341     char *tmp = malloc(SIZE_BUF);
342     strcpy(tmp, START_PATH);
343     if (!strcmp(path, strcat(tmp, "..")))
344     {
345         send_content(sock, START_PATH);
346         free(tmp);
347         return 0;
348     }
349     free(tmp);
350     if ((fp = fopen(path, "r")) == NULL)
351     {
352         perror("Opening␣of␣file␣error");
353         send_content(sock, path);
354         return -1;
355     }
356 
357     fstat(fileno(fp), &about_file);
358     if ((about_file.st_mode & S_IFMT) != S_IFDIR)
359     {
360         int ch, i;
361         for (i = 0; i < (sizeof(text) - sizeof(char)) && (ch =
                getc(fp)) != EOF; i++)
362         {
363             if (ch == '\n')
364                 ch = '|';
365             text[i] = ch;
366         }
367         strcat(text, "|");
368         if ((msg_size = send(sock, text, strlen(text), 0)) < 0)
369         {
370             perror("SEND␣content␣of␣file␣error");
371             exit(1);
372         }
373         printf("SEND␣␣[%d␣bytes]:␣content␣of␣file\n", msg_size)
                ;
374 
375         dirname(path);
376         sendPath_recvReport(sock, path);
377 
378     }
379     else
380         send_content(sock, path);
381     fclose(fp);
382     return 0;
383 
384 }
```

```
385
386 void sendPath_recvReport(int sock, char *path)
387 {
388    int msg_size;
389    recv_report(sock);
390    if (path[strlen(path) - 1] != '/')
391       strcat(path, "/");
392    if ((msg_size = send(sock, path, strlen(path), 0)) < 0)
393    {
394       perror("SEND␣current␣path␣error");
395       exit(1);
396    }
397    printf("SEND␣␣[%d␣bytes]:␣current␣path␣'%s'\n", msg_size,
       path);
398    recv_report(sock);
399 }
400
401 void send_report(int sock, char *status)
402 {
403    int msg_size;
404    if ((msg_size = send(sock, status, strlen(status), 0)) <
       0)
405    {
406       perror("SEND␣report␣message␣error");
407       exit(1);
408    }
409    printf("SEND␣␣[%d␣bytes]:␣report␣message␣'%s'\n", msg_size
       , status);
410 }
411
412
413 void recv_report(int sock)
414 {
415    char status[SIZE_CMD];
416    int msg_size;
417    bzero(status, sizeof(status));
418    if ((msg_size = recv(sock, status, sizeof(status), 0)) <
       0)
419    {
420       perror("RECV␣report␣message␣failed");
421       exit(1);
422    }
423    printf("RECV␣␣[%d␣bytes]:␣report␣message␣␣'%s'\n",
       msg_size, status);
424 }
```

article.h

```
1 /*
2  * article.h
```

```
 *
 *  Created on: Nov 7, 2014
 *      Author: user
 */

#ifndef ARTICLE_H_
#define ARTICLE_H_

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <libgen.h>



#define MAX_FILES 100
#define BUF_SIZE 128
#define MAX_SIZE 1024

typedef struct art
{
   char filename[BUF_SIZE];
   char title[BUF_SIZE];
   char author[BUF_SIZE];
} Article;

int check_file_existence(char *dir_name);
int add_article(char *dir_name, char *name, char* author,
    char *content);
int find_for_author(int sock, char *dir_name, char *author);
char *lower(char *str);

#endif /* ARTICLE_H_ */
```

article.c

```
#include "article.h"

int add_article(char *dir_name, char *name, char* author,
    char *content)
{
   FILE *fp;
   if ((fp = fopen(dir_name,  "r" )) == NULL)
      if (errno == ENOENT)
```

```
8           if ((fp = fopen(dir_name,  "w+" )) == NULL)
9           {
10              perror("File creation error");
11              return -2;
12          }
13          else
14          {
15              fputs(name, fp);
16              fputs(author, fp);
17              fputs(content, fp);
18              rewind(fp);
19              close(fp);
20              return 0;
21          }
22      close(fp);
23      return -1;
24  }
25
26  int check_file_existence(char *dir_name)
27  {
28      FILE *fp;
29      if ((fp = fopen(dir_name,  "r" )) == NULL)
30          if (errno == ENOENT)
31              return 0;
32      close(fp);
33      return -1;
34  }
35
36  int find_for_author(int sock, char *dir_name, char *author)
37  {
38      char buffer[BUF_SIZE];
39      char path[MAX_SIZE];
40      char *ptr;
41      const char *delimiter = "----------------------|";
42      int msg_size;
43      char *filename;
44      FILE *fp;
45      DIR *dir = opendir(dir_name);
46
47      bzero(buffer,sizeof(buffer));
48      struct stat about_file;
49      Article *arts=(Article *)malloc(MAX_FILES * sizeof(Article
            ));
50      int i, k = 0;
51      if(dir)
52      {
53          struct dirent *ent;
54          while((ent = readdir(dir)) != NULL)
55          {
```

```
56          strcpy(path, dir_name);
57          filename = ent->d_name;
58
59          if ((fp = fopen(strcat(path, filename), "r")) ==
                NULL)
60          {
61             printf("error␣%s\n", filename);
62             perror("Opening␣of␣file␣error");
63          }
64          fstat(fileno(fp), &about_file);
65          if ((about_file.st_mode & S_IFMT) != S_IFDIR)
66          {
67             for( i = 0; (ptr = fgets(buffer, sizeof(buffer),
                   fp)) != NULL && i <2; i++)
68             {
69                if (i == 0)
70                   strcpy(arts[k].title, ptr);
71                else if (i == 1)
72                   strcpy(arts[k].author, ptr);
73                bzero(ptr, strlen(ptr));
74             }
75             strcpy(arts[k].filename, filename);
76             k++;
77          }
78          fclose(fp);
79       }
80       closedir(dir);
81       bzero(buffer, sizeof(buffer));
82       strcat(buffer, "Search␣results␣for␣author:␣");
83       strcat(buffer, author);
84       strcat(buffer, "|");
85       strcat(buffer, delimiter);
86       for (i = 0; k >= 0; --k)
87       {
88          if (strstr(lower(arts[k].author), lower(author)) !=
                NULL)
89          {
90             strcat(buffer, arts[k].author);
91             strcat(buffer, ":␣␣␣␣");
92             strcat(buffer, arts[k].filename);
93             strcat(buffer, "|");
94             i++;
95          }
96          puts(buffer);
97       }
98       strcat(buffer, delimiter);
99       if (i == 0)
100      {
101         bzero(buffer, sizeof(buffer));
```

```
102             strcat(buffer, "There␣are␣no␣articles␣of␣");
103             strcat(buffer, author);
104             strcat(buffer, "|");
105         }
106         if ((msg_size = send(sock, buffer, strlen(buffer), 0))
                < 0)
107         {
108             perror("SEND␣found␣result␣error");
109             return -1;
110         }
111         printf("SEND␣␣␣[%d␣bytes]:␣found␣result␣'%s'\n",
                msg_size, buffer);
112         return 0;
113     }
114     free(arts);
115     return -1;
116 }
117 
118 char *lower(char *str)
119 {
120     int i;
121     char *new = strdup(str);
122     for (i = 0; i < strlen(new); i++)
123         new[i] = tolower(new[i]);
124     return new;
125 }
```

TCP Client

main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <stdbool.h>
5  #include <sys/stat.h>
6  #include <dirent.h>
7  #include <string.h>
8  #include <fcntl.h>
9  #include <sys/socket.h>
10 #include <netinet/in.h>
11 #include <libgen.h>
12 #include <stdio_ext.h>
13 
14 #define SIZE_CMD 5
15 #define SIZE_ARG 50
16 #define SIZE_STR 128
17 #define SIZE_BUF 1024
```

```
18 #define SUCCESS "000"
19 #define UNSUCCESS "111"
20 #define SIZE_CONTENT 4096
21 #define DEFAULT_PORT 5001
22
23 void output(char *str);
24 void add_article_to_system(int sock, char *path);
25 int recv_report(int sock);
26 void send_report(int sock, char *status);
27 void ValidateArgs(int argc, char **argv);
28 void usage();
29
30 int port = DEFAULT_PORT;
31 bool interface = 0;
32 char szAddress[SIZE_STR];
33
34 int main(int argc, char **argv)
35 {
36    int sock, msg_size;
37    char path[SIZE_BUF];
38    char name[SIZE_STR];
39    char author[SIZE_STR];
40    char buffer[SIZE_BUF];
41    char command[SIZE_CMD];
42    char content[SIZE_CONTENT];
43    struct sockaddr_in client;
44
45
46    ValidateArgs(argc, argv);
47
48    if (interface)
49    {
50       client.sin_addr.s_addr = inet_addr(szAddress);
51       if (client.sin_addr.s_addr == INADDR_NONE)
52          usage();
53    }
54    else
55       client.sin_addr.s_addr = htonl(INADDR_ANY);
56    client.sin_family = AF_INET;
57    client.sin_port = htons(port);
58
59    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
60    {
61       perror("Socket is not created");
62       exit(1);
63    }
64
65    if(connect(sock, (struct sockaddr *)&client, sizeof(client
       )) < 0)
```

```
66      {
67          perror("Connection error");
68          exit(2);
69      }
70
71      bzero(buffer, sizeof(buffer));
72      while(strcmp(buffer, ":start"))
73      {
74          fgets(buffer, sizeof(buffer), stdin);
75          if (buffer[strlen(buffer) - 1] == '\n')
76              buffer[strlen(buffer) - 1] = '\0';
77          if ((msg_size = send(sock, buffer, strlen(buffer), 0))
                < 0)
78          {
79              perror("SEND start message failed");
80              exit(1);
81          }
82      }
83      //printf("SEND  [%d bytes]: start message '%s'\n",
            msg_size, buffer);
84      bzero(buffer, sizeof(buffer));
85      if ((msg_size = recv(sock, buffer, sizeof(buffer), 0)) <
            0)
86      {
87          perror("RECV directory content failed");
88          exit(1);
89      }
90      //printf("RECV  [%d bytes]: directory content\n", msg_size
            );
91      output(buffer);
92      send_report(sock, SUCCESS);
93      while(1)
94      {
95          bzero(path, sizeof(path));
96          if ((msg_size = recv(sock, path, sizeof(path), 0)) < 0)
97          {
98              perror("RECV current path failed");
99              exit(1);
100         }
101 //      printf("RECV  [%d bytes]: current path '%s'\n",
        msg_size, path);
102
103         send_report(sock, SUCCESS);
104
105         bzero(buffer, sizeof(buffer));
106         if ((msg_size = recv(sock, buffer, sizeof(buffer), 0))
                < 0)
107         {
108             perror("RECV invitation message failed");
```

```
109             exit(1);
110         }
111         //printf("RECV  [%d bytes]: invitation message\n",
                msg_size);
112         output(buffer);
113         char space;
114         bzero(name, sizeof(name));
115         bzero(buffer, sizeof(buffer));
116         bzero(author, sizeof(author));
117         bzero(command, sizeof(command));
118         bzero(content, sizeof(content));
119         scanf("%5s%1c", command, &space);
120         if ((msg_size = send(sock, command, strlen(command), 0)
                ) < 0)
121         {
122             perror("SEND␣command␣failed");
123             exit(1);
124         }
125     // printf("SEND  [%d bytes]: command '%s'\n", msg_size,
            path);
126 
127         if (!strcmp(command, ":exit"))
128         {
129             bzero(buffer, sizeof(buffer));
130             if ((msg_size = recv(sock, buffer, sizeof(buffer),
                    0)) < 0) // Receive the content of file
131             {
132                 perror("RECV␣exit␣message␣failed");
133                 exit(1);
134             }
135             output(buffer);
136             break;
137         }
138         if (recv_report(sock) < 0)
139         {
140             puts("!No␣such␣command");
141             send_report(sock, SUCCESS);
142         }
143         if (!strcmp(command, "add"))
144         {
145             char str[SIZE_ARG];
146             fgets(name, sizeof(name), stdin);
147             if (name[strlen(name) - 1] == '\n')
148                 name[strlen(name) - 1] = '\0';
149             if ((msg_size = send(sock, name, strlen(name), 0)) <
                     0)
150             {
151                 perror("SEND␣command␣failed");
152                 exit(1);
```

```
153             }
154             //printf("SEND  [%d bytes]: title of article '%s'\n
                    ", msg_size, name);
155             if (recv_report(sock) < 0)
156             {
157                 puts("!Such␣file␣already␣exist");
158                 send_report(sock, SUCCESS);
159             }
160             else
161             {
162                 int length = sizeof(content) - sizeof(author) -
                        sizeof(name);
163                 printf("Input␣author:␣");
164                 fgets(author, sizeof(author), stdin);
165                 if (author[strlen(author) - 1] == '\n')
166                     author[strlen(author) - 1] = '\0';
167                 printf("name's␣read:␣%s␣[%d␣bytes]\n", name,
                        msg_size);
168                 printf("author's␣read:␣%s␣[%d␣bytes]\n", author,
                        msg_size);
169                 puts("Put␣content:");
170                 printf("[%d␣of␣%d]␣␣", (strlen(content)+strlen(
                        str)), length);
171                 while (fgets(str, sizeof(str), stdin) != NULL)
172                 {
173                     if (!strncmp(":end", str, strlen(":end")))
174                         break;
175                     if ((strlen(content)+strlen(str)) > length)
176                     {
177                         puts("!Text␣size␣will␣not␣allow.␣Type␣less␣
                                or␣:end");
178                         bzero(str, strlen(str));
179                         printf("[%d␣of␣%d]␣␣", (strlen(content)+
                                strlen(str)), length);
180                         //__fpurge(stdin);
181                     }
182                     strcat(content, str);
183                     bzero(str, strlen(str));
184                 }
185                 if ((msg_size = send(sock, author, strlen(author)
                        , 0)) < 0)
186                 {
187                     perror("SEND␣author␣of␣article␣failed");
188                     exit(1);
189                 }
190                 //printf("SEND  [%d bytes]: author of article '%s
                        '\n", msg_size, author);
191                 recv_report(sock);
192                 if ((msg_size = send(sock, content, strlen(
```

```
                content), 0)) < 0)
193             {
194                 perror("SEND␣file␣content␣failed");
195                 exit(1);
196             }
197             //printf("SEND  [%d bytes]: file content '%s'\n",
                    msg_size, content);
198             if (recv_report(sock) < 0)
199                 puts("!Such␣file␣already␣exist");
200             send_report(sock, SUCCESS);
201         }
202     }
203         gets(buffer);
204         if (!strcmp(command, "open"))
205         {
206             strcat(path, buffer);
207             if ((msg_size = send(sock, path, strlen(path), 0)
                    ) < 0)
208             {
209                 perror("SEND␣full␣path␣to␣file␣failed");
210                 exit(1);
211             }
212             //printf("SEND  [%d bytes]: full path to file '%s
                    '\n", msg_size, path);
213         }
214         else if (!strcmp(command, "find"))
215         {
216             if ((msg_size = send(sock, buffer, strlen(buffer)
                    , 0)) < 0)
217             {
218                 perror("SEND␣author␣to␣find␣failed");
219                 exit(1);
220             }
221             //printf("SEND  [%d bytes]: author to find '%s'\n
                    ", msg_size, buffer);
222         }
223         bzero(content, strlen(content));
224         if ((msg_size = recv(sock, content, sizeof(content),
                 0)) < 0)  // Receive the content of file
225         {
226             perror("RECV␣file␣or␣directory␣content␣failed");
227             exit(1);
228         }
229         //printf("RECV  [%d bytes]: file or directory
                content\n", msg_size);
230         output(content);
231         send_report(sock, SUCCESS);
232
233 }
```

```
234     close(sock);
235    return 0;
236 }
237
238 void ValidateArgs(int argc, char **argv)
239 {
240    int i;
241    for(i = 1; i < argc; i++)
242    {
243       if ((argv[i][0] == '-') || (argv[i][0] == '/'))
244       {
245          switch (tolower(argv[i][1]))
246          {
247          case 'p':
248             port = atoi(&argv[i][3]);
249             break;
250          case 'i':
251             interface = 1;
252             if (strlen(argv[i]) > 3)
253                strcpy(szAddress, &argv[i][3]);
254             break;
255          default:
256             usage();
257             break;
258          }
259       }
260    }
261 }
262
263 void usage()
264 {
265    printf("usage: server [-p:x] [-i:IP]\n\n");
266    printf(" -p:x Port number to listen on\n");
267    printf(" -i:str Interface to listen on\n");
268 }
269
270 int recv_report(int sock)
271 {
272    char status[SIZE_CMD];
273    int msg_size;
274    bzero(status, sizeof(status));
275    if ((msg_size = recv(sock, status, sizeof(status), 0)) <
       0)
276    {
277       perror("RECV report message failed");
278       exit(1);
279    }
280    //printf("RECV  [%d bytes]: report message  '%s'\n",
       msg_size, status);
```

```
281     return (!strcmp(status, SUCCESS) ? 0 : -1);
282 }
283
284 void send_report(int sock, char *status)
285 {
286     int msg_size;
287     if ((msg_size = send(sock, status, sizeof(status), 0)) <
            0)
288     {
289         perror("SEND report message failed");
290         exit(1);
291     }
292     //printf("SEND  [%d bytes]: report message  '%s'\n",
            msg_size, status);
293 }
294 void output(char *buffer)
295 {
296     int i;
297     for (i = 0; i < strlen(buffer); i++)
298         if (buffer[i] != '|')
299             printf("%c", buffer[i]);
300         else
301             printf("\n");
302     if (buffer[strlen(buffer) - 1] == '\n')
303         buffer[strlen(buffer) - 1] = '\0';
304 }
305
306
307 void add_article_to_system(int sock, char *path)
308 {
309     char buffer[SIZE_BUF];
310     char content[SIZE_CONTENT];
311     int msg_size;
312     printf("Current path is %s\n", path);
313     strcat(path, buffer);
314     if ((msg_size = send(sock, path, strlen(path), 0)) < 0)
315     {
316         perror("SEND full path to file failed");
317         exit(1);
318     }
319     //printf("SEND  [%d bytes]: full path to file '%s'\n",
            msg_size, path);
320
321     bzero(content, sizeof(content));
322     if ((msg_size = recv(sock, content, sizeof(content), 0)) <
             0)  // Receive the content of file
323     {
324         perror("RECV file or directory content failed");
325         exit(1);
```

```
326     }
327     //printf("RECV  [%d bytes]: file or directory content\n",
            msg_size);
328     output(content);
329 }
```

