SDD HSC Major Work - Messenger

Max Hamilton
35419837

Contents

1 Terminology

Throughout this report, the client who requests the application be built will be referred to as the customer. "The client" will refer to the application which "users of the client" use to send and receive messages, and "the server" will refer to the application which centrally stores messages, administrated by "users of the client", as in the client-server model.

2 Design Brief

2.1 Client and target audience

The target audience of this application is anyone who needs to securely communicate, without trusting the server they are communicating on. This could include governments, corporations cognizant of the possibility of hacking, or anyone renting servers or using a cloud provider. The target audience is one who does not mind giving up a small amount of convenience for security.

2.2 Definition of the problem

The customer requires a secure messaging application. They don't trust existing infrastructure, and existing messaging applications. They need all the code to be open source, and to be able to run their own servers. However, they don't fully trust their own sysadmins with the data, are conscious about the possibility they could be hacked, or might outsource the server hosting to a cloud provider, so need the server to know as little about the messages as possible. Ideally, an attacker could hack their system, control the server, and the worst thing they could do would be shut it down, with 0 possibility for sabotage and 0 possibility of accessing any data or metadata. The messaging service must be similar to email, with the ability to send messages to multiple people, to respond to specific messages, and to CC and BCC different accounts to messages. Users should only be able to read messages intended for them, and there should be a way for the recipient to authenticate a message came from a specific user, without the server knowing the author. The customer requires 2 applications, one client, and one server. The client application must have a GUI interface, but the server may have a CLI interface, as it will be run by system administrators and not ordinary users.

2.3 Description of the solution

2.3.1 Initialisation

The user of the server requests the server initialise a new user. The server will generate a random one-time initialisation code, and display it on-screen. The user of the client is informed of the IP address or domain name of the server by the user of the server, and inputs it to the client. A secure connection is established between the client and server via TLS over TCP/IP, and the client stores the server's IP address or domain name if dynamic DNS is used. The user will receive the initialisation code generated by the server for this initialisation, communicated to them via the user of the server, through existing contact channels. The client will then send that code back over the connection, to authenticate themselves. The client will generate a private and public key-pair, and store both. The user of the client will then input a username. The username and the public key are all sent to the server, which stores them together as an account. It is heavily recommended that the initialisation be done on a device used only for initialisations, or via a proxy, VPN, or over Tor, as otherwise the server could connect the username and public key to an IP address, which can be used to find the author or recipient of messages if they are not uploaded or downloaded over a proxy, VPN, or Tor.

Table 1: Security considerations

Possible threat	**Prevention**
Fake clients sending initialisation requests.	The server generates the initialisation code on request from the user of the server, so an attempt to initialise without prior planning will be ignored.
Code being sniffed on network and used to send a false request.	The code is sent over a secure TLS connection, and after being used once, is no longer valid.
Username and password being sniffed or altered.	The username and password are sent over a secure TLS connection.
Initialisation code being sniffed through the initial communication.	No prevention. The system administrator must be careful to ensure the correct person receives the code, and no one else does. The code must be conveyed through another secure message channel, digital or physical.
Server building profiles of usernames and public keys to IP addresses.	Initialise via an IP not used for further message sending or receiving, such as on company device used for every initialisation, or via a proxy, VPN, or over Tor.

2.3.2 Access control and authentication

Users of the client do not log in in the traditional sense, the sole factor controlling their account is their private key. That key can be put onto a USB stick and deleted from the original computer for physical control, or left as a file on the computer. To send a message to the server, a client will need to sign a string of random bits provided by the server with their private key. The possession of a private key imparts its user with 2 privileges: To decrypt the sending/receiving code broadcast by the server, allowing it to send messages to the server which will not be discarded, and pull messages from the server.

Table 2: Security considerations

Possible threat	**Prevention**
Attacker sniffs the signed response and uses it in a replay attack	It is very unlikely that the same series of random bits will be sent twice to enable a replay attack.
User leaks their private key	No prevention. If a private key is leaked, hostile actors can impersonate the owner, read the owner's messages, read the sending codes, and read the list of usernames and public keys. Users must not leak their private keys.

