Introduction.

This project utilizes UDP sockets on a UNIX system to implement a file exchanger. The system designed consists of the client and server programs. The order of which program is first launched does not matter. Next data flow occurs during the transfer:

1. The Client and Server programs are launched.
2. Each program accepts a set of required options provided via the command line interface.
3. The client program reads the data to be transferred from the 10 files specified by the user.
4. The client transfers the data to the server via a udp socket.
5. The server reconstructs the data into a single packet, saves the data as the file, and responds to the client with a single packet containing the reconstructed data.
6. The client receives the reconstructed data and saves the data.

Results:

The client and server programs were written in a manner to meet these design specifications

1. Per project requirements, the server listens on the udp port 7777 per project requirements.
2. Per project requirements, the client specifies the FILENAME, TOTAL NUMBER OF LINES, and LINE NUMBER in the application layer of each packet.
3. Per project requirements, the server using information mentioned above reconstructs the file, stores the file on the server and retransmits the reconstructed file back as a single packet.
4. Per project requirements, the server does not send a FIN to Client until all data is received.
5. Extra point, the client listens on the UDP port specified by the user via command options (in demonstration script, the 7778 port is used for the client listener since the loop back interface was used to test the two programs; no two applications can use the same port number).
6. The Client and Server programs use calloc() and malloc() to allocate the exact amount of memory as required by each line from a file instead of instantiating an array of fixed size and checking whether the number of characters per line exceeds the size of an array element. Using calloc() and malloc() to dynamically allocate the memory increases the code complexity but yields reduces memory utilization. The code complexity was also increased since each line of the file had to be manually parsed to count the characters per line and to determine the number of lines in each file.
7. Both Client and Server implement GNU's Argp() function to parse the arguments. The arguments specify such information as IP address, port, source file, destination file instead of hard typing this information into the source code. This feature added flexibility at the cost of the increased code complexity.
8. While the project requirements do not specify, the socket timeout option is implemented to improve reliability.
9. The number of files, the maximum lines per file, and the size of the receiver buffer are required to be specified using these MACROs set in the project/sw/client/makefile and project/sw/server/makefile files:
   1. -DMAX\_NUM\_OF\_FILES=10
   2. -DMAX\_NUM\_OF\_LINES=1024
   3. -RECEIVE\_BUFFER\_SIZE=9000

The client and server programs implement the two state machines shown in the figure below:

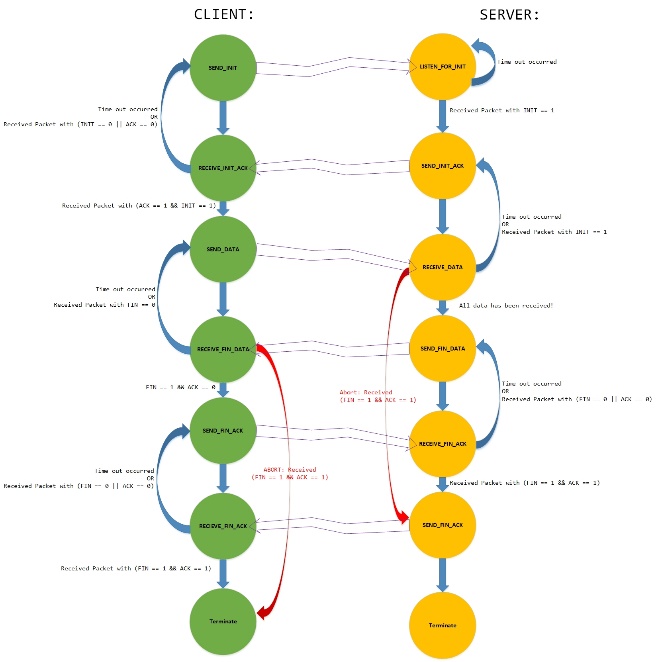


Figure 1. The Client and Server FSMs are shown in green and yellow respectively.

In the figure above, the blue state transitions show normal program flow. The red state transitions show the program flow when abort takes place. The orange double lined arrows show the data flow between the client and the server as the data exchange takes place.

Next application layer header is prepended to the payload of the packets by both client and server programs:

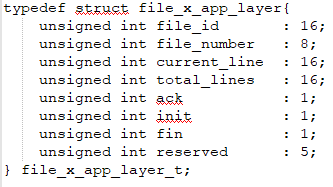


Figure 2. The data structure implemented in C programming language is shown.

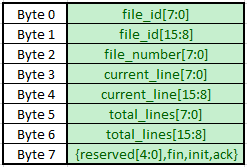


Figure 3. The memory layout produced by the implemented data structure (See figure 2 above).