UDP Server

main.c

```
1  #include <stdio.h>
2  #include <errno.h>
3  #include <fcntl.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <dirent.h>
7  #include <libgen.h>
8  #include <pthread.h>
9  #include <stdbool.h>
10 #include <sys/stat.h>
11 #include <sys/types.h>
12 #include <sys/socket.h>
13 #include <netinet/in.h>
14 #include "article.h"
15
16 #define DEFAULT_PORT 5001
17 #define MAX_CONNECT  3
18 #define SIZE_CMD     5
19 #define SIZE_BUF     1024
20 #define SIZE_CONTENT 4096
21 #define SIZE_STR     128
22 #define SUCCESS      "000"
23 #define UNSUCCESS    "111"
24 #define START_PATH   "/home/ks/workspace/
       InformationSystem_ServerUDP/Information System/"
25
26 typedef struct
27 {
28     int socket_fd;
29     struct sockaddr_in *ptr_addr;
30 } P_socket;
31
32 pthread_t t1[MAX_CONNECT];
33 char szAddress[SIZE_STR];
34 int port = DEFAULT_PORT;
35 bool interface = 0;
36
```

```
37 int send_content(int sock, char *dir_name, struct sockaddr_in
       *ptr_addr);
38 int open_file(int sock, char *path, struct sockaddr_in *
       ptr_addr);
39 void sendPath_recvReport(int sock, char *path, struct
       sockaddr_in *ptr_addr);
40 void send_input_error(int sock, struct sockaddr_in *ptr_addr)
       ;
41 void send_report(int sock, char *status, struct sockaddr_in *
       ptr_addr);
42 void recv_report(int sock, struct sockaddr_in *ptr_addr);
43 void validateArgs(int argc, char **argv);
44 void *pthread_handler(void *ptr);
45 void usage();
46
47 int main(int argc, char **argv)
48 {
49    const int on = 1;
50    int sock, i;
51    struct sockaddr_in addr[MAX_CONNECT], server;
52    P_socket p_sock[MAX_CONNECT];
53    validateArgs(argc, argv);
54
55    if (interface)
56    {
57       server.sin_addr.s_addr = inet_addr(szAddress);
58       if (server.sin_addr.s_addr == INADDR_NONE)
59          usage();
60    }
61    else
62       server.sin_addr.s_addr = htonl(INADDR_ANY);
63
64    server.sin_family = AF_INET;
65    server.sin_port = htons(port);
66
67    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
68    {
69       perror("Socket␣is␣not␣created");
70       exit(1);
71    }
72    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)
       );
73
74    for (i = 0; i < MAX_CONNECT; i++)
75    {
76       bzero(&addr[i], sizeof(addr[i]));
77       p_sock[i].socket_fd = sock;
78       p_sock[i].ptr_addr = &addr[i];
79    }
```

```
80
81      if (bind(sock, (struct sockaddr *)&server, sizeof(server))
            < 0)
82      {
83          perror("Socket␣is␣not␣bound");
84          exit(1);
85      }
86
87      for(i = 0;i < MAX_CONNECT;i++)
88      {
89          if (pthread_create(&t1[i], NULL, (void *)&
                pthread_handler, (void *)&p_sock[i]) != 0)
90          {
91              perror("Thread␣creating");
92              exit(1);
93          }
94          pthread_join(t1[i], NULL);
95      }
96
97      close(sock);
98      return 0;
99  }
100
101 void *pthread_handler(void *ptr)
102 {
103
104      P_socket *data;
105      data = (P_socket *) ptr;
106      struct sockaddr_in addr = *(data->ptr_addr);
107
108      int sock = data->socket_fd;
109     int len = sizeof(addr);
110     int msg_size;
111     const char *invite_msg = ">␣␣␣";
112     const char *exit_msg = "Bye-bye!!!|";
113     char path[SIZE_BUF];
114     char name[SIZE_STR];
115     char buffer[SIZE_BUF];
116     char author[SIZE_STR];
117     char content[SIZE_CONTENT];
118
119     while(strcmp(buffer, ":start"))
120     {
121         bzero(buffer, sizeof(buffer));
122         if ((msg_size = recvfrom(sock, buffer, sizeof(buffer),
                0, (struct sockaddr*)&addr,  (socklen_t *)&len)) <
                0)
123         {
124             perror("RECV␣start␣message␣error");
```

```
125             exit(1);
126         }
127     }
128     printf("RECV␣␣[%d␣bytes]:␣start␣message␣'%s'\n", msg_size,
            buffer);
129     send_content(sock, START_PATH, &addr);
130     strcpy(path, START_PATH);
131
132     while(1)
133     {
134         if ((msg_size = sendto(sock, invite_msg, strlen(
                invite_msg), 0,  (struct sockaddr*)&addr, sizeof(
                addr))) < 0)
135         {
136             perror("SEND␣invitation␣message␣error");
137             exit(1);
138         }
139         printf("SEND␣␣[%d␣bytes]:␣invitation␣message␣'%s'\n",
                msg_size, invite_msg);
140         bzero(buffer, sizeof(buffer));
141         if ((msg_size = recvfrom(sock, buffer, sizeof(buffer),
                0, (struct sockaddr*)&addr,  (socklen_t *)&len)) <
                0)
142         {
143             perror("RECV␣command␣error");
144             exit(1);
145         }
146         printf("RECV␣␣[%d␣bytes]:␣command␣'%s'\n", msg_size,
                buffer);
147
148         if (!strcmp(buffer, ":exit"))
149         {
150             if ((msg_size = sendto(sock, exit_msg, strlen(
                    exit_msg), 0,  (struct sockaddr*)&addr, sizeof(
                    addr))) < 0)
151             {
152                 perror("SEND␣directory␣content␣error");
153                 exit(1);
154             }
155             printf("SEND␣␣[%d␣bytes]:␣directory␣content␣'%s'\n",
                     msg_size, exit_msg);
156             break;
157         }
158         if (strcmp(buffer, "find") && strcmp(buffer, "open") &&
                strcmp(buffer, "add"))
159         {
160             send_input_error(sock, &addr);
161             send_content(sock, path, &addr);
162             continue;
```

```
163         }
164         send_report(sock, SUCCESS, &addr);
165 
166         if (!strcmp(buffer, "open"))
167         {
168             bzero(path, sizeof(path));
169             if ((msg_size = recvfrom(sock, path, sizeof(path),
                    0, (struct sockaddr*)&addr,  (socklen_t *)&len))
                    < 0)
170             {
171                 perror("RECV path to file error");
172                 exit(1);
173             }
174             printf("RECV  [%d bytes]: path to file message '%s'\
                    n", msg_size, path);
175             open_file(sock, path, &addr);
176         }
177 
178         if (!strcmp(buffer, "find"))
179         {
180             bzero(author, sizeof(author));
181             if ((msg_size = recvfrom(sock, author, sizeof(author
                    ), 0, (struct sockaddr*)&addr,  (socklen_t *)&len
                    )) < 0)
182             {
183                 perror("RECV author to find error");
184                 exit(1);
185             }
186             printf("RECV  [%d bytes]: author to find '%s'\n",
                    msg_size, author);
187 
188             find_for_author(sock, path, author, &addr);
189             sendPath_recvReport(sock, path, &addr);
190         }
191 
192         if (!strcmp(buffer, "add"))
193         {
194             bzero(name, sizeof(name));
195             if ((msg_size = recvfrom(sock, name, sizeof(name),
                    0, (struct sockaddr*)&addr,  (socklen_t *)&len))
                    < 0)
196             {
197                 perror("RECV name error");
198                 exit(1);
199             }
200             printf("RECV  [%d bytes]: name'%s'\n", msg_size,
                    name);
201 
202             char *dir = strdup(path);
```

```
203             strcat(path, name);
204             strcat(path, ".txt");
205             if (check_file_existence(path) < 0)
206             {
207                 send_report(sock, UNSUCCESS, &addr);
208                 recv_report(sock, &addr);
209             }
210             else
211             {
212                 send_report(sock, SUCCESS, &addr);
213                 bzero(author, sizeof(author));
214                 if ((msg_size = recvfrom(sock, author, sizeof(
                        author), 0, (struct sockaddr*)&addr,  (
                        socklen_t *)&len)) < 0)
215                 {
216                     perror("RECV␣author␣error");
217                     exit(1);
218                 }
219                 printf("RECV␣␣[%d␣bytes]:␣author␣'%s'\n",
                        msg_size, author);
220                 send_report(sock, SUCCESS, &addr);
221                 bzero(content, sizeof(content));
222                 if ((msg_size = recvfrom(sock, content, sizeof(
                        content), 0, (struct sockaddr*)&addr,  (
                        socklen_t *)&len)) < 0)
223                 {
224                     perror("RECV␣content␣of␣file␣error");
225                     exit(1);
226                 }
227                 printf("RECV␣␣[%d␣bytes]:␣content␣of␣file␣'%s'\n"
                        , msg_size, content);
228                 strcat(name, "\n");
229                 strcat(author, "\n\n");
230                 if (add_article(path, name, author, content) < 0)
231                     send_report(sock, UNSUCCESS, &addr);
232                 else
233                     send_report(sock, SUCCESS, &addr);
234                 recv_report(sock, &addr);
235             }
236             send_content(sock, dir, &addr);
237             dirname(path);
238             if (path[strlen(path) - 1] != '/')
239                 strcat(path, "/");
240         }
241     }
242     pthread_exit(0);
243     return 0;
244 }
245
```

```
246 void validateArgs(int argc, char **argv)
247 {
248    int i;
249
250    for(i = 1; i < argc; i++)
251    {
252       if ((argv[i][0] == '-') || (argv[i][0] == '/'))
253       {
254          switch (tolower(argv[i][1]))
255          {
256          case 'p':
257             port = atoi(&argv[i][3]);
258             break;
259          case 'i':
260             interface = 1;
261             if (strlen(argv[i]) > 3)
262                strcpy(szAddress, &argv[i][3]);
263             break;
264          default:
265             usage();
266             break;
267          }
268       }
269    }
270 }
271
272 void usage()
273 {
274    printf("usage: server [-p:x] [-i:IP]\n\n");
275    printf(" -p:x Port number to listen on\n");
276    printf(" -i:str Interface to listen on\n");
277 }
278
279 void send_input_error(int sock, struct sockaddr_in *ptr_addr)
280 {
281    send_report(sock, UNSUCCESS, ptr_addr);
282    recv_report(sock, ptr_addr);
283 }
284
285 int send_content(int sock, char *dir_name, struct sockaddr_in
       *ptr_addr)
286 {
287    struct sockaddr_in addr;
288    addr = *ptr_addr;
289    char buffer[SIZE_BUF];
290    char *filename;
291    const char *delimiter = "----------------------|";
292    int msg_size;
293    DIR *dir = opendir(dir_name);
```

```
294     bzero(buffer,sizeof(buffer));
295     if(dir)
296     {
297         struct dirent *ent;
298         strcat(buffer, delimiter);
299         while((ent = readdir(dir)) != NULL)
300         {
301             filename = ent->d_name;
302             if (strcmp(filename, ".")==0)
303                 continue;
304             strcat(filename, "|");
305             strcat(buffer, filename);
306         }
307         closedir(dir);
308         strcat(buffer, delimiter);
309         if ((msg_size = sendto(sock, buffer, strlen(buffer), 0,
                (struct sockaddr*)&addr, sizeof(addr))) < 0)
310         {
311             perror("SEND␣directory␣content␣error");
312             exit(1);
313         }
314         printf("SEND␣␣[%d␣bytes]:␣directory␣content␣'%s'\n",
                msg_size, buffer);
315         bzero(filename,sizeof(filename));
316         strcpy(filename, dir_name);
317         if (!strcmp(basename(filename), "."))
318             dirname(dir_name);
319         else if (!strcmp(basename(filename), ".."))
320             dirname(dirname(dir_name));
321     }
322     else
323     {
324         puts(dir_name);
325         const char *err_msg = "!No␣such␣file␣or␣directory|";
326         if ((msg_size = sendto(sock, err_msg, strlen(err_msg),
                0,  (struct sockaddr*)&addr, sizeof(addr))) < 0)
327         {
328             perror("SEND␣no␣file␣or␣directory␣error");
329             exit(1);
330         }
331         printf("SEND␣␣[%d␣bytes]:␣no␣file␣or␣directory␣message␣
                '%s'\n", msg_size, buffer);
332         dirname(dir_name);
333     }
334     sendPath_recvReport(sock, dir_name, &addr);
335     return 0;
336 }
337
338 int open_file(int sock, char *path, struct sockaddr_in *
```

```
    ptr_addr)
{
    struct sockaddr_in addr;
    addr = *ptr_addr;
    FILE *fp;
    int msg_size;
    struct stat about_file;

    char buffer[SIZE_STR], text[SIZE_CONTENT];
    bzero(text, sizeof(text));
    bzero(buffer, sizeof(buffer));

    char *tmp = malloc(SIZE_BUF);
    strcpy(tmp, START_PATH);
    if (!strcmp(path, strcat(tmp, "..")))
    {
        send_content(sock, START_PATH, &addr);
        free(tmp);
        return 0;
    }
    free(tmp);
    if ((fp = fopen(path, "r")) == NULL)
    {
        perror("Opening of file error");
        send_content(sock, path, ptr_addr);
        return -1;
    }

    fstat(fileno(fp), &about_file);
    if ((about_file.st_mode & S_IFMT) != S_IFDIR)
    {
        int ch, i;
        for (i = 0; i < (sizeof(text) - sizeof(char)) && (ch =
            getc(fp)) != EOF; i++)
        {
            if (ch == '\n')
                ch = '|';
            text[i] = ch;
        }
        strcat(text, "|");
        if ((msg_size = sendto(sock, text, strlen(text), 0,  (
            struct sockaddr*)&addr, sizeof(addr))) < 0)
        {
            perror("SEND content of file error");
            exit(1);
        }
        printf("SEND  [%d bytes]: content of file \n%s\n\n",
            msg_size, text);