2.3.3 Sending Messages

To send a message, the user of the client composes it, attaches any files, which are just concatenated onto the end of the message, inputs the usernames of any recipients, and requests the client to send. If the client does not want the server to know the length of the message, all messages padded to a previously arranged maximum size. The client then adds a header composing of the username of the user of the client, and a digital signature using the client's stored private key. The client then encrypts the message for multiple recipients, including itself, and every intended recipient. If the client does not want the server to know the number of recipients, they can generate junk recipients and randomly insert them into the list. The client then connects to the server, and sends this header plus encrypted message and original header to the server, along with the sending code it received earlier. If uploading messages is done directly, servers can build a profile of which IP addresses sent which message. This can be prevented by sending messages via a proxy,

VPN, or over Tor. The server verifies the code, and stores the message, as well as the time at which it was received. If the server receives a message with an incorrect code, it discards it, and logs the IP for potential blocking. The server refreshes the code being broadcast after a certain period of time, or after a certain number of messages have been sent.

Table 3: Security considerations

Possible threat	**Prevention**
Messages being forged or altered.	The messages are digitally signed by the author.
Hostile server	The server could potentially know: The timing of messages, the IP address which authored certain messages, the length of the message, and the number of recipients. To prevent the server knowing the IP address of the author, which could be used to build a profile, clients can upload via a proxy, VPN, or over Tor. By padding the message to an agreed upon maximum length, the length can be hidden, although the server would still know an upper bound on the message length. By adding junk keys, the number of recipients can be hidden, although the server would still know an upper bound on the number of recipients. The timing of messages can be concealed from the server by the sending of junk messages at random intervals, although that uses a lot of server storage space and the server still knows an upper bound on the number of messages sent.
Server being overwhelmed by fake messages from a hostile actor	Clients must log in with a username and password to establish a connection with the server and send messages.
Messages being sniffed	Messages are sent over a TLS connection, and additionally encrypted. Even if an attacker could decrypt the TLS encryption, they would know as much as the server, which is very little.

2.3.4 Receiving messages

Each client will store the most recent message that it has received, and periodically poll via TLS over TCP/IP the server for any messages sent past that time, as well as any new users. If there are, the server will send the earliest message past the received time, and a bit indicating whether there are more messages available. If there are, the client will continue polling until there are no more messages available. When a client polls for a message, it sends the timestamp it wants the messages past. The server can then send a small snippet which the client can use to determine whether the message is intended for them. If it isn't, they move on, but if it is, they request the full message. The code is again verified, the encrypted message is sent, and the client then locally decrypts it to display to the user. The client will keep a local log of past message titles, senders, timestamps, and a list of unique words, stored encrypted on the disk, to enable the user of the client to search and browse past messages. The full message is downloaded on demand by sending a request with the timestamp the message was sent to the server. If the client does not use a proxy, the server can build a profile of which IP addresses download which messages, even without names to match. To prevent this, clients can use a proxy or VPN, or connect over Tor. In the event the volume of messages becomes too much for the client to check every one, the clients can be subdivided into groups, with the server storing the messages of each group seperately, and senders including the group of the recipient in message headers. Then, only messages in the group of the recipient are presented by the server. This seperation into groups gives the server a small amount of information about recipients, in exchange for much faster performance

Table 4: Security considerations

Possible threat	Prevention
Hostile actors polling the server.	Requests for messages must authenticate, and the sent messages are completely encrypted.
Hostile actors sniffing the transmitted messages.	Messages are sent over TLS, with another layer of encryption.
Hostile clients attempting to read others messages.	Messages are encrypted with the private keys of the recipients, and can only be decrypted with their private keys. Hostile clients could download encrypted messages, but not decrypt them. They would know only what the server knows, which is very little.
Server sending fraudulent messages.	All messages are signed with the private key of the author, so it is impossible to fake or tamper with them.
Server not sending messages.	Users will be able to see if their message is not uploaded, as users must download their own messages to read them, so if a server is not sending certain messages, it can be quickly detected.
Server detecting who requests messages and building a profile.	While the server can see who requests messages, all they know is an IP address, so profile building can be circumvented with the use of a proxy, VPN, or connecting via Tor.

2.3.5 Storage

The server will store, in an database:

- Encrypted messages.
- Timestamps of encrypted messages.
- List of usernames, and the public keys they are associated with.

3 Planning the software solution

3.1 Algorithms

Algorithm 1: Loops forever, handling incoming connections concurrently

```
1  BEGIN receive_connections
2      Let PORT = 2001
3      Let listener = BindTcpListener(PORT)
4      Let fileAccessStatus = Mutex::Nothing
5      A mutex allows for a value to be safely shared and mutated between threads
6      Let userKeyMap = Mutex::BiHashmap::empty
7      A BiHashmap is a 2-way hashmap, where either type can be used as an index
8      WHILE true
9          IF listener.newConnectionAvailable THEN
10             spawn new thread with function handle_connection(listener.newConnection,
                 fileAccessStatus, userKeyMap)
11         ENDIF
12     ENDWHILE
13 END receive_connections
```

