ELEC-C7241 - Tietokoneverkot
Tietoliikenneprojekti “Tiedostonjako”

Eskelinen Jussi, 64949J, jussi.eskelinen@aalto.fi
Kopitca Artur, 474212, artur.kopitca@aalto.fi

9.4.2017

Table of contents

1. Program Description

1.1 General Information

The project programs are written on C programming language and tested on the operating system Ubuntu in the Aalto University's computer class, Y342a.

The project consists of two programs, "server.c" and "client.c", hereafter referred to as the Server and the Client. To run the programs, they should be located either on the same computer or on two or more different computers. The data to be shared should be located in the folder of arbitrary name in the Server's and the Client's directories.

The file sharing is implemented as follows: the Server creates a TCP socket and listens for the file requests from the clients. The Client initiates a TCP connection with the Server on its IP address and port number set by the user. Associated with the TCP connection, there will be a socket on the Client and a socket on the Server. The Client sends a request message via its socket to it, the Server receives the request message from its socket interface and performs the required processing: determines the type of the requested file and the method to be performed on the file.

If the method is of type GET (request to receive a file), the Server searches for the requested file in the directory set by the user, sends a response, informing the Client about the result of the file search and, if the file is found, delivers a header and the file to the Client. After delivering the requested file, the Server closes the ongoing TCP connection and continues to listen for the oncoming requests from the clients.

If the method is of type PUT (request to send a file), the Server receives the file from the Client and saves it in the directory set by the user. If the file with the same name already exists in the given directory, the old file is replaced with the new one. After receiving and saving the file, Server closes the ongoing TCP connection and continues to listen for the oncoming requests from the clients.

The Server is able to service up to 5 clients. If the Server receives the new requests before finishing the ongoing one, the new requests are queued.

As the Server services only one client at a time and the Client directly interacts with the Server without undergoing any security checks by some middleware, the Server is of iterative type and client-server architecture is two-tier.[1] Each Server-Client connection is non-persistent.[2]

File type can be any of the following: PDF, JPG, PNG, HTML or TXT.

The programs transmit the files of size up to 2 MB.

1.2 Instructions

In order to execute both programs on the operating system Ubuntu, follow the step-by-step instructions bellow.

1.2.1 Instructions for the Server

1. Place the Server program "server.c" in an arbitrary directory, e.g. in a folder named "Server" (~/Server) in the home directory.
2. Place the data to be requested from the Client in an arbitrarily named folder in the Server program's directory, e.g. in a folder named "files" (~/Server/files).
3. Open the terminal and type the following command into it to select the Server's directory:

   cd ~/Server
4. Compile the Server program by typing the following command into the terminal:

[1] https://www.tutorialspoint.com/unix_sockets/client_server_model.htm
[2] https://tools.ietf.org/html/rfc7230#section-6.3

gcc -g -Wall -Wextra -std=c99 -o server server.c

5. To assign a certain port to the Server (e.g. a port number 4444) and execute the Server program, type the following command into the terminal:

   ./server ./files 4444

   Where "./files" is the name of the folder mentioned in step 2.
   The terminal output should be as follows:

```
 % ./server "./files" 4444
Socket created...
Binding done...
-------------------------------------------------------
Waiting for a connection...
```

   Now the Server is running and listening for the oncoming requests from the clients.

If the Client connects the Server and requests to receive the file (e.g. ‘‘file.txt’’) from the directory set by the user (~/Server/files), the output should be as follows, provided the requested file was found and successfully sent to the Client:

```
Connection accepted...
Connection received from: elgar.org.aalto.fi [IP: 130.233.241.29]
Processing request...
file.txt is requested (19233781 bytes).
Sending the file...
File is sent.
-------------------------------------------------------
Waiting for a connection...
```

If the Client connects the Server and requests to send the file (e.g. ‘‘file.txt’’) to the directory set by the user (~/Server/files), the output should be as follows, provided the requested file was successfully received from the Client:

```
Connection accepted...
Connection received from: elgar.org.aalto.fi [IP: 130.233.241.29]
Processing request...
Receiving the file...
Parsing the header...
File is received.
File directory is ./files/file.txt
-------------------------------------------------------
Waiting for a connection...
```