```

```
384         dirname(path);
385         sendPath_recvReport(sock, path, &addr);
386
387     }
388     else
389         send_content(sock, path, &addr);
390     fclose(fp);
391     return 0;
392
393 }
394
395 void sendPath_recvReport(int sock, char *path, struct
        sockaddr_in *ptr_addr)
396 {
397     struct sockaddr_in addr;
398     addr = *ptr_addr;
399     int msg_size;
400     recv_report(sock, ptr_addr);
401     if (path[strlen(path) - 1] != '/')
402         strcat(path, "/");
403     if ((msg_size = sendto(sock, path, strlen(path), 0,  (
        struct sockaddr*)&addr, sizeof(addr))) < 0)
404     {
405         perror("SEND␣current␣path␣error");
406         exit(1);
407     }
408     printf("SEND␣␣[%d␣bytes]:␣current␣path␣'%s'\n", msg_size,
        path);
409     recv_report(sock, ptr_addr);
410 }
411
412 void send_report(int sock, char *status, struct sockaddr_in *
        ptr_addr)
413 {
414     struct sockaddr_in addr;
415     addr = *ptr_addr;
416     int msg_size;
417     if ((msg_size = sendto(sock, status, strlen(status), 0,  (
        struct sockaddr*)&addr, sizeof(addr))) < 0)
418     {
419         perror("SEND␣report␣message␣error");
420         exit(1);
421     }
422     printf("SEND␣␣[%d␣bytes]:␣report␣message␣'%s'\n", msg_size
        , status);
423 }
424
425
426 void recv_report(int sock, struct sockaddr_in *ptr_addr)
```

```
427 {
428    struct sockaddr_in addr;
429    addr = *ptr_addr;
430    int len =  sizeof(addr);
431    char status[SIZE_CMD];
432    int msg_size;
433    bzero(status, sizeof(status));
434    if ((msg_size = recvfrom(sock, status, sizeof(status), 0,
          (struct sockaddr*)&addr,  (socklen_t *)&len)) < 0)
435    {
436       perror("RECV␣report␣message␣failed");
437       exit(1);
438    }
439    printf("RECV␣␣[%d␣bytes]:␣report␣message␣␣'%s'\n",
          msg_size, status);
440 }
```

article.h

```
1  /*
2   * article.h
3   *
4   *  Created on: Nov 7, 2014
5   *      Author: user
6   */
7
8  #ifndef ARTICLE_H_
9  #define ARTICLE_H_
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <string.h>
14 #include <errno.h>
15 #include <sys/types.h>
16 #include <sys/stat.h>
17 #include <dirent.h>
18 #include <sys/socket.h>
19 #include <netinet/in.h>
20 #include <libgen.h>
21
22
23
24 #define MAX_FILES 100
25 #define BUF_SIZE 128
26 #define MAX_SIZE 1024
27
28 typedef struct art
29 {
30    char filename[BUF_SIZE];
31    char title[BUF_SIZE];
```

```
32      char author[BUF_SIZE];
33  } Article;
34
35  int check_file_existence(char *dir_name);
36  int add_article(char *dir_name, char *name, char* author,
        char *content);
37  int find_for_author(int sock, char *dir_name, char *author,
        struct sockaddr_in *ptr_addr);
38  char *lower(char *str);
39
40  #endif /* ARTICLE_H_ */
```

article.c

```
1   /*
2    * article.c
3    *
4    *  Created on: Nov 7, 2014
5    *      Author: user
6    */
7
8   #include "article.h"
9
10  int add_article(char *dir_name, char *name, char* author,
        char *content)
11  {
12      FILE *fp;
13      if ((fp = fopen(dir_name, "r")) == NULL)
14          if (errno == ENOENT)
15              if ((fp = fopen(dir_name, "w+")) == NULL)
16              {
17                  perror("File creation error");
18                  return -2;
19              }
20              else
21              {
22                  fputs(name, fp);
23                  fputs(author, fp);
24                  fputs(content, fp);
25                  rewind(fp);
26                  close(fp);
27                  return 0;
28              }
29      close(fp);
30      return -1;
31  }
32
33  int check_file_existence(char *dir_name)
34  {
35      FILE *fp;
```

```
36     if ((fp = fopen(dir_name,  "r" )) == NULL)
37        if (errno == ENOENT)
38           return 0;
39     close(fp);
40     return -1;
41  }
42
43  int find_for_author(int sock, char *dir_name, char *author,
       struct sockaddr_in *ptr_addr)
44  {
45     struct sockaddr_in addr;
46     addr = *ptr_addr;
47     char buffer[BUF_SIZE];
48     char path[MAX_SIZE];
49     char *ptr;
50     const char *delimiter = "----------------------|";
51     int msg_size;
52     char *filename;
53     FILE *fp;
54     DIR *dir = opendir(dir_name);
55
56     bzero(buffer,sizeof(buffer));
57     struct stat about_file;
58     Article *arts=(Article *)malloc(MAX_FILES * sizeof(Article
          ));
59     int i, k = 0;
60     if(dir)
61     {
62        struct dirent *ent;
63        while((ent = readdir(dir)) != NULL)
64        {
65           strcpy(path, dir_name);
66           filename = ent->d_name;
67
68           if ((fp = fopen(strcat(path, filename), "r")) ==
                NULL)
69           {
70              printf("error␣%s\n", filename);
71              perror("Opening␣of␣file␣error");
72           }
73           fstat(fileno(fp), &about_file);
74           if ((about_file.st_mode & S_IFMT) != S_IFDIR)
75           {
76              for( i = 0; (ptr = fgets(buffer, sizeof(buffer),
                   fp)) != NULL && i <2; i++)
77              {
78
79                 if (i == 0)
80                    strcpy(arts[k].title, ptr);
```

```
81                else if (i == 1)
82                   strcpy(arts[k].author, ptr);
83                bzero(ptr, strlen(ptr));
84             }
85             strcpy(arts[k].filename, filename);
86             k++;
87          }
88          fclose(fp);
89       }
90       closedir(dir);
91       bzero(buffer, sizeof(buffer));
92       strcat(buffer, "Search␣results␣for␣author:␣");
93       strcat(buffer, author);
94       strcat(buffer, "|");
95       strcat(buffer, delimiter);
96       for (i = 0; k >= 0; --k)
97       {
98          if (strstr(lower(arts[k].author), lower(author)) !=
               NULL)
99          {
100            strcat(buffer, arts[k].author);
101            strcat(buffer, ":␣␣␣␣");
102            strcat(buffer, arts[k].filename);
103            strcat(buffer, "|");
104            i++;
105         }
106      }
107      strcat(buffer, delimiter);
108      if (i == 0)
109      {
110         bzero(buffer, sizeof(buffer));
111         strcat(buffer, "There␣are␣no␣articles␣of␣");
112         strcat(buffer, author);
113         strcat(buffer, "|");
114      }
115      if ((msg_size = sendto(sock, buffer, strlen(buffer), 0,
              (struct sockaddr*)&addr, sizeof(addr))) < 0)
116      {
117         perror("SEND␣found␣result␣error");
118         return -1;
119      }
120      printf("SEND␣␣␣[%d␣bytes]:␣found␣result␣'%s'\n",
            msg_size, buffer);
121      return 0;
122   }
123   free(arts);
124   return -1;
125 }
126
```

```
127 char *lower(char *str)
128 {
129    int i;
130    char *new = strdup(str);
131    for (i = 0; i < strlen(new); i++)
132       new[i] = tolower(new[i]);
133    return new;
134 }
```

UDP Client

main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <stdbool.h>
5  #include <sys/stat.h>
6  #include <dirent.h>
7  #include <string.h>
8  #include <fcntl.h>
9  #include <sys/socket.h>
10 #include <netinet/in.h>
11 #include <libgen.h>
12
13 #define SIZE_CMD 5
14 #define SIZE_ARG 50
15 #define SIZE_STR 128
16 #define SIZE_BUF 1024
17 #define SUCCESS "000"
18 #define UNSUCCESS "111"
19 #define SIZE_CONTENT 4096
20 #define DEFAULT_PORT 5001
21
22 void output(char *str);
23 void add_article_to_system(int sock, char *path);
24 int recv_report(int sock);
25 void send_report(int sock, char *status);
26 void ValidateArgs(int argc, char **argv);
27 void usage();
28
29
30 int port = DEFAULT_PORT;
31 bool interface = 0;
32 char szAddress[SIZE_STR];
33 struct sockaddr_in client;
34
35 int main(int argc, char **argv)
```

```
36 {
37    int sock;
38    int msg_size;
39    char name[SIZE_STR];
40    char path[SIZE_BUF];
41    char author[SIZE_STR];
42    char buffer[SIZE_BUF];
43    char command[SIZE_CMD];
44    char content[SIZE_CONTENT];
45
46    ValidateArgs(argc, argv);
47
48    if (interface)
49    {
50       client.sin_addr.s_addr = inet_addr(szAddress);
51       if (client.sin_addr.s_addr == INADDR_NONE)
52          usage();
53    }
54    else
55       client.sin_addr.s_addr = htonl(INADDR_ANY);
56    client.sin_family = AF_INET;
57    client.sin_port = htons(port);
58    int len = sizeof(client);
59
60    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
61    {
62       perror("Socket is not created");
63       exit(1);
64    }
65
66    bzero(buffer, sizeof(buffer));
67    while(strcmp(buffer, ":start"))
68    {
69
70       fgets(buffer, sizeof(buffer), stdin);
71       if (buffer[strlen(buffer) - 1] == '\n')
72          buffer[strlen(buffer) - 1] = '\0';
73       if ((msg_size = sendto(sock, buffer, strlen(buffer), 0,
             (struct sockaddr *)&client, sizeof(client))) < 0)
74       {
75          perror("SEND start message failed");
76          exit(1);
77       }
78    }
79    //printf("SEND  [%d bytes]: start message '%s'\n",
          msg_size, buffer);
80
81    bzero(buffer, sizeof(buffer));
82    if ((msg_size = recvfrom(sock, buffer, sizeof(buffer), 0,
```

```
        (struct sockaddr *)&client, (socklen_t *)&len)) < 0)
83  {
84      perror("RECV directory content failed");
85      exit(1);
86  }
87  //printf("RECV  [%d bytes]: directory content\n", msg_size
        );
88  output(buffer);
89  send_report(sock, SUCCESS);
90
91  while(1)
92  {
93      bzero(path, sizeof(path));
94      if ((msg_size = recvfrom(sock, path, sizeof(path), 0, (
            struct sockaddr *)&client, (socklen_t *)&len)) < 0)
95      {
96          perror("RECV current path failed");
97          exit(1);
98      }
99      //printf("RECV  [%d bytes]: current path '%s'\n",
            msg_size, path);
100
101     send_report(sock, SUCCESS);
102
103     bzero(buffer, sizeof(buffer));
104     if ((msg_size = recvfrom(sock, buffer, sizeof(buffer),
            0, (struct sockaddr *)&client, (socklen_t *)&len)) <
             0)
105     {
106         perror("RECV invitation message failed");
107         exit(1);
108     }
109     //printf("RECV  [%d bytes]: invitation message\n",
            msg_size);
110     output(buffer);
111     char space;
112     bzero(name, sizeof(name));
113     bzero(buffer, sizeof(buffer));
114     bzero(author, sizeof(author));
115     bzero(command, sizeof(command));
116     bzero(content, sizeof(content));
117     scanf("%5s%1c", command, &space);
118     if ((msg_size = sendto(sock, command, strlen(command),
            0, (struct sockaddr *)&client, sizeof(client))) < 0)
119     {
120         perror("SEND command failed");
121         exit(1);
122     }
123     //printf("SEND  [%d bytes]: command '%s'\n", msg_size,
```

```
            path);
124
125         if (!strcmp(command, ":exit"))
126         {
127             bzero(buffer, sizeof(buffer));
128             if ((msg_size = recvfrom(sock, buffer, sizeof(buffer
                    ), 0, (struct sockaddr *)&client, (socklen_t *)&
                    len)) < 0) // Receive the content of file
129             {
130                 perror("RECV file or directory content failed");
131                 exit(1);
132             }
133             //printf("RECV  [%d bytes]: file or directory
                    content\n", msg_size);
134             output(buffer);
135             break;
136         }
137
138         if (recv_report(sock) < 0)
139         {
140             puts("!No such command");
141             send_report(sock, SUCCESS);
142         }
143
144         if (!strcmp(command, "add"))
145         {
146             char str[SIZE_ARG];
147             fgets(name, sizeof(name), stdin);
148             if (name[strlen(name) - 1] == '\n')
149                 name[strlen(name) - 1] = '\0';
150             if ((msg_size = sendto(sock, name, strlen(name), 0,
                    (struct sockaddr *)&client, sizeof(client))) < 0)
151             {
152                 perror("SEND command failed");
153                 exit(1);
154             }
155             //printf("SEND  [%d bytes]: title of article '%s'\n
                    ", msg_size, name);
156             if (recv_report(sock) < 0)
157             {
158                 puts("!Such file already exist");
159                 send_report(sock, SUCCESS);
160             }
161             else
162             {
163                 int length = sizeof(content) - sizeof(author) -
                        sizeof(name);
164                 printf("Input author: ");
165                 fgets(author, sizeof(author), stdin);
```

