Brno University of Technology

Faculty of Information Technology

Computer Communications and Networks

1.project (1.variant)

Client-Server Application for Obtaining User Information

March 12, 2018

Stupinský Šimon, (xstupi00)

Contents

1 Introduction **2**

2 Application Protocol **3**
2.1 Overview 3
2.2 Packet Design 3
2.3 Type of messages 4
2.3.1 Communication flow 4
2.4 Example of communication 5

3 Implementation **7**
3.1 TCP and Berkely sockets 7
3.2 Details of implementation 7

4 Demonstration of the activities of the applications **9**

Bibliography **12**

A Finite Machines of Communication Flow **13**

Chapter 1

Introduction

This documentation is focused on the description design of **Application Protocol** and implementation of **Client-Server** application. Application Protocol has the task of realizing transfer user information from the server side to the client side, based on its requirements. Design of this protocol will be introduced in Chapter 2.

The client-Server application uses this protocol to communicate between both sides of this application, respectively its implementation, will be introduced in Chapter 3.

Chapter 2

Application Protocol

2.1 Overview

Since this protocol has the task transfer data between client and server, it is obvious, that its main requirement will be reliability. The first part of this process was the study already existing protocols. We focused on the protocols with similar features and functionality that our protocol should have, one of them was a protocol **FTP** [1].

The first candidate representing the protocol was a char array. Traditionally, an array would be divided into several parts, separately specific whitespaces, where two largest part are *head* and *body*. However, the work with such array is not the easiest way to transform data between two different sides. Not only for this reason, we have decided design a protocol using some flatten of abstraction. Specifically, it will be used the *struct*, which will be representing **packet** for communication.

2.2 Packet Design

At design packet to communication between clients and server was given consideration to the nature of this communication. For each send, as well received the message it is necessary to distinguish this type of message. It is obvious, that for both participants of communication is the important resolution of the type of the receiving message, on the receiver side. On the other side of the communication channel is needed to set the correct type of the messages, with respect to the requirements to the receiver, to avoid communication noise. Not only for these required packet properties we chose as our first item of structure **msg_type**, who will be the holder of this information. The nearest information about these types of messages will be introduced in next section 2.3.

Still, we are able to represent the only type of messages and not its useful content for the second side. Just because of the need sending useful data is our second item of structure **data buffer**, which will play this role. The size of this buffer has been selected *256-bit*. With respect to the role of communication can commute to transform data with the smaller size in 2/3 of cases. Provided that no unexpected situations have been occurred, such as too long login or a name of the home directory of some users.

With the definition of both items of structure we can to represent its interconnection. The message type will describe what type of data to expect in the attached buffer. This type can be used by the sending side and the receiving side to determine what type of data

to attach to the message, and what type of data can be expected to be found in the message received. The code below represents the described packet structure.

`Packet Structure`

```
struct message {
    msg_type type;
    char data[BUFF_SIZE];
};
```

2.3 Type of messages

This process of designing the type of messages was the next an important part in the design of the whole protocol. Typically are represented by an integer and are located at the beginning of the packet, in our case at the beginning of the packet structure. A good way to implement this is by using an enumerator, what was also in our case. Each message can be handled differently based on the type and both sides of communication decide about its further activities on that basis.

The individual types of the message were designed to clearly define the exchange of information between sender and receiver. The goal was to ensure that both participants of communication always knew, what the other side requires from them.

`Type of Messages`

```
enum msg_type {
    BEGIN_CODE = 01, ACCEPT = 02, NAMES = 10,
    HOME_DIR = 20, USERS = 30, NAMES_END = 11,
    HOME_DIR_END = 21, USERS_END = 31, DATA = 40,
    DATA_END = 41, ERROR = 99,
};
```

2.3.1 Communication flow

The first type is *BEGIN_CODE*, which the client uses after connecting to the server. On the other side, the server responds to this type of message by sending *ACCEPT* back to the client. It is a type of control of the right connection between the two sides. After exchanging these two messages the client is re-run. For these types is attached buffer empty.

Its task is to send requirements, which is required, to the server. For these purposes are types *NAMES*, *HOME_DIR*, and *USERS*. Each describes a different type of request, depending on the given argument on the client side. As you can guess, *NAMES* indicates the requirements for **User ID Info** for specific login, *HOME_DIR* indicates the requirements for **Home directory** of specific login. The type *USERS means* that is required the specific group of users. Types equivalent to these, only with different postfix *_END* indicates the end of the sending of the request by the sender. On the basis of these types, the receiver, respectively server, discovered that momentarily does not receive another message and will be can fulfill the requirements. For these types contains attached buffer data with login, which was entered by the user on the client side.

