Pavel Berkovich

N0NaMe_

Peer-to-peer conversation security using KleeQ

Computer Science Tripos, Part II

St John's College

13th May, 2016

Proforma

Name:	**Pavel Berkovich**
Title:	**NoNaMe—Peer-to-peer conversation security using KleeQ**
Examination:	**Computer Science Tripos, Part II, June 2016**
Approx. Word Count:	**11,992**
Project Originator:	**Pavel Berkovich**
Project Supervisor:	**Dr. Richard Clayton**
Special Difficulties:	**None**

Original Aims of the Project

The aim of this project has been to produce an implementation of KleeQ, a peer-to-peer conversation security protocol designed for devices with limited connectivity, and evaluate its performance and practicality in the broader context of Internet messaging. The intention was to preserve the security guarantees of the original design as well as understand its limitations that could potentially be used as attack vectors. It was also expected that a useable prototype of a messaging application would be produced, to evaluate the usability implications of the design as well as demonstrate the work of the implementation.

Summary of the Work Completed

Each of the project's aims has been successfully achieved. Despite it taking more time than expected, the protocol has been implemented in full, according to the specification, retaining all of the security properties of the design. The implementation has demonstrated some impressive performance characteristics, highlighting some of the practical benefits that the use of the protocol can offer. A prototype of a messaging system, comprising a client application as well as some external components, has been implemented and tested.

Declaration of Originality

I, Pavel Berkovich of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Chapter 1

Introduction

1.1 Overview of secure messaging

The era of global communication has presented humanity with many opportunities, but has also made our private lives more vulnerable to intrusion. Given the broad range of cyber-threats that our society is facing today, there is now significant demand for secure communication systems. The recent disclosures about widespread state surveillance programmes demonstrated the massive scale of the resources that a potential adversary might possess, and led to fundamental change in public perception of what security means. As a result, we are now witnessing an unprecedented level of effort to develop messaging systems emphasising security and privacy. Some of these systems have enjoyed considerable popularity whilst others remained largely unknown to non-experts, but each of them has been found to have some security flaws[1] or usability problems.

Discussing secure messaging in more detail and comparing different solutions requires a clearly defined threat model and a systematic evaluation framework. One possible approach was proposed by Unger *et al.* [1] in a recent survey of the existing secure messaging systems which identified three key problems that any secure messenger must solve, namely:

Problem 1: Trust Establishment
How do we know that our peers are who they say they are? How do we make sure that they are *not being impersonated* by a malicious adversary?

Problem 2: Conversation Security
Once we are sure that we are talking to the right parties, how do we protect the security and privacy of the *conversation's content*? In other words, how do we encrypt the messages, what format do we transmit them in, and what security protocols do we perform?

Problem 3: Transport Privacy
Once we have secured the content of our messages, how do we actually

[1]`https://www.eff.org/secure-messaging-scorecard`

send them so as to *hide their metadata* (*e.g.* sender identity, recipient identity, conversation to which the message belongs *etc*)?

Security is a comprehensive concept, and therefore building a truly secure messaging application requires solving all of these problems. Due to time constraints this project will only focus on *conversation security* (Problem 2 above).

1.2 Aims of the project

This project aims to build on KleeQ [2]—a peer-to-peer conversation security protocol that protects the content of conversation in a fully connected group (clique) of trusted participants communicating in a broadcast manner.

The scheme relies on Diffie-Hellman key exchange for deriving a common key which is used for encryption and message authentication. The protocol uses the *patching algorithm* to exchange messages and converge on a global transcript. To ensure integrity of the conversation, the transcript is then verified in blocks of several messages via the process of *sealing*. KleeQ achieves forward and backward secrecy by asynchronously rotating keys after a block is sealed, meaning that a compromised key only gives the adversary the ability to read messages within one block, but not in the previous or subsequent ones. The protocol enables users to repudiate authorship of specific messages as well as participation in a given conversation, since it only uses the common keys derived as part of the conversation to authenticate messages and makes no use of public keys. The asynchronous nature of the protocol makes it resilient to unstable network conditions and makes it possible to write messages while offline and then re-converge on a single transcript when connection is re-established.

As compared to other conversation security solutions [1], KleeQ finds a good balance between security and usability characteristics which makes it a promising candidate for use in general-purpose messaging applications.

The authors of KleeQ (Reardon *et al.*, [2]) omitted some details in the protocol's description and only provided an unstable proof-of-concept implementation. This project aims to take their work further by re-implementing KleeQ from scratch preserving all of its security guarantees, and evaluating its performance and practicality for use in everyday messaging. More specifically, the objectives are as follows:

Implementation
Fill out the gaps in the protocol's description and construct a messaging application (codename `N0NaMe`) based on it, preserving the security features of the original design.

Evaluation of Performance
Use the newly created application to evaluate the performance of the protocol and determine what limits it would impose on potential deployment.

Evaluation of Usability
Understand the usability implications of the protocol and assess the feasibility of its use in different kinds of consumer messaging systems.

1.3 Summary

This chapter has shown where this project fits in the field of secure messaging, and described the objectives it expects to meet. The specifics of how these objectives are addressed are described in the following chapters.

Chapter 2

Preparation

This section familiarises the reader with the work that was done before any programming began. It presents the details of the KleeQ protocol (§ 2.1), as well as the formal requirements for what needs to be accomplished (§ 2.2). Then it describes some preliminary design work (§ 2.3) and the expected implementation strategy (§ 2.4). At the end of this section, there is a brief discussion of the technology and tools that were used in this project (§ 2.5).

2.1 Protocol description

2.1.1 Assumptions

KleeQ takes its name from the word "clique", which is a graph-theoretic term for a fully connected graph (Figure 2.1). The protocol assumes that conversation partici-

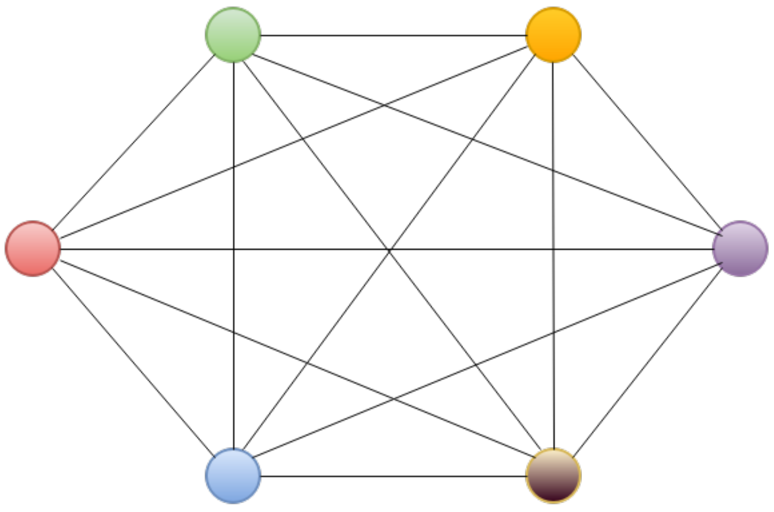


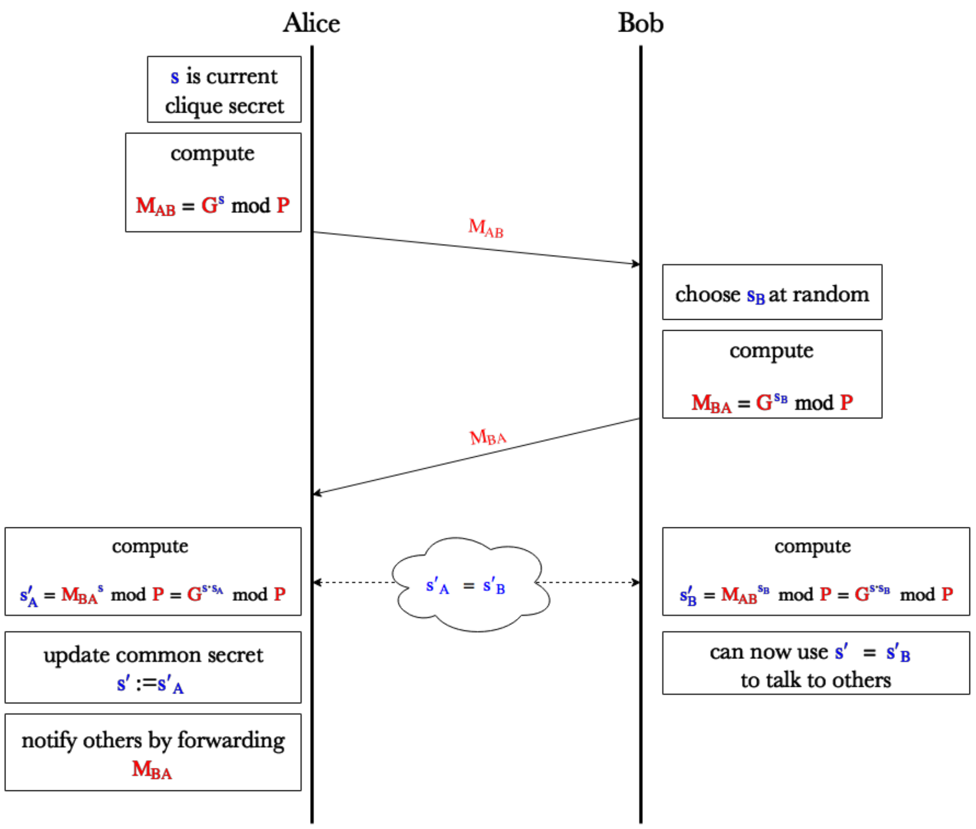
Figure 2.1: Clique of size $n = 6$

pants form a clique (*i.e.* no two of them are strangers) and that all clique members are trustworthy (*i.e.* that the problem of Trust Establishment from § 1.1 has already been solved via other means).

2.1.2 Clique formation

Every clique has a *common secret* that all participants share and which is regularly updated. When a clique is started, members are added one-by-one. One of the members first creates a singleton clique with a random secret, and then adds other participants. To add a user, one of the clique members performs a Diffie-Hellman key exchange [3] with them, using the current clique secret as the secret exponent, thereby negotiating a new common secret based on the previous one. When other clique members are notified of the new user, they also update their version of the secret.

Diffie-Hellman is a key exchange protocol that allows two parties to establish a common secret over an insecure channel. It uses two fixed publicly known parameters G and P, such that G is a generator of the cyclic group of order P. Let's say user Alice is a member of an existing clique with common secret s, and she wants to invite user Bob to the conversation.



Alice
Bob
s is current clique secret
compute $M_{AB} = G^{s}$ mod P
M_{AB}
choose s_B at random
compute $M_{BA} = G^{s_B}$ mod P
M_{BA}
compute $s'_A = M_{BA}{}^{s}$ mod P = $G^{s \cdot s_A}$ mod P
$s'_A = s'_B$
compute $s'_B = M_{AB}{}^{s_B}$ mod P = $G^{s \cdot s_B}$ mod P
update common secret $s' := s'_A$
can now use $s' = s'_B$ to talk to others
notify others by forwarding M_{BA}

Figure 2.2: Adding a new user to an existing clique using Diffie-Hellman key exchange. Public information is in red, secret—in blue.

She sends Bob the modular exponent $M_{AB} = G^s \bmod P$. Upon receiving this message, Bob generates a random number s_B and sends Alice $M_{BA} = G^{s_B} \bmod P$. Bob now computes $s'_B = M_{AB}^{s_B} \bmod P = G^{s \cdot s_B} \bmod P$. When Alice receives Bob's response, she computes $s'_A = M_{BA}^{s} \bmod P = G^{s \cdot s_B} \bmod P$ and updates the clique's common secret to be this new value. Clearly, $s'_A = s'_B = s'$, so Bob now also knows the secret and can use it to participate in the conversation.

An adversary who can observe this exchange and see both M_{AB} and M_{BA} has no computationally feasible way to determine the values of s and s_B and cannot compute the common secret $s' = G^{s \cdot s_B}$. The security of this protocol relies on the difficulty of the Discrete Logarithm Problem [4, sl. 77] in the chosen cyclic group, so it makes sense to choose parameters G and P to be very large [4, sl. 83]. It is also worth pointing out that this scheme is clearly vulnerable to a man-in-the-middle attack where an active adversary modifies the values of M_{AB} and M_{BA}, essentially impersonating each user to the other one. This can be solved by some form of authentication, but this problem belongs to the realm of Trust Establishment (§ 1.1) and is therefore outside the scope of KleeQ.

2.1.3 Exchanging messages

Clique members exchange messages through the procedure of *patching*. This process needs two pieces of information that each user maintains – the current *transcript* of the conversation and the *version vector*. The version vector is a vector of same size as the clique, where each entry corresponds to the number of messages authored by a particular conversation participant. The *patching algorithm* is then as follows:

Algorithm 1 The Patching Algorithm

Alice

1. Sends her version vector $v_A = (v_{A1}, v_{A2}, ..., v_{An})$ to Bob, where v_{Ai} is the number of messages authored by user i as seen by Alice

Bob

1. Calculates the difference between his own version vector and the one sent by Alice, to see which messages she is missing:

$$v_\Delta = v_B - v_A$$

2. Generates a patch to provide the missing messages:

```
$Patch \leftarrow \emptyset$
for $i$ from 1 to $n$ do
  if $v_{\Delta i} > 0$ then
    $Patch.add$(last $v_{\Delta i}$ messages authored by participant $i$)
```

3. Sends the *Patch* and version vector v_B to Alice.

Alice

1. Inserts the messages provided by Bob into transcript.
2. As above, computes difference between v_A and v_B, to see which messages Bob is missing.
3. As above, generates a *Patch* and send it to Bob.

Bob

1. Inserts the messages provided by Alice into transcript.

The above exchange happens regularly between every pair of clique members, and makes sure that their views of the transcript contain the same messages.

In order to ensure identical *ordering* of messages amongst clique members, each participant also maintains a *Lamport timestamp* which is initially set to 0. When two users exchange patches as shown above, they update their timestamps to the maximum of their Lamport times plus one. Every time a message is written, it is given the current Lamport time of the author which is then incremented by one. It is then possible to define a *total order* $<_L$ on messages as follows:

$$<_L (m_1, m_2) = \begin{cases} \texttt{TS}(m_1) < \texttt{TS}(m_2) & \texttt{TS}(m_1) \neq \texttt{TS}(m_2) \\ <_{lex} (\texttt{Author}(m_1), \texttt{Author}(m_2)) & \text{otherwise} \end{cases}$$

where TS is the Lamport timestamp of a message and $<_{lex}$ denotes lexicographic comparison of authors' names. The $<_L$ relation allows the clique members to arrange all messages in a linear sequence, sorting them by Lamport time and breaking ties by comparing the names of authors lexicographically.

It can be formally proven [2] that exchanging messages as specified by Algorithm 1 and ordering them using the $<_L$ relation results in all participants converging on the *same version of the transcript*.

2.1.4 Transcript verification

In an ideal situation, just using the scheme above will allow the clique to converge on the correct single transcript. However, in the more realistic setting where network errors can occur or a malicious adversary can try to inject fake content, measures must be taken to verify the *integrity* of the conversation. In other words, it is necessary to make sure that all users have the same view of the conversation. This is done using the process of *block sealing*.

In brief, the scheme involves the clique members independently dividing the transcript into blocks of messages in a deterministic fashion and *sealing* them by comparing the hashes of their content. For this to work correctly, it is necessary to choose blocks to be sealed such that all messages in them have been received and none are missing. This task is performed by the *block finding algorithm* (Algorithm 2 in the next page).

Intuitively, in Stage 1 the algorithm scans the unsealed part of the transcript (its *tail*) backwards, starting with most recent messages, until it has seen at least one message authored by each of the clique members. Because of how the patching algorithm works, it is certain that the remaining, less recent part of the transcript (also referred to as the *sealable set*) contains no missing messages. In Stage 2, blocks are extracted from the sealable set by scanning it in chronological order and taking the smallest

Algorithm 2 The Block Finding Algorithm

Input

- C: clique
- M: sequence of unsealed messages (*tail*)

Output

- B: prefix of M which is a sealable block, or $\emptyset$ if no such block can be found

Variables

- SeenSet: set of uniquely seen authors

Procedure

Stage 1: calculating the sealable set

Purpose: Finds part of the transcript with no missing messages.

```
1. SeenSet ← $\emptyset$

2. for $i = |M|$ down to 1 do
       $m \leftarrow M[i]$
       $SeenSet \leftarrow SeenSet \cup \{Author(m)\}$
       if $|SeenSet| = |C|$ then
           $S \leftarrow M[1..i]$
           break

3. if $|SeenSet| \neq |C|$ then
       return $\emptyset$
```

Stage 2: finding a block

Purpose: Computes the next block to be sealed.

```
4. SeenSet ← $\emptyset$

5. for $i$ from 1 to $|S|$ do
       $m \leftarrow S[i]$
       $SeenSet \leftarrow SeenSet \cup \{Author(m)\}$
       if $|SeenSet| = |C|$ then
           return $S[1..i]$

6. return $\emptyset$
```

message subsequence in which every user has authored at least one message. There is a formal proof [2], omitted here for brevity, that this algorithm allows the clique members to compute identical blocks independently.

Every block is given a sequential number. When a block is found, the concatenation of its sequential number and its messages are hashed to compute the *fingerprint* of the block, which the clique members then exchange and verify with each other. After verification, the sealed blocks are *safely deleted*, which is partly how forward secrecy is achieved (newly joined users do not receive messages from sealed blocks).

