CSCD58 - Final Project Report

Group members:

Denis Dekhtyarenko 1006316675 dekhtyar
Joshua Carrasco Sousa 1004948934 carras28

Description of project:

Our project is an instant-chat messaging application that allows users to communicate with each other via the command line.

Note: the video demo for the project includes a bit with E2E encryption disabled. It works as intended in the final implementation

Rationale of project:

An instant-chat messaging application requires secure internetwork communication. During the implementation of the project we gained a hands-on understanding of the higher level network stacks such as the Application, Presentation, Session, and Transport layers.

How the project relates to the field of “Computer Networks”:

Our project will require us to interact with the TCP protocol and interact extensively with an application layer protocol to send and receive data via the network. During implementation of our project, we will be exposed to the Application, Presentation, Session layers of the network stack which have been thoroughly studied during the course.

Goals:

1. A client capable of accepting user input and sending user input to a central server. This goal provides a basic understanding and hands-on experience with the TCP protocol.
2. Designing and implementing a simple request-based application level protocol over TCP that is tailored to our use cases with our instant-chat messaging application. This goal provides a better understanding of the application layer from the perspective of an application layer protocol designer.
3. A central server capable of handling connections to multiple clients. This goal serves to increase our understanding of the client-server model.
4. A server capable of receiving messages from clients and sending them to the respective clients. This goal provides us with a greater understanding of the difficulties and approaches to data transfer between clients with the client-server model.

5. Implementing end-to-end encryption of messages using public-key cryptography, so that only the sender and recipient of a message will know message content. This goal serves to give us hands-on experience with the session layer and give us insight into difficulties faced when dealing with encryption.

Contributions of each member of the team

Joshua:

- Basic client-server TCP connection via socket.
- Designed D58P protocol.
- Multithreaded server side logic for handling D58P requests.
- Long polling logic for sending messages from client to client (through server).
- Server side logic for authentication, and messaging.
- Server side logic for ensuring most recent connections are used for handling Get Message requests from clients.
- netutils.c library for creating various types of D58P requests, responses.

Denis:

- Client side logic for handling and parsing user input to create D58P requests.
- Client side logic to send D58P requests and process responses.
- Multi threaded client side logic for having a separate thread constantly getting messages by sending D58P Get Message requests.
- Implementation of end-to-end public key encryption using the OpenSSL development library (RSA)

How to compile and run the project

Our project is written in C. Our project uses libraries available on linux, and is untested on Windows and Mac. The executables for client and server are provided in case compiling has some issues.

Dependencies: Our project uses OpenSSL encryption library. So install OpenSSL on your machine to compile the application. Some machines come with OpenSSL already installed.

To install it you can run:

```
$ sudo apt-get install libssl-dev
```

Note: the UTSC Mathlab machines have all the dependencies and can compile easily with make.

Included is a Makefile which will compile the .c files into the client and server executables.
To compile simply run:

```
$ make
```

To run the server, run:

```
$ ./server
```

To run the client, run:

```
$ ./client <host>
```

Where <host> is the hostname / IP address of the host running the server.

For example,

```
$ ./client localhost
```

How to test the implementation

Run a central server, and 2 clients.

The first thing you should do with the client is identify the clients with the server. This is done with the /user <user> <password> command.

In client 1 run:

```
/user Bob password
```

Client 1 is now authenticated with the server as “Bob”.

In client 2 run:

```
/user Alice password2
```

Client 2 is now authenticated with the server as “Alice”.

Notice the server logs new connections and prints request data.
All network data transfer within our application uses a custom Application layer protocol that we designed that we named “D58P”. The server prints out the D58P requests made by the client.
To see more information about D58P, see the documentation section on D58P, or “D58P_specification.md” in the submission code.

In client 1 run:

```
/msg Alice
```

Bob is now chatting with Alice. All further messages will be directed to Alice.

In client 1 run:

```
Hey Alice, what's up?
```

Alice has received the message from Bob in her client.