```
166             if (author[strlen(author) - 1] == '\n')
167                 author[strlen(author) - 1] = '\0';
168             printf("name's␣read:␣%s␣[%d␣bytes]\n", name,
                    msg_size);
169             printf("author's␣read:␣%s␣[%d␣bytes]\n", author,
                    msg_size);
170
171             puts("Put␣content:");
172             printf("[%d␣of␣%d]␣␣", (strlen(content)+strlen(
                    str)), length);
173             while (fgets(str, sizeof(str), stdin) != NULL)
174             {
175                 if (!strncmp(":end", str, strlen(":end")))
176                     break;
177                 if ((strlen(content)+strlen(str)) > length)
178                 {
179                     puts("!Text␣size␣will␣not␣allow.␣Type␣less␣
                            or␣:end");
180                     bzero(str, strlen(str));
181                     printf("[%d␣of␣%d]␣␣", (strlen(content)+
                            strlen(str)), length);
182                     //__fpurge(stdin);
183                 }
184                 strcat(content, str);
185                 bzero(str, strlen(str));
186             }
187             if ((msg_size = sendto(sock, author, strlen(
                    author), 0, (struct sockaddr *)&client, sizeof
                    (client))) < 0)
188             {
189                 perror("SEND␣author␣of␣article␣failed");
190                 exit(1);
191             }
192             //printf("SEND  [%d bytes]: author of article '%s
                    '\n", msg_size, author);
193             recv_report(sock);
194
195             if ((msg_size = sendto(sock, content, strlen(
                    content), 0, (struct sockaddr *)&client,
                    sizeof(client))) < 0)
196             {
197                 perror("SEND␣file␣content␣failed");
198                 exit(1);
199             }
200             //printf("SEND  [%d bytes]: file content '%s'\n",
                     msg_size, content);
201             if (recv_report(sock) < 0)
202                 puts("!Such␣file␣already␣exist");
203             send_report(sock, SUCCESS);
```

```
204             }
205         }
206 
207         gets(buffer);
208         if (!strcmp(command, "open"))
209         {
210             strcat(path, buffer);
211             if ((msg_size = sendto(sock, path, strlen(path), 0,
                    (struct sockaddr *)&client, sizeof(client))) < 0)
212             {
213                 perror("SEND full path to file failed");
214                 exit(1);
215             }
216             //printf("SEND  [%d bytes]: full path to file '%s'\n
                    ", msg_size, path);
217         }
218         else if (!strcmp(command, "find"))
219         {
220             if ((msg_size = sendto(sock, buffer, strlen(buffer),
                     0, (struct sockaddr *)&client, sizeof(client)))
                    < 0)
221             {
222                 perror("SEND author to find failed");
223                 exit(1);
224             }
225             //printf("SEND  [%d bytes]: author to find '%s'\n",
                    msg_size, buffer);
226         }
227 
228         bzero(content, sizeof(content));
229         if ((msg_size = recvfrom(sock, content, sizeof(content)
                , 0, (struct sockaddr *)&client, (socklen_t *)&len))
                 < 0)  // Receive the content of file
230         {
231             perror("RECV file or directory content failed");
232             exit(1);
233         }
234         //printf("RECV  [%d bytes]: file or directory content\n
                ", msg_size);
235         output(content);
236         send_report(sock, SUCCESS);
237     }
238      close(sock);
239     return 0;
240 }
241 
242 void ValidateArgs(int argc, char **argv)
243 {
244     int i;
```

```
245
246     for(i = 1; i < argc; i++)
247     {
248        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
249        {
250           switch (tolower(argv[i][1]))
251           {
252           case 'p':
253              port = atoi(&argv[i][3]);
254              break;
255           case 'i':
256              interface = 1;
257              if (strlen(argv[i]) > 3)
258                 strcpy(szAddress, &argv[i][3]);
259              break;
260           default:
261              usage();
262              break;
263           }
264        }
265     }
266  }
267
268  void usage()
269  {
270     printf("usage: server [-p:x] [-i:IP]\n\n");
271     printf(" -p:x Port number to listen on\n");
272     printf(" -i:str Interface to listen on\n");
273  }
274
275  int recv_report(int sock)
276  {
277     char status[SIZE_CMD];
278     int msg_size;
279     int len = sizeof(client);
280     bzero(status, sizeof(status));
281     if ((msg_size = recvfrom(sock, status, sizeof(status), 0,
           (struct sockaddr *)&client, (socklen_t *)&len)) < 0)
282     {
283        perror("RECV report message failed");
284        exit(1);
285     }
286     //printf("RECV  [%d bytes]: report message  '%s'\n",
           msg_size, status);
287     return (!strcmp(status, SUCCESS) ? 0 : -1);
288  }
289
290  void send_report(int sock, char *status)
291  {
```

```
292     int msg_size;
293     if ((msg_size = sendto(sock, status, sizeof(status), 0, (
            struct sockaddr *)&client, sizeof(client))) < 0)
294     {
295         perror("SEND report message failed");
296         exit(1);
297     }
298     //printf("SEND  [%d bytes]: report message  '%s'\n",
            msg_size, status);
299 }
300
301 void output(char *buffer)
302 {
303     int i;
304     for (i = 0; i < strlen(buffer); i++)
305         if (buffer[i] != '|')
306             printf("%c", buffer[i]);
307         else
308             printf("\n");
309     if (buffer[strlen(buffer) - 1] == '\n')
310         buffer[strlen(buffer) - 1] = '\0';
311 }
312
313
314 void add_article_to_system(int sock, char *path)
315 {
316     char buffer[SIZE_BUF];
317     char content[SIZE_CONTENT];
318     int msg_size;
319     int len = sizeof(client);
320     printf("Current path is %s\n", path);
321     strcat(path, buffer);
322     if ((msg_size = sendto(sock, path, strlen(path), 0, (
            struct sockaddr *)&client, sizeof(client))) < 0)
323     {
324         perror("SEND full path to file failed");
325         exit(1);
326     }
327     //printf("SEND  [%d bytes]: full path to file '%s'\n",
            msg_size, path);
328
329     bzero(content, sizeof(content));
330     if ((msg_size = recvfrom(sock, content, sizeof(content),
            0,  (struct sockaddr *)&client, (socklen_t *)&len)) <
            0) // Receive the content of file
331     {
332         perror("RECV file or directory content failed");
333         exit(1);
334     }
```

```
335     //printf("RECV  [%d bytes]: file or directory content\n",
            msg_size);
336     output(content);
337 }
```

WINDOWS

TCP Server

main.c

```
1  #include <winsock2.h>
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <dirent.h>
6  #include <string.h>
7  #include <sys/types.h>
8  #include <sys/stat.h>
9
10 #define DEFAULT_PORT 5001
11 #define SIZE_CMD 5
12 #define SIZE_BUF 1024
13 #define SIZE_CONTENT 4096
14 #define SIZE_STR 128
15 #define MAX_FILES 20
16 #define MAX_CONNECT 3
17 #define SUCCESS "000"
18 #define UNSUCCESS "111"
19 #define START_PATH "C:/Users/Kseniya/workspace/test_server/
       Information System/"
20
21 int port = DEFAULT_PORT;
22 BOOL binterface = 0;
23 char szAddress[SIZE_STR];
24
25 int send_content(SOCKET sock, char *dir_name);
26 int open_file(SOCKET sock, char *path);
27 void sendPath_recvReport(SOCKET sock, char *path);
28 void send_input_error(SOCKET sock);
29 void send_report(SOCKET sock, char *status);
30 void recv_report(SOCKET sock);
31 void ValidateArgs(int argc, char **argv);
32 void usage();
33 DWORD WINAPI ClientThread(LPVOID lpParam);
34
35 int main(int argc, char **argv)
36 {
```

```
37    WSADATA        wsd;
38    SOCKET         sock,
39    sClient;
40    int            iAddrSize;
41    HANDLE         hThread;
42    DWORD          dwThreadId;
43    struct sockaddr_in local,
44    client;
45
46    ValidateArgs(argc, argv);
47    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
48    {
49       printf("Failed to load Winsock!\n");
50       return 1;
51    }
52    sock = socket(AF_INET, SOCK_STREAM, IPPROTO_IP);
53    if (sock == SOCKET_ERROR)
54    {
55       printf("socket() failed: %d\n", WSAGetLastError());
56       return 1;
57    }
58    if (binterface)
59    {
60       local.sin_addr.s_addr = inet_addr(szAddress);
61       if (local.sin_addr.s_addr == INADDR_NONE)
62       usage();
63       }
64    else
65       local.sin_addr.s_addr = htonl(INADDR_ANY);
66    local.sin_family = AF_INET;
67    local.sin_port = htons(port);
68
69    if (bind(sock, (struct sockaddr *)&local,
70    sizeof(local)) == SOCKET_ERROR)
71    {
72       printf("bind() failed: %d\n", WSAGetLastError());
73       return 1;
74    }
75    listen(sock, 8);
76
77    while (1)
78    {
79       iAddrSize = sizeof(client);
80       sClient = accept(sock, (struct sockaddr *)&client,
81       &iAddrSize);
82       if (sClient == INVALID_SOCKET)
83       {
84          printf("accept() failed: %d\n", WSAGetLastError());
85          break;
```

```
86          }
87          printf("Accepted␣client:␣%s:%d\n",
88          inet_ntoa(client.sin_addr), ntohs(client.sin_port));
89
90          hThread = CreateThread(NULL, 0, ClientThread,
91          (LPVOID)sClient, 0, &dwThreadId);
92          if (hThread == NULL)
93          {
94              printf("CreateThread()␣failed:␣%d\n", GetLastError()
                    );
95              break;
96          }
97          CloseHandle(hThread);
98      }
99      closesocket(sock);
100
101     WSACleanup();
102     return 0;
103 }
104
105 void send_input_error(SOCKET sock)
106 {
107     send_report(sock, UNSUCCESS);
108     recv_report(sock);
109 }
110
111 int send_content(SOCKET sock, char *dir_name)
112 {
113     char buffer[SIZE_BUF];
114     char *filename;
115     const char *delimiter = "----------------------|";
116     int msg_size;
117     DIR *dir = opendir(dir_name);
118     memset(buffer, 0, sizeof(buffer));
119     if(dir)
120     {
121         struct dirent *ent;
122         strcat(buffer, delimiter);
123         while((ent = readdir(dir)) != NULL)
124         {
125             filename = ent->d_name;
126             if (strcmp(filename, ".")==0)
127                 continue;
128             strcat(filename, "|");
129             strcat(buffer, filename);
130         }
131         closedir(dir);
132         strcat(buffer, delimiter);
133         if ((msg_size = send(sock, buffer, strlen(buffer), 0))
```

```
            == SOCKET_ERROR)
134         {
135             printf("SEND␣directory␣content␣error:␣%d\n",
                    WSAGetLastError());
136             exit(1);
137         }
138         printf("SEND␣␣[%d␣bytes]:␣directory␣content␣'%s'\n",
                msg_size, buffer);
139         memset(filename, 0,sizeof(filename));
140         strcpy(filename, dir_name);
141         if (!strcmp(basename(filename), "."))
142             dirname(dir_name);
143         else if (!strcmp(basename(filename), ".."))
144             dirname(dirname(dir_name));
145     }
146     else
147     {
148         const char *err_msg = "!No␣such␣file␣or␣directory|";
149         if ((msg_size = send(sock, err_msg, strlen(err_msg), 0)
                ) == SOCKET_ERROR)
150         {
151             printf("SEND␣no␣file␣or␣directory␣error:␣%d\n",
                    WSAGetLastError());
152             exit(1);
153         }
154         printf("SEND␣␣[%d␣bytes]:␣no␣file␣or␣directory␣message␣
                '%s'\n", msg_size, buffer);
155         dirname(dir_name);
156     }
157     sendPath_recvReport(sock, dir_name);
158     return 0;
159 }
160 DWORD WINAPI ClientThread(LPVOID lpParam)
161 {
162     SOCKET        sock=(SOCKET)lpParam;
163     int msg_size;
164     const char *invite_msg = ">␣␣␣";
165     const char *exit_msg = "Bye-bye!!!|";
166     char path[SIZE_BUF];
167     char name[SIZE_STR];
168     char buffer[SIZE_BUF];
169     char author[SIZE_STR];
170     char content[SIZE_CONTENT];
171     while(strcmp(buffer, ":start"))
172     {
173         memset(buffer, 0, sizeof(buffer));
174         if ((msg_size = recv(sock, buffer, sizeof(buffer), 0))
                == SOCKET_ERROR)
175         {
```

```
176             printf("Receive : start_msg failed: %d\n",
                    WSAGetLastError());
177             exit(1);
178         }
179     }
180     buffer[msg_size] = '\0';
181     send_content(sock, START_PATH);
182     strcpy(path, START_PATH);
183
184     while(1)
185     {
186         if ((msg_size = send(sock, invite_msg, strlen(
                invite_msg), 0)) == SOCKET_ERROR)
187         {
188             printf("SEND invitation message error: %d\n",
                    WSAGetLastError());
189             exit(1);
190         }
191         printf("SEND  [%d bytes]: invitation message '%s'\n",
                msg_size, invite_msg);
192
193         memset(buffer, 0, sizeof(buffer));
194         if ((msg_size = recv(sock, buffer, sizeof(buffer), 0))
                == SOCKET_ERROR)
195         {
196             printf("RECV command error: %d\n", WSAGetLastError()
                    );
197             exit(1);
198         }
199         printf("RECV  [%d bytes]: command '%s'\n", msg_size,
                buffer);
200         if (!strcmp(buffer, ":exit"))
201         {
202             if ((msg_size = send(sock, exit_msg, strlen(exit_msg
                    ), 0)) == SOCKET_ERROR)
203             {
204                 printf("SEND directory content error: %d\n",
                        WSAGetLastError());
205                 exit(1);
206             }
207             printf("SEND  [%d bytes]: directory content '%s'\n",
                     msg_size, exit_msg);
208             break;
209         }
210         if (strcmp(buffer, "find") && strcmp(buffer, "open") &&
                 strcmp(buffer, "add"))
211         {
212             send_input_error(sock);
213             send_content(sock, path);
```