2.1.5 Key management

As described in § 2.1.2, each clique has a common secret. This secret is used to derive the *key* which is then used to encrypt and authenticate the communication. The secret and the key are updated on multiple occasions. Initially, the key is calculated as:

$$k \leftarrow MAC_s(cliqueName)$$

where s is the initial random secret and $cliqueName$ is the public name of the clique that is known to all participants.

When a new participant is added, the secret gets renegotiated as per § 2.1.2, and the key is also recomputed:

$$s_{i+1} \leftarrow G^{s_i \cdot s_{other}} \bmod P$$
$$k_{i+1} \leftarrow MAC_{s_{i+1}}(cliqueName)$$

The key also gets rotated when a block is sealed:

$$s_{i+1} \leftarrow MAC_{k_i}(s_i)$$
$$k_{i+1} \leftarrow MAC_{s_{i+1}}(block\_content)$$

The resulting chain of secrets and keys can be schematically summarised as follows:

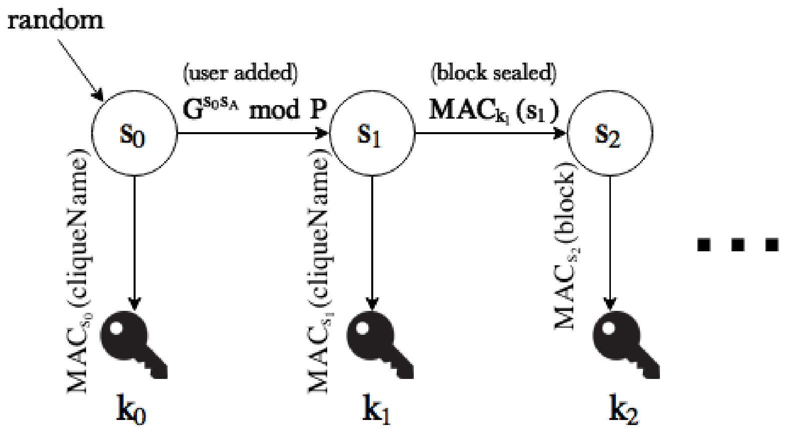


Figure 2.3: Schematic diagram of the key rotation process.

Note that the key rotation scheme helps achieve both forward and backward secrecy—an adversary who discovers the key at a particular time will not be able to decrypt messages from previous or future blocks. Given a compromised key, extracting the previous one is essentially equivalent to online inversion of a hash function, whereas computing the next key in the chain would require the adversary to also know the current secret.

2.1.6 Message format

Another issue that needs to be solved is addressing. When a KleeQ instance receives an encrypted message, it needs to determine which clique it belongs to. This is done using *address tags*, which is something that each clique has at all times. It is always equal to:

$$c_{id} = MAC_k(cliqueName)$$

This tag is at the beginning of every transmission and in conjunction with a look-up table of address tags can be used by a KleeQ instance to demultiplex messages into appropriate cliques as they arrive. As keys are rotated, old address tags are removed from the look-up table and new ones are added.

In addition, to protect the content of the communication from being viewed and/or modified by the adversary, it is necessary to encrypt and authenticate all messages using the current clique key. Following the recommended practices, we first encrypt the message and then MAC the concatenation of the ciphertext and the clique name. This allows to discard messages if they do not pass integrity checks, without attempting to decrypt them first.
Eventually, for a message m the following is transmitted:

$$T = c_{id}||E_k(m)||MAC_k(cliqueName||E_k(m))$$

Replay attacks (where an intercepted message is repeated by the adversary) are prevented by each m containing the name of the author, name of the recipient and the sender's Lamport time.

2.2 Requirements analysis

The requirements for the project follow directly from the aims outlined in § 1.2 and success criteria mentioned in the proposal.[1] They can be summarised as follows:

Implementing the protocol
Implement the protocol exactly as specified in § 2.1, ensuring that all of its security guarantees still hold, namely:

- confidentiality of message content
- message integrity
- forward secrecy
- backward secrecy
- message authorship repudiation
- conversation participation repudiation

Constructing a useable messaging application
Build a simple messaging application based on the protocol implementation.

It is worth pointing out that these two items are mutually dependent—implementing and debugging the protocol requires some kind of messenger prototype to be in place. This circumstance was identified early on and taken into account when planning the work (§ 2.4).

2.3 Preliminary system design

Some early architecture design was done before any code was written. In particular, the main classes of the implementation were conceived and thought through, and some of the additional external components necessary for operation of the protocol were engineered.

2.3.1 Secondary components

In order to implement and run KleeQ, it was necessary to address some secondary practical issues first. These issues are:

Contact Discovery
How do we know where our peers reside? Which IP addresses do we send messages to?

Transport
How do we send messages to peers who are not publicly addressable on the Internet? What if they have a non-public IP address (*e.g.* if they are behind a NAT)?

Both of these issues can be solved in a completely distributed fashion in a "pure" P2P way (*e.g.* using distributed hash tables), but in order to meet the time constraints of this project it was decided to use a centralised server-based solution. For the purpose of running KleeQ two server "scaffolding" applications were designed.

[1]See Appendix A

Address Book

This component keeps track of users' IP-addresses. All KleeQ instances perform a *check-in* every once in a while, thereby updating the records in the address book.

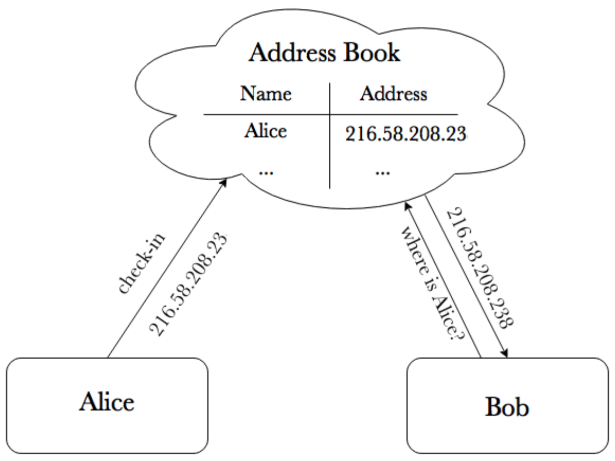


Figure 2.4: Operation of the Address Book

Other users query the address book when they need to send a message, and get the current IP-address of the destination.

Store-and-Forward Service

This application is used as a relay station to leave messages to be collected by other users later. It operates as a collection of "pigeonholes" where users put messages when they want to communicate and which are emptied by recipient every once in a while.

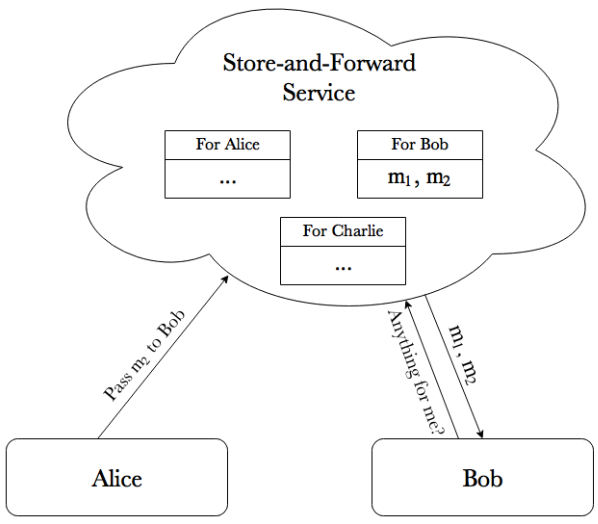


Figure 2.5: Operation of the Store-and-Forward Service

This scheme allows communication between users who do not have public IP-addresses, since the connection to the store-and-forward is always initiated by the client.

2.3.2 Object-oriented design

In order to achieve a modular and extensible design, a rough object-oriented design of the core part of the client application was produced. It consists of the three most important classes—`Communicator`, `Client` and `Clique`.

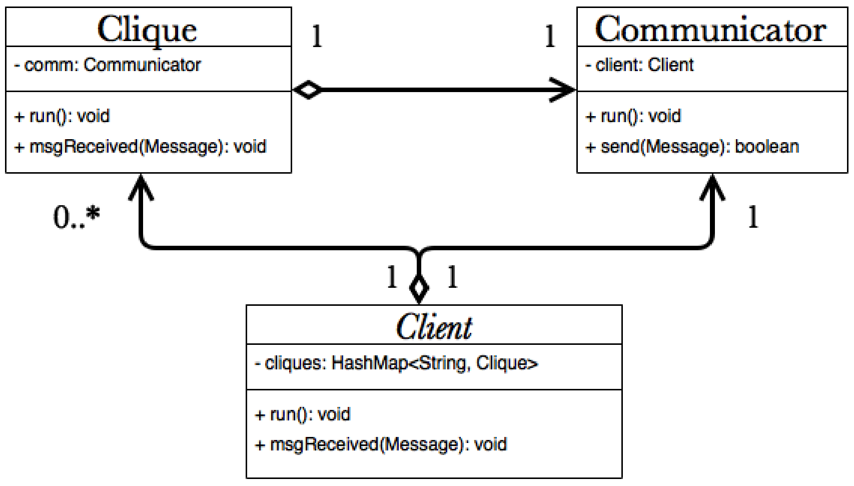


Figure 2.6: Core part of the client application

Each of these three classes are *active*, *i.e.* they are running as separate threads. `Communicator` is in charge of sending and receiving messages (directly via TCP or via Store-and-Forward). When a message is received, the `Communicator` gives a callback to class `Client` which demultiplexes the message into the appropriate Clique (as specified in § 2.1.6) by redirecting the callback to it. Class Clique is responsible for performing most of the actual protocol (as per § 2.1), including patching, sealing, key management *etc*. When Clique needs to send a message, it uses the `Communicator` for this. Apart from re-directing messages to the right cliques, the `Client` class also handles the interaction with the user (*e.g.* command-line interface or GUI).

2.4 Implementation strategy

It became clear early on that before commencing the implementation of the actual protocol the following preparatory steps needed to be taken:

- Building and testing the "scaffolding" server components, *i.e.* the Address Book and the Store-and-Forward.
- Implementing the transport functionality on the client side (class `Communicator` in Figure 2.6).
- Preparing a simple command-line interface to be able to test different parts of the code without changing it (part of class `Client` in Figure 2.6).

Given the small size and relatively low complexity of the tasks above, it was decided to implement these parts of the project following the *waterfall model*—do the design work upfront and write the code to fit the specifications.

Implementing KleeQ itself required a more flexible approach and the *iterative model* was chosen. The plan was to produce a series of prototypes, adding features incrementally:

Prototype 1: Clique Formation
: Implement clique formation (§ 2.1.2). Make sure users compute the shared secret correctly. Add encryption and authentication.

Prototype 2: Patching
: Instead of sending messages directly, use the patching algorithm (§ 2.1.3). Make sure users converge on a single transcript.

Prototype 3: Sealing
: Verify the global transcript via the process of block sealing (§ 2.1.4). Make sure the blocks are calculated, verified and deleted correctly.

Prototype 4: Key Management
: Improve key management by implementing key rotation (§ 2.1.5). Make sure rotation of address tags (§ 2.1.6) does not disrupt communication.

Prototype 5: Interface
: Enhance user interaction by building a GUI. Prepare for automated evaluation by constructing a "machine interface" for use by testing scripts.

2.5 Technical choices

Server-based components were implemented in Python, using the PyCharm IDE. For the purpose of running the code, a cloud server was rented from the PaaS provider PythonAnywhere[2]. As noted before, most of the server-side development needed to be done before writing any client code, so the testing was performed manually, using a simple utility called Postman[3] which allows sending custom HTTP requests and receiving responses.

The client-side code was written in Java, using the IntelliJ IDE. Messages were serialised into JSON strings using the `json.simple`[4] library. The graphical interface was built using the Swing framework.

The project relied on `git` for source code management, and the back-up strategy involved regular uploads to GitHub and copying to an external hard drive.

2.6 Summary

This chapter has described the KleeQ protocol in full detail (§ 2.1), presented the formal requirements for this project (§ 2.1), familiarised the reader with the early design work (§ 2.3) and the planning (§ 2.4) that was performed before any code was written. The next chapter provides further detail of design and elaborates on some interesting aspects of the implementation.

[2]`https://www.pythonanywhere.com/`

[3]`https://www.getpostman.com/`

[4]`https://github.com/fangyidong/json-simple`

Chapter 3

Implementation

This chapter describes in detail how the plan presented in § 2.4 was carried out. I start by looking at some auxiliary components (§ 3.1) that were implemented before KleeQ itself, namely the server infrastructure (§ 3.1.1) and their client-side API (§ 3.1.2), the client transport code (§ 3.1.3) and a simple command-line interface (§ 3.1.3). I then proceed to discussing the implementation of KleeQ (§ 3.2), elaborating in detail on each of the incremental prototypes that were constructed.

3.1 Preparatory coding

As explained in the previous chapter, implementing KleeQ as described in § 2.1 required some other code to be written first. In particular, the following components needed to be built and tested:

- server Python applications (Address Book and Store-and-Forward)
- client-side code for interacting with the server
- a command-line interface (CLI) for basic group chat operations

The next few sections describe in detail how these tasks were performed.

3.1.1 Server-side applications

This part of the work was regarded as a necessary diversion from the original goal of implementing and evaluating KleeQ, and therefore the design decisions were directed towards simplicity and quickness of implementation. Both server components are small HTTP services—the choice of protocol has been motivated by a good selection of third-party libraries for HTTP communication and the fact that it is not usually blocked by firewalls.

The Address Book and the Store-and-Forward (SaF) service were implemented using Flask[1], a RESTful Python framework for building HTTP server applications. Essentially, it provides an easy and efficient way to write server code which accepts and replies to HTTP requests. Here is a quick example of how it works:

[1] `http://flask.pocoo.org`

```python
from flask import Flask
app = Flask(__name__)

@app.route("/hello/<userID>")
def welcome(userID):
    return "Hello and welcome, %s!\n" % (userID)

if __name__ == "__main__":
    app.run()
```

The framework permits the defining of callback functions which get invoked when an HTTP request is received for a particular URL. For example, if the program above were running on `www.example.com` (on port 80), then typing `www.example.com/hello/Pavel` into the browser would result in the function above being called and the text "Hello and welcome, Pavel!" displayed.

Any complex server behaviour can be achieved in a similar way. This kind of system design is called *Representational State Transfer* (REST)—requests are encoded as access requests for some resource. All functions of the server applications are defined in this RESTful way—they are callbacks whose invocation is triggered by HTTP requests for specific URLs, which makes the code very easy to debug using a browser.

Address Book

This application is essentially an online look-up table which is used by client applications to record their *network addresses* (IP-address and port) from time to time (*"check-in"*), so that their peers knew where to send messages for them. The service has 3 commands:

Command	Arguments	Function
check-in	userID, IP:port	Records address of a particular user
lookup	userID	Returns the address of a given user
display	—	Returns addresses of all known users

Table 3.1: Functionality of the Address Book

All of these commands are defined as Flask callbacks, similar to the example above. The implementation is based on a hash map (Python dictionary) which maps unique user names (*userIDs*) to network address strings. When a user checks-in, their record is updated. In addition, every record contains the time of last check-in which is used to clear out stale records as users go offline. If a check-in happens from an IP-address belonging to the private range, then the system records it as "0.0.0.0:0", which serves as a signal for everyone that this user cannot be reached directly and all communication with them should be happening through the store-and-forward service.

Store-an-Forward

The operation of this component is analogous to that of a pigeonhole area at a Cambridge college. Users leave their communications (encrypted or otherwise) in “pigeonholes" marked with unique userIDs which are regularly emptied by their owners. The purpose of this arrangement is to facilitate communication in cases when a direct TCP connection cannot be easily made (*e.g.* when the destination resides behind a NAT). Table 3.2 presents the function set of the service.

Command	Arguments	Function
store	userID, msg	Stores a message for a given user
retrieve	userID	Returns all messages for user, clears postbox
view	userID	Returns all messages for a specific user

Table 3.2: Functionality of the Address Book

As before, each command is implemented as a Flask callback procedure. The implementation is based on a hash map where userIDs are keys and lists of base64-encoded message strings are values.

3.1.2 Client-side API for server applications

As mentioned before, the server applications use HTTP to communicate with the outside world. To access the commands of the services from the client application, a Java API has been built. It is a collection of static methods contained in 3 classes, which provide a convenient interface to the server components by sending the appropriate HTTP requests.

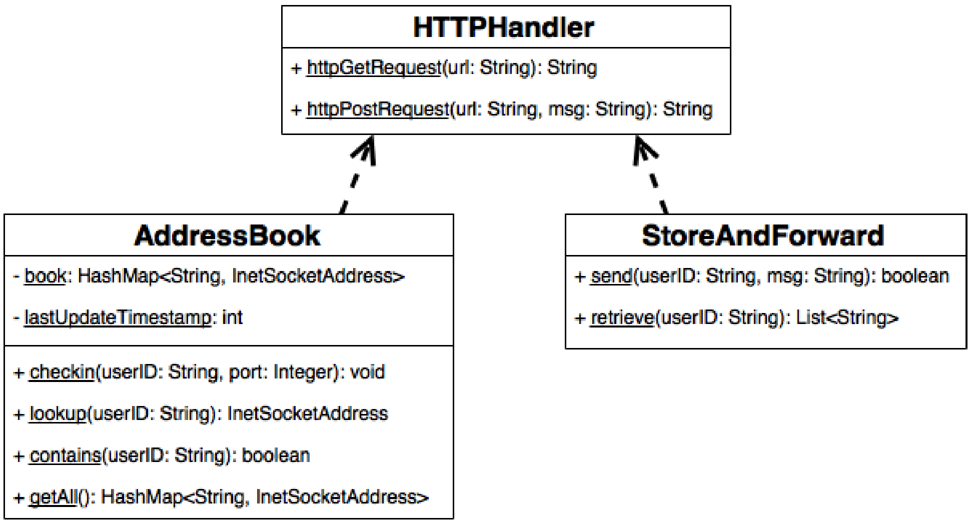

HTTPHandler
+ httpGetRequest(url: String): String
+ httpPostRequest(url: String, msg: String): String
AddressBook
- book: HashMap<String, InetSocketAddress>
- lastUpdateTimestamp: int
+ checkin(userID: String, port: Integer): void
+ lookup(userID: String): InetSocketAddress
+ contains(userID: String): boolean
+ getAll(): HashMap<String, InetSocketAddress>
StoreAndForward
+ send(userID: String, msg: String): boolean
+ retrieve(userID: String): List<String>

Figure 3.1: Class diagram of the API to server components

In Figure 3.1, class `HTTPHandler` implements the functionality for sending HTTP GET and POST requests. These two methods are used by the classes `AddressBook` and `StoreAndForward` to access the commands of the server applications. The rest of the client-side code can therefore use these classes to communicate with the server components as if they were available locally (*access transparency*).