Algorithm 2: Handles a single connection, authenticating then dispatching it off the the dedicated function to handle it's request

```
Input : A connection to a client, the file access lock, and the map between users and keys
BEGIN handle_connection
    Let randBytes = random_bytes(8)
    clientConnection.send(randBytes)
    clientConnection.read(publicKey, signature)
    IF publicKey not in userKeyMap THEN
        clientConnection.send(DENIED)
        Return
    ENDIF
    IF signature invalid THEN
        clientConnection.send(DENIED)
        Return
    ENDIF
    clientConnection.send(ACCEPTED)
    clientConnection.read(intent)
    CASEWHERE intent
        CASE WriteData
            WHILE true
                IF fileAccess = NotInUse THEN
                    fileAccess = InUseWriting
                    Break
                ENDIF
                Sleep(1 millisecond)
            ENDWHILE
            receive_and_store(clientConnection)
            fileAccess = NotInUse
        CASE DetermineRecipiency
            WHILE true
                IF fileAccess = NotInUse OR fileAccess = InUseReading THEN
                    fileAccess = InUseReading
                    Break
                ENDIF
                Sleep(1 millisecond)
            ENDWHILE
            handle_recipiency_query(clientConnection)
            fileAccess = NotInUse
        CASE FetchData
            WHILE true
                IF fileAccess = NotInUse OR fileAccess = InUseReading THEN
                    fileAccess = InUseReading
                    Break
                ENDIF
                Sleep(1 millisecond)
            ENDWHILE
            send_message_data(clientConnection)
            fileAccess = NotInUse
        CASE Otherwise:
            Return Error
    ENDCASE
END handle_connection
```

3.2 Prototype

For the prototype, I submit a working set of libraries for the client and server, as well as an extensive test suite, to confirm it is working and bug-free. The client GUI application and the server CLI application will rely on these libraries for the sending and receiving of messages.

3.3 Data Dictionary

This data dictionary is completed for the function "handle_recipiency_query", which handles a request from a client as to whether a particular message is for them.

Table 5: Data Dictionary

Identifier	Data Type	Description	Scope	Example
read_count	usize	Holds the amount of bytes received from client	Local	let read_count = client_stream.read(&mut buffer)
write_count	usize	Holds the amount of bytes sent to client	Local	let write_count = client_stream.write(buffer)
timestamp_buf	[u8; 8]	Holds the timestamp of the message the client is enquiring about, in byte form, after being sent over a network	Local	let read_count = client_stream.read(&mut timestamp_buf)
timestamp	u64	Holds the timestamp of the message the client is enquiring about	Local	let timestamp = u64::from_be_bytes (timestamp_buf)
existence_confirmation_buffer	[u8; 1]	Sent to client to let them know if the message they are asking about exists	Local	client_stream.write(existence_confirma tion_buffer)
ENCRYPTED_KEY _LEN	u64	The length in bytes of the symmetric key, encrypted asymmetrically. A constant	Global	if read_count < ENCRYPTED_KEY_LEN {bail!("Database file at path {database_path} is malformed. Could not read keys. Tried to read {ENCRYPTED_KEY_LEN} bytes but received {read_count}")}