```
214             continue;
215         }
216         send_report(sock, SUCCESS);
217         if (!strcmp(buffer, "open"))
218         {
219             memset(path, 0, sizeof(path));
220             if ((msg_size = recv(sock, path, sizeof(path), 0))
                    == SOCKET_ERROR)
221             {
222                 printf("RECV path to file error: %d\n",
                        WSAGetLastError());
223                 exit(1);
224             }
225             printf("RECV  [%d bytes]: path to file message '%s'\
                    n", msg_size, path);
226             open_file(sock, path);
227         }
228         if (!strcmp(buffer, "find"))
229         {
230             memset(author, 0,sizeof(author));
231             if ((msg_size = recv(sock, author, sizeof(author),
                    0)) == SOCKET_ERROR)
232             {
233                 printf("RECV author to find error: %d\n",
                        WSAGetLastError());
234                 exit(1);
235             }
236             printf("RECV  [%d bytes]: author to find '%s'\n",
                    msg_size, author);
237             find_for_author(sock, path, author);
238             sendPath_recvReport(sock, path);
239         }
240         if (!strcmp(buffer, "add"))
241         {
242             memset(name, 0,sizeof(name));
243             if ((msg_size = recv(sock, name, sizeof(name), 0))
                    == SOCKET_ERROR)
244             {
245                 printf("RECV name error: %d\n", WSAGetLastError()
                        );
246                 exit(1);
247             }
248             printf("RECV  [%d bytes]: name '%s'\n", msg_size,
                    name);
249             char *dir = strdup(path);
250             strcat(path, name);
251             strcat(path, ".txt");
252             if (check_file_existence(path) < 0)
253             {
```

```
254             send_report(sock, UNSUCCESS);
255             recv_report(sock);
256         }
257         else
258         {
259             send_report(sock, SUCCESS);
260             memset(author,0, sizeof(author));
261             if ((msg_size = recv(sock, author, sizeof(author)
                    , 0)) == SOCKET_ERROR)
262             {
263                 printf("RECV␣author␣error:␣%d\n",
                        WSAGetLastError());
264                 exit(1);
265             }
266             printf("RECV␣␣[%d␣bytes]:␣author␣'%s'\n",
                    msg_size, author);
267             send_report(sock, SUCCESS);
268             memset(content, 0,sizeof(content));
269             if ((msg_size = recv(sock, content, sizeof(
                    content), 0)) == SOCKET_ERROR)
270             {
271                 printf("RECV␣content␣of␣file␣error:␣%d\n",
                        WSAGetLastError());
272                 exit(1);
273             }
274             printf("RECV␣␣[%d␣bytes]:␣content␣of␣file␣'%s'\n"
                    , msg_size, content);
275             strcat(name, "\n");
276             strcat(author, "\n\n");
277             if (add_article(path, name, author, content) < 0)
278                 send_report(sock, UNSUCCESS);
279             else
280                 send_report(sock, SUCCESS);
281             recv_report(sock);
282         }
283         send_content(sock, dir);
284         dirname(path);
285         if (path[strlen(path) - 1] != '/')
286             strcat(path, "/");
287     }
288 }
289 return 0;
290 }
291
292 int open_file(SOCKET sock, char *path)
293 {
294     FILE *fp;
295     int msg_size;
296     struct stat about_file;
```

```
297
298     char buffer[SIZE_STR], text[SIZE_CONTENT];
299     memset(text, 0, sizeof(text));
300     memset(buffer, 0, sizeof(buffer));
301
302     char *tmp = malloc(SIZE_BUF);
303     strcpy(tmp, START_PATH);
304     if (!strcmp(path, strcat(tmp, "..")))
305     {
306         send_content(sock, START_PATH);
307         free(tmp);
308         return 0;
309     }
310     free(tmp);
311     if ((fp = fopen(path, "r")) == NULL)
312     {
313         printf("Opening␣of␣file␣error:␣%d\n", WSAGetLastError()
                );
314         send_content(sock, path);
315         return -1;
316     }
317
318     fstat(fileno(fp), &about_file);
319     if ((about_file.st_mode & S_IFMT) != S_IFDIR)
320     {
321         int ch, i;
322         for (i = 0; i < (sizeof(text) - sizeof(char)) && (ch =
                getc(fp)) != EOF; i++)
323         {
324             if (ch == '\n')
325                 ch = '|';
326             text[i] = ch;
327         }
328         strcat(text, "|");
329         if ((msg_size = send(sock, text, strlen(text), 0)) ==
                SOCKET_ERROR)
330         {
331             printf("SEND␣content␣of␣file␣error:␣%d\n",
                    WSAGetLastError());
332             exit(1);
333         }
334         printf("SEND␣␣[%d␣bytes]:␣content␣of␣file␣\n%s\n\n",
                msg_size, text);
335
336         dirname(path);
337         sendPath_recvReport(sock, path);
338
339     }
340     else
```

```
341         send_content(sock, path);
342     fclose(fp);
343     return 0;
344 
345 }
346 
347 void sendPath_recvReport(SOCKET sock, char *path)
348 {
349     int msg_size;
350     recv_report(sock);
351     if (path[strlen(path) - 1] != '/')
352         strcat(path, "/");
353     if ((msg_size = send(sock, path, strlen(path), 0)) ==
        SOCKET_ERROR)
354     {
355         printf("SEND␣current␣path␣error:␣%d\n", WSAGetLastError
            ());
356         exit(1);
357     }
358     printf("SEND␣␣[%d␣bytes]:␣current␣path␣'%s'\n", msg_size,
        path);
359     recv_report(sock);
360 }
361 
362 void send_report(SOCKET sock, char *status)
363 {
364     int msg_size;
365     if ((msg_size = send(sock, status, strlen(status), 0)) ==
        SOCKET_ERROR)
366     {
367         printf("SEND␣report␣message␣error:␣%d\n",
            WSAGetLastError());
368         exit(1);
369     }
370     printf("SEND␣␣[%d␣bytes]:␣report␣message␣'%s'\n", msg_size
        , status);
371 }
372 
373 
374 void recv_report(SOCKET sock)
375 {
376     char status[SIZE_CMD];
377     int msg_size;
378     memset(status, 0, sizeof(status));
379     if ((msg_size = recv(sock, status, sizeof(status), 0)) ==
        SOCKET_ERROR)
380     {
381         printf("RECV␣report␣message␣failed:␣%d\n",
            WSAGetLastError());
```

```
382         exit(1);
383     }
384     printf("RECV␣␣[%d␣bytes]:␣report␣message␣␣'%s'\n",
            msg_size, status);
385 }
386
387 void usage()
388 {
389 printf("usage:␣server␣[-p:x]␣[-i:IP]\n\n");
390 printf("-p:x␣␣␣␣␣␣Port␣number␣to␣listen␣on\n");
391 printf("-i:str␣␣␣␣Interface␣to␣listen␣on\n");
392 ExitProcess(1);
393 }
394
395 void ValidateArgs(int argc, char **argv)
396 {
397     int i;
398
399     for(i = 1; i < argc; i++)
400     {
401         if ((argv[i][0] == '-') || (argv[i][0] == '/'))
402         {
403             switch (tolower(argv[i][1]))
404             {
405             case 'p':
406                 port = atoi(&argv[i][3]);
407                 break;
408             case 'i':
409                 binterface = 1;
410                 if (strlen(argv[i]) > 3)
411                     strcpy(szAddress, &argv[i][3]);
412                 break;
413             default:
414                 usage();
415                 break;
416             }
417         }
418     }
419 }
```

article.h

```
1 #ifndef ARTICLE_H_
2 #define ARTICLE_H_
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <string.h>
7 #include <errno.h>
8 #include <sys/types.h>
```

```
9  #include <sys/stat.h>
10 #include <dirent.h>
11 #include <winsock2.h>
12 #include <libgen.h>
13
14 #define MAX_FILES 100
15 #define BUF_SIZE 128
16 #define MAX_SIZE 1024
17
18 typedef struct art
19 {
20    char filename[BUF_SIZE];
21    char title[BUF_SIZE];
22    char author[BUF_SIZE];
23 } Article;
24
25 int check_file_existence(char *dir_name);
26 int add_article(char *dir_name, char *name, char* author,
      char *content);
27 int find_for_author(SOCKET sock, char *dir_name, char *author
      );
28 char *lower(char *str);
29
30 #endif /* ARTICLE_H_ */
```

article.c

```
1  /*
2   * article.c
3   *
4   *  Created on: Nov 7, 2014
5   *      Author: user
6   */
7
8  #include "article.h"
9
10 int add_article(char *dir_name, char *name, char* author,
      char *content)
11 {
12    FILE *fp;
13    if ((fp = fopen(dir_name, "r" )) == NULL)
14       if (errno == ENOENT)
15          if ((fp = fopen(dir_name, "w+" )) == NULL)
16          {
17             perror("File creation error");
18             return -2;
19          }
20          else
21          {
22             fputs(name, fp);
```

```
23              fputs(author, fp);
24              fputs(content, fp);
25              rewind(fp);
26              close(fp);
27              return 0;
28          }
29    close(fp);
30    return -1;
31 }
32
33 int check_file_existence(char *dir_name)
34 {
35    FILE *fp;
36    if ((fp = fopen(dir_name,  "r" )) == NULL)
37       if (errno == ENOENT)
38          return 0;
39    close(fp);
40    return -1;
41 }
42
43 int find_for_author(SOCKET sock, char *dir_name, char *author
      )
44 {
45    char buffer[BUF_SIZE];
46    char path[MAX_SIZE];
47    char *ptr;
48    const char *delimiter = "----------------------|";
49    int msg_size;
50    char *filename;
51    FILE *fp;
52    DIR *dir = opendir(dir_name);
53    memset(buffer,0, sizeof(buffer));
54    struct stat about_file;
55    Article *arts=(Article *)malloc(MAX_FILES * sizeof(Article
         ));
56    int i, k = 0;
57    if(dir)
58    {
59       struct dirent *ent;
60       while((ent = readdir(dir)) != NULL)
61       {
62          strcpy(path, dir_name);
63          filename = ent->d_name;
64          if ((fp = fopen(strcat(path, filename), "r")) ==
               NULL)
65          {
66             printf("error␣%s\n", filename);
67             perror("Opening␣of␣file␣error");
68          }
```

```
69          fstat(fileno(fp), &about_file);
70          if ((about_file.st_mode & S_IFMT) != S_IFDIR)
71          {
72             for( i = 0; (ptr = fgets(buffer, sizeof(buffer),
                fp)) != NULL && i <2; i++)
73             {
74                if (i == 0)
75                   strcpy(arts[k].title, ptr);
76                else if (i == 1)
77                   strcpy(arts[k].author, ptr);
78                memset(ptr, 0, strlen(ptr));
79             }
80             strcpy(arts[k].filename, filename);
81             k++;
82          }
83          fclose(fp);
84       }
85       closedir(dir);
86       memset(buffer, 0, sizeof(buffer));
87       strcat(buffer, "Search␣results␣for␣author:␣");
88       strcat(buffer, author);
89       strcat(buffer, "|");
90       strcat(buffer, delimiter);
91       for (i = 0; k >= 0; --k)
92       {
93          if (strstr(lower(arts[k].author), lower(author)) !=
             NULL)
94          {
95             strcat(buffer, arts[k].author);
96             strcat(buffer, ":␣␣␣␣␣");
97             strcat(buffer, arts[k].filename);
98             strcat(buffer, "|");
99             i++;
100         }
101      }
102      strcat(buffer, delimiter);
103      if (i == 0)
104      {
105         memset(buffer, 0, sizeof(buffer));
106         strcat(buffer, "There␣are␣no␣articles␣of␣");
107         strcat(buffer, author);
108         strcat(buffer, "|");
109      }
110      if ((msg_size = send(sock, buffer, strlen(buffer), 0))
         == SOCKET_ERROR)
111      {
112         printf("SEND␣found␣result␣error:␣%d\n",
             WSAGetLastError());
113         return -1;
```

```
114         }
115         printf("SEND␣␣[%d␣bytes]:␣found␣result␣'%s'\n",
                msg_size, buffer);
116         return 0;
117     }
118     free(arts);
119     return -1;
120 }
121
122
123 char *lower(char *str)
124 {
125     int i;
126     char *new = strdup(str);
127     for (i = 0; i < strlen(new); i++)
128         new[i] = tolower(new[i]);
129     return new;
130 }
```

TCP Client

main.c

```
1  // Module Name: Client.c
2  //
3  // Description:
4  //     This sample is the echo client. It connects to the TCP
       server,
5  //     sends data, and reads data back from the server.
6  //
7  // Compile:
8  //     cl -o Client Client.c ws2_32.lib
9  //
10 // Command Line Options:
11 //     client [-p:x] [-s:IP] [-n:x] [-o]
12 //             -p:x       Remote port to send to
13 //             -s:IP      Server's IP address or hostname
14 //             -n:x       Number of times to send message
15 //             -o         Send messages only; don't receive
16 //
17 #include <winsock2.h>
18 #include <stdio.h>
19 #include <stdlib.h>
20 #include <sys/types.h>
21 #include <stdbool.h>
22 #include <sys/stat.h>
23 #include <dirent.h>
24 #include <string.h>
```

```
25 #include <fcntl.h>
26 #include <libgen.h>
27
28 #define SIZE_CMD 5
29 #define SIZE_ARG 50
30 #define SIZE_STR 128
31 #define SIZE_BUF 1024
32 #define SUCCESS "000"
33 #define UNSUCCESS "111"
34 #define SIZE_CONTENT 4096
35 #define DEFAULT_PORT 5001
36
37 int port = DEFAULT_PORT;
38 bool binterface = 0;
39 char szAddress[SIZE_STR];
40
41 int recv_report(SOCKET sock);
42 void send_report(SOCKET sock, char *status);
43 void ValidateArgs(int argc, char **argv);
44 void usage();
45 void output(char *buffer);
46
47
48 int main(int argc, char **argv)
49 {
50    WSADATA wsd;
51    SOCKET sock;
52    int msg_size;
53    char name[SIZE_STR];
54    char path[SIZE_BUF];
55    char author[SIZE_STR];
56    char buffer[SIZE_BUF];
57    char command[SIZE_CMD];
58    char content[SIZE_CONTENT];
59    struct sockaddr_in client;
60
61    ValidateArgs(argc, argv);
62    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
63    {
64       printf("Failed to load Winsock library!\n");
65       return 1;
66    }
67    if (binterface)
68       {
69       client.sin_addr.s_addr = inet_addr(szAddress);
70          if (client.sin_addr.s_addr == INADDR_NONE)
71             usage();
72       }
73       else
```