In an early implementation of the `AddressBook` class, the methods `lookup()` and `contains()` were sending an HTTP request to the server upon every invocation which considerably slowed down the whole program. To solve this problem, a simple client-side *caching system* was built-in—the class downloads the whole address book from the server every few seconds, and the methods use this downloaded copy to compute their results. This means that the methods may return results that are slightly stale, but this does not cause any issues for KleeQ due to its inherently asynchronous nature.

3.1.3 Client-side transport code

The interface described above is primarily used by the `Communicator` class (see Figure 2.6), which is responsible for transport and connectivity. The class contains three threads that are perpetually running:

Address-reporting thread
Calls the `AddressBook` `checkin()` method every few seconds, to update/renew the server records about the address of the client. The client is assumed to be offline if the Address Book is unreachable.

SaF Querying Thread
Calls the `StoreAndForward` `retrieve()` function every few seconds, and gives a callback to the `Client` class for each of the new messages.

TCP Socket Server Thread
Listens on a port (randomly chosen at start up) for direct TCP connections, also giving callbacks to the `Client` class as messages arrive.

At this point, the reader may be wondering why threads for *both* direct TCP transmission and for SaF run at the same time. Indeed, as of now communication happens only though one of the two—it is direct (via TCP) if the client is running on a public IP-address, and via SaF otherwise. However, it is worth noticing that it may be possible to connect two clients directly even when they are not publicly addressable if they happen to reside on the *same local network*. This feature has not been implemented so far, but would definitely be useful in the future.

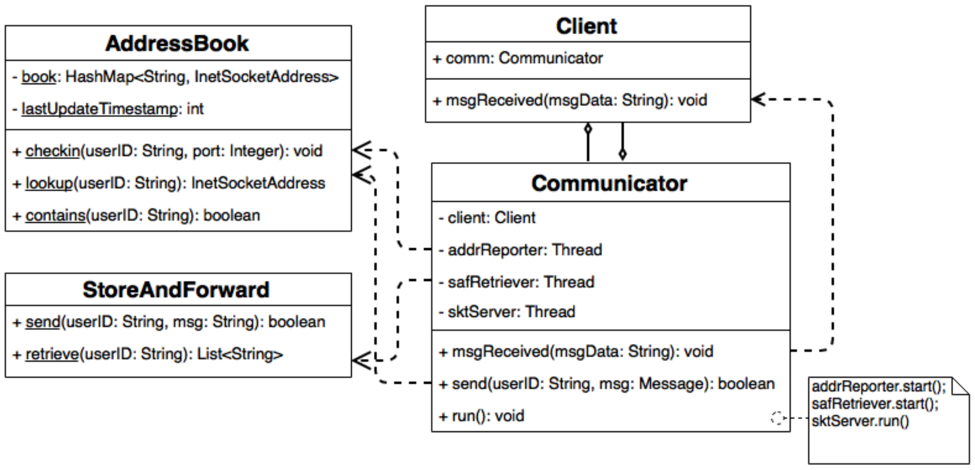

AddressBook
- book: HashMap<String, InetSocketAddress>
- lastUpdateTimestamp: int
+ checkin(userID: String, port: Integer): void
+ lookup(userID: String): InetSocketAddress
+ contains(userID: String): boolean
StoreAndForward
+ send(userID: String, msg: String): boolean
+ retrieve(userID: String): List<String>
Client
+ comm: Communicator
+ msgReceived(msgData: String): void
Communicator
- client: Client
- addrReporter: Thread
- safRetriever: Thread
- sktServer: Thread
+ msgReceived(msgData: String): void
+ send(userID: String, msg: Message): boolean
+ run(): void
addrReporter.start();
safRetriever.start();
sktServer.run()

Figure 3.2: Operation of the `Communicator` class

The `Communicator` class also contains a `send()` method which uses the data in the Address Book to decide which of the two transport options (direct TCP or SaF) to use for the given destination, and sends the message. The method returns a boolean value, depending on whether transmission has been successful (*e.g.* sending could fail if the client is offline).

3.1.4 A simple CLI

The primary reason for building this component was to be able to test various aspects of the KleeQ's operation just by typing commands, without changing the source code. In addition, upfront design of a CLI helped understand what use-cases need to be handled and make the implementation process more systematic.

The design involved defining a minimal set of operations that a group messaging application needs to have. The corresponding commands are shown in Table 3.3.

Command	Arguments	Function
	Main menu commands	
<empty>	–	Refresh
create	groupName	Create a new group with given name
add	userID, groupName	Add a specific user to a given group
view	groupName	View a particular group (*group view*)
help	–	Display a help message
exit	–	Terminate the program
	Group view commands	
<empty>	–	Refresh
msg	msgText	Send a message to current group
back	–	Go back to main menu
add	userID	Add a specific user to current group
help	–	Display a help message
exit	–	Terminate the program

Table 3.3: Command-line interface

The implementation of the interface above is contained in the `CLIClient` class which is a subclass of `Client` (Figure 3.3).

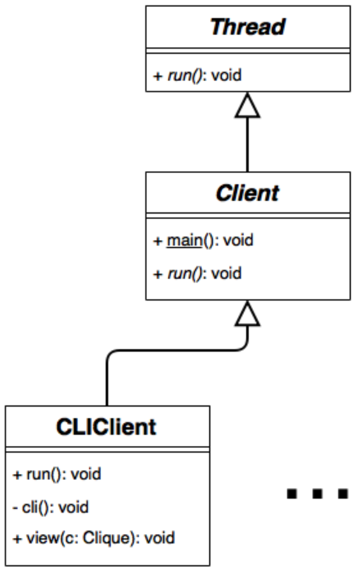


Figure 3.3: The CLI is implemented in the class `CLIClient`

This proved to be very convenient later on, since implementing other types of interfaces (*e.g.* GUI) could be neatly done by creating another subclass of `Client` and writing a different version of the `run()` method.

3.2 Protocol implementation

Once the "scaffolding" (server components and CLI) was completed, it became possible to proceed with the implementation of KleeQ itself, as described in § 2.1.

3.2.1 Prototype 1: Clique formation

Building this first prototype involved implementing the procedure for creating a new clique and adding users to it. At this point, messages between clients were to be sent directly, rather than via patching, to be able to implement and test the cryptography more easily. This required the following tasks to be completed:

- Designing an object-oriented representation for a clique such that it neatly fits into the previous design.
- Working out the exact mechanics for adding a new user, based on the description in § 2.1.2. Designing the classes to model this process.
- Choosing the cryptographic methods to be used for Diffie-Hellman key exchange, encryption and message authentication. Implementing them in the object-oriented fashion that would work well with the rest of the design.

The next few sections address these three tasks in more detail.

Object-oriented clique model

A clique is modelled by the `Clique` class (Figure 3.4) which contains all the group-specific information and implements most of the protocol's logic. Instances of the class are stored in a hash map (clique name to `Clique` instance) contained in the `Client` class, and its methods are called by the UI code when the user issues a command. In addition, `Client` is responsible for de-multiplexing the incoming messages into appropriate cliques—this is done using another hash map that maps *address tags* (§ 2.1.6) to clique names. To re-direct a message into a specific clique, the `Client` calls the `datagramReceived()` method for the relevant instance of `Clique`.

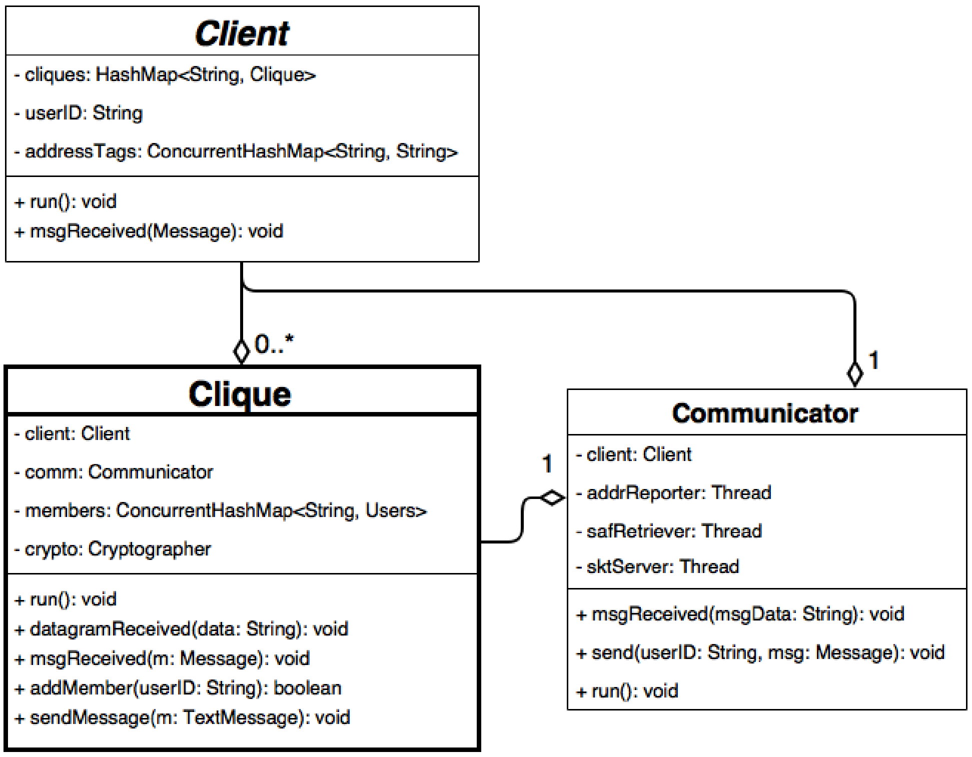

Client
- cliques: HashMap<String, Clique>
- userID: String
- addressTags: ConcurrentHashMap<String, String>
+ run(): void
+ msgReceived(Message): void
0..*
1
Clique
- client: Client
- comm: Communicator
- members: ConcurrentHashMap<String, Users>
- crypto: Cryptographer
+ run(): void
+ datagramReceived(data: String): void
+ msgReceived(m: Message): void
+ addMember(userID: String): boolean
+ sendMessage(m: TextMessage): void
1
Communicator
- client: Client
- addrReporter: Thread
- safRetriever: Thread
- sktServer: Thread
+ msgReceived(msgData: String): void
+ send(userID: String, msg: Message): void
+ run(): void

Figure 3.4: The `Clique` class models a communication group.

Messages for adding users

As mentioned in § 2.1, KleeQ solves the key distribution problem via the Diffie-Hellman key exchange protocol. However, apart from converging on a common secret adding a new user involves *distribution of group membership information*. In other words, when a new user is added by one of the current conversation participants, they need to be notified of what *other* users are part of the group. There exist multiple ways of doing it, but no particular method was suggested by KleeQ's authors in the original paper. The method that was chosen as part of this project is illustrated in Figure 3.5.

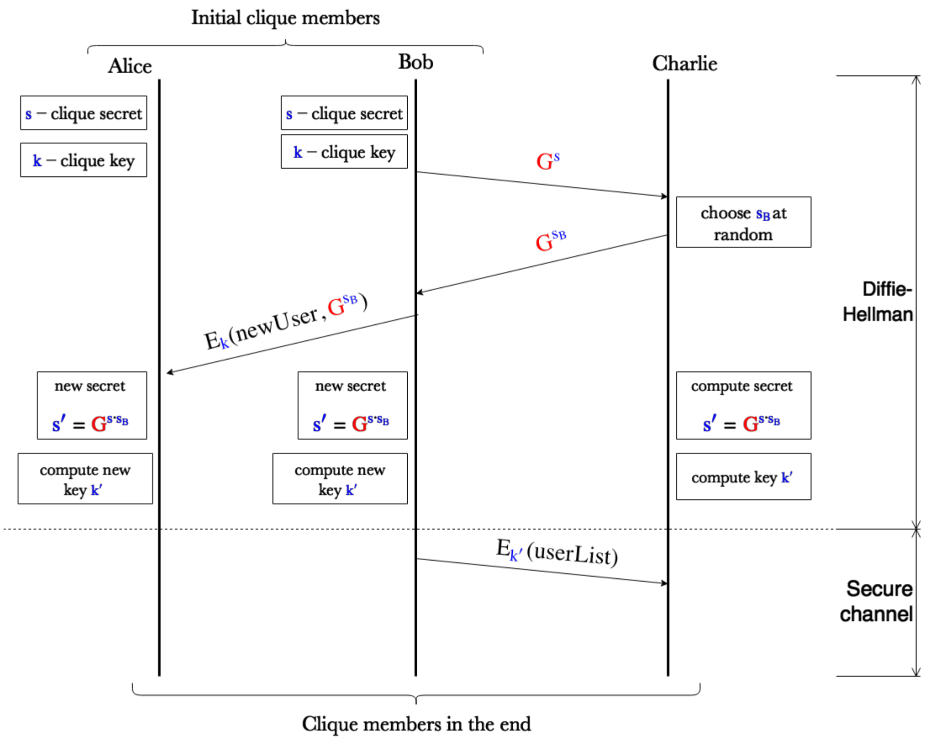


Figure 3.5: Schematic procedure for adding a new user to a clique. Public information is in red, private—in blue.

The figure shows how one of the current members (Bob) adds a new user (Charlie) to an existing clique also containing user Alice. Bob sends an invitation containing a Diffie-Hellman negotiation parameter based on the current clique secret. Charlie replies with his own negotiation parameter, and computes the new group secret and key. Upon receiving Charlie's response, Bob notifies the existing group members (Alice) of the newly added user using the *old* key, also forwarding the new user's negotiation parameter. Then Bob computes the *new* group secret, and uses the new key to provide Charlie with the current list of clique members ($userList = [Alice, Bob]$, in this example).

In terms of programming, each of the messages in Figure 3.5 is modelled as a Java class (see Figure 3.6). All message classes are derived from the abstract Message parent class which contains information and behaviour common to all messages.

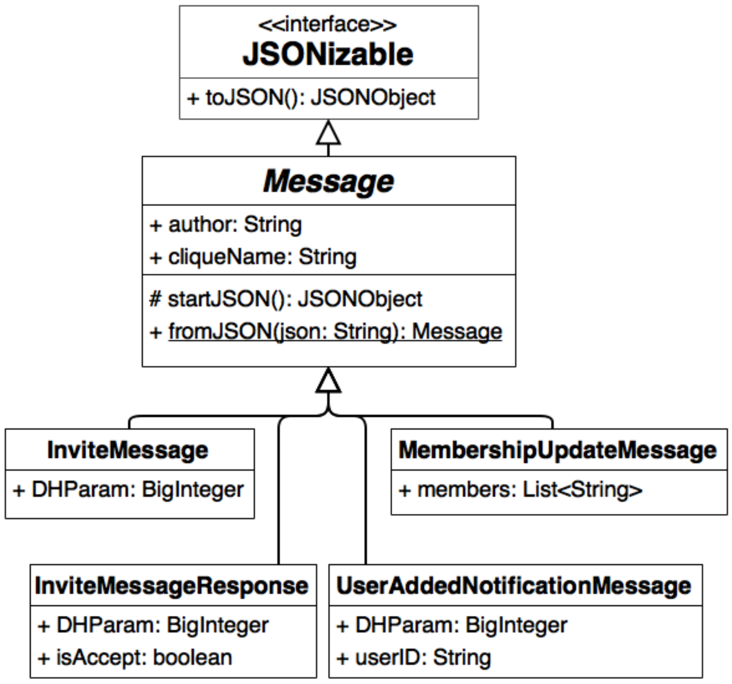


Figure 3.6: Classes modelling messages necessary to add a new user to a clique.

For transmission, instances of these classes are serialised into JSON strings[2]. Using JSON was preferred over designing a custom byte-by-byte message layout because it is more flexible (can add and remove fields without changing much code), and was favoured over Java serialisation because it is not limited to Java and can make interoperability with other languages easier if needed. Conversion to JSON is performed using the `toJSON()` method, declared in the `JSONizable` interface and implemented in each of the child classes. When a message is received, the appropriate object is reconstructed by calling the static `fromJSON()` method in the base `Message` class.

Cryptography

The `Clique` class contains an instance of the `Cryptographer` (Figure 3.7) class that holds the key material and implements the relevant cryptographic procedures (Diffie-Hellman key exchange, encryption/decryption, MAC, message digest).