In the opposite direction are sent types *DATA* and *DATA_END*. The server sends them useful obtained data to the client. Types *DATA* it means for the client will receiving the message until the receive type *DATA_END*. For these types contains attached buffer just mentioned obtained useful data.

The last type of messages is *ERROR*. It is sent in unexpected situations with an error message in the attached buffer. On the client side means receiving this types an immediate exit. On both sides, the error message from the buffer is output to the standard error output.

Communication flow of client and server represented by *Finite Machine*, you can see in the Appendices A.1 and A.2. Transitions between individual states of these Machines are representing by received or sent types of messages. It is based on their foundations what actions will be both sides executing after received or the sent message.

2.4 Example of communication

For better imagination, we give the example of the mentioned communication between client and server, you can see it in picture 2.1. As we have described in the previous section 2.3.1, at the beginning of communication is occurring control, which includes two introduction messages from both sides. After the client received *ACCEPT* type of message from the server, will be sending requirements.

In our example it is requirements for **USER ID INFO** from */etc/passwd/* file from the server side. We assume the length of entered login exceeds the size of the attached buffer, to use as many types of messages as possible. When the client will send the last part of login, will set the message type on the *HOME_END*, which tells to the server that it is the last message.

The server on the basics this type of message passes from receiver state. Once the required data is obtained, it fits into the role of the sender. Data is sent to the client after the part of the size of the attached buffer, with message type *DATA*. Just as it did for the client, also the server at the last part of sending data will set the type of message to *DATA_END*. After the client received *DATA_END* type of message from the server and to end all the activities with the received data, then end of its existence.

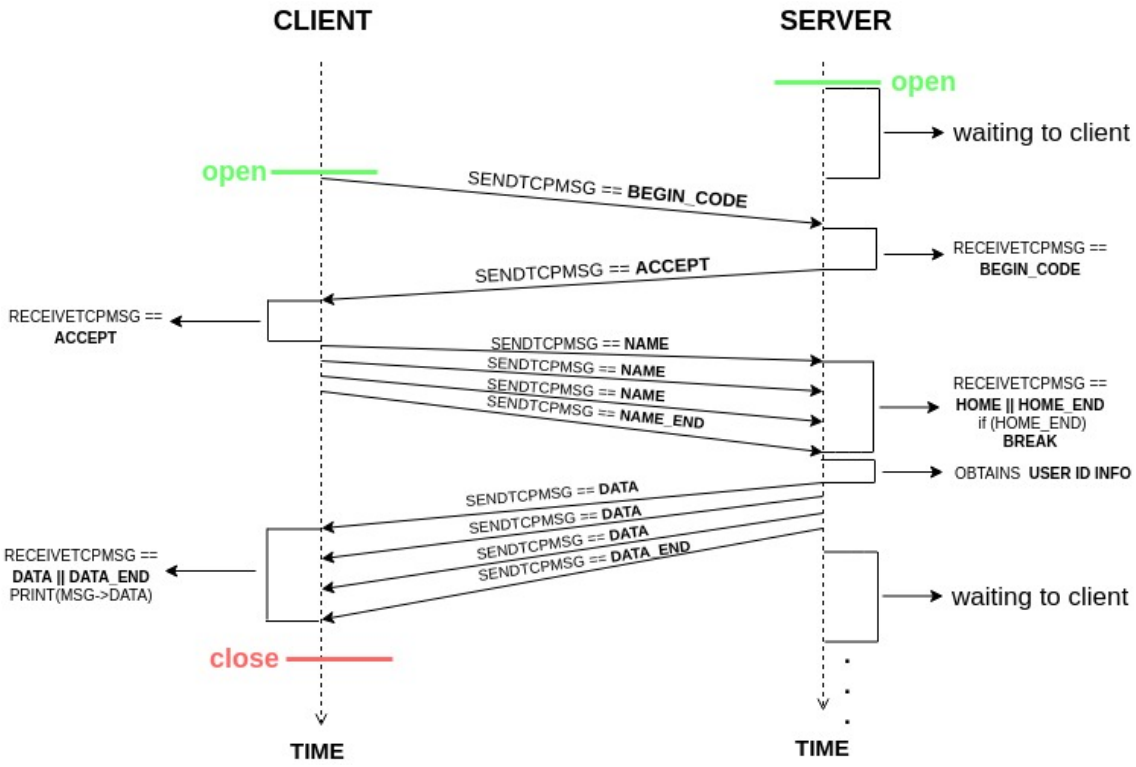

Figure 2.1: Example of communication between client and server.