Notice the server logs the D58P /Message request that Bob sent, the message reads “Hey Alice, what’s up?”. Also notice how the server receives a new D58P /Get Message request from Alice. Our implementation uses a long polling approach to sending messages from client to client, so after a message is received it asks the server for another message immediately.

In client 2 run:

```
/msg Bob
```

Alice is now chatting with Bob. All further messages will be directed to Bob.

In client 2 run:

```
Nothing much, how about you?
```

Bob has received the message from Alice in her client.

Open a new client.
In client 3 authenticate as John and say hi to Alice.
In client 3 run:

```
/user John password
/msg Alice
Hello Alice!
```

Alice has received the message from John in her client.

Now reply from Bob to Alice.
In client 1 run:

```
I'm doing well thanks
```

Alice has received the message from Bob in her client.
Alice is now chatting with 2 people. To send a message to John, begin chatting with him first.

In client 2 run:

```
/msg John
Hello John!
```

John receives Alice's message and Bob does not.

This concludes the main functionality of the application.

Note that our implementation of long polling allows for connection drops and connection switching. That is, when a client "logs out" by exiting the client, they will receive all the messages they should have received when they log back in. Users are also able to switch connections.

Continuing from the previous step,
Terminate client 2 (Alice) with Ctrl+C or /exit

Now Bob sends Alice 5 messages while she is offline,
In client 1 (Bob) run:

```
Message 1
Message 2
Message 3
Message 4
Message 5
```

Now Alice can log back in, by running the client again and using

```
/user Alice password2
```

Alice will immediately receive the messages Bob sent her while she was offline. Note that Alice could be running the client from a different host this time.

Implementation details and Documentation

server.c: Source code for the server side logic

```
int main()
```

Will bind to SERVER_PORT and will infinitely loop server_loop().

```
void server_loop()
```

Performs high level server logic. Accepting a connection, and dispatching a thread to handle the connection.

```
void *handle_connection(void *arg)
```

The routine a new thread runs to handle a connection. This function takes in a client socket file descriptor (passed as void *arg), receives data over the socket, and parses it as a D58P request. Then calls the corresponding request handler.

```
void user_handler(int client_socket, struct D58P *req, int len)
```

Handles D58P /User requests. Takes in the client socket file descriptor, and the request data in the form of a struct D58P *req. Extracts a username and password from the request data. If the username does not exist in the server, then registers a new user. If the user exists, then tries to log in. The username/password is compared with a global structure of authenticated users called “struct user *user_list”. Then responds to the client with an appropriate D58P response.

```
void message_handler(int client_socket, struct D58P *req, int len)
```

Handles D58P /Message requests. Takes in the client socket file descriptor, and the request data in the form of a struct D58P *req. This function will ensure the user is authenticated and the target user exists. Then adds the message to a global structure of in-flight messages called “struct message *message_list”. Then responds to the client with an appropriate D58P response.

```
void get_message_handler(int client_socket, struct D58P *req, int len)
```

Handles D58P /Get Message requests. Takes in the client socket file descriptor, and the request data in the form of a struct D58P *req.

Since our implementation for transferring messages from user to user uses long polling, this function has most of the logic for sending messages to users.

The user info is extracted from the request data and the function will ensure that the user is authenticated.

Then the global structure of in-flight messages called “struct message *message_list” is searched for a message destined for the requesting user. This list is iterated through infinitely until such a message is found. When found the function sends a D58P response with the message.

There is also logic to handle connection switches in this function. It utilizes another global structure of threads handling D58P /Get Message requests called “struct server_thread *thread_list”. If the get_message_handler(...) function determines that there is already a thread handling a D58P /Get Message for the calling user, it will signal to that thread to terminate and use the current thread to handle the request instead. This ensures that the D58P /Get Message request is handled by the thread with the most recent connection.

```
void get_key_handler(int client_socket, struct D58P *req, int len)
```

This method handles a user’s request to get another user’s public key. It retrieves the requested key and creates a D58P response.

