Master’s Degree in Software Engineering
Dissertation

Secure share of information in web groups

André Manuel Casaca Lizardo
alizardo@student.dei.uc.pt

Supervisor
Prof. Dr. Raul Barbosa

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Abstract

Nowadays, billions of people use distributed systems on the Internet, including services to make online payments, to share information and to communicate, making it crucial to guarantee privacy and even their safety. Unfortunately, most of these systems receive user information in plain text (even if the channel is encrypted) and persist that information in servers where it remains accessible and readable, therefore compromising the human right to privacy.

In this context, secure group communication through untrusted servers is the fundamental research question addressed in this thesis. This research question presents several challenges. Namely, providing confidentiality and integrity of stored information, key distribution among geographically distributed clients, ensuring availability of private information, and meeting the necessary scalability.

This thesis proposes a security protocol, named Sharelock, that provides end-to-end encryption as the means to securely communicate through untrusted servers. The protocol scales to large groups and large amounts of data by applying fast cryptographic algorithms in the client-side application. Moreover, the protocol supports diverse applications by being agnostic in relation to the information shared between the users.

An experimental evaluation of the protocol was performed, focusing on the scalability of the most frequently used and the most complex protocol primitives. Namely, persisting data, fetching data, and removing users from group (the heaviest operation with respect to the key distribution problem). The results show that the protocol can scale to large group sizes and group data volumes. Therefore, this work shows that it is feasible to have secure group communication through untrusted servers by shifting the cryptographic tasks to the client-side without compromising scalability.

Keywords: Security, Cryptography, Distributed Systems, End-to-end Encryption, Group Communication.

Acknowledgements

My gratitude to my supervisor Prof. Dr. Raúl Barbosa who always challenged me with cutting-edge technologies, prompted me to think about security problems in distributed systems and supported me during this year.

I would like to thank Prof. Dr. Filipe Araújo, Jaime Correia and Samuel Neves who shared their vast expertise with me in order to write a remarkable article related with end-to-end security in Internet of Things systems.

To my parents who always supported me during these six years and allowed me to take a master's degree in Software Engineering. Finally, last but no least, a very special thank you to my sister, my brother-in-law and my little niece for your unconditional love and support.

Acronyms

API Application Programming Interface. 2, 33, 34, 36, 38, 39

CRUD Create, Read, Update, Delete. 2

CSS Cascading Style Sheets. 40

DOS Denial of Service. 16

ER Entity-Relationship. 37

HTML Hyper Text Markup Language. 40

HTTP Hyper Text Transfer Protocol. 7

HTTPS Hyper Text Transfer Protocol Secure. 1, 7, 10, 11, 35, 39, 43

IoT Internet of Things. 2, 50

J2EE Java Platform, Enterprise Edition. 34, 35

JPA Java Persistence API. 34

JS Javascript. 40

JSON Javascript Object Notation. 12, 50

MITM Man In The Middle. 15

NIST National Institute of Standards and Technology. 11

ORM Object-Relational Mapping. 34–36

OSN Online Social Networks. 2, 9, 10

OTR Off-The-Record. 8, 10

PGP Pretty Good Privacy. 7, 10

REST Representational State Transfer. 35, 36

RESTful Representational State Transfer. 2, 34, 36, 39

TLS Transport Layer Security. 7

UML Unified Modelling Language. 33

W3C World Wide Web Consortium. 38

Contents

List of Tables

List of Diagrams

List of Figures

Chapter 1

Introduction

The Internet has grown substantially is used by billions of people around the world. During this significant growth, several distributed systems, such as Facebook, Amazon, Gmail and Paypal, started providing sensitive services on the Internet. Currently, people use services on the Internet to make online payments, to share information, to communicate, etc. Therefore, services handle sensitive information of their users, which should be kept private.

Taking into account the human right to privacy, security in services on the Internet is recognised as a fundamental problem. In order to ensure privacy, the service provider should not have the right to read and use users' information without their consent.

Nowadays, most of services on the Internet use secure channels between clients and servers, like Hyper Text Transfer Protocol Secure (HTTPS), to prevent disclosure of information from unauthorised parties. However, this strategy does not guarantee privacy at the server as the information is handled in plain text allowing the server to read, mine and update the data. Therefore, to achieve security we have to prevent accesses and modifications to information from unauthorised parties, while making it only available to authorised entities.