Chapter 3

Implementation

This implementation using to communication between two computers **Berkeley UNIX sockets** and **TCP transport layer protocol**. This combination is the way to reliably transform data between two sides. Data is formatted and transmitted in the application layer, which operates on the top of TCP.

We will gradually introduce the most interesting facts about implementation, which includes three files. File *ipk-client.cpp* contains implementations of the client application and *ipk-server.cpp* contains implementations of the server application. The third file is *interface.h*, which contains common declarations and definitions methods and functions, which using both mentioned modules. Before that, however, let us say something about using TCP protocol and Berkeley sockets.

3.1 TCP and Berkely sockets

TCP is a standard network protocol used to reliably transform data between two computers in the network. TCP is the foundation for the modern internet, and without it - or without a similar protocol - there would be no way to send large amounts of data over a network accurately. The protocol can be easily implemented by any programmer not only for these type of application. How was mentioned above, it is the best way to ensure reliable transfer of data between two different sides in the network [2].

BSD Sockets generally relies upon client/server architecture. For TCP communications, one host listens for incoming connection requests. When a request arrives, the server host will accept it, at which point data can be transferred between the hosts. A socket is an abstract representation (handle) for the local endpoint of a network communication path. The Berkeley sockets API represents it as a file descriptor (file handle) in the Unix philosophy that provides a common interface for input and output to streams of data [4].

3.2 Details of implementation

Design of Berkeley UNIX sockets was taken from [2]. As mentioned in this article, this way of implementation by using of class brings some advantages connect with this type of applications. The class itself has been stolen by some attributes according to our needs. Similarly, its methods were also adapted according to the specifications and our needs. For more information, you can browse to above cited article or to our file *interface.h*, where is this class implemented.

The first area we look at will be processing of the file */etc/passwd/*. From this file can be necessary to acquire **USER ID INFO** or **Home Directory** for specific login entered by users on the client side. For this purpose was used library function *getpwnam_r*. This function returns a pointer to a structure containing the broken-out fields of the record in the password database, that matches the username name which was entered [3]. Properties of this function are suitable for our required functionality. From returned pointer to a structure will choose the necessary data.

As well may be a request from the client for specific groups of users of that file. This processing is solving manually, by browsing the file, respectively scrolling its lines. Because the format is so favorable for processing its properties, was selected the following method. We scroll the line by line and we are starving the prefix match between the received login entered by users on the client side and username on the actual line. The comparison is executing lexicographically char by char.

Another area we will look at will be the resolution of the concurrently non-blocking server. This requirement is necessary because the server must be able to meet the requirements of multiple clients concurrently. This kind of parallelism is solved in our implementation as follows. The main process waiting to the connection of the client. At the moment when the client connects, using *fork()* is creating a new process that will serves the client. This child process serve the client and then extinction. The main process will continue in waiting to connect the new client.

Chapter 4

Demonstration of the activities of the applications

NAME mode

```
$ ./ipk-client -h merlin.fit.vutbr.cz -p 9998 -n rysavy
Rysavy Ondrej,UIFS,541141118
```

NAME mode

```
$ ./ipk-client -h merlin.fit.vutbr.cz -p 9999 -n xstupi00
Stupinsky Simon,FIT BIT 2r
```

NAME mode (no login)

```
$ ./ipk-client -h merlin.fit.vutbr.cz -p 9999 -n xstupi
Login was not found!
```

NAME mode (no info)

```
$ ./ipk-client -h 127.0.0.1 -p 9999 -n syslog
Any information not found!
```

FILE mode

```
$ ./ipk-client -h merlin.fit.vutbr.cz -p 9998 -f rysavy
/homes/kazi/rysavy
```

FILE mode

```
$ ./ipk-client -h merlin.fit.vutbr.cz -p 9998 -f xmarci10
/homes/eva/xm/xmarci10
```

FILE mode

```
$ ./ipk-client -h merlin.fit.vutbr.cz -p 9998 -f xmarci100
Login was not found!
```

USERS mode

```
$ ./ipk-client -h merlin.fit.vutbr.cz -p 9998 -l xlis
xlisci01
xlisci02
xlisie00
xliska16
xlisti00
```

USERS mode

```
$ ./ipk-client -h 127.0.0.1 -p 9999 -l s
sys
sync
systemd-timesync
systemd-network
systemd-resolve
systemd-bus-proxy
syslog
speech-dispatcher
saned
simon
```

USERS mode (no users)

```
$ ./ipk-client -h merlin.fit.vutbr.cz -p 9998 -l xstupi01
Any information not found!
```

USERS mode

```
$ ./ipk-client -h merlin.fit.vutbr.cz -p 9998 -l xstupi00
xstupi00
```