Other functions in server.c:

```
int is_authenticated(char *username, char *password)
struct user* find_user(char *username)
void register_user(char *username, char *password)
void insert_message(struct message *msg)
void add_message(struct user *from, struct user *to, char *message)
struct server_thread *find_thread(struct user *user)
void remove_thread(struct server_thread *t)
struct server_thread* add_thread(int client_socket, struct user *user)
```

Are all helper functions to handle access to global structures such as
struct user *user_list; // list of authenticated users.
struct message *message_list; // list of messages in flight
struct server_thread *thread_list; // list of threads handling /Get Message requests

client.c: Source code for the client side logic

```
int main(int argc, char * argv[])
```

Resolves the passed hostname to an IP address, sets up encryption keys, starts a new thread to get messages, and loops client_loop() forever.

```
void client_loop()
```

Handles high level client logic. Will receive input from the client, parse the input, determine what action should be taken, forwards to appropriate function.

```
void user_handler(char buf[MAX_LINE], int len)
```

Handles user input of form “/user <user> <pass>”. Creates a D58P /User request from inputted user data, and sends the request to the server to authenticate. Sets a global variable called “struct D58P_auth auth” with the authentication information (used for future requests).

```
void msg_handler(char buf[MAX_LINE], int len)
```

Handles user input of form “/msg <recipient>”. Will set the global variable char target_user[MAX_LINE] to the recipient inputted by the user. Future messages will be sent with the recipient as the target_user.

```
void send_message_handler(char buf[MAX_LINE], int len)
```

Handles messages inputted by the user. send_message_handler(...) is only called when the client is authenticated with “/user <user> <pass>” and chatting with someone using “/msg <recipient>”. The function will take user input and create a D58P /Message request to the current recipient. Then the request is sent to the server. The server will then forward the message to the correct recipient.

```
void* get_messages(void *aux)
```

The routine for the thread launched by the main() function. This thread will keep sending D58P /Get Message requests to the server. The D58P /Get Message request is held open by the server until a message is received. So once the client receives a response and prints to the client, it immediately sends another request.

```
void exit_handler(char buf[MAX_LINE], int len)
```

Handles user input of form “/exit”. Terminates the program with EXIT_SUCCESS.

netutils.c: Handles socket connections and D58P requests and responses

```
int create_connection(struct sockaddr_in *sin)
```

Creates connection to socket address. Returns socket file descriptor for connection. Terminates the program if it could not connect.

```
int accept_connection(int sfd, struct sockaddr_in *sin)
```

Accepts a connection on specified socket file descriptor (sfd). Sets “sin” to the address of the connecting peer. Returns new socket file descriptor for the client connection. Terminates program if could not accept.

```
void create_message_request(struct D58P *req, struct D58P_auth *auth,
struct D58P_message_data *data)
```

Builds a D58P /Message request from provided auth information and message information.

```
void create_user_request(struct D58P *req, struct D58P_auth *auth)
```

Builds a D58P /User request from provided auth information.

```
void create_get_messages_request(struct D58P *req, struct D58P_auth *auth)
```

Builds a D58P /Get Message request from provided auth information.

```
void create_response(struct D58P *res, char *type, enum D58P_ResponseCode
code)
```

Creates generic D58P response of specified type with specified response code.

```
void create_get_message_response(struct D58P *res, enum D58P_ResponseCode
code, char *from, char buf[MAX_LINE])
```

Creates D58P /Get Message response given response code and the message to be returned.

```
int send_D58P_request(struct sockaddr_in *sin, struct D58P *req, struct
D58P *res)
```

Sends a D58P request given its struct to the provided server. Parses the response as a D58P struct and stores in *res. Returns length of response.

```
void send_D58P_response(int sfd, struct D58P *res)
```