1.2.2 Instructions for the Client

1. Place the Client program ‘‘client.c’’ in an arbitrary directory, e.g. in a folder named ‘‘Client’’ (~/Client) in the home directory.
2. Place the data to be shared in an arbitrarily named folder in the Client program’s directory, e.g. in a folder named ‘‘files’’ (~/Client/files).
3. Open the terminal and type the following command into it to select the Client’s directory:

   cd ~/Client

4. Compile the Client program by typing the following command into the terminal:

   gcc -g -Wall -Wextra -std=c99 -o client client.c

5. To assign a certain port to the Client (e.g. a port number 4444) and receive a file (e.g. ‘‘file.txt’’) from the Server with a certain IP address (e.g. 130.233.241.29), type the following command into the terminal:

   ./client 130.233.241.29/file.txt ./files 4444 GET

   where ‘‘./files’’ is the name of the folder mentioned in step 2, 4444 is the Server’s port number, and ‘‘130.233.241.29/file.txt’’ is the Server’s IP address and the name of the requested file, separated by slash.
   The output should be as follows, provided the requested file was successfully received from the Server:

```
% ./client 130.233.241.29/file.txt ./files 4444 GET
Socket created...
Trying to connect 130.233.241.29...
Connection successful.
Requesting file.txt...
Parsing the header...
File is received.
File directory is ./files/file.txt
```

   To assign a certain port to the Client (e.g. a port number 4444) and send a file (e.g. ‘‘file.txt’’) to the Server, type the following command into the terminal:

   ./client 130.233.241.29/file.txt ./files 4444 PUT

   where ‘‘./files’’ is the name of the folder mentioned in step 2, 4444 is the Server’s port number, and ‘‘130.233.241.29/file.txt’’ is the Server’s IP address and the name of the requested file, separated by slash.
   The output should be as follows, provided the requested file was successfully sent to the the Server:

```
% ./client 130.233.241.29/file.txt ./files 4444 PUT
Socket created...
Trying to connect 130.233.241.29...
Connection successful.
Sending the file...
File size is 19534277 bytes.
File is sent.
```

1.3 Protocols

The programs use TCP (Transmission Control Protocol) in the transport layer, IP addresses in the network layer and elements of HTTP/1.1 (Hypertext Transfer Protocol) in the application layer to exchange files and requests.

1.3.1 HTTP

The programs make use of particular elements of The Hypertext Transfer Protocol (HTTP/1.1) to construct and exchange the requests of methods PUT and GET, responses (‘‘200 OK’’, ‘‘400 Bad Request’’ and ‘‘404 File Not Found’’) and headers.

GET

Following the standard of HTTP/1.1, the GET method means retrieve whatever information (e.g. file) is identified by the Request-URI.[3]

When requesting to receive the file, the Client constructs a file request which includes a method token[4] GET and request-URI (file name),[5] partially following the standard of HTTP/1.1. However, unlike in HTTP/1.1, the Client's file request line does not contain HTTP-Version of the protocol.[6] Client's file request is always terminated by the newline. No additional information is specified. File name includes relative or absolute path, only if the name of the file on the Server does.

An example GET request message would be:

GET book.pdf

PUT

Partially following the standard of HTTP/1.1, the PUT method requests that the incoming file be stored under the supplied name (or Request-URI).[7] On the other hand, unlike in HTTP/1.1, it is not the enclosed entity that PUT requests to be stored,[8] but the incoming file, which is going to be sent by the Client to the Server later, after the HTTP handshake.

Analogous to the GET method procedure, when sending the file, the Client also constructs a file request which includes a method toke PUT and request-URI (file name) terminated by the newline with no additional information specified.

An example PUT request message would be:

PUT image.jpg

Header

Header is constructed and sent either by the Server, when one has received a file request of method type GET from the Client, or by the Client, when one has sent a file request of method type PUT to the Server.

Header fields semantics partially follows the standard of HTTP/1.1.[9] The header fields are Date, Location and Content-Type.