```
74          client.sin_addr.s_addr = htonl(INADDR_ANY);
75      client.sin_family = AF_INET;
76      client.sin_port = htons(port);
77
78      if ((sock = socket(AF_INET, SOCK_STREAM, 0)) ==
            SOCKET_ERROR)
79      {
80          printf("Create␣socket␣failed:␣%d\n", WSAGetLastError())
                ;
81          exit(1);
82      }
83      if (connect(sock, (struct sockaddr *)&client, sizeof(
            client))  == SOCKET_ERROR)
84      {
85          printf("Connect␣failed:␣%d\n", WSAGetLastError());
86          exit(1);
87      }
88      memset(buffer, 0, sizeof(buffer));
89      while(strcmp(buffer, ":start"))
90      {
91
92          fgets(buffer, sizeof(buffer), stdin);
93          if (buffer[strlen(buffer) - 1] == '\n')
94              buffer[strlen(buffer) - 1] = '\0';
95          if ((msg_size = send(sock, buffer, strlen(buffer), 0))
                == SOCKET_ERROR)
96          {
97              printf("SEND␣start␣message␣failed:␣%d\n",
                    WSAGetLastError());
98              exit(1);
99          }
100     }
101     //printf("SEND  [%d bytes]: start message '%s'\n",
            msg_size, buffer);
102
103     memset(buffer, 0, sizeof(buffer));
104     if ((msg_size = recv(sock, buffer, sizeof(buffer), 0)) ==
            SOCKET_ERROR)
105     {
106         printf("RECV␣directory␣content␣failed:␣%d\n",
                WSAGetLastError());
107         exit(1);
108     }
109     //printf("RECV  [%d bytes]: directory content\n", msg_size
            );
110     output(buffer);
111     send_report(sock, SUCCESS);
112
113     while(1)
```

```
114     {
115         memset(path, 0, sizeof(path));
116         if ((msg_size = recv(sock, path, sizeof(path), 0)) ==
                SOCKET_ERROR)
117         {
118             printf("RECV␣current␣path␣failed:␣%d\n",
                    WSAGetLastError());
119             exit(1);
120         }
121         //printf("RECV  [%d bytes]: current path '%s'\n",
                msg_size, path);
122 
123         send_report(sock, SUCCESS);
124 
125         memset(buffer, 0, sizeof(buffer));
126         if ((msg_size = recv(sock, buffer, sizeof(buffer), 0))
                == SOCKET_ERROR)
127         {
128             printf("RECV␣invitation␣message␣failed:␣%d\n",
                    WSAGetLastError());
129             exit(1);
130         }
131         //printf("RECV  [%d bytes]: invitation message\n",
                msg_size);
132         output(buffer);
133         char space;
134         memset(name, 0, sizeof(name));
135         memset(buffer, 0, sizeof(buffer));
136         memset(author, 0, sizeof(author));
137         memset(command, 0, sizeof(command));
138         memset(content, 0, sizeof(content));
139         scanf("%5s%1c", command, &space);
140         if ((msg_size = send(sock, command, strlen(command), 0)
                ) == SOCKET_ERROR)
141         {
142             printf("SEND␣command␣failed:␣%d\n", WSAGetLastError
                    ());
143             exit(1);
144         }
145         //printf("SEND  [%d bytes]: command '%s'\n", msg_size,
                path);
146 
147         if (!strcmp(command, ":exit"))
148         {
149             memset(buffer, 0, sizeof(buffer));
150             if ((msg_size = recv(sock, buffer, sizeof(buffer),
                    0)) == SOCKET_ERROR) // Receive the content of
                    file
151             {
```

```
152             printf("RECV file or directory content failed: %d
                    \n", WSAGetLastError());
153             exit(1);
154         }
155         //printf("RECV  [%d bytes]: file or directory
                content\n", msg_size);
156         output(buffer);
157         break;
158     }
159
160     if (recv_report(sock) < 0)
161     {
162         puts("!No such command");
163         send_report(sock, SUCCESS);
164     }
165
166     if (!strcmp(command, "add"))
167     {
168         char str[SIZE_ARG];
169         fgets(name, sizeof(name), stdin);
170         if (name[strlen(name) - 1] == '\n')
171             name[strlen(name) - 1] = '\0';
172         if ((msg_size = send(sock, name, strlen(name), 0))
                == SOCKET_ERROR)
173         {
174             printf("SEND command failed: %d\n",
                    WSAGetLastError());
175             exit(1);
176         }
177         //printf("SEND  [%d bytes]: title of article '%s'\n
                ", msg_size, name);
178         if (recv_report(sock) < 0)
179         {
180             puts("!Such file already exist");
181             send_report(sock, SUCCESS);
182         }
183         else
184         {
185             int length = sizeof(content) - sizeof(author) -
                    sizeof(name);
186             printf("Input author: ");
187             fgets(author, sizeof(author), stdin);
188             if (author[strlen(author) - 1] == '\n')
189                 author[strlen(author) - 1] = '\0';
190             printf("name's read: %s [%d bytes]\n", name,
                    msg_size);
191             printf("author's read: %s [%d bytes]\n", author,
                    msg_size);
192
```

```
193             puts("Put content:");
194             printf("[%d of %d]  ", (strlen(content)+strlen(
                    str)), length);
195             while (fgets(str, sizeof(str), stdin) != NULL)
196             {
197                 if (!strncmp(":end", str, strlen(":end")))
198                     break;
199                 if ((strlen(content)+strlen(str)) > length)
200                 {
201                     puts("!Text size will not allow");
202                     memset(str, 0, strlen(str));
203                     printf("[%d of %d]  ",(strlen(content)+
                            strlen(str)), length );
204                 }
205                 strcat(content, str);
206                 memset(str, 0, strlen(str));
207             }
208             if ((msg_size = send(sock, author, strlen(author)
                    , 0)) == SOCKET_ERROR)
209             {
210                 printf("SEND author of article failed: %d\n",
                        WSAGetLastError());
211                 exit(1);
212             }
213             //printf("SEND  [%d bytes]: author of article '%s
                    '\n", msg_size, author);
214             recv_report(sock);
215
216             if ((msg_size = send(sock, content, strlen(
                    content), 0)) == SOCKET_ERROR)
217             {
218                 printf("SEND file content failed: %d\n",
                        WSAGetLastError());
219                 exit(1);
220             }
221             //printf("SEND  [%d bytes]: file content '%s'\n",
                     msg_size, content);
222             if (recv_report(sock) < 0)
223                 puts("!Such file already exist");
224             send_report(sock, SUCCESS);
225         }
226     }
227
228         gets(buffer);
229         if (!strcmp(command, "open"))
230         {
231             strcat(path, buffer);
232             if ((msg_size = send(sock, path, strlen(path), 0)
                    ) == SOCKET_ERROR)
```

```
233             {
234                 printf("SEND full path to file failed: %d\n",
                        WSAGetLastError());
235                 exit(1);
236             }
237             //printf("SEND  [%d bytes]: full path to file '%s
                    '\n", msg_size, path);
238         }
239         else if (!strcmp(command, "find"))
240         {
241             if ((msg_size = send(sock, buffer, strlen(buffer)
                    , 0))== SOCKET_ERROR)
242             {
243                 printf("SEND author to find failed: %d\n",
                        WSAGetLastError());
244                 exit(1);
245             }
246             //printf("SEND  [%d bytes]: author to find '%s'\n
                    ", msg_size, buffer);
247         }
248 
249         memset(content, 0, sizeof(content));
250         if ((msg_size = recv(sock, content, sizeof(content),
                 0)) == SOCKET_ERROR)  // Receive the content of
                file
251         {
252             printf("RECV file or directory content failed: %d
                    \n", WSAGetLastError());
253             exit(1);
254         }
255         //printf("RECV  [%d bytes]: file or directory
                content\n", msg_size);
256         output(content);
257 
258         send_report(sock, SUCCESS);
259 
260     }
261 
262     closesocket(sock);
263 
264     WSACleanup();
265     return 0;
266 }
267 
268 void ValidateArgs(int argc, char **argv)
269 {
270     int i;
271     for(i = 1; i < argc; i++)
272     {
```

```
273         if ((argv[i][0] == '-') || (argv[i][0] == '/'))
274         {
275             switch (tolower(argv[i][1]))
276             {
277             case 'p':
278                 port = atoi(&argv[i][3]);
279                 break;
280             case 'i':
281                 binterface = 1;
282                 if (strlen(argv[i]) > 3)
283                     strcpy(szAddress, &argv[i][3]);
284                 break;
285             default:
286                 usage();
287                 break;
288             }
289         }
290     }
291 }
292 
293 void usage()
294 {
295     printf("usage: server [-p:x] [-i:IP]\n\n");
296     printf(" -p:x Port number to listen on\n");
297     printf(" -i:str Interface to listen on\n");
298 }
299 
300 void output(char *buffer)
301 {
302     int i;
303     for (i = 0; i < strlen(buffer); i++)
304         if (buffer[i] != '|')
305             printf("%c", buffer[i]);
306         else
307             printf("\n");
308     if (buffer[strlen(buffer) - 1] == '\n')
309         buffer[strlen(buffer) - 1] = '\0';
310 }
311 
312 int recv_report(SOCKET sock)
313 {
314     char status[SIZE_CMD];
315     int msg_size;
316     memset(status, 0, sizeof(status));
317     if ((msg_size = recv(sock, status, sizeof(status), 0)) ==
        SOCKET_ERROR)
318     {
319         perror("RECV report message failed");
320         printf("connect() failed: %d\n", WSAGetLastError());
```

```
321         exit(1);
322     }
323     //printf("RECV   [%d bytes]: report message   '%s'\n",
            msg_size, status);
324     return (!strcmp(status, SUCCESS) ? 0 : -1);
325 }
326
327 void send_report(SOCKET sock, char *status)
328 {
329     int msg_size;
330     if ((msg_size = send(sock, status, sizeof(status), 0)) ==
            SOCKET_ERROR)
331     {
332         perror("SEND report message failed");
333         printf("connect() failed: %d\n", WSAGetLastError());
334         exit(1);
335     }
336     //printf("SEND   [%d bytes]: report message   '%s'\n",
            msg_size, status);
337
338 }
```

UDP Server

main.c

```
1  #include <winsock2.h>
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <dirent.h>
6  #include <string.h>
7  #include <stdbool.h>
8  #include <sys/types.h>
9  #include <sys/stat.h>
10 #include "article.h"
11
12 #define DEFAULT_PORT 5001
13 #define MAX_CONNECT  3
14 #define SIZE_CMD     5
15 #define SIZE_BUF     1024
16 #define SIZE_CONTENT 4096
17 #define SIZE_STR     128
18 #define SUCCESS      "000"
19 #define UNSUCCESS    "111"
20 #define START_PATH "C:/Users/Kseniya/workspace/server_udp/
       Information System/"
21
```

```
22 int port = DEFAULT_PORT;
23 bool binterface = 0;
24 char szAddress[SIZE_STR];
25
26 int send_content(SOCKET sock, char *dir_name, struct
      sockaddr_in *ptr_addr);
27 int open_file(SOCKET sock, char *path, struct sockaddr_in *
      ptr_addr);
28 void sendPath_recvReport(SOCKET sock, char *path, struct
      sockaddr_in *ptr_addr);
29 void send_input_error(SOCKET sock, struct sockaddr_in *
      ptr_addr);
30 void send_report(SOCKET sock, char *status, struct
      sockaddr_in *ptr_addr);
31 void recv_report(SOCKET sock, struct sockaddr_in *ptr_addr);
32 void validateArgs(int argc, char **argv);
33 DWORD WINAPI ClientThread(LPVOID lpParam);
34 void usage();
35
36 typedef struct
37 {
38    SOCKET socket_fd;
39    struct sockaddr_in *ptr_addr;
40 } P_socket;
41
42 int main(int argc, char **argv)
43 {
44    WSADATA        wsd;
45    SOCKET         sock, i;
46    HANDLE         hThread;
47    DWORD          dwThreadId;
48    struct sockaddr_in server, addr[MAX_CONNECT];
49    P_socket p_sock[MAX_CONNECT];
50    const int on = 1;
51
52    validateArgs(argc, argv);
53    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
54    {
55       printf("Failed␣to␣load␣Winsock!\n");
56       return 1;
57    }
58
59    if (binterface)
60    {
61       server.sin_addr.s_addr = inet_addr(szAddress);
62       if (server.sin_addr.s_addr == INADDR_NONE)
63          usage();
64    }
65    else
```

```
66          server.sin_addr.s_addr = htonl(INADDR_ANY);
67      server.sin_family = AF_INET;
68      server.sin_port = htons(port);
69
70
71      if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) ==
            SOCKET_ERROR)
72      {
73          printf("socket() failed: %d\n", WSAGetLastError());
74          exit(1);
75      }
76      setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)
            );
77      for (i = 0; i < MAX_CONNECT; i++)
78      {
79          memset(&addr[i], 0, sizeof(addr[i]));
80          p_sock[i].socket_fd = sock;
81          p_sock[i].ptr_addr = &addr[i];
82      }
83      if (bind(sock, (struct sockaddr *)&server, sizeof(server))
             == SOCKET_ERROR)
84      {
85          printf("bind() failed: %d\n", WSAGetLastError());
86          return 1;
87      }
88
89      for(i = 0;i < MAX_CONNECT;i++)
90      {
91          printf("Accepted client: %s:%d\n",  inet_ntoa(server.
                sin_addr), ntohs(addr[i].sin_port));
92          if ((hThread = CreateThread(NULL, 0, ClientThread, (
                LPVOID)&p_sock[i], 0, &dwThreadId)) == NULL)
93          {
94              printf("CreateThread() failed: %d\n", GetLastError()
                    );
95              break;
96          }
97          WaitForSingleObject(hThread, INFINITE);
98          CloseHandle(hThread);
99      }
100     closesocket(sock);
101
102     WSACleanup();
103     return 0;
104 }
105 DWORD WINAPI ClientThread(LPVOID lpParam)
106 {
107     P_socket *data;
108     data = (P_socket *) lpParam;
```

```
109     struct sockaddr_in addr = *(data->ptr_addr);
110      SOCKET sock = data->socket_fd;
111     int len = sizeof(addr);
112     int msg_size;
113     const char *invite_msg = ">␣␣␣";
114     const char *exit_msg = "Bye-bye!!!|";
115     char path[SIZE_BUF];
116     char name[SIZE_STR];
117     char buffer[SIZE_BUF];
118     char author[SIZE_STR];
119     char content[SIZE_CONTENT];
120
121     while(strcmp(buffer, ":start"))
122     {
123        memset(buffer, 0, sizeof(buffer));
124        if ((msg_size = recvfrom(sock, buffer, sizeof(buffer),
              0, (struct sockaddr*)&addr,  &len))== SOCKET_ERROR)
125        {
126           printf("Receive␣:start_msg␣failed:␣%d\n",
                 WSAGetLastError());
127           exit(1);
128        }
129     }
130     buffer[msg_size] = '\0';
131     send_content(sock, START_PATH, &addr);
132     strcpy(path, START_PATH);
133
134     while(1)
135     {
136        if ((msg_size = sendto(sock, invite_msg, strlen(
              invite_msg), 0,  (struct sockaddr*)&addr, sizeof(
              addr))) == SOCKET_ERROR)
137        {
138           printf("SEND␣invitation␣message␣error:␣%d\n",
                 WSAGetLastError());
139           exit(1);
140        }
141        printf("SEND␣␣[%d␣bytes]:␣invitation␣message␣'%s'\n",
              msg_size, invite_msg);
142
143        memset(buffer, 0, sizeof(buffer));
144        if ((msg_size = recvfrom(sock, buffer, sizeof(buffer),
              0, (struct sockaddr*)&addr, &len)) == SOCKET_ERROR)
145        {
146           printf("RECV␣command␣error:␣%d\n", WSAGetLastError()
                 );
147           exit(1);
148        }
149        printf("RECV␣␣[%d␣bytes]:␣command␣'%s'\n", msg_size,
```