Sends a D58P response given its struct to the client connected to the provided socket file descriptor.

```
void send_D58P_response_ack(int sfd)
```

Helper to send a D58P response acknowledgement.

```
int verify_acknowledgement(int sfd);
```

Verifies an acknowledgement on the socket file descriptor. Returns 1 if acknowledgement received, 0 otherwise.

```
void create_get_key_response(struct D58P *res, enum D58P_ResponseCode code, char *user, char* target_user, char *e, char *n)
```

Creates a D58P response for \Get Key in res, given the code and all of the lines.

```
void create_get_key_request(struct D58P *req, struct D58P_auth *auth, char *target_user)
```

Creates a D58P request for /Get Key in req, given the sending user's info in auth and the name of the target user.

Other functions in netutils.c:

```
void parse_D58P_buf(struct D58P *data, char buf[MAX_REQUEST]);
void dump_D58P(struct D58P *data);
```

Utils for parsing a D58P struct from a buffer of characters, and printing D58P structs to stdout.

D58P Specification: The application layer protocol used by client and server

Requests

Any data sent over the TCP socket of the below form will be interpreted as a D58P request.

```
D58P /RequestName
<data1>
<data2>
```

```
...
```

First the “D58P”, then followed by forward-slash with the type of request.

Then on each following line separated by newline characters are data required for the particular D58P request.

Examples:

```
D58P /User
<user>
<pass>
```

Represents a D58P /User request. When the server receives this it will try to authenticate the user using specified username and password.

```
D58P /Message
<user>
<password>
<target user>
<message>
```

Represents a D58P /Message request. When the server receives this, it will try to send “message” from “user” to “target user”. The server will use the <user>/<password> pair to authenticate the request.

```
D58P /Get Message
<user>
<password>
```

Represents a D58P /Get Message request. When the server receives this, it will search for a message destined for “user”. The server will use the <user>/<password> pair to authenticate the request.

Responses

Below represents a D58P response.

```
D58P \ResponseName
<code>
<data1>
<data2>
...
```

First the /D58P, then a back-slash followed by the response name. The response name is typically the same as the request name, the only difference is the \ instead of /.

On the second line is the response code. D58P response codes are analogous to HTTP response codes. So `code=200` indicates OK, `code=400` indicates BAD REQUEST.

Then on each following line separated by newline characters are data required for the particular D58P response.
Examples:

```
D58P \User
<code>
```

Represents a D58P \User response. `code` refers to the D58P code for the response. `code=200` indicates a successful login. `code=201` indicates successful register. `code=401` indicates unauthorized. This is sent as a response to D58P /User requests.

```
/D58P \Message
<code>
```

Represents a D58P \Message response. `code` refers to the D58P code for the response. `code=200` indicates the message will be sent to the recipient. This is sent as a response to D58P /Message requests.

```
D58P \Get Message
<code>
<from>
<message>
```

Represents a D58P \Get Message response. `code` refers to the D58P code for the response. `code=200` indicates a message was returned. `from` is the sender of the

message. `message` is the message that was sent. This is sent as a response to D58P /Get Message requests.

Acknowledgements

After a client sends a request and receives a response, the server expects to receive an acknowledgement from the client. This is so that the server can be sure that the client received the response.