In order to overcome these problems, security protocols implemented mechanisms where the information is encrypted by the sender and can only be decrypted by the receiver, this is known as end-to-end encryption which is the main focus of this thesis. Moreover, end-to-end encryption follows such a mechanism using a key which is only known by the sender and the receiver. The protocols may also implement end-to-end encryption when multiple users share information, i.e., in group communications. Signal and Snake are examples of such protocols that provide end-to-end encryption through untrusted servers, in other words servers which may access and manipulate the information maliciously.

Signal supports synchronous and asynchronous group communications, availability and application independence. Firstly, synchronous communications means that the sender and the receivers can communicate in a continuous stream at a constant rate, [11]. Secondly, asynchronous communications means that both parties can communicate without being online at the same time. Thirdly, availability means that the information persisted by the users is only available for them and the server cannot access it. Lastly, application independence represents information that comes from the client-side is always processed and persisted equally, regardless of its data type.

Snake was designed for Online Social Networks (OSN) providing untraceability, anonymity and unlinkability (see Section 2). When compared with Signal, Snake supports user, group and data management through Create, Read, Update, Delete (CRUD) primitives.

Despite the increase in security granted by these two protocols could be improved. On the one hand, Signal does not provide an abstract data model nor specification of the protocol as we only know that Signal has primitives to send and receive persisted data but we do not know if Signal provides user and group management primitives as well. Signal has recently started to support application independence as when this research begun, they only supported end-to-end encryption for text messages. On the other hand, Snake does not provide application independence as it is only developed for Online Social Networks (OSN).

To overcome the aforementioned problem, we propose a security protocol, named Sharelock, which provides end-to-end encryption in asynchronous group communication, access control, application independence and database abstraction through untrusted servers as well as user, group and data management primitives. We consider end-to-end encryption as one of the thesis goals.

Sharelock shifts all cryptographic work to the client side adopting high-speed cryptographic algorithms to decrease the client processing time without compromising system security. Additionally, the protocol provides application independence which allows multiple message data types and data models. Considering application independence as one of the thesis goals.

Scalability is relevant as it refers to the capacity of the system to handle different amounts of work without compromising the whole system thus we test the protocol through a web implementation measuring the most used and the most complex primitives, and comparing the execution time of requests with payload in plain text or encrypted with Sharelock. We also consider performance and scalability as one of the thesis goals.

This thesis contributes with four artefacts. Firstly, a specification of Sharelock for any client-server architecture. Secondly, a adapatation of Sharelock protocol for Internet of Things (IoT) systems which manuscript is in preparation for a submission to a international journal. Thirdly, a web implementation of the protocol which provides a javascript library, a Representational State Transfer (RESTful) Application Programming Interface (API) and a database for fast integration. Finally, a web chat, named Hedwig, which uses the aforementioned implementation to achieve end-to-end security.

This document is divided into Chapters and Sections. Chapter 2 details the state of the art. Chapter 3 details the Protocol including sections like architecture and design, the security goals, the main features and a security evaluation. Chapter 4 details the web implementation of the protocol. Chapter 5 details the performance evaluation in order to test the most used primitives and the most complex primitive of the web implementation of the protocol. Finally, Chapter 6 details the future work and the main conclusions.

Chapter 2

State of the art

This chapter describes the current status of web security explaining the most used protocols and focusing on those related to our work. It also provides a summary which matches the analysed protocols and the following attributes, namely end-to-end encryption, confidentiality, integrity, availability, group communication and database abstraction.

2.1 Notation

Based on [10], an entity involved in a scenario is a Principal and Table 2.1 refers the familiar names used to portray these principals in scenarios.

Table 2.1: Familiar names for the protagonists in security protocols

Alice	First participant
Bob	Second participant
Carol	Participant in three- and four-party protocols
Dave	Participant in four-party protocols
Eve	Eavesdropper
Mallory	Malicious attacker
Sara	A server

In addition, Table 2.2 details the main threats classes: leakage, tampering and vandalism. The knowledge of classic attacks improves the security of new designs. Table 2.3 explains the classic attacks defined in the literature.