```
            buffer);
150
151         if (!strcmp(buffer, ":exit"))
152         {
153             if ((msg_size = sendto(sock, exit_msg, strlen(
                    exit_msg), 0,  (struct sockaddr*)&addr, sizeof(
                    addr))) == SOCKET_ERROR)
154             {
155                 printf("SEND␣directory␣content␣error:␣%d\n",
                        WSAGetLastError());
156                 exit(1);
157             }
158             printf("SEND␣␣[%d␣bytes]:␣directory␣content␣'%s'\n",
                     msg_size, exit_msg);
159             break;
160         }
161
162         if (strcmp(buffer, "find") && strcmp(buffer, "open") &&
                 strcmp(buffer, "add"))
163         {
164             send_input_error(sock, &addr);
165             send_content(sock, path, &addr);
166             continue;
167         }
168         send_report(sock, SUCCESS, &addr);
169
170         if (!strcmp(buffer, "open"))
171         {
172             memset(path, 0, sizeof(path));
173             if ((msg_size = recvfrom(sock, path, sizeof(path),
                    0, (struct sockaddr*)&addr,  &len)) ==
                    SOCKET_ERROR)
174             {
175                 printf("RECV␣path␣to␣file␣error:␣%d\n",
                        WSAGetLastError());
176                 exit(1);
177             }
178             printf("RECV␣␣[%d␣bytes]:␣path␣to␣file␣message␣'%s'\
                    n", msg_size, path);
179             open_file(sock, path, &addr);
180         }
181
182         if (!strcmp(buffer, "find"))
183         {
184             memset(author, 0,sizeof(author));
185             if ((msg_size = recvfrom(sock, author, sizeof(author
                    ), 0, (struct sockaddr*)&addr, &len)) ==
                    SOCKET_ERROR)
186             {
```

```
187             printf("RECV␣author␣to␣find␣error:␣%d\n",
                    WSAGetLastError());
188             exit(1);
189         }
190         printf("RECV␣␣[%d␣bytes]:␣author␣to␣find␣'%s'\n",
                msg_size, author);
191         find_for_author(sock, path, author, &addr);
192         sendPath_recvReport(sock, path, &addr);
193     }
194
195     if (!strcmp(buffer, "add"))
196     {
197         memset(name, 0,sizeof(name));
198         if ((msg_size = recvfrom(sock, name, sizeof(name),
                0, (struct sockaddr*)&addr,  &len)) ==
                SOCKET_ERROR)
199         {
200             printf("RECV␣name␣error:␣%d\n", WSAGetLastError()
                    );
201             exit(1);
202         }
203         printf("RECV␣␣[%d␣bytes]:␣name'%s'\n", msg_size,
                name);
204         char *dir = strdup(path);
205         strcat(path, name);
206         strcat(path, ".txt");
207         if (check_file_existence(path) < 0)
208         {
209             send_report(sock, UNSUCCESS, &addr);
210             recv_report(sock, &addr);
211         }
212         else
213         {
214             send_report(sock, SUCCESS, &addr);
215             memset(author,0, sizeof(author));
216             if ((msg_size = recvfrom(sock, author, sizeof(
                    author), 0, (struct sockaddr*)&addr,  &len))
                    == SOCKET_ERROR)
217             {
218                 printf("RECV␣author␣error:␣%d\n",
                        WSAGetLastError());
219                 exit(1);
220             }
221             printf("RECV␣␣[%d␣bytes]:␣author␣'%s'\n",
                    msg_size, author);
222             send_report(sock, SUCCESS, &addr);
223             memset(content, 0,sizeof(content));
224             if ((msg_size = recvfrom(sock, content, sizeof(
                    content), 0, (struct sockaddr*)&addr,  &len))
```

```
                == SOCKET_ERROR)
225             {
226                 printf("RECV content of file error: %d\n",
                        WSAGetLastError());
227                 exit(1);
228             }
229             printf("RECV  [%d bytes]: content of file '%s'\n"
                    , msg_size, content);
230
231             strcat(name, "\n");
232             strcat(author, "\n\n");
233             if (add_article(path, name, author, content) < 0)
234                 send_report(sock, UNSUCCESS, &addr);
235             else
236                 send_report(sock, SUCCESS, &addr);
237             recv_report(sock, &addr);
238             }
239             send_content(sock, dir, &addr);
240             dirname(path);
241             if (path[strlen(path) - 1] != '/')
242                 strcat(path, "/");
243         }
244     }
245     return 0;
246 }
247
248 void send_input_error(SOCKET sock, struct sockaddr_in *
        ptr_addr)
249 {
250     send_report(sock, UNSUCCESS, ptr_addr);
251     recv_report(sock, ptr_addr);
252 }
253
254 int send_content(SOCKET sock, char *dir_name, struct
        sockaddr_in *ptr_addr)
255 {
256     struct sockaddr_in addr;
257     addr = *ptr_addr;
258     char buffer[SIZE_BUF];
259     char *filename;
260     const char *delimiter = "----------------------|";
261     int msg_size;
262     DIR *dir = opendir(dir_name);
263     memset(buffer, 0, sizeof(buffer));
264     if(dir)
265     {
266         struct dirent *ent;
267         strcat(buffer, delimiter);
268         while((ent = readdir(dir)) != NULL)
```

```
269         {
270             filename = ent->d_name;
271             if (strcmp(filename, ".")==0)
272                 continue;
273             strcat(filename, "|");
274             strcat(buffer, filename);
275         }
276         closedir(dir);
277         strcat(buffer, delimiter);
278         if ((msg_size = sendto(sock, buffer, strlen(buffer), 0,
                  (struct sockaddr*)&addr, sizeof(addr))) ==
                SOCKET_ERROR)
279         {
280             printf("SEND␣directory␣content␣error:␣%d\n",
                    WSAGetLastError());
281             exit(1);
282         }
283         printf("SEND␣␣[%d␣bytes]:␣directory␣content␣'%s'\n",
                msg_size, buffer);
284         memset(filename, 0, sizeof(filename));
285         strcpy(filename, dir_name);
286         if (!strcmp(basename(filename), "."))
287             dirname(dir_name);
288         else if (!strcmp(basename(filename), ".."))
289             dirname(dirname(dir_name));
290     }
291     else
292     {
293         const char *err_msg = "!No␣such␣file␣or␣directory|";
294         if ((msg_size = sendto(sock, err_msg, strlen(err_msg),
                0,  (struct sockaddr*)&addr, sizeof(addr))) ==
                SOCKET_ERROR)
295         {
296             printf("SEND␣no␣file␣or␣directory␣error:␣%d\n",
                    WSAGetLastError());
297             exit(1);
298         }
299         printf("SEND␣␣[%d␣bytes]:␣no␣file␣or␣directory␣message␣
                '%s'\n", msg_size, buffer);
300         dirname(dir_name);
301     }
302
303     sendPath_recvReport(sock, dir_name, &addr);
304     return 0;
305 }
306
307 int open_file(SOCKET sock, char *path, struct sockaddr_in *
        ptr_addr)
308 {
```

```
309     struct sockaddr_in addr;
310     addr = *ptr_addr;
311     FILE *fp;
312     int msg_size;
313     struct stat about_file;
314 
315     char buffer[SIZE_STR], text[SIZE_CONTENT];
316     memset(text, 0, sizeof(text));
317     memset(buffer, 0, sizeof(buffer));
318 
319     char *tmp = malloc(SIZE_BUF);
320     strcpy(tmp, START_PATH);
321     if (!strcmp(path, strcat(tmp, "..")))
322     {
323         send_content(sock, START_PATH, &addr);
324         free(tmp);
325         return 0;
326     }
327     free(tmp);
328     if ((fp = fopen(path, "r")) == NULL)
329     {
330         printf("Opening␣of␣file␣error:␣%d\n", WSAGetLastError()
              );
331         send_content(sock, path, &addr);
332         return -1;
333     }
334 
335     fstat(fileno(fp), &about_file);
336     if ((about_file.st_mode & S_IFMT) != S_IFDIR)
337     {
338         int ch, i;
339         for (i = 0; i < (sizeof(text) - sizeof(char)) && (ch =
              getc(fp)) != EOF; i++)
340         {
341             if (ch == '\n')
342                 ch = '|';
343             text[i] = ch;
344         }
345         if ((msg_size = sendto(sock, text, strlen(text), 0,  (
              struct sockaddr*)&addr, sizeof(addr))) ==
              SOCKET_ERROR)
346         {
347             printf("SEND␣content␣of␣file␣error:␣%d\n",
                  WSAGetLastError());
348             exit(1);
349         }
350 
351         dirname(path);
352         sendPath_recvReport(sock, path, &addr);
```

```
353
354     }
355     else
356         send_content(sock, path, &addr);
357     fclose(fp);
358     return 0;
359
360 }
361
362 void sendPath_recvReport(SOCKET sock, char *path, struct
        sockaddr_in *ptr_addr)
363 {
364     struct sockaddr_in addr;
365     addr = *ptr_addr;
366     int msg_size;
367     recv_report(sock, ptr_addr);
368     if (path[strlen(path) - 1] != '/')
369         strcat(path, "/");
370     if ((msg_size = sendto(sock, path, strlen(path), 0,  (
        struct sockaddr*)&addr, sizeof(addr))) == SOCKET_ERROR)
371     {
372         printf("SEND␣current␣path␣error:␣%d\n", WSAGetLastError
            ());
373         exit(1);
374     }
375     printf("SEND␣␣[%d␣bytes]:␣current␣path␣'%s'\n", msg_size,
        path);
376     recv_report(sock, ptr_addr);
377 }
378
379 void send_report(SOCKET sock, char *status, struct
        sockaddr_in *ptr_addr)
380 {
381     struct sockaddr_in addr;
382     addr = *ptr_addr;
383     int msg_size;
384     if ((msg_size = sendto(sock, status, strlen(status), 0,  (
        struct sockaddr*)&addr, sizeof(addr))) == SOCKET_ERROR)
385     {
386         printf("SEND␣report␣message␣error:␣%d\n",
            WSAGetLastError());
387         exit(1);
388     }
389     printf("SEND␣␣[%d␣bytes]:␣report␣message␣'%s'\n", msg_size
        , status);
390 }
391
392
393 void recv_report(SOCKET sock, struct sockaddr_in *ptr_addr)
```

```
394 {
395    struct sockaddr_in addr;
396    addr = *ptr_addr;
397    int len =  sizeof(addr);
398    char status[SIZE_CMD];
399    int msg_size;
400    memset(status, 0, sizeof(status));
401    if ((msg_size = recvfrom(sock, status, sizeof(status), 0,
          (struct sockaddr*)&addr,  &len)) == SOCKET_ERROR)
402    {
403       printf("RECV␣report␣message␣failed:␣%d\n",
             WSAGetLastError());
404       exit(1);
405    }
406    printf("RECV␣␣[%d␣bytes]:␣report␣message␣␣'%s'\n",
          msg_size, status);
407 }
408
409 void usage()
410 {
411    printf("usage:␣server␣[-p:x]␣[-i:IP]\n\n");
412    printf("␣-p:x␣Port␣number␣to␣listen␣on\n");
413    printf("␣-i:str␣Interface␣to␣listen␣on\n");
414    ExitProcess(1);
415 }
416
417 void validateArgs(int argc, char **argv)
418 {
419    int i;
420
421    for(i = 1; i < argc; i++)
422    {
423       if ((argv[i][0] == '-') || (argv[i][0] == '/'))
424       {
425          switch (tolower(argv[i][1]))
426          {
427          case 'p':
428             port = atoi(&argv[i][3]);
429             break;
430          case 'i':
431             binterface = 1;
432             if (strlen(argv[i]) > 3)
433                strcpy(szAddress, &argv[i][3]);
434             break;
435          default:
436             usage();
437             break;
438          }
439       }
```

```
440     }
441 }
```

article.h

```
1  #ifndef ARTICLE_H_
2  #define ARTICLE_H_
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <errno.h>
8  #include <sys/types.h>
9  #include <sys/stat.h>
10 #include <dirent.h>
11 #include <winsock2.h>
12 #include <libgen.h>
13
14 #define MAX_FILES 100
15 #define BUF_SIZE 128
16 #define MAX_SIZE 1024
17
18 typedef struct art
19 {
20    char filename[BUF_SIZE];
21    char title[BUF_SIZE];
22    char author[BUF_SIZE];
23 } Article;
24
25 int check_file_existence(char *dir_name);
26 int add_article(char *dir_name, char *name, char* author,
       char *content);
27 int find_for_author(SOCKET sock, char *dir_name, char *author
       , struct sockaddr_in *ptr_addr);
28 char *lower(char *str);
29
30 #endif /* ARTICLE_H_ */
```

article.c

```
1  #include "article.h"
2
3  int add_article(char *dir_name, char *name, char* author,
       char *content)
4  {
5     FILE *fp;
6     if ((fp = fopen(dir_name,  "r" )) == NULL)
7        if (errno == ENOENT)
8           if ((fp = fopen(dir_name,  "w+" )) == NULL)
9           {
```

```
10              perror("File creation error");
11              return -2;
12          }
13          else
14          {
15              fputs(name, fp);
16              fputs(author, fp);
17              fputs(content, fp);
18              rewind(fp);
19              close(fp);
20              return 0;
21          }
22      close(fp);
23      return -1;
24  }
25
26  int check_file_existence(char *dir_name)
27  {
28      FILE *fp;
29      if ((fp = fopen(dir_name,  "r" )) == NULL)
30          if (errno == ENOENT)
31              return 0;
32      close(fp);
33      return -1;
34  }
35
36  int find_for_author(SOCKET sock, char *dir_name, char *author
        , struct sockaddr_in *ptr_addr)
37  {
38      struct sockaddr_in addr;
39      addr = *ptr_addr;
40      char buffer[BUF_SIZE];
41      char path[MAX_SIZE];
42      char *ptr;
43      const char *delimiter = "----------------------|";
44      int msg_size;
45      char *filename;
46      FILE *fp;
47      DIR *dir = opendir(dir_name);
48
49      memset(buffer,0, sizeof(buffer));
50      struct stat about_file;
51      Article *arts=(Article *)malloc(MAX_FILES * sizeof(Article
            ));
52      int i, k = 0;
53      if(dir)
54      {
55          struct dirent *ent;
56          while((ent = readdir(dir)) != NULL)
```