3.4 Modelling tools

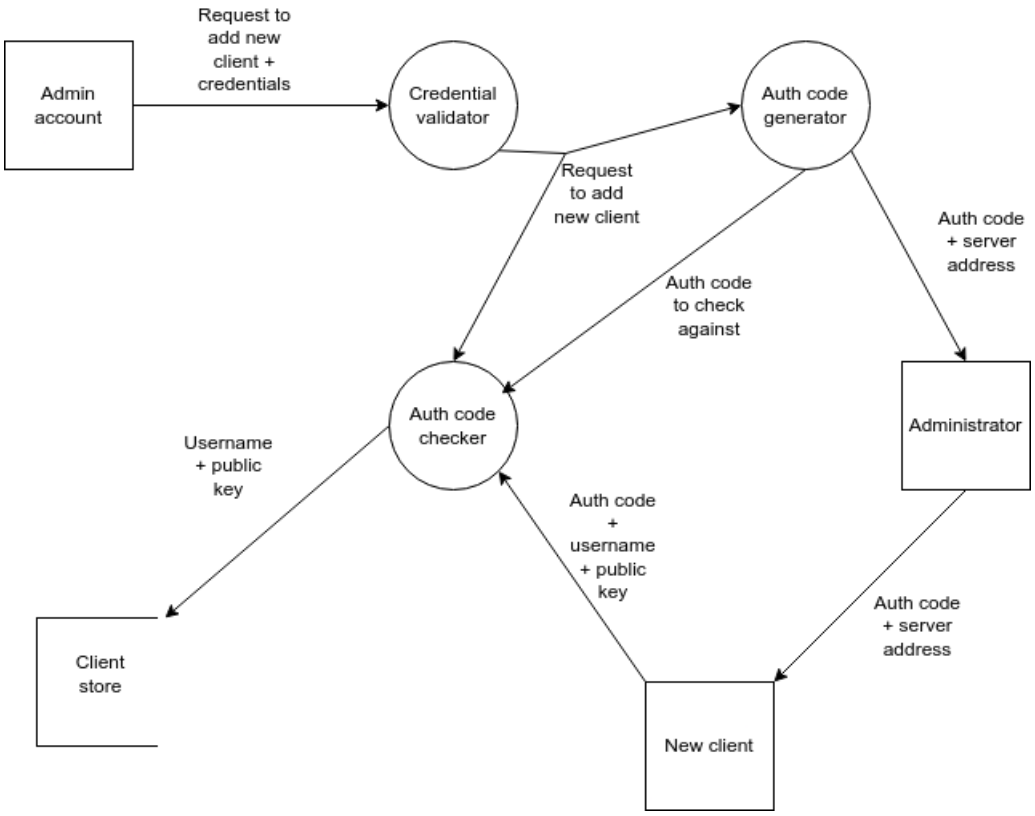

Figure 1: Data flow diagram for the process of adding a new client

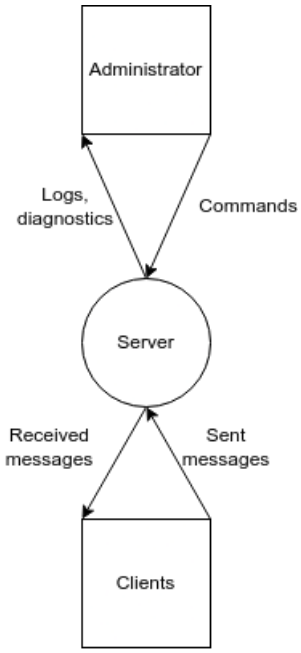

Figure 2: Context diagram for the server handling messages

3.5 Programming language

My selected programming language is Rust. I have chosen rust for its ability to send and manipulate individual bits easily (essential for sending and receiving messages over TCP, encrypting messages, and writing messages to disk), its mature networking and encryption libraries, essential for the operation of the project, which make it simple to both encrypt and send messages, its memory safe nature, which utilises a borrowing and ownership system to eliminate race conditions, use-after free errors, double-free errors, unexpected mutation, and many other memory-related problems, and its ergonomic and useful tools, such as trait-based generics, functional programming tools such as iterators, and well-documented standard library, which aid in ease and speed of programming, and its speed, which ensures that the final product is fast and responsive for all users.

3.6 Logbook

For the full logbook, see section 6.2, the end of this report.

4 Building the software solution

4.1 Source code

The source code is submitted online. It should be noted that in the submitted code, the client application is not yet developed, so future references to it, and its planned command line interface, refer to the plan, and how it will behave when developed.

4.2 Feedback

Table 6: Feedback form

Question	Answer 1	Answer 2
How easy do you find the administration of the server?	Easy. The command line commands are very simple, and the provided online help file alleviates any confusion.	The server is easy to initialise and start, but the process of adding or removing users is confusing.
How do you find the process of sending messages?	It is annoying to have to type the whole message in a terminal, I would prefer a GUI or the ability to attach pre-written text files.	Alright, although I often find myself typing a message in another tab, and then copy-pasting it into the terminal.

In response to the feedback, I added an explicit mention of the ability of the linux command line to pipe the contents of a file into the standard input to the online help file accessed via the command line, so users do not need to type out full messages in the terminal.

5 Documenting to meet installation of software solution

5.1 Installation guide

5.1.1 Server installation

To install the server, download the distributed binary, and run "server init pub_key_path username", as superuser, to initialise the server, with the first admin account. Then, run "server start", also as superuser, to start the server.

5.1.2 Client installation

Once the client application folder is downloaded, navigate to it and run the installer, as superuser. Once done, the client can be run by running the command: "client start", or the online help file can be viewed by running "client help".

5.2 User manual

5.2.1 Admin

As an administrator, you will need to run the server, and add or remove users. In order to run the server, once it is installed, simply run the command "server start", as superuser. Then, in order to add or remove users, open the client application, navigate to the admin panel, then click on "Add or Remove User". If you intend to add a user, the application will give you a code, which you must give to the user to add, so they can register themselves. If you intend to remove a user, simply enter that user's username.