[2]The third-party library `json-simple` was used for this (`https://github.com/fangyidong/json-simple`)

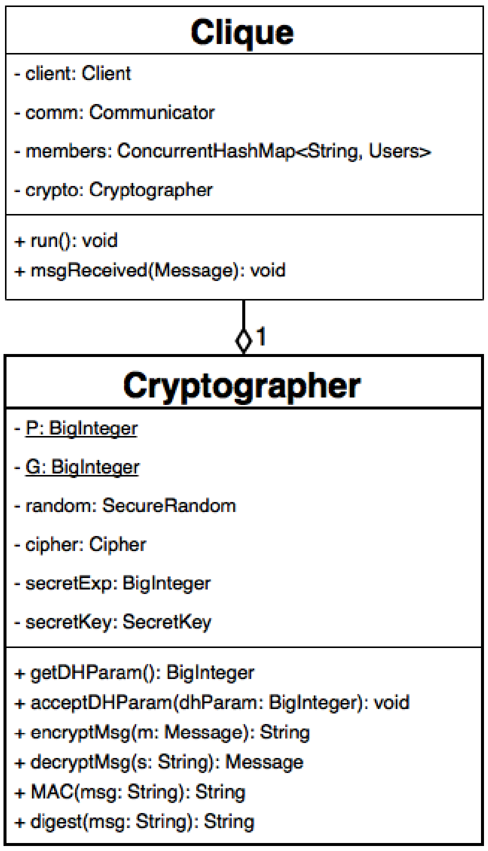

Clique
- client: Client
- comm: Communicator
- members: ConcurrentHashMap<String, Users>
- crypto: Cryptographer
+ run(): void
+ msgReceived(Message): void
1
Cryptographer
- P: BigInteger
- G: BigInteger
- random: SecureRandom
- cipher: Cipher
- secretExp: BigInteger
- secretKey: SecretKey
+ getDHParam(): BigInteger
+ acceptDHParam(dhParam: BigInteger): void
+ encryptMsg(m: Message): String
+ decryptMsg(s: String): Message
+ MAC(msg: String): String
+ digest(msg: String): String

Figure 3.7: An instance of `Cryptographer` class is contained in `Clique`, and handles all group-specific cryptographic procedures.

The implementation of the key exchange is based on modular arithmetic with values of generator G and group order P taken to be:

```
P  =  FFFFFFFF FFFFFFFF C90FDAA2 2168C234 C4C6628B 80DC1CD1
      29024E08 8A67CC74 020BBEA6 3B139B22 514A0879 8E3404DD
      EF9519B3 CD3A431B 302B0A6D F25F1437 4FE1356D 6D51C245
      E485B576 625E7EC6 F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
      EE386BFB 5A899FA5 AE9F2411 7C4B1FE6 49286651 ECE45B3D
      C2007CB8 A163BF05 98DA4836 1C55D39A 69163FA8 FD24CF5F
      83655D23 DCA3AD96 1C62F356 208552BB 9ED52907 7096966D
      670C354E 4ABC9804 F1746C08 CA18217C 32905E46 2E36CE3B
      E39E772C 180E8603 9B2783A2 EC07A28F B5C55DF0 6F4C52C9
      DE2BCBF6 95581718 3995497C EA956AE5 15D22618 98FA0510
      15728E5A 8AAAC42D AD33170D 04507A33 A85521AB DF1CBA64
      ECFB8504 58DBEF0A 8AEA7157 5D060C7D B3970F85 A6E1E4C7
      ABF5AE8C DB0933D7 1E8C94E0 4A25619D CEE3D226 1AD2EE6B
      F12FFA06 D98A0864 D8760273 3EC86A64 521F2B18 177B200C
      BBE11757 7A615D6C 770988C0 BAD946E2 08E24FA0 74E5AB31
      43DB5BFC E0FD108E 4B82D120 A9210801 1A723C12 A787E6D7
      88719A10 BDBA5B26 99C32718 6AF4E23C 1A946834 B6150BDA
      2583E9CA 2AD44CE8 DBBBC2DB 04DE8EF9 2E8EFC14 1FBECAA6
      287C5947 4E6BC05D 99B2964F A090C3A2 233BA186 515BE7ED
      1F612970 CEE2D7AF B81BDD76 2170481C D0069127 D5B05AA9
      93B4EA98 8D8FDDC1 86FFB7DC 90A6C08F 4DF435C9 34063199
      FFFFFFFF FFFFFFFF

G  =  2
```

which is a 4096-bit group recommended for use in Internet key exchange (IKE) in RFC3526 [5].

The Diffie-Hellman procedures were implemented manually rather than using a library solution, for didactic purposes. A more advanced library implementation (*e.g.* based on elliptic curves) can always be easily substituted in place of the existing one if necessary—this would only require changing the `getDHParam()` and `acceptDHParam()` methods.

For encryption, the `Cryptographer` uses AES—the industry-standard symmetric block cipher at the moment. I use keys of length 128 bits, as recommended by the German Office of Information Security (BSI) for use up until year 2021 [6]. Stronger cryptography (AES-192 or AES-256) can be enabled with almost no changes to the code (just need to change the key size constant), but this requires manual installation of the Java Cryptography Extension (JCE) which would have a negative usability effect or would necessitate the creation of an installer program. Our implementation uses AES-128 in *counter mode* (CTR) with the initial vector generated by a secure pseudo-random number generator. This choice was made because the messages are generally of variable length and CTR is space-efficient in this case since it requires no padding.

For message authentication, I use keyed-hash message authentication codes (HMAC) based on SHA-256. According to a recent report by the German Office of Information Security (BSI), this cryptographic hash function can currently be considered secure and is expected to remain in active use at least until 2021 [6]. As of 2014, the best known attack could practically find a collision on 28 out of 64 rounds of SHA-256 [7]. This project also uses SHA-256 for computing message digests. As per the recommended practices[3], I use *encrypt-then-MAC* (EtM), to not have to decrypt messages in order to verify their authenticity and integrity.

The same key is used for encryption and MACing. Whilst not recommended in general, this is secure in this case because I calculate:

$$MAC_k(cliqueName||E_k(m))$$

and there are no known interactions between AES-CTR and HMAC-SHA256, since the underlying primitives are sufficiently different.

3.2.2 Prototype 2: Patching

As mentioned before, the first prototype relied on exchanging messages by sending them directly. Building the second prototype involved using the process of patching (§ 2.1.3) for communication. To do this, the following tasks needed to be accomplished:

- Designing a specialised data structure to store the message history, that would enforce the $<_L$ ordering as defined in § 2.1.3

[3]ISO/IEC 19772:2009

- Working out what messages and in what order exactly need to be sent, and model this process using classes
- Use result of the previous two tasks to implement the patching algorithm

The next several sections elaborate on the details of how the above was done.

Data structure for Message History

In the first prototype, the messages were sent directly and message history was stored in a list. This simple solution was sufficient because the history was only used to show it to the user, in order of arrival time. However, as was described in § 2.1.3, patching (and later sealing) requires the messages to be ordered according to a specific total order ($<_L$). In addition, the message history needs to be cheap to iterate, since the patching algorithm will require it to answer queries of the form "last N messages by user X", and also iteration will be the basis of *block finding* as part of sealing § 2.1.4. Apart from the actual messages, a *version number* and a *Lamport timestamp* need to be kept at all times.

A logical way to satisfy the requirements above would be to model message history as a class, that contains the Lamport timestamp, the version number and the data structure storing messages according to some ordering (Figure 3.8). The Java Standard Library contains a class for just such a data structure—it is the `TreeSet<T>`. `TreeSet` is a sorted, bidirectionally iterable collection that stores its items in an order specified by a `Comparator`.

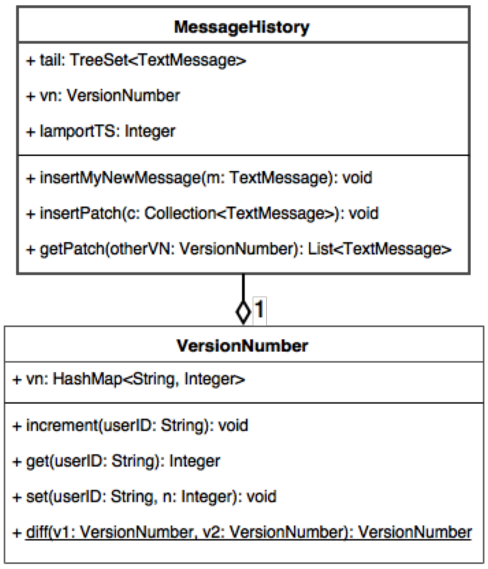

MessageHistory
+ tail: TreeSet<TextMessage>
+ vn: VersionNumber
+ lamportTS: Integer
+ insertMyNewMessage(m: TextMessage): void
+ insertPatch(c: Collection<TextMessage>): void
+ getPatch(otherVN: VersionNumber): List<TextMessage>
1
VersionNumber
+ vn: HashMap<String, Integer>
+ increment(userID: String): void
+ get(userID: String): Integer
+ set(userID: String, n: Integer): void
+ diff(v1: VersionNumber, v2: VersionNumber): VersionNumber

Figure 3.8: `MessageHistory` models conversation history and provides an interface to it which is necessary for patching.

The implementation of VersionNumber is based on a hash map which keeps the number of messages authored by each user (if user is not in the hash map, 0 is assumed).

Messages needed for Patching

Algorithm 1 in § 2.1.3 shows the process of patching that is used by KleeQ to exchange messages. Essentially, each user updates every other user on the messages that they have not yet seen. Despite being somewhat inefficient (quadratic complexity), this scheme seems to be the only one to allow communication even when some of the group members are offline. The algorithm specifies the following kind of message exchange:

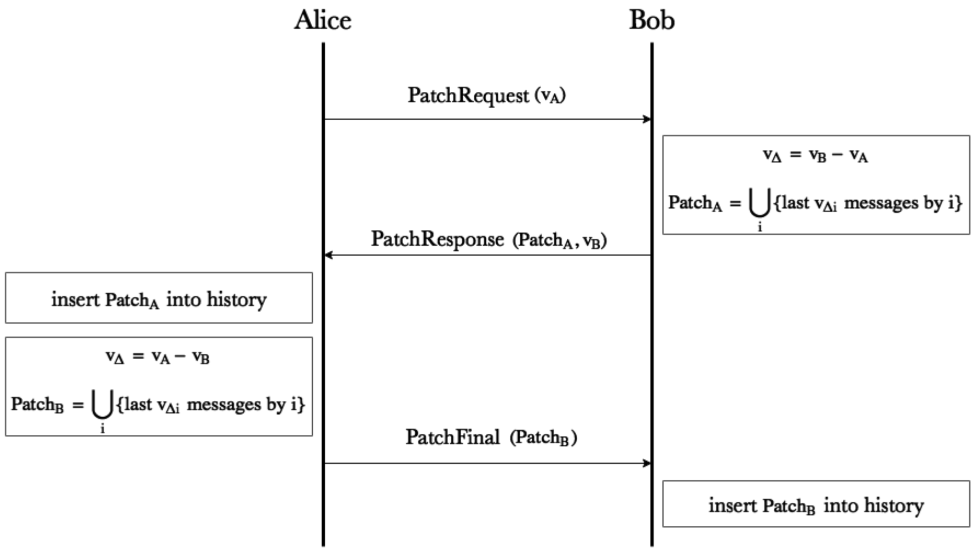

Alice
Bob
PatchRequest (v_A)
$v_\Delta = v_B - v_A$
$\text{Patch}_A = \bigcup_i \{\text{last } v_{\Delta i} \text{ messages by i}\}$
PatchResponse (Patch_A, v_B)
insert Patch_A into history
$v_\Delta = v_A - v_B$
$\text{Patch}_B = \bigcup_i \{\text{last } v_{\Delta i} \text{ messages by i}\}$
PatchFinal (Patch_B)
insert Patch_B into history

Figure 3.9: Patching algorithm.

Implementing the scheme as shown above would require having three different message types (one for each communication). To make things simpler, I decomposed the exchange above into two request-response interactions:

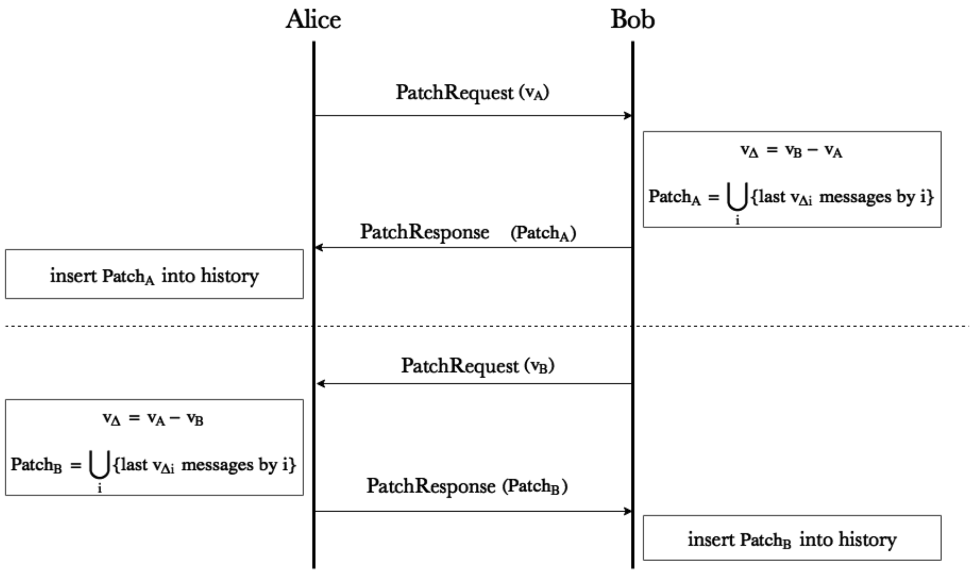

Alice
Bob
PatchRequest (v_A)
$v_\Delta = v_B - v_A$
$\text{Patch}_A = \bigcup_i \{\text{last } v_{\Delta i} \text{ messages by i}\}$
PatchResponse (Patch_A)
insert Patch_A into history
PatchRequest (v_B)
$v_\Delta = v_A - v_B$
$\text{Patch}_B = \bigcup_i \{\text{last } v_{\Delta i} \text{ messages by i}\}$
PatchResponse (Patch_B)
insert Patch_B into history

Figure 3.10: Message scheme for the patching algorithm.

This only requires two message types, and is therefore easier to implement. The overhead of transmitting four messages instead of three is minimal, since the messages also become smaller. As before, each of the message types is modelled with a subclass of the `Message` class (Figure 3.11).

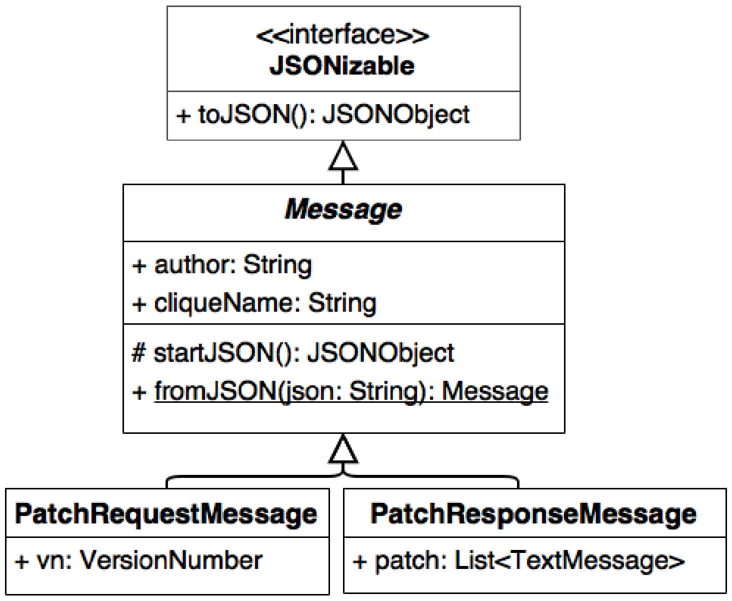


Figure 3.11: Message classes for patching scheme.

The messages are converted to JSON strings as before.

Details of implementation

As mentioned before, the `Clique` class is *active*, *i.e.* it contains a `run()` method whose code is executed in a separate thread. This thread is used to send out patch requests—there is an infinite loop that regularly sends out a *patching burst*, where a patch request is sent to each of the conversation participants. To make sure that users do not query each other in lockstep, the process is randomised—the time between bursts is a random uniformly distributed quantity, as is the spacing between requests within a single burst. In addition, in order to reduce the amount of consumed traffic and not send too many redundant requests, I have implemented a timeout mechanism—it keeps the time of the last request sent to each user in a hash map, and re-transmission happens only after a specific period of time.

This scheme has a lot of parameters that can be varied—inter-burst time spacing, intra-burst time spacing, request timeout period. These parameters can be changed depending on how intensive we want the patching process to be. The trade-off here is time-to-delivery vs traffic/energy consumption.

3.2.3 Prototype 3: Sealing

The patching algorithm allows clique members to converge on a single transcript. To make sure that no messages in the transcript were damaged or maliciously modified, users independently split it into *blocks* in a deterministic way (as per § 2.1.4) and compare hashes of the blocks. Once it has been established that everybody has the same version of a block, it can be safely deleted. To implement this process, two tasks needed to be performed:

- Working out the exact protocol that clique members must use to compare hashes (what messages to sent, when to send them and what bookkeeping to do).
- Coming up with an object-oriented implementation for the above that would integrate well with the rest of the design.

The next few sections describe how these tasks were accomplished in more detail.

Messages needed for Sealing

The original designers of the protocol left this part unspecified, only going as far as saying that hashes need to be compared once a block has been found by the block finding algorithm (Algorithm 2 in § 2.1.4). This section describes the solution that was used to fill this gap.