```
57          {
58              strcpy(path, dir_name);
59              filename = ent->d_name;
60
61              if ((fp = fopen(strcat(path, filename), "r")) ==
                    NULL)
62              {
63                  printf("error␣%s\n", filename);
64                  perror("Opening␣of␣file␣error");
65              }
66              fstat(fileno(fp), &about_file);
67              if ((about_file.st_mode & S_IFMT) != S_IFDIR)
68              {
69                  for( i = 0; (ptr = fgets(buffer, sizeof(buffer),
                        fp)) != NULL && i <2; i++)
70                  {
71
72                      if (i == 0)
73                          strcpy(arts[k].title, ptr);
74                      else if (i == 1)
75                          strcpy(arts[k].author, ptr);
76                      memset(ptr, 0, strlen(ptr));
77                  }
78                  strcpy(arts[k].filename, filename);
79                  k++;
80              }
81              fclose(fp);
82          }
83          closedir(dir);
84          memset(buffer, 0, sizeof(buffer));
85          strcat(buffer, "Search␣results␣for␣author:␣");
86          strcat(buffer, author);
87          strcat(buffer, "|");
88          strcat(buffer, delimiter);
89          for (i = 0; k >= 0; --k)
90          {
91              if (strstr(lower(arts[k].author), lower(author)) !=
                    NULL)
92              {
93                  strcat(buffer, arts[k].author);
94                  strcat(buffer, ":␣␣␣␣");
95                  strcat(buffer, arts[k].filename);
96                  strcat(buffer, "|");
97                  i++;
98              }
99          }
100         strcat(buffer, delimiter);
101         if (i == 0)
102         {
```

```
103             memset(buffer, 0, sizeof(buffer));
104             strcat(buffer, "There are no articles of ");
105             strcat(buffer, author);
106             strcat(buffer, "|");
107         }
108         if ((msg_size = sendto(sock, buffer, strlen(buffer), 0,
                  (struct sockaddr*)&addr, sizeof(addr))) ==
                SOCKET_ERROR)
109         {
110             printf("SEND found result error: %d\n",
                    WSAGetLastError());
111             return -1;
112         }
113         printf("SEND   [%d bytes]: found result '%s'\n",
                msg_size, buffer);
114         return 0;
115     }
116     free(arts);
117     return -1;
118 }
119
120 char *lower(char *str)
121 {
122     int i;
123     char *new = strdup(str);
124     for (i = 0; i < strlen(new); i++)
125         new[i] = tolower(new[i]);
126     return new;
127 }
```

UDP Client

main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <stdbool.h>
5  #include <sys/stat.h>
6  #include <dirent.h>
7  #include <string.h>
8  #include <fcntl.h>
9  #include <libgen.h>
10 #include <winsock2.h>
11
12 #define SIZE_CMD 5
13 #define SIZE_ARG 50
14 #define SIZE_STR 128
```

```
15 #define SIZE_BUF 1024
16 #define SUCCESS "000"
17 #define UNSUCCESS "111"
18 #define SIZE_CONTENT 4096
19 #define DEFAULT_PORT 5001
20
21 void output(char *str);
22 void add_article_to_system(SOCKET sock, char *path);
23 int recv_report(SOCKET sock);
24 void send_report(SOCKET sock, char *status);
25 void ValidateArgs(int argc, char **argv);
26 void usage();
27
28
29 int port = DEFAULT_PORT;
30 bool binterface = 0;
31 char szAddress[SIZE_STR];
32 struct sockaddr_in client;
33
34 int main(int argc, char **argv)
35 {
36    WSADATA wsd;
37    SOCKET sock;
38    int msg_size;
39    char path[SIZE_BUF];
40    char name[SIZE_STR];
41    char buffer[SIZE_BUF];
42    char author[SIZE_STR];
43    char command[SIZE_CMD];
44    char content[SIZE_CONTENT];
45
46    ValidateArgs(argc, argv);
47    if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
48    {
49       printf("Failed to load Winsock library!\n");
50       return 1;
51    }
52    if (binterface)
53    {
54       client.sin_addr.s_addr = inet_addr(szAddress);
55       if (client.sin_addr.s_addr == INADDR_NONE)
56          usage();
57    }
58    else
59       client.sin_addr.s_addr = htonl(INADDR_ANY);
60    client.sin_family = AF_INET;
61    client.sin_port = htons(port);
62    int len = sizeof(client);
63
```

```
64    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
65    {
66       printf("Socket is not created: %d\n", WSAGetLastError()
         );
67       exit(1);
68    }
69
70    memset(buffer, 0, sizeof(buffer));
71    while(strcmp(buffer, ":start"))
72    {
73
74       fgets(buffer, sizeof(buffer), stdin);
75       if (buffer[strlen(buffer) - 1] == '\n')
76          buffer[strlen(buffer) - 1] = '\0';
77       if ((msg_size = sendto(sock, buffer, strlen(buffer), 0,
             (struct sockaddr *)&client, sizeof(client))) ==
          SOCKET_ERROR)
78       {
79          printf("SEND start message failed: %d\n",
             WSAGetLastError());
80          exit(1);
81       }
82    }
83    //printf("SEND  [%d bytes]: start message '%s'\n",
       msg_size, buffer);
84
85    memset(buffer, 0, sizeof(buffer));
86    if ((msg_size = recvfrom(sock, buffer, sizeof(buffer), 0,
       (struct sockaddr *)&client, &len)) == SOCKET_ERROR)
87    {
88       printf("RECV directory content failed: %d\n",
          WSAGetLastError());
89       exit(1);
90    }
91    //printf("RECV  [%d bytes]: directory content\n", msg_size
       );
92    output(buffer);
93    send_report(sock, SUCCESS);
94
95    while(1)
96    {
97       memset(path, 0, sizeof(path));
98       if ((msg_size = recvfrom(sock, path, sizeof(path), 0, (
          struct sockaddr *)&client, &len)) == SOCKET_ERROR)
99       {
100         printf("RECV current path failed: %d\n",
             WSAGetLastError());
101         exit(1);
102      }
```

```
103         //printf("RECV  [%d bytes]: current path '%s'\n",
                msg_size, path);
104
105         send_report(sock, SUCCESS);
106
107         memset(buffer, 0, sizeof(buffer));
108         if ((msg_size = recvfrom(sock, buffer, sizeof(buffer),
                0, (struct sockaddr *)&client, &len)) ==
                SOCKET_ERROR)
109         {
110             printf("RECV invitation message failed: %d\n",
                    WSAGetLastError());
111             exit(1);
112         }
113         //printf("RECV  [%d bytes]: invitation message\n",
                msg_size);
114         output(buffer);
115         char space;
116         memset(name, 0, sizeof(name));
117         memset(buffer, 0, sizeof(buffer));
118         memset(author, 0, sizeof(author));
119         memset(command, 0, sizeof(command));
120         memset(content, 0, sizeof(content));
121         scanf("%5s%1c", command, &space);
122         if ((msg_size = sendto(sock, command, strlen(command),
                0, (struct sockaddr *)&client, sizeof(client))) ==
                SOCKET_ERROR)
123         {
124             printf("SEND command failed: %d\n", WSAGetLastError
                    ());
125             exit(1);
126         }
127         //printf("SEND  [%d bytes]: command '%s'\n", msg_size,
                path);
128
129         if (!strcmp(command, ":exit"))
130         {
131             memset(buffer, 0, sizeof(buffer));
132             if ((msg_size = recvfrom(sock, buffer, sizeof(buffer
                    ), 0, (struct sockaddr *)&client, &len)) ==
                    SOCKET_ERROR)  // Receive the content of file
133             {
134                 printf("RECV file or directory content failed: %d
                        \n", WSAGetLastError());
135                 exit(1);
136             }
137             //printf("RECV  [%d bytes]: file or directory
                    content\n", msg_size);
138             output(buffer);
```

```
139             break;
140         }
141 
142         if (recv_report(sock) < 0)
143         {
144             puts("!No such command");
145             send_report(sock, SUCCESS);
146         }
147 
148         if (!strcmp(command, "add"))
149         {
150             char str[SIZE_ARG];
151             fgets(name, sizeof(name), stdin);
152             if (name[strlen(name) - 1] == '\n')
153                 name[strlen(name) - 1] = '\0';
154             if ((msg_size = sendto(sock, name, strlen(name), 0,
                    (struct sockaddr *)&client, sizeof(client))) ==
                    SOCKET_ERROR)
155             {
156                 printf("SEND command failed: %d\n",
                        WSAGetLastError());
157                 exit(1);
158             }
159             //printf("SEND  [%d bytes]: title of article '%s'\n
                    ", msg_size, name);
160             if (recv_report(sock) < 0)
161             {
162                 puts("!Such file already exist");
163                 send_report(sock, SUCCESS);
164             }
165             else
166             {
167                 int length = sizeof(content) - sizeof(author) -
                        sizeof(name);
168                 printf("Input author: ");
169                 fgets(author, sizeof(author), stdin);
170                 if (author[strlen(author) - 1] == '\n')
171                     author[strlen(author) - 1] = '\0';
172                 printf("name's read: %s [%d bytes]\n", name,
                        msg_size);
173                 printf("author's read: %s [%d bytes]\n", author,
                        msg_size);
174 
175                 puts("Put content:");
176                 printf("[%d of %d]  ", (strlen(content)+strlen(
                        str)), length);
177                 while (fgets(str, sizeof(str), stdin) != NULL)
178                 {
179                     if (!strncmp(":end", str, strlen(":end")))
```

```
180                 break;
181             if ((strlen(content)+strlen(str)) > length)
182             {
183                 puts("!Text␣size␣will␣not␣allow");
184                 memset(str, 0, strlen(str));
185                 printf("[%d␣of␣%d]␣␣",(strlen(content)+
                        strlen(str)), length );
186             }
187             strcat(content, str);
188             memset(str, 0, strlen(str));
189         }
190         if ((msg_size = sendto(sock, author, strlen(
                author), 0, (struct sockaddr *)&client, sizeof
                (client))) == SOCKET_ERROR)
191         {
192             printf("SEND␣author␣of␣article␣failed:␣%d\n",
                    WSAGetLastError());
193             exit(1);
194         }
195         //printf("SEND  [%d bytes]: author of article '%s
                '\n", msg_size, author);
196         recv_report(sock);
197 
198         if ((msg_size = sendto(sock, content, strlen(
                content), 0, (struct sockaddr *)&client,
                sizeof(client))) == SOCKET_ERROR)
199         {
200             printf("SEND␣file␣content␣failed:␣%d\n",
                    WSAGetLastError());
201             exit(1);
202         }
203         //printf("SEND  [%d bytes]: file content '%s'\n",
                 msg_size, content);
204         if (recv_report(sock) < 0)
205             puts("!Such␣file␣already␣exist");
206         send_report(sock, SUCCESS);
207     }
208 }
209 
210 gets(buffer);
211 if (!strcmp(command, "open"))
212 {
213     strcat(path, buffer);
214     if ((msg_size = sendto(sock, path, strlen(path), 0,
            (struct sockaddr *)&client, sizeof(client))) ==
            SOCKET_ERROR)
215     {
216         printf("SEND␣full␣path␣to␣file␣failed:␣%d\n",
                WSAGetLastError());
```

```
217             exit(1);
218          }
219          //printf("SEND  [%d bytes]: full path to file '%s'\n
                ", msg_size, path);
220       }
221       else if (!strcmp(command, "find"))
222       {
223          if ((msg_size = sendto(sock, buffer, strlen(buffer),
                 0, (struct sockaddr *)&client, sizeof(client)))
                == SOCKET_ERROR)
224          {
225             printf("SEND author to find failed: %d\n",
                   WSAGetLastError());
226             exit(1);
227          }
228          //printf("SEND  [%d bytes]: author to find '%s'\n",
                msg_size, buffer);
229       }
230
231       memset(content, 0, sizeof(content));
232       if ((msg_size = recvfrom(sock, content, sizeof(content)
             , 0, (struct sockaddr *)&client, &len)) ==
             SOCKET_ERROR)   // Receive the content of file
233       {
234          printf("RECV file or directory content failed: %d\n"
                , WSAGetLastError());
235          exit(1);
236       }
237       //printf("RECV  [%d bytes]: file or directory content\n
             ", msg_size);
238       output(content);
239       send_report(sock, SUCCESS);
240    }
241
242
243    closesocket(sock);
244
245    WSACleanup();
246    return 0;
247 }
248
249 void ValidateArgs(int argc, char **argv)
250 {
251    int i;
252
253    for(i = 1; i < argc; i++)
254    {
255       if ((argv[i][0] == '-') || (argv[i][0] == '/'))
256       {
```

```
257             switch (tolower(argv[i][1]))
258             {
259             case 'p':
260                 port = atoi(&argv[i][3]);
261                 break;
262             case 'i':
263                 binterface = 1;
264                 if (strlen(argv[i]) > 3)
265                     strcpy(szAddress, &argv[i][3]);
266                 break;
267             default:
268                 usage();
269                 break;
270             }
271         }
272     }
273 }
274
275 void usage()
276 {
277     printf("usage: server [-p:x] [-i:IP]\n\n");
278     printf(" -p:x Port number to listen on\n");
279     printf(" -i:str Interface to listen on\n");
280 }
281
282 int recv_report(SOCKET sock)
283 {
284     char status[SIZE_CMD];
285     int msg_size;
286     int len = sizeof(client);
287     memset(status, 0, sizeof(status));
288     if ((msg_size = recvfrom(sock, status, sizeof(status), 0,
            (struct sockaddr *)&client, &len)) == SOCKET_ERROR)
289     {
290         printf("RECV report message failed: %d\n",
                WSAGetLastError());
291         exit(1);
292     }
293     //printf("RECV  [%d bytes]: report message  '%s'\n",
            msg_size, status);
294     return (!strcmp(status, SUCCESS) ? 0 : -1);
295 }
296
297 void send_report(SOCKET sock, char *status)
298 {
299     int msg_size;
300     if ((msg_size = sendto(sock, status, sizeof(status), 0, (
            struct sockaddr *)&client, sizeof(client))) ==
            SOCKET_ERROR)
```

```
301     {
302         printf("SEND report message failed: %d\n",
                WSAGetLastError());
303         exit(1);
304     }
305     //printf("SEND  [%d bytes]: report message  '%s'\n",
            msg_size, status);
306 }
307
308 void output(char *buffer)
309 {
310     int i;
311     for (i = 0; i < strlen(buffer); i++)
312         if (buffer[i] != '|')
313             printf("%c", buffer[i]);
314         else
315             printf("\n");
316     if (buffer[strlen(buffer) - 1] == '\n')
317         buffer[strlen(buffer) - 1] = '\0';
318 }
319
320
321 void add_article_to_system(SOCKET sock, char *path)
322 {
323     char buffer[SIZE_BUF];
324     char content[SIZE_CONTENT];
325     int msg_size;
326     int len = sizeof(client);
327     printf("Current path is %s\n", path);
328     strcat(path, buffer);
329     if ((msg_size = sendto(sock, path, strlen(path), 0, (
            struct sockaddr *)&client, sizeof(client))) ==
            SOCKET_ERROR)
330     {
331         printf("SEND full path to file failed: %d\n",
                WSAGetLastError());
332         exit(1);
333     }
334     //printf("SEND  [%d bytes]: full path to file '%s'\n",
            msg_size, path);
335
336     memset(content, 0, sizeof(content));
337     if ((msg_size = recvfrom(sock, content, sizeof(content),
            0,  (struct sockaddr *)&client, &len))== SOCKET_ERROR)
338     {
339         printf("RECV file or directory content failed: %d\n",
                WSAGetLastError());
340         exit(1);
341     }
```

```
342     //printf("RECV  [%d bytes]: file or directory content\n",
            msg_size);
343     output(content);
344 }
```