5.2.2 User

In order to start the client application, which regularly polls for new messages, simply run "client start", in a command line. In order to send a message, run the command "client send recipient1 recipient2 recipient3", with an arbitrarily long list of recipients, before typing your message into the standard input. You can also pipe the contents of a file with a pre-written message into standard input, so as to not have to type the message into the terminal. In order to view your messages, run the command "client view", in order to browse the most recent messages.

5.3 Testing methods

Included in the submitted source code is a full testing module for the operation of sending messages between server and client, covering many concurrent clients all interfacing with 1 server, and testing many aspects, such as file content integrity, false negatives and positives on message existence, and load testing with many clients at once, some of whom unexpectedly disconnect. These tests can be run with the command "cargo test".

6 Evaluate and suggest modifications

6.1 Evaluation

The main issue with my project is that the final product is only a prototype, with the underlying logic, but not a proper user interface for the client. In order to improve this in future, I would reduce the large scope of the project, or explicitly allocate working time, so as to fully complete the project within the required time frame. In aspects other than time management and scope, there are no outstanding issues with my project or implementation. The vast majority of the project is completed, but the lack of a user interface leads to difficulties demonstrating. Due to this, the demonstration of the program will be in the testing, which is designed to simulate the proccess of clients sending, and receiving messages.

6.2 Logbook

Table 7: Log Book

Date	Summary
22/10/2022	Came up with idea for the project. A messenger app. I put a bit of thought into how it would work, and the client. The client would be a company that needs a secure messaging service similar to email. I would provide a client software, and a server software. The client software would be a GUI, and the server may be a GUI or a CLI. The server software will not be able to read messages, and will have the minimum possible information about the messages. This allows for secure outsourcing of the server hosting if the company cannot run their own servers. To add a new user, the server would generate a code which the system administrator could communicate to the new user, which the client uses to send a username and password to the server. On initialisation, the client would generate a public and private key. The client keeps the private key, and sends the public key to the server. To send a message, the client requests the recipient's public key, and encrypts the message with it. The encrypted message is then sent to the server, where it is stored indefinitely. I think I can make it anonymous who exactly is sending and receiving messages. The message on the server is then available for download by all clients, but only the recipient can use their private key to decrypt it. I can have anonymous uploads by encrypting the name of the sender with the rest of the message, and anonymous downloads by having a constant magic number at the start of each message. Every client, on seeing a new available message, would try to decrypt it with their private key, and if the constant magic number didn't match, they could discard it, meaning only the recipient would be able to access it. Additionally, messages would be signed with the sender's public key, then the signature encrypted with the rest. That way, once the message was decrypted, the identity of the sender could be verified. A bad actor could download all encrypted messages, but the would not know the intended recipient, sender, or content. A lot of aspects could be controlled by the server software, such as file size caps, message size limits, and length of saved history.
22/10/2022	Formulated my thoughts by writing the description of the solution, in a lot of detail. Might have to cut it down later.

Table 7: Log Book (Continued)

22/10/2022	I have finished the first draft of the design brief, and have a complete plan for the high-level design of my program.
23/10/2022	I have done some research and begun to implement the core functionality of the client. Experiencing some difficulties deciding to encrypt files on disk, or to pull them into memory.
29/10/2022	I have located the cryptography libraries to use, as well as encrypting files bit by bit so they do not take up huge amounts of memory.
16/11/2022	I have written the code for clients sending messages, and the server receiving and storing them.
23/11/2022	I have written the code for clients asking about the recipiency of messages, and the server answering.
30/11/2022	I have written the code for clients downloading message contents.
05/2/2022	I have started writing test code for the general sending and receiving of messages.
10/2/2022	I have finished writing the test code, and am debugging numerous issues relating to encryption, error handling, and bugs in test code.
16/2/2022	I have gotten single-threaded tests working, and am now writing multi-threaded tests.
20/1/2022	I have added mutex-style guards on the files storing messages, which should eliminate most multi-threaded bugs.
25/11/2022	I have finished debugging the multi-threaded code, and the full test suite is passing without bugs.
26/11/2022	I have started work on the user-facing CLI application for the server.
16/3/2023	I have begun a redesign of the user model, including methods of adding and removing users, and keeping the user list up to date
20/3/2022	Still working on the user model
28/3/2022	Finalising user model, working on code for updating users from user file
7/4/2022	Finalised updating users, working on code for adding and removing users
14/4/2022	Still working on adding and removing users
29/4/2022	Finished adding and removing users, finalising server interface
3/4/2022	Finished server interface
6/4/2022	Finalising testing and final product