When a user finds a block is found, they *signal* this event to all of their peers by sending the hash code of the block. After the signal has been sent out, the user waits to receive the same signal from *every other clique member*. When the appropriate signal has been received from everyone, the user considers the block sealed.

Due to the asynchronous nature of communication, it is inevitable that users will be discovering blocks at different times. Thinking systematically, there are two cases that need to be considered:

1. Users discover a block at almost the same time
2. Users discover a block with a significant time difference

Figures 3.12 and 3.13 illustrate each of these cases for the case of clique of size 2.

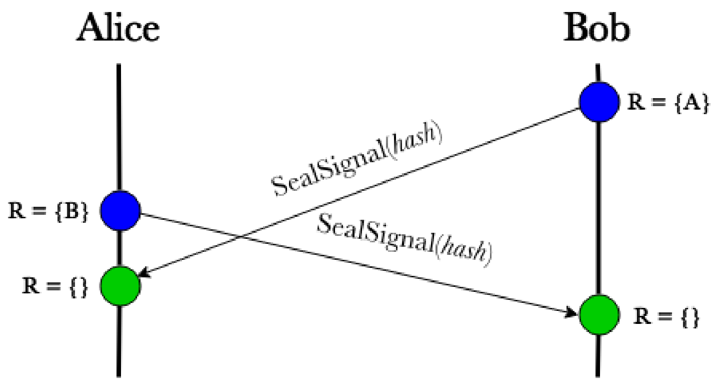


Figure 3.12: Case 1. Alice and Bob find the block almost simultaneously.

Figure 3.12 shows the case where participants discover (blue circle in figure) a block almost at the same time and signal each other. For each signalled hash, every participant keeps track of R—subset of clique members whose signals for the corresponding blocks have not yet been received. When the set R becomes empty, the block is sealed (green circle in figure).

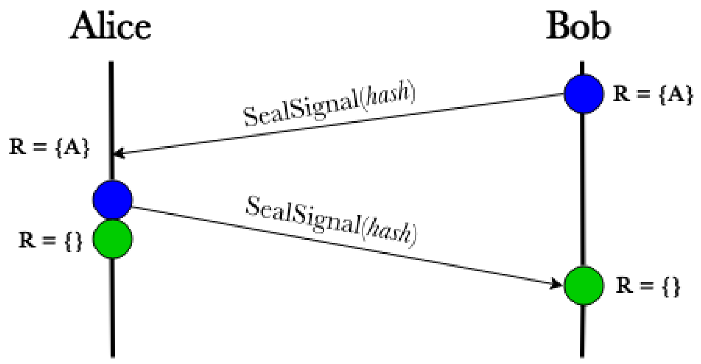


Figure 3.13: Case 2. Bob finds a block much earlier, and his signal is received by Alice before she finds it herself.

Figure 3.13 shows what happens when the participants find a block at significantly different times. Alice receives a signal for a block she has not discovered yet, but her actions are effectively same as before—she keeps a set of users who still have not found the block *which includes herself*. When she finds the block, she signals this and then considers it sealed.

As can be seen, in both cases 2 messages are sufficient to seal a block, which is much better than the straightforward solution where hashes are verified by each user in a request-response way, similarly to patching. For a general clique of size n, the message count is $n(n-1)$.

The scheme described above is sufficient under the assumption of *reliable transport* which may be valid as part of this project (both TCP and HTTP are reliable) but not generally. For completeness, the cases when messages get lost or intercepted have also been handled. It is in fact quite straightforward—if someone's signal gets lost, they are reminded about it by those who failed to receive it. An example of this is shown in Figure 3.14.

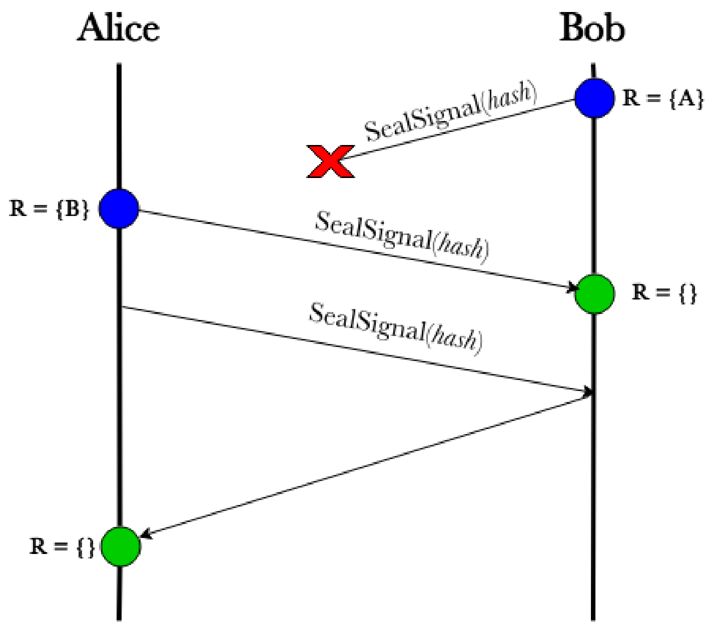


Figure 3.14: One of the signals is lost. Unlucky recipient requests a retransmission explicitly.

Bob discovers a block and signals it to Alice, but the message is unfortunately lost in transit. Alice finds the block herself and notifies Bob about it—Bob now regards the block sealed. After some time, Alice sees that she is missing a confirmation from Bob and signals him again to ask for a re-transmission. Upon receiving the notification, Bob sees that it is *for a block that has already been sealed*, and re-sends the signal to Alice.

Details of implementation

The basis of the scheme described above is the block finding algorithm (Algorithm 2 from § 2.1.4). It was implemented as the `getNextSealableBlock()` method of the `MessageHistory` class. At this point, some of the advance planning done for the previous prototype paid off—the fact that `TreeSet` had been chosen as the storage structure for messages helped implement the block finding algorithm in an efficient and straightforward way. In particular, the algorithm heavily relies on iterating over the history in both directions and taking sub-lists, and both of these operations are cheap and provided as standard methods in `TreeSet`. Also, a simple caching mechanism was used to avoid re-computing the blocks every time the `getNextSealableBlock()` is called—the method finds the next block only once, and then re-uses its previous work on following calls when possible.

A discovered block is represented by the class `SealableBlock` (Figure 3.15).

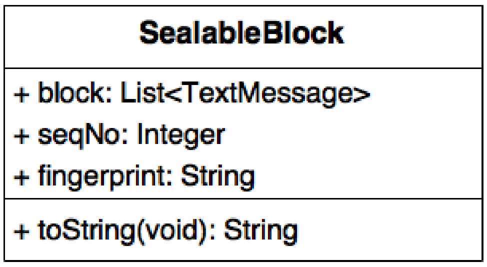


Figure 3.15: `SealableBlock` class.

The class contains the block's fingerprint that is calculated as follows:

$$\text{fingerprint} = \text{SHA256}\Big(block.number + \sum_{m \in block} (''|'' + m.author + '' : '' + m.text + ''@'' + m.lamportTime)\Big)$$

where *block.number* is the sequence number of the block.

All of the signalling logic as described in the previous section is contained in the `Clique` class. Similarly to patching, every few seconds we attempt to discover a block in the `MessageHistory`. If the `getNextSealableBlock()` returns a non-empty block, the application sends out a signal message containing the hash of this block to each of the clique members. We keep a hash map that maps a block hash code to the set of users who still have not confirmed its discovery (set R from the previous section). Every once in a while, a signal is re-sent to everyone in the set, in case they did not get it the last time or their signal did not reach us. As before, received messages are handled in the `msgReceived()` method—the logic for what happens when a signal is *received* is implemented there.

Similarly to other message types, the signalling message class (`SealSignalMessage`) is just another subclass of `Message` (Figure 3.16) and is converted to JSON for transmission in a similar way.

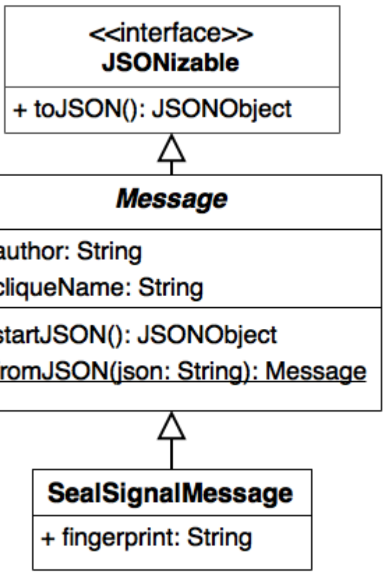


Figure 3.16: `SealSignalMessage` class models the block discovery signal.

3.2.4 Prototype 4: Key management

Key management as described in § 2.1.5 proved to be fairly easy to implement, thanks to the provident design and flexible programming performed at the previous stages. The key rotation required when adding a new user to the clique is automatically handled by the implementation of the clique formation routines based on Diffie-Hellman key exchange (see Prototype 1 in § 3.2.1).

Rotating the key after a block has been sealed was implemented by just adding one extra method to the `Cryptographer` class:

```
synchronized public void rotateKey(SealableBlock sBlock) {
    /* updates the shared secret and rotates the key on block seal
    *       secret <--- MAC(K, secret)
    *       K <--- MAC(secret, block_contents)
    * */
    // save current key, for sealing retransmissions
    prevKey = secretKey

    // update the shared secret
    secretExp = new BigInteger(macBytes(secretExp.toByteArray()));

    // temporary update for secret key, to calc MAC(secret, block_content)
    secretKey = new SecretKeySpec(secretExp.toByteArray(), 0, encBitLength / 8, encAlgo);
    byte[] newSecretKeyBytes = macBytes(sBlock.toString().getBytes());

    // final update to secret key
    secretKey = new SecretKeySpec(newSecretKeyBytes, 0, encBitLength / 8, encAlgo);
}
```

As noted in § 3.2.3, my sealing protocol requires users to be able to receive and send signals regarding previously sealed blocks (the case shown in Figure 3.14). This is implemented by always keeping the previous key in memory, in case someone needs a retransmission of the sealing signal.[4]

[4] It is easy to show that keeping more than one old key is not necessary—it is impossible to "fall behind" by more than one block.

3.2.5 Prototype 5: Interface

This was the final prototype whose aim was to prepare the application for evaluation and demonstration. For these purposes, two tasks needed to be accomplished:

- Building a *machine interface*, to allow easy interaction of the application with the evaluation scripts.
- Constructing a simple graphical interface (GUI) with the same functionality as the existing command-line interface (CLI).

Both of these components were fairly straightforward to add to the existing messaging application due to the modular approach that was chosen when writing the CLI (§ 3.1.4). The required functionality could be easily added with two more extensions of the abstract `Client` class (Figure 3.17).

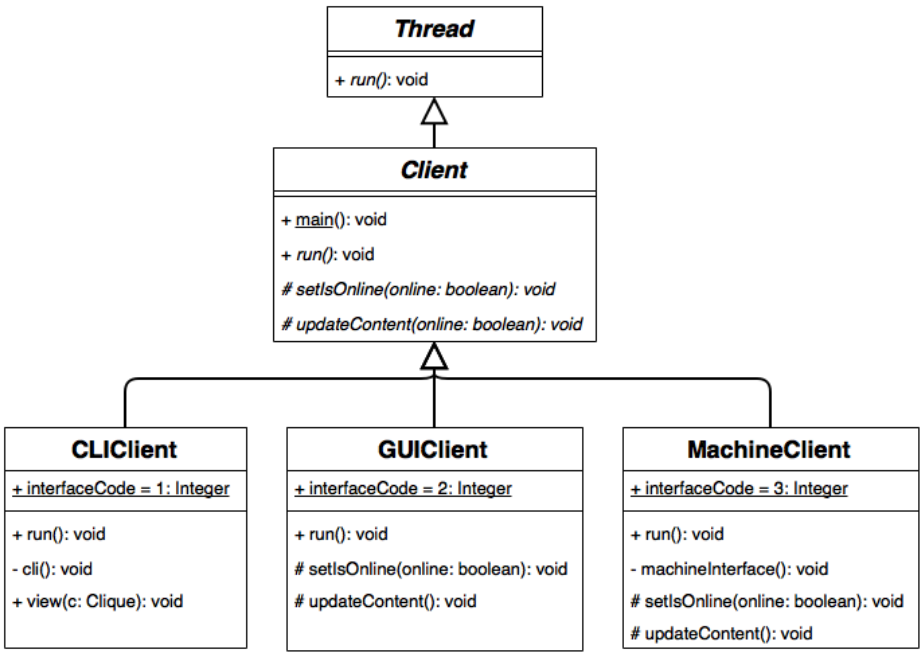

Thread
+ run(): void
Client
+ main(): void
+ run(): void
# setIsOnline(online: boolean): void
# updateContent(online: boolean): void
CLIClient
+ interfaceCode = 1: Integer
+ run(): void
- cli(): void
+ view(c: Clique): void
GUIClient
+ interfaceCode = 2: Integer
+ run(): void
# setIsOnline(online: boolean): void
# updateContent(): void
MachineClient
+ interfaceCode = 3: Integer
+ run(): void
- machineInterface(): void
# setIsOnline(online: boolean): void
# updateContent(): void

Figure 3.17: Class hierarchy implementing different interfaces.

The next two sections below present a more detailed account of how the interfaces have been implemented.

Machine Interface

Manually evaluating the newly created application at a large scale would hardly have been possible—it would require too much human input. The machine interface was built for the purpose of evaluating the application in an automated way. It was expected that evaluation would be accomplished by running a single script that would start-up several instances of `N0NaMe` and control them by sending commands (Figure 3.18).

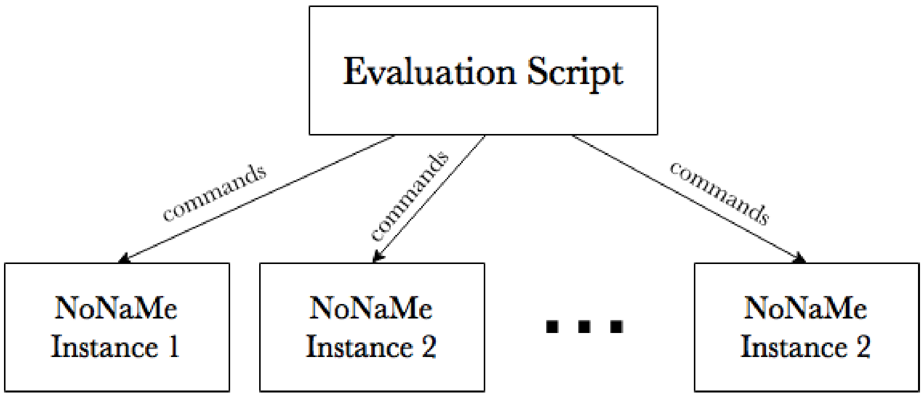


Figure 3.18: Evaluation script interacts with several `N0NaMe` instances.

The required kind of inter-process communication (IPC) can be easily achieved by redirecting the *standard streams* (`stdin`, `stdout` and `stderr`). The programming of the machine interface is not much different from that of the CLI. Essentially, it is a simplified version of the CLI with both commands and output optimised for interaction with a script rather than a human. Table 3.4 summarises the command set.

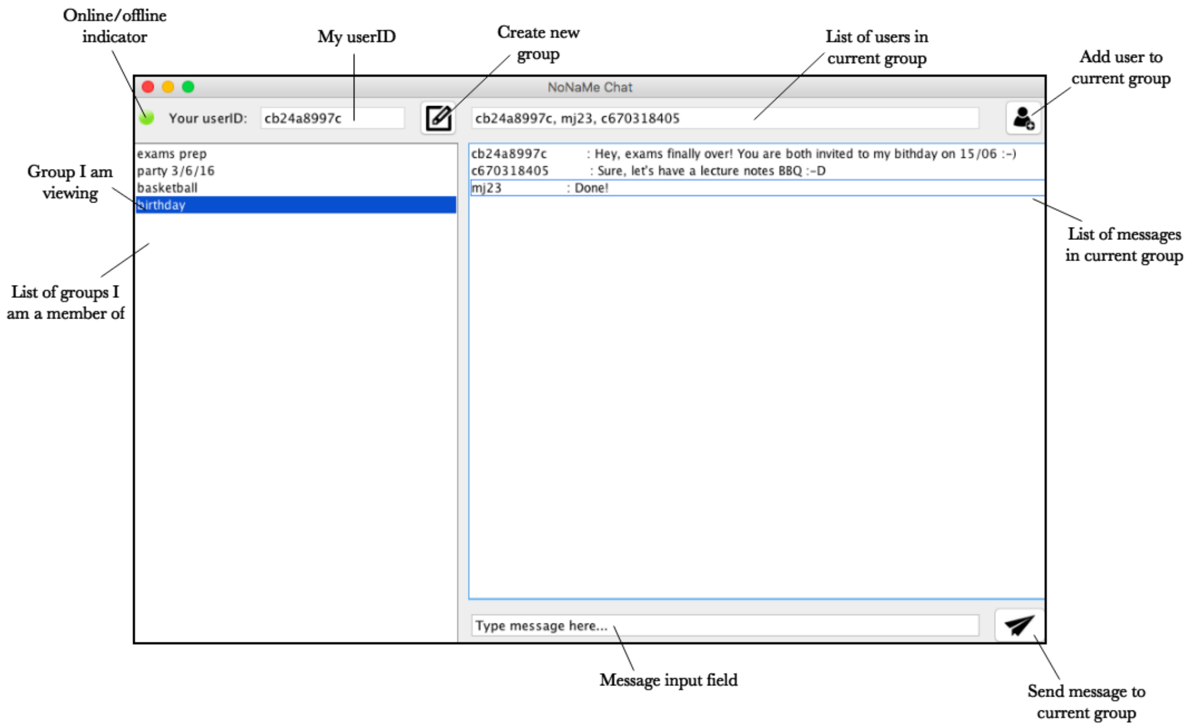
Command	Arguments	Output
groups	–	space-separated list of groups
peers	–	space-seprated list of list of peers online
status	–	online/offline
create	groupID	ACK if successfully created, Error otherwise
add	userID, groupID	ACK is added, Error otherwise
msg	groupID, msgText	ACK if sent, Error otherwise
history	groupID	JSON([(author, text), ...])
members	groupID	space separated list of users in a clique
exit	–	ACK (program shuts down)

Table 3.4: Machine interface commands

As can be seen, the machine interface is *mode-free*—this was done to reduce the amount of state that an evaluation script would need to hold. As in the case of the CLI, the implementation is contained in the `machineInterface()` method of `MachineClient`, and consists of a simple infinite loop where the input is taken from `stdin` and parsed by a few conditional statements, and the output is written into `stdout`.