Moreover, interaction diagrams portray the interaction between nodes and servers over time, where the communication between them is represented by arrows symbolising requests on a secure channel. Furthermore, each arrow must contain a message, a function or an acknowledge, namely *ack*, stands for an acknowledge. $functionName(param1, param2, ...)$ for a function call with n parameters. $\{Data\}$, for a request payload. $hash(param1)$, for a hashed variable. In addition, the encryption equations follows this

Table 2.2: Security Threats Classes

Leakage	Refers to the acquisition of information by unauthorised recipients.
Tampering	Refers to the unauthorised alteration of information.
Vandalism	Refers to interference with the proper operation of a system without gain to the perpetrator.

Table 2.3: Security Classic Attacks

Eavesdropping	Obtaining copies of messages without authority.
Masquerading	Sending or receiving messages using the identity of another principal without their authority.
Message tampering	Intercepting messages and altering their contents before passing them on the intended recipient.
Replaying	Storing intercepted messages and sending them at a later date.
Denial of service	Flooding a channel or other resource with messages in order to deny access for others.

notation, namely K_s, stands for a symmetric key. K^+_{Alice}, for Alice's public key. K^-_{Alice}, for Alice's private key. M, for the information. $\{M\}_{K_s}$, for an object encrypted with a symmetric key. $\{K_s\}_{K^+_{Bob}}$, for a symmetric key encrypted with Bob's public key. $\{K^+_{Bob}\}_{pwd}$, for Bob's public key encrypted with a password. $[M]_{K^-_{Bob}}$, for signed information.

2.2 Security

In [2], security is defined by confidentiality, availability and integrity. Firstly, confidentiality means the prevention of unauthorised access to private information. Secondly, availability means that the information must be readily accessible only for authorised entities. Lastly, integrity ensures that the information remains unmodified by unauthorised entities.

The most effective way to ensure security in a distributed system is by encrypting the data and the channels. To that end, the usage of cryptographic keys grew in distributed systems. Nowadays, techniques like symmetric and asymmetric keys, digital signatures and certificates are essential to provide and achieve full security on the Internet.

According to [20], an asymmetric key is a pair of two different keys which represent a public and a private key. A public key can decrypt information that was encrypted by a private key and vice versa. Conversely, symmetric keys are not a set of keys, but only one key which is able to encrypt and decrypt information.

Signatures sign and validate documents ensuring authenticity and trustworthiness. Additionally, Digital Signatures sign information confirming the authenticity of the received information.

The certification authorities are trusted by the clients and by the servers and, their job is to certificate and verify the authenticity of these external entities. For instance, HTTPS uses certificates in order to identify the servers and, in some cases, the clients.

2.3 Protocols

This section details the most used security protocols on the Internet which ensure at least one of the following security aspects, namely integrity, confidentiality, availability, group communication and end-to-end encryption.

2.3.1 HTTPS

HTTPS [18, 19] is a security protocol which implements a secure layer on the Hyper Text Transfer Protocol (HTTP) [14]. Technically, this protocol implements a security layer, based on Transport Layer Security (TLS) [13, 12], which provides confidentiality and integrity in the communication channel between the clients' browser and the server. On the one hand, the messages are delivered in plain text to the server side, on the other hand, the channel which carries these messages is encrypted.

Most web applications on the Internet implement this protocol as their only security measure. As a consequence, they are able to read, mine and update the users' information.

2.3.2 PGP

Pretty Good Privacy (PGP) [15, 8] is a security protocol which provides asynchronous secure communications through end-to-end encryption. It was introduced to provide privacy in email conversations, achieved with confidentiality, integrity and authenticity of exchanged data by using asymmetric encryption.

When a user wants to send an encrypted email, they initially have to get the receiver's public key in order to encrypt the email. It provides confidentiality and integrity because the information can only be decrypted with the receiver private key. To speed up the process of getting the receiver's public key, there are many services which provide features to publish public keys and to get public keys from other entities.

This protocol shifts all key management work to the client who is responsible for persisting their own public and private keys. As a consequence, if they lose their keys, they have to generate a new key pair and notify their recipients. This protocol was mainly designed for point-to-point communications.

2.3.3 Off-The-Record Messaging

Off-The-Record (OTR) [6, 1] is a security protocol which provides end-to-end encryption and represents the idea of two people chatting face-to-face in a private room. This protocol provides confidentiality through symmetric key encryption.