To avoid confusion that may occur due to the difference between a 32-bit and a 64-bit machines, the memory layout shown in Figure 3 is provided. It should be noted that the data is not converted to big endian byte order prior being transmitted via a UDP socket. Thus both client and server expect the data to be received in the little endian order.

Referencing the state machines provided in Figure 1 and the data fields shown in Figure 2 and Figure 3, the client and server communication is descrided:

1. The server listens for INIT packets to be received.
2. Client sends an INIT packet specifying next parameters using these field:
   1. **file\_id**: specifies the unique transaction number. Currently, the server and client programs are launched and terminated once per file exchange. However, should the server program functionality to be extended in the future to continuously monitor the incoming connections, this field can uniquely tag a transaction per given file.
   2. **file\_number**: not used in INIT packet.
   3. **current\_line**: specifies the port the Clients listens receives udp packets from (the recvfrom() port).
   4. **total\_lines**: specifies the number of files to be transferred.
   5. **ack**: not used
   6. **init**: set to specify it’s an INIT packet.
   7. **fin**: not used.
   8. **reserved**: not used.
3. The Server responds to INIT packet with INIT|ACK packet by setting the **ack** field of the INIT packet and retransmitting the resultant packet back to Client.
4. The Client receives the INIT|ACK packet from the Server and responds by sending the packets containing each line of text from each file to the Server. In these packet, the application layer fields contain these data:
   1. **file\_id**: specifies the unique transaction number. Currently, the server and client programs are launched and terminated once per file exchange. However, should the server program functionality to be extended in the future to continuously monitor the incoming connections, this field can uniquely tag a transaction per given file.
   2. **file\_number**: specifies the file number the payload corresponds to.
   3. **current\_line**: specifies the line number of the text contained in the payload.
   4. **total\_lines**: specifies the number of lines per this file. Only used on initial receival of the packet for a given file to determine the number of lines to expect for the file.
5. The Client transitions to the RECEIVE\_FIN\_DATA state listening for the Server to acknowledge the received data.
6. If the Server does not receive all the data, the Client does not receive an acknowledgement which causes the Client to retransmit the data to the Server due to a timeout that takes place. If the Server receivs all the data, the server sends the FIN|DATA packet with the **fin** field set and containing the reconstructed data as a payload.
7. Upon receival of the reconstructed data by the Client, the Client responds with the FIN|ACK packet as an acknowledgement of the successful data transaction. The fin and ack fields are set.
8. The Server then responds to the FIN|ACK packet with the copy of the FIN|ACK packet and terminates its execution.
9. The Client upon receival of the FIN|ACK packet from the Server also terminates. If not, the Client retransmits the FIN|ACK packet until the acknowledgement is received or the process is terminated by the user. At this point, both the Client has definitely received all the data and we only care that the Server has been terminated by the client’s FIN|ACK packet.

Instruction:

The code is provided using next directory tree:

1. project/ directory contains these scripts:
   1. demonstration.sh: builds the client and server executables, generates 9 text files from file0.txt, launches the wireshark application, launches the client and server executables producing “project/sw/client/file.txt” file containing reconstructed data.
   2. makeall.sh: builds the client and server executables in the project/sw/client/ and project/sw/server directories where the executables are to be run manually by the user.
   3. makeclean.sh: removes executables and cleans directories in the project/sw/client/ and project/sw/server/ directories.
   4. run\_client.sh: generates the 9 text files from file0.txt, builds the client program, launches the client program using loopback interface and the 7778 recvfrom port.
   5. run\_server.sh builds the server program, and launches the server application.

To begin, the user may choose first to run the client and server programs automatically as a demonstration:

1. Go to project/ directory.
2. Launch demonstration.sh script
3. Observe the wireshark’s output.
4. Observe the output on the command line for information regarding how to run the client and server applications manually or read the section below.

To run the Client and Server application manually:

1. Run project/makeall.sh script.
2. Run project/sw/client/client.run application with these options:
   1. --source-port 7778
   2. --dest-ip 127.0.0.1
   3. --dest-port 7777
   4. --output-file file.txt
   5. Followed by the 10 filenames.
3. Run project/sw/client/server.run application with these options:
   1. --source-ip 127.0.0.1
   2. --source-port=7777
4. Example:
   1. project/makeall.sh
   2. project/sw/client/duplicate\_file0.sh
   3. project/sw/server/server.run --source-ip 127.0.0.1 --source-port=7777 -v -x
   4. project/sw/client/client.run --source-ip 127.0.0.1

--source-port 7778

--dest-ip 127.0.0.1

--dest-port 7777

--output-file file.txt

-v -x

file0.txt file1.txt file2.txt file3.txt file4.txt file5.txt file6.txt file7.txt file8.txt file9.txt