Graphical User Interface (GUI)

A GUI was built to assess the usability implications of using KleeQ, as well as to demonstrate the newly constructed messaging application. Figure 3.19 shows its layout and explains the functionality of various components.


Online/offline indicator
My userID
Create new group
List of users in current group
Add user to current group
Group I am viewing
List of groups I am a member of
List of messages in current group
Message input field
Send message to current group
NoNaMe Chat
Your userID:
Type message here...

cb24a8997c

cb24a8997c, mj23, c670318405

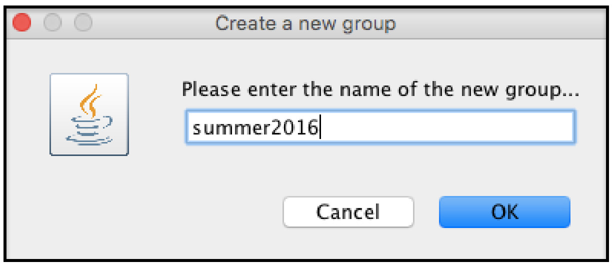
exams prep
party 3/6/16
basketball
birthday

```
cb24a8997c     : Hey, exams finally over! You are both invited to my bithday on 15/06 :-)
c670318405     : Sure, let's have a lecture notes BBQ :-D
mj23           : Done!
```

Figure 3.19: Interface layout and functionality.

Most modern messengers seem to have relatively similar interface layouts, so there seems to be certain consensus in the field regarding what can be considered intuitive. The GUI was designed to be similar to that of other messaging applications (*e.g.* Viber, Telegram), to make sure it is familiar to anyone who has ever used messengers before.

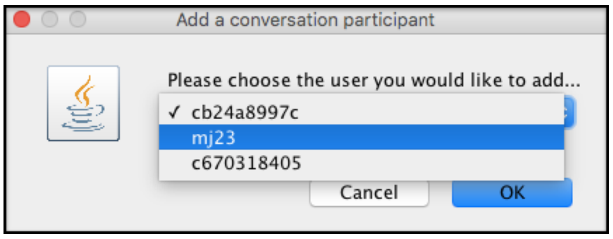
In terms of design, the interface is based on a single screen with functionality for creating new groups and adding users implemented as pop-up windows (Figure 3.20 and Figure 3.21).



Please enter the name of the new group...

summer2016

Figure 3.20: Pop-up window for creating a new group.



Please choose the user you would like to add...

✓ cb24a8997c
mj23
c670318405

Figure 3.21: Pop-up window for add a new user to the current group.

The interface is based on the Swing framework, and was constructed using the visual design tool built into IntelliJ IDEA. Most of the GUI logic is contained in the `GUIClient` class (Figure 3.22).

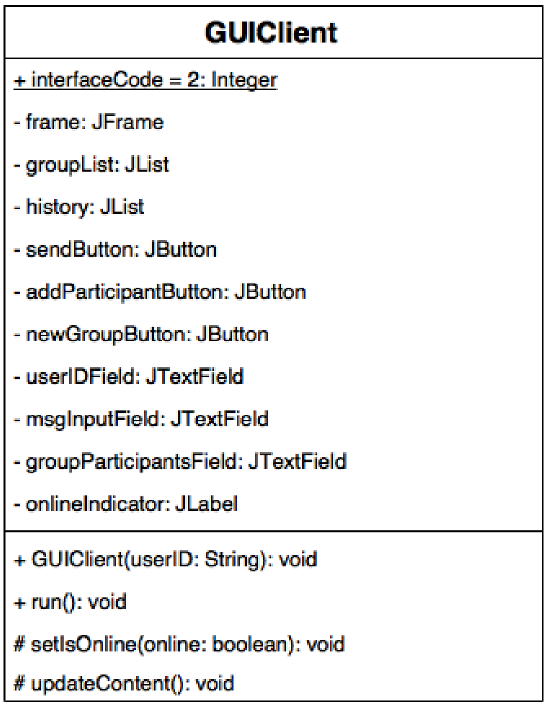

GUIClient
+ interfaceCode = 2: Integer
- frame: JFrame
- groupList: JList
- history: JList
- sendButton: JButton
- addParticipantButton: JButton
- newGroupButton: JButton
- userIDField: JTextField
- msgInputField: JTextField
- groupParticipantsField: JTextField
- onlineIndicator: JLabel
+ GUIClient(userID: String): void
+ run(): void
# setIsOnline(online: boolean): void
# updateContent(): void

Figure 3.22: `GUIClient` implements the GUI logic.

In brief, the constructor of the class instantiates `JFrame` and adds callbacks to all the GUI elements contained in it (text fields, buttons *etc*), which handle user input and output. The inherited `run()` method makes the frame visible to the user, allowing them to interact with the program. The fact that `GUIClient` is a subclass of `Client` (see Figure 3.17) enables a simple communication model between the two—`GUIClient` can get the necessary information by accessing `protected` fields and methods in `Client`, whereas `Client` can talk to `GUIClient` by calling the abstract methods that `GUIClient` implements (*e.g.* `setIsOnline()`, `updateContent()`). This kind of communication between classes makes the UI very efficient—the `Client` explicitly requests an interface refresh when some significant change happens (*e.g.* a new message arrives), which requires much less computing power than busy-waiting on several data structures.

3.3 Summary

In this chapter, the reader was familiarised with how `N0NaMe` was taken from a conceptual description to a concrete implementation. In particular, in § 3.1 I described the auxiliary "scaffolding" components that were constructed before the core protocol, and then (in § 3.2) went over the series incremental steps that were taken to implement KleeQ itself. At this stage, some measures were taken to simplify the process of evaluation, whose details and findings are presented in the next chapter.

Chapter 4

Evaluation

This chapter presents the results of systematic summative evaluation of the constructed system. More specifically, it analyses its performance characteristics (§ 4.1), considers how robust the system is (§ 4.2), assesses its usability properties (§ 4.3) and evaluates its security properties (§ 4.4). Based on these results, I then determine whether the initial project requirements have been satisfied (§ 4.5).

4.1 Performance

4.1.1 Evaluation framework

Discussing performance of a system requires a systematic evaluation framework. In particular, it is necessary to decide which *performance characteristics* are of interest, and then determine a set of *input parameters* whose variations can affect them.

In the case of messaging applications, the key performance characteristics include:

Messaging latency
: How long it takes for a new message to reach all clique members. The significance of this parameter varies with the type of communication (*e.g.* not very important for email, but very significant for instant messaging).

Traffic consumption
: How much traffic (uplink and downlink) is generated by the application per unit time, on average. This characteristic can have an impact on broadband bills and energy consumption, both of which are of particular relevance to mobile devices.

CPU usage
: How much CPU resources, on average, the application needs. This metric is interesting on its own, but can also be used as a proxy for energy consumption.

Memory footprint
: How much RAM, on average, the application takes up. Just as CPU usage, this property is important from the multitasking viewpoint, and is of particular significance in mobile and embedded devices.

In the case of `N0NaMe`, the performance characteristics above largely depend on the following 2 system parameters:

Number of clique participants
: `N0NaMe` is a peer-to-peer solution, so a key concern is its *scalability* (*i.e.* how its performance is affected as the number of communicating parties increases).

Patching period
: As described in § 3.2.2, every `N0NaMe` instance regularly emits a burst of patch requests. The time spacing between patches clearly affects the performance, and is therefore worth investigating.

Table 4.1 summarises the evaluation framework chosen as part of this project.

Input parameters	Performance Characteristics
• Number of clique members	• Messaging latency
• Patching period	• Traffic consumption
	• CPU usage
	• Memory footprint

Table 4.1: Summary of the performance evaluation framework

The effects of variations in the input parameters on the performance characteristics above, as well some key trade-offs, are presented in § 4.1.3.

4.1.2 Testing automation

`N0NaMe` is a complex distributed system which contains multiple sources of uncertainty, including but not limited to:

- pseudo-random behaviour as part of the design (§ 3.2.2, § 3.2.3);
- inherently unpredictable network conditions (*long-range dependence*);
- random variations in server-side system load due to shared resources;
- uncertain behaviour of the JVM garbage collector.

Given such an environment, making reasonably accurate estimates of the performance characteristics requires averaging over a large number of repeated measurements. Whilst possible to do manually, these observations can be made much more comfortably and reliably by automated means.

This was achieved by writing Python scripts capable of launching multiple instances of `N0NaMe` and manipulating them in different ways, meanwhile performing measurements of the relevant parameters. Building on the Machine Interface described in § 3.2.5, a simple Python class was written to permit full range of automated control of a `N0NaMe` instance:

```python
class NonameInstance:

    def __init__(self, userID=None, patch_period=3000):
        '''Run a new NONaMe instance with the given userID and patching period'''
        if userID is None:
            userID = ''.join(random.choice('0123456789abcdef') for i in range(10))

        self.userID = userID
        self.patch_period = patch_period
        self.proc = subprocess.Popen(["java", "-jar", "../part2proj.jar", "-m", str(patch_period),
                                      userID], stdin=subprocess.PIPE, stdout=subprocess.PIPE)

    def __command(self, cmd):
        '''Issue a command to the instance. Used by other methods in class.'''
        self.proc.stdin.write((cmd + "\n").encode("UTF-8"))
        self.proc.stdin.flush()
        return self.proc.stdout.readline().decode("UTF-8").strip()

    def groupList(self):
        '''Returns the list of all groups this instance is a member of.'''
        return re.split("\s+", self.__command("groups"))

    def peerList(self):
        '''Returns a list of all NoNaMe users currently online'''
        return re.split("\s+", self.__command("peers"))

    def isOnline(self):
        '''Checks if this instance is online'''
        raw = self.__command("status")
        return (raw == "online")

    def create(self, groupName):
        '''Create a group with a given name'''
        raw = self.__command("create %s" % (groupName))
        return (raw == "ACK")

    def memberList(self, groupName):
        '''Returns a list of all members of a given group'''
        return re.split("\s+",self.__command("members %s" % (groupName)))

    def add(self, userID, groupName):
        '''Adds a user to the specified group and blocks until user replies'''
        raw = self.__command("add %s %s" % (userID, groupName))
        if raw == "ACK":
            # wait for DH to go through
            while(userID not in self.memberList(groupName)):
                sleep(0.3)
            return True
        else:
            return False

    def sendMessage(self, groupName, txt):
        '''Sends a message to the specified group'''
        raw = self.__command("msg %s %s" % (groupName, txt))
        return (raw == "ACK")

    def getHistory(self, groupName):
        '''Returns list if (author, message_text) tuples in chronological order'''
        return json.loads(self.__command("history %s" % (groupName)))

    def exit(self):
        '''Shuts down the NoNaMe instance'''
        raw = self.__command("exit")
        return (raw == "ACK")
```

The class above launches an instance of `N0NaMe` when it is constructed, and interacts with it via the standard streams (`stdin` and `stdout`). The evaluation scripts use the `NonameInstance` class to play out different usage scenarios and measure the performance characteristics of interest.

4.1.3 Results and discussion

This section presents the findings for each of the performance characteristics in § 4.1.1, and discusses some of the identified trade-offs and patters.

Message latency

In theory, when one of the clique members authors a message, the time it takes to get delivered it to everyone in the clique (message latency L) can be modelled as:

$$L = P + 2 \cdot T + Q \tag{4.1}$$

where:

- P – time between message is authored and the *last* patch request for it is sent
- T – time it takes the patch request to reach the destination
- Q – time the patch request is queueing at the destination, plus processing time

Figure 4.1 shows each of these times on a timing diagram.

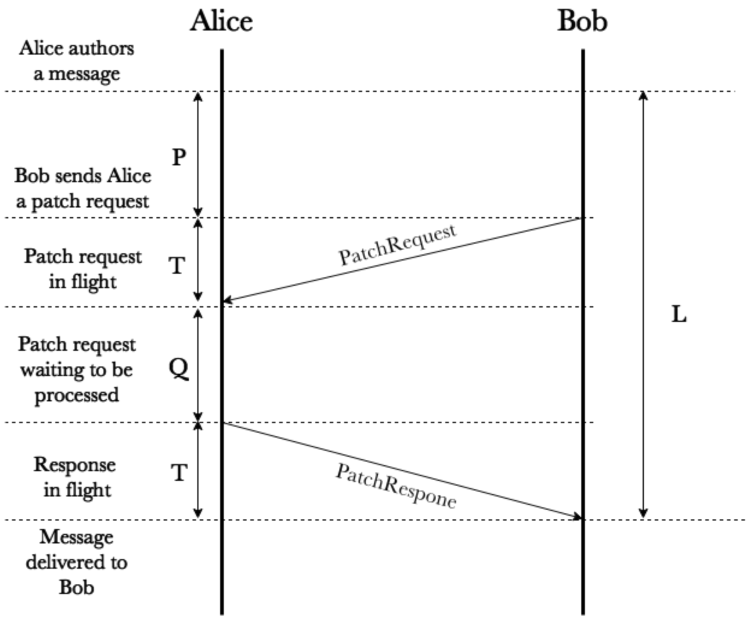


Figure 4.1: Breakdown of the message latency.

To empirically find out how quickly messages are delivered (latency) in cliques of different sizes, a specialised evaluation script has been written. It starts by launching a *leader instance* which creates a clique. Then, new `N0NaMe` instances are created and added to the clique iteratively, thereby increasing the number of clique members (N). At each iteration (*i.e.* for each value of N), the delay is measured—a clique member, chosen uniformly at random, authors a message and the algorithm measures the time it takes for it to propagate to all other instances. For every N, the latency is estimated as the average of $5N$ such measurements (*i.e.* every clique member authors roughly 5 messages). Figure 4.3 shows the results of this experiment.

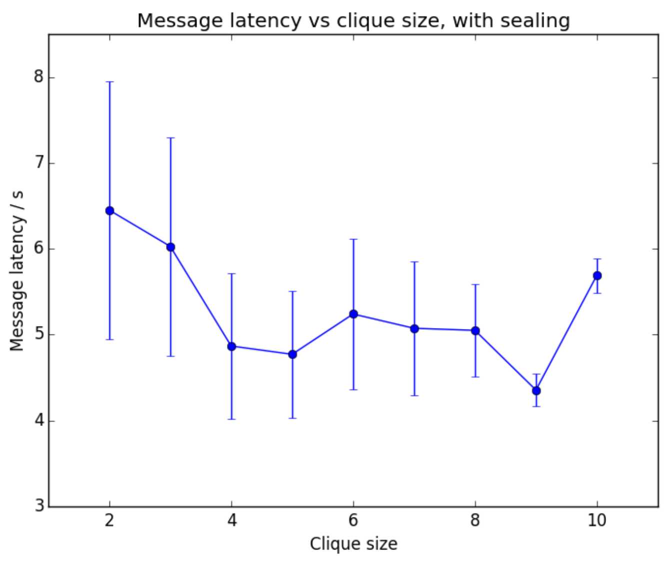


Figure 4.2: Messaging latency as a function of clique size, with patching every 3s. Authors chosen at random. 95% confidence intervals.

At the first glance, the results appear paradoxical and seem to disobey the previously suggested model. Messages take longer to propagate in a clique of size 2, than in clique of size 9! Moreover, the error is larger for smaller cliques. Upon some deliberation, however, the reason becomes obvious—since authors are chosen *at random*, smaller cliques end up performing much more sealing than larger ones (sealable blocks occur more frequently). The abundance of sealing traffic results in patch requests waiting to be processed for longer (Q in Equation 4.1 is larger) which means higher message latency.

To confirm the validity of this hypothesis, the evaluation script was amended so as to completely eliminate sealing. As before, the clique was created and gradually expanded by the leader instance. However, for each clique size, only the newly added instance was now authoring messages, with the average of 30 delay measurements taken as representative of clique latency. Figure 4.3 (in the next page) presents the resulting pattern.

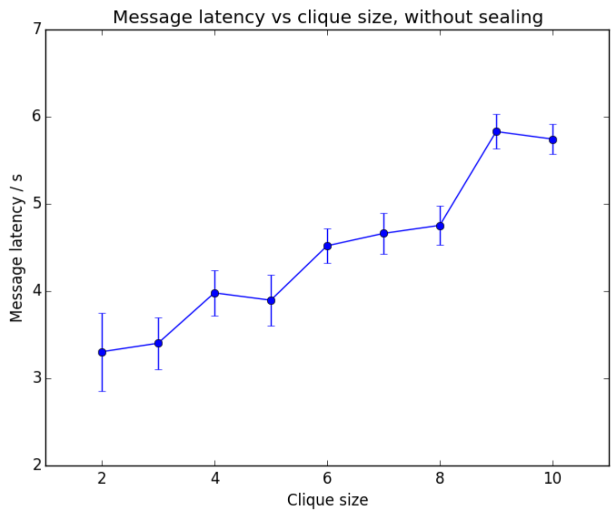


Figure 4.3: Messaging latency as a function of clique size, with patching every 3s. No sealing happens. 95% confidence intervals.