The protocol ensures deniability, confidentiality and integrity of the shared messages. However, the protocol does not persist shared messages and does not support group communication.

Deniability means that there is no proof of the sender's identity. For instance, Bob cannot prove that he received a message from Alice. This happens because after the session key exchanging, it is impossible to prove that someone who has the session key sent any message.

2.3.4 Signal

Signal [9, 17] is a security protocol which provides end-to-end encryption. It supports synchronous and asynchronous group communication. Signal has recently started to support multiple message formats such as video, text or images.

In terms of cryptography, Signal supports ephemeral keys, future and forward secrecy of the shared messages and implements the Double-Ratchet Protocol. Firstly, ephemeral keys are symmetric keys which are individually generated for each message. Secondly, forward secrecy means that if a key is compromised, the information encrypted with previous keys from the same keychain cannot be decrypted with this key. Thirdly, future secrecy means that if a key is compromised, the future information will not be compromised either. Finally, Double-Ratchet Protocol is a key management algorithm that manages the renewal of ephemeral keys used to encrypt messages. Moreover, this protocol also prevents previous messages from being compromised by future keys.

2.3.5 MTProto

Alongside with WhatsAPP and Messenger, Telegram is a popular chat which supports end-to-end encryption in group communications by implementing the MTProto protocol. According to [16], MTProto is a symmetric encryption protocol which has some security vulnerabilities, namely the lack of *authenticated encryption* and the possibility of *chosen-ciphertext attack*. The former, *authenticated encryption* provides confidentiality, integrity and authenticity of the encrypted data. The latter, *chosen-ciphertext attack* is an attack where the attacker can infer on the information by decrypting some of its pieces.

In addition, messages are encrypted in client-side with symmetric keys and sent to the server to be relayed to the recipients. During this process, the server does not persist the message. Therefore, MTProto do not support availability of the shared information.

2.3.6 Snake

Snake [4, 3] is a security web protocol for Online Social Networks (OSN) providing end-to-end encryption along with untraceability, anonymity and unlinkability. Firstly, untraceability is the prevention of tracing user's actions. Secondly, anonymity is the ability to allow anonymous information. Lastly, unlinkability means that the information cannot be associated with the source entities. The protocol also provides confidentiality even when faced with a leakage of persisted data.

Regarding features, Snake provides user, group and data management as well as a public key authentication through Web of Trust mechanisms or through the Socialist Millionaire Protocol in order to preserve privacy. On the one hand, Web of Trust is a concept which creates a graph of trust where the nodes trust the connected nodes of their connected nodes. For instance, if Alice trusts Bob and Bob trusts Carol, Alice trusts Carol. On the other hand, Socialist Millionaire Protocol is based on the socialist millionaire problem in which two millionaires try to compare their fortunes without being the first to disclose the information.

Focusing on group management, the protocol ensures confidentiality of the group data by using symmetric encryption. When someone joins the group, the symmetric key is shared with them. While when someone leaves the group, the key is refreshed and spread to the group members. Group features, like invite or remove members, are only performed by group administrators. For data management, the protocol provides availability by persisting encrypted information, which was previously encrypted in client-side, making it available only for authorised entities.

2.4 Summary

Table 2.4 summarises the state of the art matching the protocols with the security requirements and with group communications, namely end-to-end encryption, confidentiality, integrity, availability, group communications and database abstraction. Firstly, end-to-end encryption ensures confidentiality and integrity of the shared information. Secondly, confidentiality represents the requirement of preventing unauthorised access of the persisted and shared information. Thirdly, integrity means that the information shared in the system cannot be modified from unauthorised entities. Fourthly, availability means that the information must be available only for authorised entities. Fifthly, group communications means the ability of the protocol to support users communicating through a group. Finally, a database abstraction is the ability of the protocol to easily allow multiple and different data models in their architecture.