Run-time error in Client

```
$ ./ipk-client -h merlin.fit.vutbr.cz -p 9998
Usage : $ ./ipk-client -h <host> -p <port> [-n|-f|-l] login

MANDATORY OPTIONS:
-h host -> IP address or fully-qualified DNS name, identifying the server
-p port -> Destination port number in the range (0,65535)
login Login specifies the login name for the following operation

OPTIONAL OPTIONS: (1/3)
-n name -> The full user name is returned, including any additional
           information for that login (User ID Info)
-f file -> Return home directory information for the specified login
           (Home directory)
-l list -> If the login was entered, will be searching users with the same prefix.
           If the login was not entered, will be returned all users.

EXAMPLES OF USAGE: $ ./ipk-client -h eva.fit.vutbr.cz -p 55555 -n rysavy
                   $ ./ipk-client -h host -p port -l
```

Run-time error in Server

```
$ ./ipk-server -p
Usage : $ ./ipk-server -p <port>

MANDATORY OPTION:
-p port -> Destination port number in the range (0,65535)

EXAMPLE OF USAGE: $ ./ipk-server -p 55555
```

Bibliography

[1] File Transfer Protocol. RFC 959. October 1985. doi:10.17487/RFC0959.
Retrieved from: https://rfc-editor.org/rfc/rfc959.txt

[2] Lattrel, R.: Designing and Implementing an Application Layer Network Protocol Using UNIX Sockets and TCP. online. November 2012. 480 ECE Design Team 2.
Retrieved from: https://www.egr.msu.edu/classes/ece480/capstone/fall12/group02/documents/Ryan-Lattrel_App-Note.pdf

[3] Linux: getpwnam(3). online. man page.
Retrieved from: https://linux.die.net/man/3/getpwnam

[4] Wikipedia: Berkeley sockets. online.
Retrieved from: https://en.wiikipedia.org/wiki/Berkeley_sockets

Appendix A

Finite Machines of Communication Flow

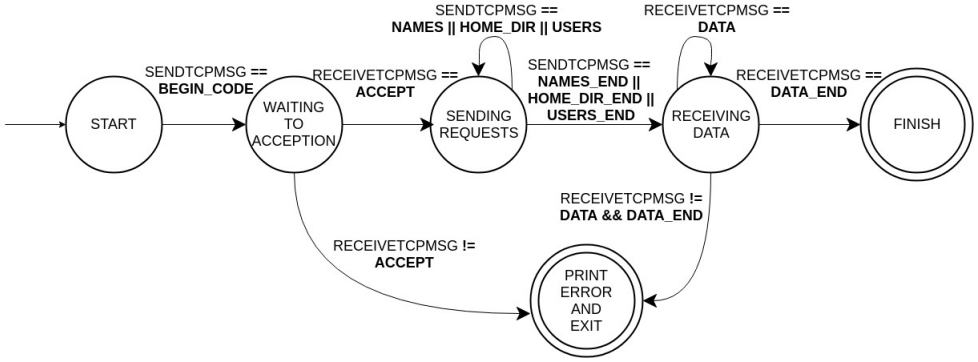

Figure A.1: Finite Machine representing communication flow on the client side.

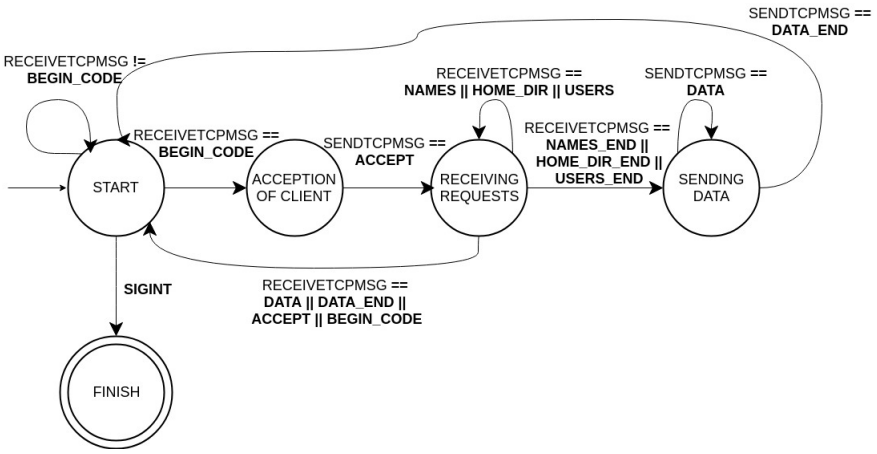

Figure A.2: Finite Machine representing communication flow on the server side.