The data suggests that the latency now increases linearly with the clique size, which is in line with the suggested latency model (Equation 4.1). To understand why this is so, remember that the application handles all incoming traffic sequentially—arriving messages form a queue, where the waiting time Q is proportional to the long-term rate of incoming traffic (by Little's theorem [8]). The latter grows linearly with clique size (as will be empirically shown in the next section), therefore so does the processing time Q for a patch request and, consequently, message latency L.

From the previously described delay model (Equation 4.1) it follows that latency also grows linearly with the *patching period*. Figure 4.4 shows how this dependency looks in practice for different clique sizes, with sealing eliminated as before.

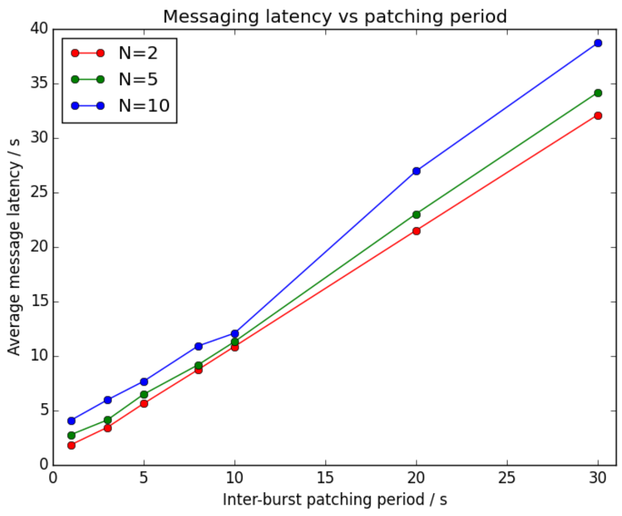


Figure 4.4: Messaging latency as a function of patching period, for different clique sizes. No sealing happens.

The figure clearly suggests a linear pattern, with larger cliques having a higher "DC offset" due to message queueing. This exactly fits the predictions of the theoretical model (Equation 4.1).

Traffic consumption

When estimating the amount of traffic consumed by the application, the following assumptions were made:

Symmetry
It is assumed that, on average, `N0NaMe` sends roughly as much traffic as it receives. Given the symmetry properties of both patching (§ 3.2.2) and sealing (§ 3.2.2), this assumption is deemed reasonably accurate.

Zero-cost transport
Size of transport headers (*e.g.* HTTP headers) is negligible compared to the size of transmitted payload (*i.e.* JSON representations of message classes). Not necessarily accurate for the Store-and-Forward-based implementation, but permits better estimate of intrinsic traffic consumption of KleeQ.

Based on these assumptions, the total traffic used up by a `N0NaMe` instance can be estimated as the amount of data it *receives* through the SaF. To access this information the server-side Python code was slightly modified to record the number of bytes a user downloads every time they empty their "postbox", and the time of download.

Since each user in a clique of size n communicates with $n - 1$ peers, it is intuitive to expect the *total* traffic flowing in the system to grow quadratically with n. The empirical data fits this expectation quite well (Figure 4.5).

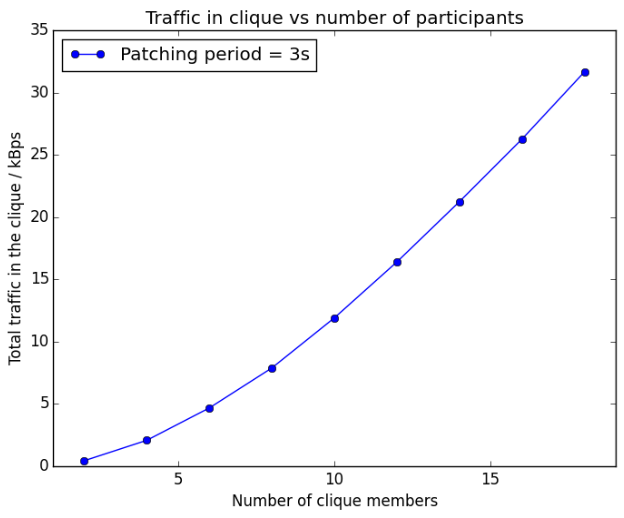


Figure 4.5: Total bandwidth usage as a function of clique size.

The result above was obtained by iteratively forming cliques of increasing size, and making randomly chosen clique members author messages every 3 to 6 seconds. For a clique of size N, the total system bandwidth usage was estimated as the average over time for a conversation of length $5N$ (*i.e.* each member authors 5 messages on average).

Since the experimental setup results in approximately symmetric operation of all instances, the amount of traffic consumed by each *individual* user can be estimated just by dividing the total average consumption by the clique size (Figure 4.6).

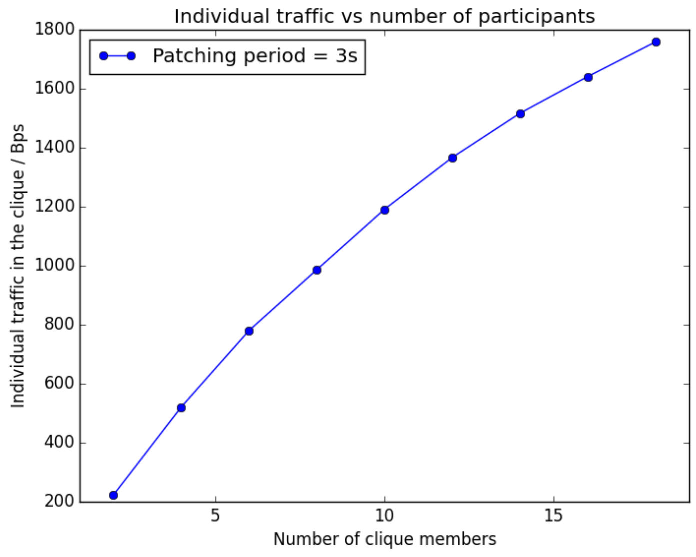


Figure 4.6: Individual bandwidth usage as a function of clique size.

From previously explained intuition, a linear pattern is expected here. In reality, however, the bandwidth usage grows sublinearly, which is particularly apparent for larger clique sizes. This can be explained by the fact the the response times of the Store-and-Forward service get longer as the clique gets larger, thereby slowing down the transmission.

It would be reasonable to hypothesise that traffic consumption is inversely proportional to the patching period (*i.e.* directly proportional to patching frequency). Figure 4.7 shows the empirically obtained data.

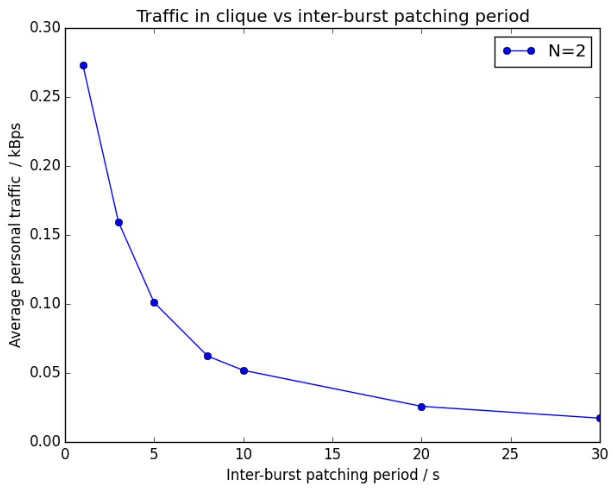


Figure 4.7: Personal traffic vs patching period.

The data above was obtained by recording the time-averaged traffic values for cliques of size 2 with different patching periods. The resulting pattern is in line with the inverse proportionality hypothesis.

CPU usage

CPU usage for different clique sizes and patching periods was measured using the `psutil`[1] Python library which gives access to some statistics about running processes. To estimate average CPU load associated with running `N0NaMe` for a given clique size and patching period, the evaluation script makes randomly chosen members author messages every 3 to 6 seconds, recording the percentage of CPU capacity[2] used over 30 patching periods and calculating the average of these measurements in the end. Figure 4.8 presents the resulting data.

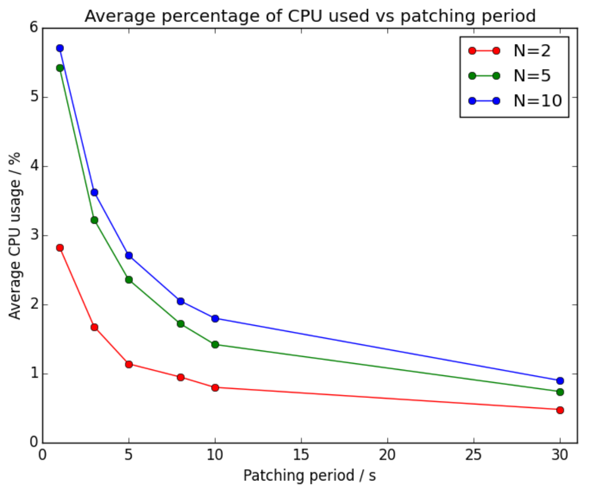


Figure 4.8: CPU usage as a function of patching period.

[1]https://pythonhosted.org/psutil/

[2]All figures were obtained on a dual-core 1.3GHz Intel I5-4250U chip with Turbo Boost up to 2.6 GHz.

As with the traffic consumption, the data suggests that the CPU usage is inversely proportional to the patching period. As one would expect, participation in larger cliques requires more computing power.

Memory footprint

Overall, the amount of memory required by N0NaMe has been found to greatly vary depending on the JVM heap parameters and garbage collection policies. The patterns of memory consumption over time, as well as for different patching periods and clique sizes, have been found very unstable due to irregularities in the work of JVM garbage collector. By altering the initial and maximum heap sizes (`-Xms` and `-Xmx` JVM options), values in the range from 60 to 250 MB were observed.

The trade-off here is that smaller heap size allows to save space, but results in more frequent garbage collection which costs CPU time and energy. On the other hand, giving the program more heap space to operate in allows to run the garbage collector less frequently and thereby save CPU cycles. In practice, the JVM garbage collection parameters need to be tuned for particular workload and usage scenario.

Key trade-offs and trends

One of the key conclusions that can be drawn from the data above is trade-off between the performance of the system and the amount of resources required.

For example, the system can achieve relatively low-latency communication by making the patching period small, but this has a cost in terms of high traffic consumption, memory requirements and CPU usage. Conversely, N0NaMe can operate consuming very little traffic, memory and power, but this results in communication not being real-time any more (Figure 4.9).

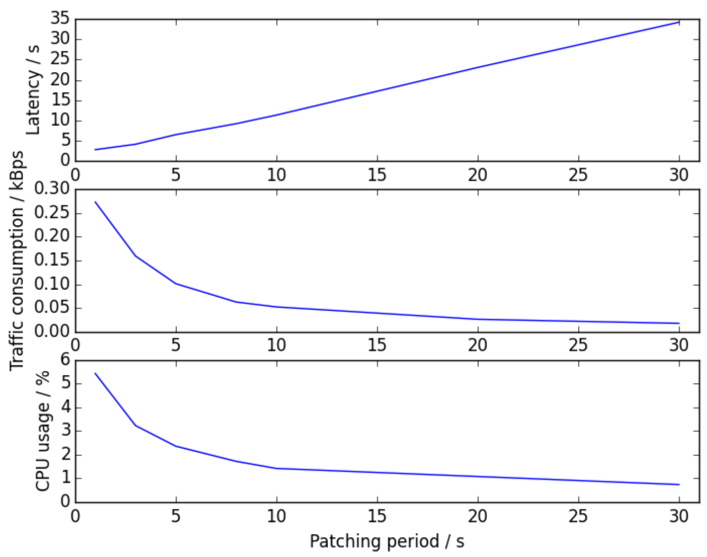


Figure 4.9: Trade-off between latency and system requirements.

The best way to tackle this problem depends on the specific setting where the application is to be used and associated priorities. If low latency is crucial and resources are readily available (*e.g.* instant messaging on desktop machines), it may be reasonable to accept the high costs and run the application with the short patching period. If, on the other hand, traffic and energy consumption are serious concern and low latency is less important (*e.g.* email on mobile devices), the patching period can be made high. Alternatively, an *adaptive* approach can be used where the patching period is dynamically adjusted – one can imagine this *e.g.* in a mobile instant messenger which communicates with low delay when open and actively used, but switches to resource-saving mode when running in background.

4.2 Robustness

This section examines `N0NaMe` from the viewpoint of stability and robustness. Of particular interest are issues of performance under high load and in the setting of transient connectivity.

4.2.1 High-load performance

Initially, the plan was to stress-test the protocol by constructing a clique of a large size out of instances with low patching period and making each of them author many messages per second. However, after carrying out the performance evaluation (§ 4.1) it became clear that the main performance bottleneck in the system is the Store-and-Forward service. Indeed, playing out the aforementioned stress-test scenario would mostly test the Store-and-Forward, rather than the intrinsic performance limitations of the protocol design. This problem arises from the somewhat naïve implementation and deployment of the server components. The Address Book and Store-and-Forward service are implemented as a single Python application which is deployed on the Flask's built-in development server platform, without any redundancy and load-balancing. Such a construction is acceptable for the purposes of building and running KleeQ, but cannot support high enough load to stress-test it. This problem could be partially fixed by implementing the Address Book and SaF as two separate applications and deploying them on a commercial-grade HTTP server platform (*e.g.* Apache or NGINX).

4.2.2 Transient connectivity

As mentioned in § 3.1.3, the functionality for handling the cases when the application goes offline due to connectivity issues was implemented as part of the address-reporting thread, with the client considering itself offline when it is unable to reach the Address Book. When the client is offline, it does not attempt sending out patch requests. Given this setup, how quickly a client can "catch up" with the rest of the clique after they go offline depends on how frequently a connection to the Address Book is attempted. This is another example of the the previously mentioned trade-off between performance and resource requirements—busy-waiting on the Address Book costs a lot of energy but allows to come back from network glitches quickly, whereas making infrequent attempts to reconnect saves resources but makes reconnection times longer.

In practice connectivity issues are often more complex than just absence of connection to the whole of Internet. Individual transmissions can be intercepted, delayed or lost, and it takes some careful planning to handle these situations. As noted before, `N0NaMe` only uses reliable transport protocols (TCP and HTTP), but the Store-and-Forward cannot be considered reliable (*e.g.* it can reboot or be accessed by a malicious adversary), so it is worth analysing different parts of the implementation for resilience to unreliable transport.

The implementation of both patching (§ 3.2.2) and sealing (§ 3.2.3) contains a timeout and re-transmission mechanism that makes them insensitive to loss of communications. The implementation of the clique formation scheme, however, does not re-transmit lost messages, meaning that network instability can result in disruption of the clique formation process. This problem has not been fixed as part of this project due to time constraints, but it would be useful to do it in the future.

4.3 Usability

One of the reasons why `N0NaMe` was given a graphical interface (GUI) was the aspiration to make it easier to use. To find out whether this goal has been achieved, the usability of the GUI was compared with that of the command-line interface (CLI) using the Keystroke-level Model (KLM)—one of the key analytic summative evaluation techniques in the field of Human-Computer Interaction.

In brief, KLM is a way to estimate how much time an average user needs to perform a routine task when using a given interface. This is done by encoding each of the actions that are needed to complete a task as one of the standard *operators*. Each operator in the model has a fixed time cost (Table 4.2), and total time to complete a task can be estimated as the sum of costs of its constituent operators.

Operator	Description	Time cost (sec)
K	keystroke or button press	≈ 0.28[3]
P	pointing at something with a mouse	1.1
H	homing hands on keyboard or mouse	0.4
M	mentally preparing for action	1.35

Table 4.2: KLM operators and associated time costs [9, 10].

For instance, typing the command `add pavel supergroup` and hitting Enter can be encoded as the sequence of operators M, H, 4K, 5K, K, 10K, K or just M, H, 20K, and

[3] estimate for "average non-secretary typist" [9]

therefore has the cost $1.35 + 0.4 + 20 \cdot 0.28 = 7.35$ seconds.

The standard tasks that one may want to perform using the system correspond to each of the CLI commands, described in § 3.1.4. Table 4.3 shows how these tasks can be encoded as KLM operators in CLI and GUI.

	CLI operators	GUI operators
Create a clique	M, H, 6K, $\|C\|$·K, K	M, H, P, H, $\|C\|$·K, K
Add user to clique	M, H, 4K, $\|U\|$·K, K, $\|C\|$·K, K	M, H, 3P, H, K
View history in clique	M, H, 5K, $\|C\|$·K, K	M, H, P
Sends text message	M, H, 4K, $\|M\|$·K, K	M, H, $\|M\|$·K, K
Exit `N0NaMe`	M, H, 5K	M, H, P

Table 4.3: Task encodings in KLM.

where $|C|$, $|U|$ and $|M|$ are length of clique name, user name and message respectively. Table 4.4 shows the resulting costs for each task for the two interfaces.

	CLI time cost (sec)	GUI time cost (sec)
Create a clique	$3.71 + 0.28 \cdot \|C\|$	$3.53 + 0.28 \cdot \|C\|$
Add user to clique	$3.43 + 0.28 \cdot (\|U\| + \|C\|)$	5.73
View history in clique	$3.43 + 0.28 \cdot \|C\|$	2.85
Sends text message	$3.15 + 0.28 \cdot \|M\|$	$2.03 + 0.28 \cdot \|M\|$
Exit `N0NaMe`	3.15	2.75

Table 4.4: Costs of tasks in `N0NaMe`.