Table 2.4: State of the art summary

	End-to-End Encryption	**Confidentiality**	**Integrity**	**Availability**	**Group Communication**	**Database Abstraction**
HTTPS[18, 19]	✗	✓	✓	✗	✗	✗
PGP[15, 8]	✓	✓	✓	✗	✗	✗
OTR[6, 1]	✓	✓	✓	✗	✗	✗
MTProto[16]	✓	✓	✓	✗	✓	✗
Signal[9, 17]	✓	✓	✓	✓	✓	✗
Snake[4, 3]	✓	✓	✓	✓	✓	✗
Sharelock	✓	✓	✓	✓	✓	✓

HTTPS only supports confidentiality and integrity as it encrypts the communication channel instead of encrypting the data. PGP and OTR also provide end-to-end encryption by encrypting the data with cryptographic keys only known by the end users, which are asymmetric and symmetric keys, respectively. Moreover, OTR ensures deniability of the information. MTProto adds group communication but not availability as the information is encrypted in client-side and sent directly to the end users. Signal and Snake are very similar providing all the above requirements and availability by encrypting information in client-side with symmetric keys which are encrypted with the group members' public keys. However, Snake is an OSN while Signal is a communication protocol.

Note that none of the aforementioned protocols implements a database abstraction, which is the ability to apply different data models in multiple scopes of group communications. This feature speeds up the integration process of new systems with the protocol, regardless of their data model. Chapter 3 includes a suggestion on how to implement the feature.

The last line represents Sharelock which provides end-to-end encryption in group communications through untrusted servers and is completely defined in the following chapter.

Chapter 3

Protocol

The protocol, named Sharelock, is based on a client-server architecture. The clients and the server are connected through a secure channel which in web scenarios could simply be a HTTPS channel. Based on that, the protocol provides an end-to-end encryption for group communication through untrusted servers.

The following sections describe, firstly, the system architecture by explaining how the design choices meet the requirements. Secondly, they describe the security goals detailing the used cryptographic algorithms and the protocol threat model. Thirdly, they describe all features provided by Sharelock, namely *User* management, *Data* management and *Group* management. Lastly, they describe a security evaluation of the protocol detailing some possible attacks.

3.1 Architecture and Design

The design choices rely in the requirements, namely end-to-end encryption in asynchronous group communication through untrusted servers, application independence, scalability, availability as well as user, group and data management.

To meet end-to-end encryption in asynchronous group communication through untrusted servers, Sharelock shifts all cryptographic work to the client side. The clients are responsible for encrypting and decrypting their group information as well as managing their keys. As a consequence, the server only persists encrypted information sent by the clients. In order to achieve high scalability and to decrease the client processing time, the protocol adopts high-speed cryptographic algorithms featured in National Institute of Standards and Technology (NIST) Recommendations [5], namely *SHA-256* for hash functions, *RSA-OEAP* for asymmetric key generation, *AES-GCM* for symmetric key generation and *PBKDF2* for cryptographic operations with passwords. Despite using these cryptographic algorithms, for some protocol primitives, this design choice could increase the client processing time and the network latency for large group states (data shared) and sizes (number of members). However, the application developer can implement some adjustments to decrease the client processing time in order to achieve higher scalability.

In spite of providing end-to-end encryption in group communication, Sharelock does not support identity verification. It means that each identity have to be verified out of band by the *User* through third party mechanisms. Unfortunately, it also means that the information sent by the *Server* must be verified in the client side to prevent tampering (see Section 3.2). To address this, Sharelock implements a trust list where each *User* could persist in the *Server* their out of band *Users'* identities validations (see Section 3.3).

Access control is obtained through group keys, which are only known by group members, preventing unauthorised access from the server and other entities. Furthermore, the group state is always encrypted with a group key, which, in turn, is encrypted with the group members' public keys. Consequently, there is no unauthorised access to the group key and the group state.

Sharelock provides application independence allowing different data types which are converted to *Objects*, supporting multiple data models. Furthermore, *Object* is defined between brackets and contains key value structures. These structures are attributes separated by commas and each of them contains a key and a value separated by a colon. The key has to be a string while the value could be a string, an integer, a double, a boolean, a list or even an object with the same structure definition than it. In addition, the list has to be surrounded by square brackets and could contain the same objects and data types as the Object. When the Object is converted to a string, it only appends double quotes around the object. This Object follows the same structure as the Javascript Object Notation (JSON) definition. Therefore, when the *Group* state is persisted, the Object is converted to a string, encrypted with the *Group* key and tagged with a specific label.