Following the standard of HTTP/1.1, the Date field represents date and time at which the header was originated. Date format follows ANSI C's asctime() standard, also supported in HTTP/1.1.[10]

Partially following the standard of HTTP/1.1,[11] the Location field represents the directory of the requested file either in the Server's folder, if the Client is receiving the file, or in the Client's folder, if the Server is receiving the file. An example Location field would be:

Location: ./files/image.png

Partially following the standard of HTTP/1.1, the Content-Type header field indicates the media-type of the file, which is going to be transmitted. The Content-Type field follows the media-type values

---

[3] https://tools.ietf.org/html/rfc2068#section-9.3
[4] https://tools.ietf.org/html/rfc2068#section-5.1.1
[5] https://tools.ietf.org/html/rfc2068#section-5.1.2
[6] https://tools.ietf.org/html/rfc2068#section-5
[7] https://tools.ietf.org/html/rfc2068#section-9.6
[8] https://tools.ietf.org/html/rfc2068#section-7
[9] https://tools.ietf.org/html/rfc2068#section-14
[10] https://tools.ietf.org/html/rfc2068#section-3.3
[11] https://tools.ietf.org/html/rfc2068#section-14.30

registered with the Internet Assigned Number Authority (IANA),[12] also supported in HTTP/1.1.[13] The programs share the files of any of the following media-types: text/html, application/pdf, image/jpeg image/png.

All three fields are present in the header in the declared sequence (Date, Location, Content-Type) and separated by the newline, and, following the standard HTTP/1.1, each header field consists of a name followed by a colon (":") and the field value.[14] An example header would be:

Date: Wed Mar 29 10:30:14 2017
Location: ./files/page.html
Content-Type: text/html

200 OK

Following the standard HTTP/1.1, ‘‘200 OK’’ response indicates that the request has succeeded.[15] ‘‘200 OK’’ responses are sent by the programs to each other to confirm that the previous message (e.g. response, request or header) from the sender was received and processed.

400 Bad Request

Partially following the standard HTTP/1.1, ‘‘400 Bad Request’’ response indicates that the header could not be understood due to malformed syntax and sent to the one who constructed the header.[16]

404 Not Found

Partially following the standard HTTP/1.1, ‘‘404 Not Found’’ response indicates that the Server has not found the file matching the file name encapsulated in the file request by the Client.[17]

1.3.2 TCP

TCP (Transmission Control Protocol) was selected to implement the data transfer in the transport layer between the Server and the Client within this project, primarily because TCP provides a reliable data transfer service to HTTP. This implies that each HTTP request message sent by the Client process eventually arrives intact at the Server; similarly, each HTTP response message sent by the Server process eventually arrives intact at the Client, unlike UDP (User Datagram Protocol), which is connectionless and sends independent packets of data from one end system to the other, without any guarantees about delivery.[18]

2. Program Functions and Algorithms

This section describes purposes, contents, arguments and return values of either program’s functions and algorithms followed by the Server and the Client in case of two different method types.

---

[12] https://www.iana.org/assignments/media-types/media-types.xhtml
[13] https://tools.ietf.org/html/rfc2068#section-3.7
[14] https://tools.ietf.org/html/rfc2068#section-4.2
[15] https://tools.ietf.org/html/rfc2068#section-10.2.1
[16] https://tools.ietf.org/html/rfc2068#section-10.4.1
[17] https://tools.ietf.org/html/rfc2068#section-10.4.5
[18] Computer networking: a top-down approach, p. 94-95, 157

2.1 Server

2.1.1 Server Functions

- `int createSocket(int port)`

This function creates a TCP/IPv4 socket of type SOCK_STREAM and, with the aid of the function bind(),[19] assigns to it a port provided by the user and received as an argument and sockaddr_in structure[20] which returns socket addresses for any address family (AF_UNSPEC), IPv4 or iPv6 for example, and contains the "wildcard address" (INADDR_ANY), which means that the Server's IP address will be assigned automatically,[21] making it possible for the socket to accept connections from the clients over the port set by the user.

The return is a file descriptor of the socket (sockfd).