The D58P Acknowledgment looks like:

```
D58P /Acknowledge
```

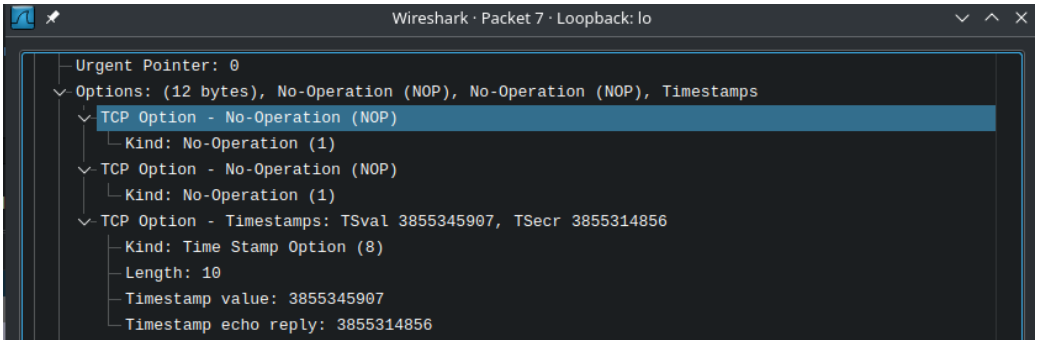
It is a single line with “D58P” followed by a forward slash then the word “Acknowledge”. The acknowledgement is sent to the server immediately after the client receives the response from the server. The acknowledgement is sent on the same TCP connection that the initial request was sent on.

Analysis and Discussion on the project’s, results and implementation

When we started working on the project, we hoped to learn more about the layers of the network stack that we had outlined in the “Rationale of project” and “Goals” section. We got what we expected and more.

When working on this project became more familiar with the types of questions and problems an application programmer faces when creating applications that perform internetwork communication. Especially in the client-server model. We initially decided on the client-server model, but thinking back, many of the decisions we had to make would have been different if we had taken a peer-to-peer model instead.

Our project uses the long polling approach to send messages from client to client (through the server). We studied other approaches to sending data from client to client in the client-server model, but decided that this approach was best fit for our application.



Wireshark screenshot of message from Alice to John (unencrypted)

We managed to accomplish all of the goals we set in the “Goals” section of this report and we are happy with the results from our project. While implementing the functionality of the project we encountered a lot of problems that we had not identified initially when choosing our implementation path.

During implementation of the project, we changed our approach to constantly thinking of the simplest but most robust way to solve problems. We identified choices made early during development could come to make future problems much more difficult. We found ourselves refactoring netutils.c often, and rewriting the functions used to create D58P requests and responses.

For transferring data across the network, initially we were thinking of implementing an HTTP library to send the requests we need for our application. While this was definitely a possible course of action, we decided to design our own application layer protocol and tailor it to our needs. We created “D58P” which is a very simple request-based protocol.

The format of D58P requests and responses hadn’t changed since we initially created it, but as we were working on the project, we modified some workflows to allow us to do things that we needed for our use cases. For example, when handling a D58P /Get Message request, if a client connection was closed and reopened later, we found that a singular message to the client would be lost. To ensure that this message was not lost, and would be delivered to the client, we modified the D58P workflow to add Acknowledgements after the server response. This allowed the server to verify that the client received the message and resulted in no lost messages, even after a client closes and reopens the application.

This gave us more insight into the design of protocols. The problem of solving these lost messages would have been really difficult with a different approach. We understand that as an application layer protocol designer, once your protocol is published, it should not be revised (or at least very seldom revised). In solving the problem we modified our protocol, and came to realize one of the major problems and questions an application layer protocol designer faces. That is, designing protocols with all use cases in mind, and with the future in mind.

<u>Note on technical debt and global structures:</u>

Our implementation of the server uses global structures such as linked lists to store lists of currently authenticated users, list of in-flight messages, and list of threads handling D58P /Get Message requests.

Currently our application is not scalable, since it depends on these global data structures, but we felt that for the purpose of the project these implementations were sufficient to achieve the goals we had set in the “Goals” section of the report.

Integrating the server with an external central database would have been a better option. As it would have allowed us to scale the chat application by adding more instances of servers connected to the database.

Concluding Remarks

concluding remarks, lessons learned.

During implementation of our instant-messaging chat application, we learned about the higher levels of the network stack. We were exposed to the application, presentation, session, and transport layers of the network stack, and gained hands-on experience with these layers.

We also learned about some of the technical decisions application developers make when creating application layer protocols, and understand some of the problems general protocol designers face when designing protocols.