**APPLICATION PROTOCOL**

**Architecture**

Client-server architecture is used for the network application. Server is given an IP address which does not change throughout the session. They don’t need to establish connection before they can communicate. There is no handshaking between UDP sender and receiver. Each UDP segment is handled independently of others. All the network communication uses UDP with possible segment loss involved. Therefore, the stop-and-wait protocol is implemented on both client and server side in order to have data reliably transferred. Client requests services from the server and server responds to it.

* **Types of Messages**

**Request**: client sends a request to the server.

**Response:** To answer to request of client, the server either sends client data or do the data translation job.

* **Syntax and Semantics of message type**

All these messages (request and response) are sent in packet buffers whose individual size is MAX\_LINE (1000 bytes) plus 1 addition byte. The first byte of each packet is a sequence number. In the format and meaning of the messages below, we talk about the actual data of the packet buffer we need.

* **Request**

Format of the message

[file size] [data] [file format] [output filename size] [output filename]

[file size]: 8 bytes

[data]: 1000 bytes

[file format]: 4 bytes

[output filename size]: 8 bytes

[output filename]: (filename size) bytes of ASCII characters

Meaning of each field in the format

[file size]: size of the file

[data]: data of the file stored in buffer

[file format]: format number that specifies the different types of translation

[output filename size]: size of the output file name

[output filename]: name of the output file where the translated data are written

The value range of each field

**Size of primitive data types may differ in different platform**

[file size]:

[data]: file size

[file format]:

[output filename size]:

[output filename]: output filename size

* **Response**

Format of the message

[acknowledgment] [ error message]

[acknowledgment]: 1 byte

[error message]: 1 byte

Meaning of each field in the format

[acknowledgment]: acknowledgment for receiving message from the client

[error message]: char that tells whether the format of the units in the file was ok or not

The value range of each field

[file size]:

[error message]:

* **Rules of sending messages**
* Client always sends the request to the server
* Server responds to the request of the server
* There is always one response for one request.
* Messages must follow the format and syntax mentioned above.
* If not followed, the program will throw an error.

**TEST CASES**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Input | Expected Output | Actual Output | Rationale | Content | Errors |
| practice\_project\_test\_file\_1 | Success | Success | Normal functionality | Data from test file 1 from practice project | No error |
| practice\_project\_test\_file\_2 | Success | Success | Normal functionality | Data from test file 2 from practice project | No error |
| largeFile | Format error | Format error | Sending and receiving file larger than buffer size. | Data in largeFile.txt | No error |
| wrongType | Format error | Format error | Testing for the incorrect type (incorrect format) | Data in wrongType.txt | No error |

**USAGE OF CLIENT AND SERVER PROGRAM**

* **Usage of Client**

The following command invokes the client:

<client> <server IP> <server port> <file path> <to format> <to name> <loss probability>

<random seed>

Where

* <client> is the name of the client executable file name,
* <server IP> is the IP address of the server
* <server port> is the TCP of the server
* <file path> is the path of the file to be sent to the server (the file path indicates the location of the file in the system on which the server runs. It includes the file name, and possible the hierarchy of directories.)
* <to format> indicates how the server should translate the received units.

0 means no translation, 1 means to only translate type 0 units to type 1 with type 1 units unchanged, and 3 means to translate type 0 to 1 and type 1 to 0.

* <to name> is the name of the file the server should save the units to
* <loss probability> is the probability of segment loss between 0 and 1.
* <random seed> is an integer to control random number generation.
* Setting the same random seed and loss probability will provide the same segment losses across multiple executions.

If there are not enough or more than enough arguments, format is not in the specified range (0 to 3), then the program is terminated. After creating the socket, setting the remote IP address, client sends the file size to the client after calculating the file size. Followed by that, client sends data in the file to the server. Thereafter, client sends file format to the server. Client sends size of the output file name and also sends the output file name. Lastly, the client sends a message saying that it’s done. For each messages client sends to server, the message contains a sequence number. For the following acknowledgment client receives from the server, client expects to receive the same sequence number or else client will retransmit the packet. Right after receiving the acknowledgment, the client updates the sequence number, either 1 or 0.

Finally, the client receives the error message from the server. 0 is a confirmation message saying that the data was translated and saved in the destination file by the server. -1 means that there was an error translating the file. After that, the client sends a message saying that it is done. Finally, the client closes the TCP connect and returns EXIT\_SUCCESS.

* **Usage of Server**

The following command invokes the server:

<server> <port> <loss probability> <random seed>

Where

* <server> is the name of the server executable file name
* <port> is the port the server listens to
* <loss probability> and <random seed> servers the same purpose as explained for the client.

If enough arguments (4) are not passed, then the server program is terminated. After the necessary setup is made (creating listening socket, filling relevant data members in socket address structure, binding socket address to listening socket), the program enters into an infinite loop to listen to client requests. The client will send the units in the file from the path on the server. The server will check the received units. If any unit has wrong format, the server will simply send back an error message and close the connection. If everything is right, the server will translate type 0 units to type1 and type 1 units to type 0, then save them to the specified file in the directory which the server application is in and send a confirmation message and close the connection.

For every acknowledgment packet the server sends, server expects to receive the same sequence number that the acknowledgment packet it sent had. Since the client won’t send an acknowledgement for the error message the server sends, the server sends the message for MAX\_TRIES just to make sure that at least one of them reaches the client. Followed by that, we close the listening socket.

**INSTRUCTIONS TO COMPILE CLIENT**

The client can be complied by the following command in the command line:

gcc <client.c> <helper.\*> <sendlib.\*> -o <filename>

where,

* <client.c> is the client c file
* <helper.\*> is the file that contains helper.c and helper.h that contains implementation and definitions of helper functions
* <sendlib.\*> contains sendlib.c and sendlib.h that implements and declares sendlib function
* <filename> is the compiled executable file name

**INSTRUCTIONS TO COMPLIE SERVER**

The server can be compiled by the following command in the command line:

gcc <server.c> <helper.\*> <sendlib.\*> -o <filename>

where,

* <server.c> is the server c file
* <sendlib.\*> same as in client.
* <helper.\*> same as in client.
* <filename> is the compiled executable file name

**KNOWN PROBLEMS**

As of now, the file translation is wrong.

**SINGNIFICANT REFERENCES**

For the helper functions, significant references were made from:

**Donahoo, M. J., & Calvert, K. L. (2001). TCP/IP SOCKETS IN C: Practical Guide for Programmers (The Practical Guide Series). San Francisco, CA: Morgan Kaufmann.**

**GITHUB LINK**

You can go to the following link to access the network program.

https://github.com/henchhing-limbu/UDP\_Socket\_Programming