- `int listenForRequest(int sockfd)`

With the aid of the function listen(), this function marks the file descriptor of the socket sockfd, received as an argument, as a passive socket, that is, as a socket that will be used to accept incoming connection requests from the clients and specifies the maximum number of connections the kernel should queue for this socket (five).[22]

The function accept() is called to return the next completed connection from the Client. The function accept() creates a new connected socket without affecting the original socket, which was created in the previous function createSocket(), and returns a new file descriptor referring to that socket (conn) with sockaddr_in structure bound to it.[23]

IP address of the Client which is connecting the Server is converted from a binary number to a printable form (character string) by using the function inet_ntop().[24] The character string IP is then converted into a network address structure (n_addr) by using the function inet_pton,[25] whereupon the ‘‘official’’ name of the host (i.e. the Client) corresponding the initial IP address is extracted by using the function gethostbyaddr()[26] and the structure of type hostent.[27] (The technique of ‘‘mapping back and forth between host names and IP addresses’’ is described in Beej's Guide to Network Programming.[28]) Host name of the Server (if there is one) is printed.

The return is a file descriptor referring to the socket (conn).

- `int HeaderParser(char * header)`

This function parses the header received as an argument to determine if all the necessary fields (Date, Location and Content-Type) are present in the header. Every line of the header is compared to a two-dimensional array containing all the necessary header fields.

Function returns 0, if the header’s fields are adequate. Otherwise the return is 1.

---

[19] http://man7.org/linux/man-pages/man2/bind.2.html
[20] http://beej.us/guide/bgnet/output/html/multipage/sockaddr_inman.html
[21] https://www.tutorialspoint.com/unix_sockets/socket_core_functions.htm
[22] https://linux.die.net/man/2/listen
[23] http://man7.org/linux/man-pages/man2/accept.2.html
[24] http://man7.org/linux/man-pages/man3/inet_ntop.3.html
[25] http://man7.org/linux/man-pages/man3/inet_pton.3.html
[26] https://linux.die.net/man/3/gethostbyaddr
[27] https://www.gnu.org/software/libc/manual/html_node/Host-Names.html
[28] http://beej.us/guide/bgnet/output/html/multipage/gethostbynameman.html

- `char * FileType(char * file)`

This function determines the type of the file received as an argument. As mentioned in the section 1.3.1 HTTP, file types correspond the media-type values registered with the Internet Assigned Number Authority (IANA), and can take any of the following values: "text/html", "application/pdf", "image/jpeg" or "image/png".

Function returns the media-type of the file as a string, if one was determined. Otherwise the return is NULL.

The rest of the Server program deals with exchanging HTTP requests and responses with the Client, receiving and sending the header as well as receiving and sending the requested file, which is all implemented in the main function and described in more detail in the section 2.1.2 Server algorithms.

2.1.2 Server Algorithms

Here are two basic algorithms followed by the Server when one is receiving a file (PUT method) and sending a file (GET method), with no errors occurring. Both algorithms are preceded by the same operations though. All the functions called inside the main functions are described in the section 2.1.1 Server Functions.

The Server receives a port number and directory of the files to be shared as the arguments from the user, dynamically allocates size of 1024 bytes for a header, request from the Client and name of the requested file (two strings).

The Server creates a TCP socket by calling the function createSocket() and starts listening for the oncoming requests from the clients by calling the function listenForRequest(). Sending and receiving of the requests & responses is implemented by using the functions recv()[29] and send().[30]

After receiving the file request from the Client, the Server parses it by extracting the type of the file and a method type (GET or PUT).

1. Method is of type GET

After determining the type of the file with aid of the function FileType(), the Server constructs a header, which contains the fields Date, Location and Content-Type.

The Server attempts to open the requested file from the directory provided by the user and, if unsuccessful there, sends the error message "404 Not Found" to the Client, who, having received it, closes the TCP connection immediately. If, however, the file is found, the Server determines and prints the size of the file in bytes, sends "200 OK" to the Client, receives "200 OK" back from the Client and sends the header. If the Client detects no errors in the header, it sends "200 OK" back to the Server, who, having received this confirmation, sends the requested file. When the file is sent, Server closes both the socket and the file and continues to listen for the oncoming the requests from the clients.

2. Method is of type PUT

The Server sends "200 OK" to confirm that the command (i.e. method PUT) is clear, receives "200 OK" and the header back from the Client. The header is verified by the function HeaderParser() and, if no errors were detected, "200 OK" is sent to the Client, whereupon the Server receives the file, closes both the socket and the file and continues to listen for the oncoming requests from the clients.