From the data above it is easy to see that, on average, the GUI helps perform each of the tasks faster as compared to the CLI. It is important to remember, however, that KLM estimates are only valid for *expert* users and do not give any indication of how long it takes to become one (no account of learning time). Moreover, KLM estimates are based on error-free execution of action sequences which often cannot be a valid assumption. Nevertheless, it would be reasonable to assume that the CLI takes longer to learn and easier to make mistakes in, so it can be tentatively concluded that the GUI achieves the goal of making `N0NaMe` more usable.

4.4 Security

This section analyses the security of `N0NaMe`. In particular, it explains how the security properties of the KleeQ protocol are delivered by the application (§ 4.4.1), and describes some of its limitations (§ 4.4.2).

4.4.1 Security properties

As mentioned in § 2.2, KleeQ provides a range of security guarantees. Let us go over each of them and evaluate the extent to which the implementation manages to deliver them.

Confidentiality of conversation

This property is achieved by using symmetric-key cryptography, as per KleeQ's design. Encryption is performed using the industry-standard AES block cipher in CTR mode, with a key of length 128-bits derived from a shared secret established via Diffie-Hellman key exchange (§ 3.2.1).

Communication integrity

Integrity of individual messages is assured using message authentication codes (MACs), based on keying the secure hash function SHA-256 (§ 3.2.1). Integrity of the whole conversation is verified as part of the sealing process (§ 3.2.3).

Forward secrecy

Forward secrecy is achieved by regularly rotating keys (§ 2.1.5) and deleting sealed blocks. Because of how keys are rotated, a compromised key gives the adversary the ability to eavesdrop on messages in the current block but does not allow them to read the *previous* messages.

Backward secrecy

The key rotation scheme also helps ensure that a compromised key does not allow the adversary to listen to *future* communications. Given the current key and therefore the ability to read messages within the current block, computing the next key also requires the knowledge of the current clique secret.

Authorship and participation repudiation

As per the original design, messages are encrypted and authenticated only using the keys derived as part of the conversation (no long-term personal keys are used). This means that, given access to the conversation transcript as well as all the cryptographic keys, one has no evidence that a particular person authored a given message. Similarly, there is no way to show that a particular person participated in the conversation.

4.4.2 Limitations

As mentioned before, by implementing KleeQ, `N0NaMe` only solves the problem of *conversation security* (§ 1.1)—it protects the *content* of the conversation from the adversary, preventing eavesdropping and injection of fake messages. It does not attempt to protect the user from impersonation or transport metadata-related attacks, that need to be addressed separately.

For example, by analysing the Store-and-Forward traffic one can fairly accurately determine the identities of conversation participants, thereby breaking the repudiation properties. This attack relies on analysing transport metadata (*e.g.* IP-addresses) whose protection belongs to the realm of *transport privacy* (§ 1.1) and is therefore outside the scope of this project.

Similarly, an adversary can gain full access to the conversation transcript and even author new messages by performing a man-in-the-middle attack when a new user is added to the clique. This is an impersonation attack, whose prevention is the responsibility of *trust establishment* (§ 1.1) mechanisms which are not considered as part of this project.

4.5 Satisfaction of requirements

The project has been able to satisfy each of its requirements (§ 2.2). In particular:

- KleeQ has been implemented according to the specification, with all gaps in the description carefully filled. All the original security guarantees have been preserved.
- A working messaging application based on KleeQ has been developed and tested. The application has some good performance characteristics, such as scalability, relatively low latency, low traffic consumption *etc*. For improved usability, a graphical interface has been produced.

Based on these facts, the project has been a success.

4.6 Summary

This chapter has presented the results of the evaluation process. In particular, the issues of performance (§ 4.1), robustness (§ 4.2), usability (§ 4.3) and security (§ 4.4) have been discussed in detail. Not only did the project achieve all of its initial goals, but has also demonstrated some very impressive evaluation results. Its accomplishments, as well as some ideas for further work, are discussed in more detail in the next chapter.

Chapter 5

Conclusions

5.1 Accomplishments

Overall, the project has been successful in achieving each of its goals. The KleeQ protocol has been implemented, in full accordance with the original description [2]. Although the paper looked complete and example code existed, numerous gaps had to be filled out along the way, taking special care to preserve the security guarantees provided by the original design, keeping in mind the issues of performance and resilience to network instability. Each of these challenges has been solved successfully.

Based on the protocol's implementation as well as some additional auxiliary components, a prototype of a messaging application, `N0NaMe`, has been constructed. The messenger has been thoroughly evaluated for performance, security and usability. In terms of performance, `N0NaMe` achieves some very impressive characteristics (§ 4.1), suggesting that KleeQ is a potentially promising solution for ubiquitous secure messaging. The cryptographic primitives used by `N0NaMe` have been carefully chosen to avoid implementation-level vulnerabilities, and deliver the security properties of KleeQ. In an effort to make the application more usable, it has been given a graphical interface which has been shown to help perform routine tasks faster.

The project has greatly benefitted from comprehensive and methodical planning at the initial stage. Not only did it make the implementation process more organised and systematic, but also helped achieve elegance, modularity and extensibility in system design.

5.2 Further work

As mentioned before, the main goal of this project was to implement and evaluate KleeQ. Whilst this goal has been successfully achieved, there are many other promising directions of work that would be useful to look at:

- *Distributed contact discovery.* The server-based Address Book is one way to solve the problem of contact discovery. To make the messenger completely distributed and eliminate the single point of failure, it needs to be replaced with a peer-to-peer mechanism. A viable solution would be to use a distributed hash table (DHT), as done by other P2P systems (*e.g.* BitTorrent).

- *Transport privacy.* At present `N0NaMe` does not attempt to hide conversation metadata (*e.g.* sender, recipient, time of transmission *etc*), and the adversary can obtain some information by analysing the Store-and-Forward traffic. One way to mitigate this problem would be to implement SaF as a *Tor hidden service* [11], which would ensure anonymity of conversation participants. In practice, however, using Tor will incur additional delay, so more performance evaluation would be necessary.

- *Trust Establishment.* `N0NaMe` would benefit from a routine for peer identity verification which would protect the user from impersonation attacks. One promising new trust establishment technique is the *self-auditable transparency log* [12], which is an authority-based trust mechanism that prevents MitM-attacks by the network and permits detection of MitM-attacks by the trust authority. A more secure but less user-friendly way to do it would be to use one of the manual key verification techniques, as done by OTR[1] and Threema[2].

- *Further performance improvements*. Whilst acceptable already, `N0NaMe`'s performance could be further enhanced. For example, an effort can be made to reduce the amount of memory it consumes, as well as replace JSON with a more compact custom-designed message format in order to generate less traffic. Given the iterative nature of patching and sealing, it would be interesting to investigate the possibility to communicate over the more lightweight UDP protocol.

5.3 Final remarks

The emergence of new types of cyber-threats in the recent years has brought secure messaging into the spotlight of academic research and industrial effort. This project makes a practical contribution by implementing and evaluating KleeQ—an existing conversation security protocol that possesses some promising security properties and, as demonstrated as part of this project, some extremely impressive performance characteristics as well.

The project has fully achieved its goals. Despite the numerous gaps left in the original description by its authors, the protocol has been fully implemented and tested. This work was later used to construct an easy-to-use messaging application, whose impressive performance highlights KleeQ's great potential as a building block for future generations of secure messengers.

[1]https://otr.cypherpunks.ca/
[2]https://threema.ch/en

Bibliography

[1] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. Sok: Secure messaging. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 232–249. IEEE, 2015.

[2] Joel Reardon, Alan Kligman, Brian Agala, and Ian Goldberg. KleeQ: Asynchronous key management for dynamic ad-hoc networks. *University of Waterloo, Tech. Rep*, 2007.

[3] Whitfield Diffie and Martin E Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[4] Markus Kuhn. Security II: Cryptography, 2016. University of Cambridge Part II CST course notes.

[5] Tero Kivinen. More modular exponential (MODP) diffie-hellman groups for internet key exchange (IKE). 2003.

[6] Kryptographische verfahren: Empfehlungen und schlüssellangen. *Technische Richtlinie TR-02102-1, Bundesamt fur Sicherheit in der Informationstechnik (BSI)*, 2016.

[7] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. Analysis of sha-512/224 and sha-512/256. In *Advances in Cryptology–ASIACRYPT 2015*, pages 612–630. Springer, 2014.

[8] John DC Little. A proof for the queuing formula: L= λ w. *Operations research*, 9(3):383–387, 1961.

[9] Stuart K Card, Thomas P Moran, and Allen Newell. The keystroke-level model for user performance time with interactive systems. *Communications of the ACM*, 23(7):396–410, 1980.

[10] Jeff Sauro. Estimating productivity: Composite operators for keystroke level modeling. In *Human-Computer Interaction. New Trends*, pages 352–361. Springer, 2009.

[11] Steven Murdoch. Anonymity and censorship resistance, 2016. University of Cambridge Part II CST course notes.

[12] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Michael J Freedman, and Edward W Felten. Coniks: A privacy-preserving consistent key service for secure end-to-end communication. *IACR Cryptology ePrint Archive*, 2014:1004, 2014.

Appendices

Appendix A

Project Proposal

A.1 Introduction and Description of the Work

The recent public outcry in response to mass surveillance programs carried out by governments around the world led to creation of multiple messaging systems which emphasised security of communication. Some have been more successful than others, but each of them has been found to have some flaws or deficiencies. Broadly speaking, there are four problems that each security-oriented messenger needs to solve:

Problem 1: Contact Discovery
How do we find out at which IP addresses our peers currently reside? How do we know where to send our messages?

Problem 2: Trust Establishment
Once we know where our peers are, how do we know that they are who they say they are? How do we make sure that they are not being impersonated by a malicious adversary?

Problem 3: Conversation Security
Once we are sure that we are talking to the right parties, how do we protect the security and privacy of the messages' content? In other words, how do we encrypt the messages, what data do we attach to them, and what security protocols do we perform?

Problem 4: Transport Privacy
What is the mechanics for actually sending the message so as to hide the message metadata (*e.g.* sender identity, recipient identity, conversation to which the message belongs *etc*).

Multiple protocols, with different threat definitions and varying levels of security, have been developed for each of these tasks. The aim of this project is to explore and build on KleeQ, one of the protocols aimed at providing conversation security (Problem 3 in the list above). Given a group of trusted parties residing at known addresses, it gives the following security guarantees for their conversation:

- confidentiality of message content

- message integrity
- forward secrecy
- backward secrecy
- message authorship repudiation
- conversation participation repudiation

KleeQ has been designed for devices with transient connectivity (*e.g.* communication via Bluetooth or wireless), but the ideas introduced by this protocol can be used in the general setting of group messaging over a network.

In this project, KleeQ will be completely re-implemented into a general-purpose network conversation security protocol, preserving the security properties mentioned above. The performance characteristics will be tested, and the use of the protocol will be demonstrated by constructing a simple demo messaging app.

Optionally, if time allows, an attempt shall be made to make use of the most recent and secure third-party solutions to solve Problems 1, 2 and 4 from the list above, thereby producing a full-blown secure messaging application.

A.2 Resources Required

All the development will be based on the resources provided by my own machine and the PWF. In addition, it is expected that a commercial cloud hosting service (DigitalOcean, Heroku or similar) will be used, for the purpose of hosting the server side of the application. The backup strategy involves periodically pushing the code to a private GitHub repository and keeping a hand-written project log. No other resources will be required.

A.3 Starting Point

As of the starting date of the project, the following relevant background:

- Experience of programming in Java, at the level of University courses *Programming in Java* and *Further Java*.
- Understanding of cryptographic primitives and computer security fundamentals, as taught in Part IB *Security I* course.
- Limited experience of writing server-side code in Python using Flask.

The successful completion of the core part of the project will require the following:

- Learning about the fundamentals of conversation security, understanding what security properties it entails and what the common attack strategies are.

- Understanding the algorithms and data structures introduced by KleeQ.
- Learning to write security-oriented code in Java.

To complete the optional part, the following will need to be done:

- Understanding the most common attack strategies employed by adversaries against secure messengers.
- Reading research papers and technical documentation describing the APIs presented by the third-party libraries in use.

A.4 Substance and Structure of the Project

A.4.1 Core Part

The core part of the project will involve re-implementing KleeQ, to make it a universal conversation security protocol. Initially, it will be necessary to obtain a deep understanding of how KleeQ operates which will be done by reading the relevant research material, as well as studying the source code of the current implementation. Then, the parts of the protocol which require re-designing to allow network communication (as opposed to ad-hoc communication via Bluetooth or similar) will be identified, and the necessary design decisions will be made. The next step will be implementing the protocol in Java and testing it locally. At that point, additional checks will be made to ensure that the new implementation retains all the security properties of the original one and, if necessary, eliminate the possible loopholes in the code.

The next step will be turning the result of the above into a simple desktop P2P messaging application. It is important to understand that producing a full-blown secure messaging system is outside the scope of the core part of this project—at this stage, the aim will be to write some simple and not necessarily secure "scaffolding" code to produce a prototype, for the purposes of demonstrating the work of the new protocol implementation. In particular, it is expected that a minimalistic graphical user interface and a simple server-based contact discovery component will be implemented at this point.

A.4.2 Optional Extensions

If time allows, the most recent and secure approaches may be used to convert the aforementioned demo application into a proper secure messaging system. As mentioned in Section A.1, this would require solving the problems of secure contact discovery, trust establishment and transport privacy. These problems can be solved in the following ways:

- DHT with query anonymisation for contact discovery
- self-auditable public key logs for trust establishment (using CONIKS)

- Tor-based hidden service for transport privacy and metadata hiding

All of the above are available in the form of open-source libraries which can be used in the project without major modifications.

In addition, an effort may be made to improve the usability of the application. With this in mind, the GUI will be refined to match the usability of the most ubiquitous messenger applications (*e.g.* WhatsApp, Viber, Telegram *etc*).

A.5 Success Criteria

There shall be two measures of success for this project:

1. Implementation of a working messaging system which would be possible to use.
2. Preservation of the security properties of the original KleeQ implementation, namely:
    - confidentiality of message content
    - message integrity
    - forward secrecy
    - backward secrecy
    - message authorship repudiation
    - conversation participation repudiation

A.6 Timetable and Milestones

It is important to structure the project in a way which would allow to evenly distribute the workload over the available time and minimise risks of missing deadlines. In addition, as pointed out by previous Part II students, it would be beneficial to finish the dissertation write-up at least two weeks in advance of the official deadline, to allow more time for Tripos revision. With this in mind, the following schedule has been set:

Weeks 1-2 (Oct 25 – Nov 6)

Do preliminary reading and understand the algorithms and data structures used by KleeQ, referring to the existing implementation as necessary. Identify the parts of the protocol which require modification, and design the object-oriented structure of the new implementation. Decide on which standard Java classes will need to be used. Find out what common implementation mistakes result is security loopholes. Arrange the necessary infrastructure, such as a back-up repository and cloud hosting space.

Milestone: The necessary knowledge of the protocol acquired, design and infrastructure ready – can begin writing code.

Weeks 3-7 (Nov 7 – Dec 11)

Implement the protocol in Java. Test locally. Watch out for most common implementation vulnerabilities. Make sure the original security guarantees are in place.

Milestone: The conversation security protocol implemented and tested.

Weeks 9-10 (Dec 12 – Dec 25)

Turn the protocol implementation into a library which would be usable by third parties in their applications.

Week 11 (Dec 26 – Jan 1)

A week-long holiday is planned for these dates. No work will be done during this time.

Weeks 12-13 (Jan 2 – Jan 15)

Implement the "address book" server application to allow contact discovery. Make sure the server has the most current information about user's status. Test the practicality of KleeQ's conversation concepts.

Week 14 (Jan 16 – Jan 22)

Write-up the progress report and submit it one week ahead of the deadline.

Weeks 15-16 (Jan 23 – Feb 5)

Create a minimalistic GUI. Make sure that the cases of asynchrony (device going offline or coming back online) are adequately handled.

Milestone: An operational messaging application is ready. Core part of the project finished, the success criteria satisfied.

Weeks 17-19 (Feb 6 – Feb 26)

Evaluate how the protocol performs over a network under high load, and try to tune it to perform better. Refine the GUI of the prototype to allow usage by non-experts.

Milestone: The application looks good and the performance is satisfactory.

Weeks 20-21 (Feb 27 – Mar 11)

Implement the optional extensions as time allows. Make sure that none of the previously implemented security guarantees are broken.

Week 22 (Mar 12 – Mar 18)

A week-long holiday is planned for these dates. No work will be done during this time.

Weeks 23-27 (Mar 19 – Apr 22)

Write up the dissertation. Use the project log to recollect the details of what work has been done and how. Pay special attention to the Evaluation section, describing in detail the previously made performance measurements. Arrange regular meetings with the supervisor to receive feedback and iteratively refine the work. This part of work will take a long time, since it will be interleaved with Tripos revision over the Easter Vacation.

Milestone: Dissertation largely written. The supervisor and the overseers are satisfied with the content.

Weeks 28-29 (Apr 23 – May 13)

No project work is scheduled for these days – the intention is to use this time for Tripos revision. It also provides a "safety buffer" in case more time is required, for one reason or another. If necessary, final adjustments to the text of the dissertation will be made at this time.

Milestone: Dissertation printed, bound and submitted at least two weeks ahead of the deadline.