[29] http://man7.org/linux/man-pages/man2/recv.2.html
[30] http://man7.org/linux/man-pages/man2/send.2.html

Both algorithms (sending the receiving) branch out of the infinite for-loop as the conditions of the if-statements. If any errors occur during the exchange of HTTP messages, the Server prints the possible cause of the error, closes the socket and passes to the beginning of the for-loop, thus continuing to listen for the oncoming requests.

2.2 Client

2.2.1 Client Functions

Two of the Client's functions, HeaderParser() and FileType(), are the duplicates of the Server's functions of the same name. These functions are not described here. See "Server Functions" for more information.

- `int ValidIP(char * ip)`

This function attempts to convert the IP address provided by the user and received as an argument into a network address structure of IPv4 address family, thus verifying the validity of the IP address.

The return is 0, if the IP address is invalid. Otherwise, the return is 1.

- `int request(char * url, int port, char * command)`

This function receives the Server's IP address & file name as one string, a port number and a request method. The IP address is separated from the file name and verified by the function ValidIP(). If the IP address is valid (i.e. of version IPv4), the Client determines the request method (either GET or PUT) and constructs a corresponding request which contains a method type (either GET or PUT) and a name of the requested file.

The Client creates a TCP/IPv4 socket of type SOCK_STREAM and binds to it the port number and the Server's IP address provided by the user.[31] The socket is then connected to the address structure (sockaddr_in) by using the function connect(),[32] whereupon the Client sends the request to the Server by using the function write().[33]

The return is a file descriptor of the socket (sockfd).

- `int HeaderParser(char * header)`

See 2.1.1 Server Functions, HeaderParser().

- `char * FileType(char * file)`

See 2.1.1 Server Functions, FileType().

2.2.2 Client Algorithms

Similar to the Server, the Client also follows two basic algorithms: when one is receiving a file (GET method) and sending a file (PUT method), with no errors occurring. However, in contrast to the Server, the

[31] http://man7.org/linux/man-pages/man2/bind.2.html
[32] http://man7.org/linux/man-pages/man2/connect.2.html
[33] https://linux.die.net/man/2/write

Client operates in accord with the methods set by the user. Both algorithms are also preceded by the same operations.

The Client receives the Server's IP address & file to be requested as one string, a directory of the files, a port number and a method type and dynamically allocates size of 1024 bytes for a header and a directory of the files. Directory of the files is then separated from the Server's IP address and combined with the name of the file to be requested, whereupon the Client determines the type of the file by using the function FileType() and calls the function request() to send the file request corresponding to the method, set by the user, to the Server.

1. Method is of type GET

The Client receives "200 OK" from the Server, thus confirming that the request is clear, replies "200 OK" back, receives the header and verifies it by calling the function HeaderParser(). If no errors were detected in the header, "200 OK" is sent to back the Server, whereupon the Client receives the file. Both the socket and the program are closed afterwards.

2. Method is of type PUT

The Client attempts to open the file to be sent, determines and prints its size in bytes. "200 OK" is then received from the Server, thus confirming that the request is clear. The Client replies "200 OK" back, constructs a header and sends it to the Server, who replies "200 OK" back, if no errors were detected in the header, whereupon the Client sends the file to the Server. Both the socket and the program are closed afterwards.

Both algorithms (sending the receiving) are executed as the conditions of the if-statements in the main function. If any errors occur during the exchange of HTTP messages, the Client prints the possible cause of the error and closes the program.

3. Errors and Exception Handling

The Server and the Client are provided with several error handling mechanisms which print the most possible cause of the error and, if possible, prevent the programs from shutting down (e.g. due to a segmentation fault or the Client's disconnection).

Here are all possible errors, which may occur during the execution of the programs, and error handling associated with them.

3.1 Server

If the number of arguments set by the user does not equal three, the error description is printed ("Not enough arguments!") and program is closed.

If the directory of the files to be shared is not found, the error description is printed ("Directory is not found!") and program is closed.

If the file requested by the Client is not found, "404 Not Found" is sent to the Client, the error description is printed ("Requested file is not found!"), whereupon the Server closes the socket and continues to listen for the oncoming requests.

If the Client does not reply "200 OK", when one is expected to reply that, the error description is printed ("Operation aborted by the Client!"), whereupon the Server closes the socket and continues to listen for the oncoming requests.

If the Client does not send the header, when one is expected to send it, the error description is printed ("Error receiving HTTP header!"), whereupon the Server closes the socket and continues to listen for the oncoming requests.

If the header received from the Client does not contain all the necessary fields or the fields are inadequate, "400 Bad Request" is sent to the Client, the error description is printed ("Error in HTTP header!"), whereupon the Server closes the socket and continues to listen for the oncoming requests.

If the Server fails to open the file for writing, the error description is printed ("Error opening file!"), whereupon the Server closes the socket and continues to listen for the oncoming requests.

If the Server fails to create the socket, the error description is printed ("Error creating socket!") and program is closed.

If the Server fails to bind the socket to an address structure or a port, the error description is printed ("Error binding socket to port!") and program is closed.

If the Server fails to accept the connection from the Client, the error description is printed ("Error accepting connection!") and Server closes the socket and continues to listen for the oncoming connections.

3.2 Client

If the request method set by the user does not correspond either GET type or PUT type, the error description is printed ("Invalid command!") and program is closed.

If the port number set by the user exceeds the range of all possible UNIX ports (from 1 to 65535),[34] the error description is printed ("Invalid port number!") and program is closed.

If the number of arguments set by the user does not equal five, the error description is printed ("Not enough arguments!") and program is closed.

If the Server does not reply "200 OK", when one is expected to reply that, the error description is printed ("Operation aborted by the Server!") and program is closed.

If the header received from the Server does not contain all the necessary fields or the fields are inadequate, "400 Bad Request" is sent to the Server, the error description is printed ("Error in HTTP header!") and program is closed.

If the file is failed to open for writing, the error description is printed ("Error opening file!") and program is closed.

If the Server does not send the header, when one is expected to send it, the error description is printed ("Error receiving HTTP header!") and program is closed.

If the Server's IP address set by the user is not of version IPv4, the error description is printed ("Invalid IP!") and program is closed.

If the Server has not found the file requested by the user, error description is printed ("File is not found!") and program is closed.

If the Client fails to create the socket, the error description is printed ("Error creating socket!") and program is closed.

If the Client fails to connect the Server, the error description is printed ("Connection error!") and program is closed.

4. Tests

The programs have been tested and shown to be running properly on Aalto University's computers with the operating system Ubuntu. Tests have also shown that the programs fulfill the project requirements. Here is a complete report of test results:

- Files of types PDF, HTML, TXT, JPEG and PNG with size less than 2 megabytes are successfully transmitted between the Server and the Client without losses.

[34] https://www.tutorialspoint.com/unix_sockets/ports_and_services.htm

- Relatively large files with size of, say, 40 megabytes damage and become unreadable after about the third iterative exchange between the Server and the Client.
- For some unknown reason, the Server fails to bind a socket to the same port set by the user about 1 out of 10 times. In this case, it is recommended to recompile and re-execute the program.
- The programs run properly on IPv6, but, because neither the Server nor the Client has IPv4/IPv6 dual-stack capability, to make it possible for the Client to bind the IPv6 sockets and for the Server to accept IPv6 connections, several modifications are to be made manually in both the Server and the Client programs beforehand.
- In theory, the Server can service at most 5 clients at a time. However, putting this theory to the test on the university's computers is relatively challenging, because every transmission of small files is normally done in few milliseconds and queues do not simply form in such little time, no matter how close to synchronously several clients request different files from the Server. To test this theory, transmission distance between the Client and the Server should be, say, at least a few hundreds of kilometers to heavily increase propagation delay.
- The Server does not normally shut down, when the Client suddenly disconnects during the HTTP handshaking or file transmission, and simply continues to listen for the oncoming requests.

5. Reference List

- https://www.tutorialspoint.com/unix_sockets, Unix Socket Tutorial
- http://man7.org/linux, Linux manual pages
- http://beej.us/guide/bgnet/, Beej's Guide to Network Programming
- https://tools.ietf.org/html/rfc7230, RFC 7230 - Hypertext Transfer Protocol (HTTP/1.1)
- https://www.gnu.org/software/libc/manual, The GNU C Library
- https://linux.die.net, Linux documentation
- Kurose, Ross: Computer Networking, 6th edition (Addison-Wesley)