Regarding the Sharelock data model, Diagram 3.1 details how the entities were modelled to meet the aforementioned requirements. Firstly, Table User has a key pair, a session token, a salt, a password and an email address. Each *User* also have a list of trusted people. Despite the protocol does not manage these lists, it persists and makes these trusted identities available for the *Users*. Secondly, Table Group represents a set of *Users* who share *Data*. It also has a symmetric key associated which represents the *Group* key. Finally, Table Data represents the *Group* state which allows multiple scopes with different label names.

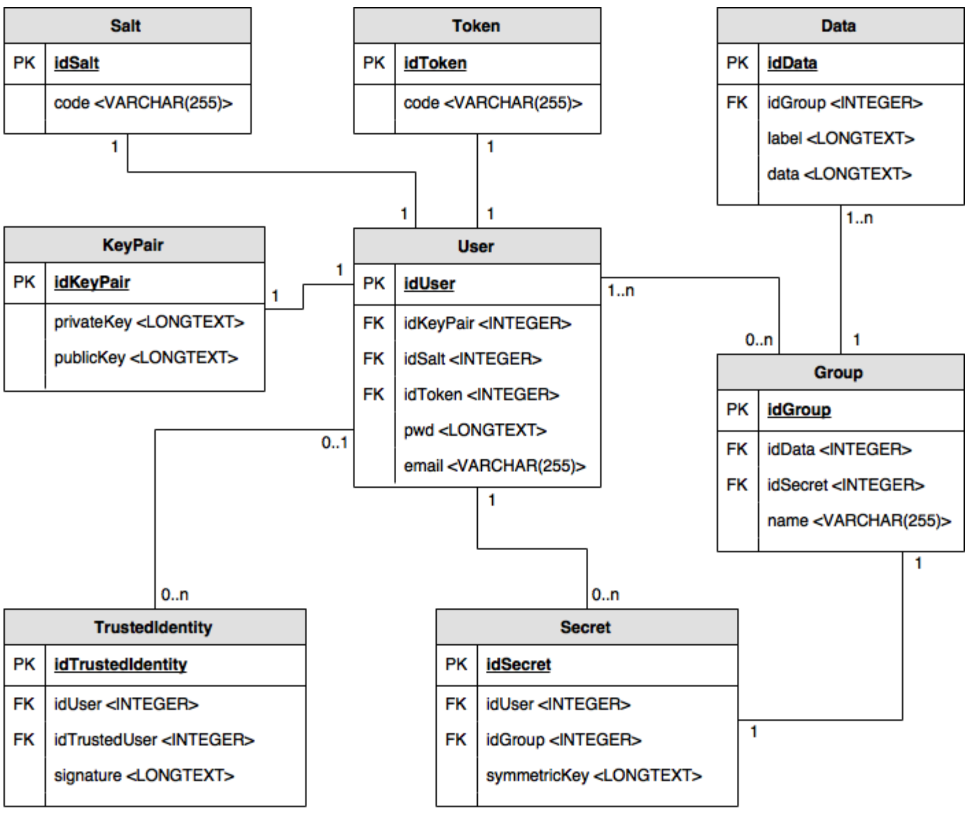


Diagram 3.1: Protocol Data Model. *VARCHAR(255)* for a string with a maximum length of 255 characters. *LONGTEXT* for a string with a maximum length of 4 gigabyte. *FK* for foreign key and *PK* for primary key.

3.2 Security Goals

The protocol provides an end-to-end encryption where the *Server* is considered as an untrusted entity. As a consequence, the information must always be encrypted and signed in client side through group keys.

Accordingly, when a *User* joins the system, the User generates a key pair, encrypts the generated private key and persists the key pair in the *Server*. Furthermore, the key pair is generated through *RSA-OEAP* algorithm with 2048 length.

Diagram 3.2 illustrates how the protocol encrypts and persists a key pair. Firstly, the password is imported using the *PBKDF2* algorithm. Secondly, the key is derived. Thirdly, the second half of this key is used to encrypt the private key using *AES-GCM* algorithm. Lastly, the key pair is persisted.

Additionally, Diagram 3.3 details how the *Client* fetches their key pair. Firstly, they requests the *Server* by sending the email and a hash of the first half of the aforementioned derived key. Secondly, the *Server* validates the

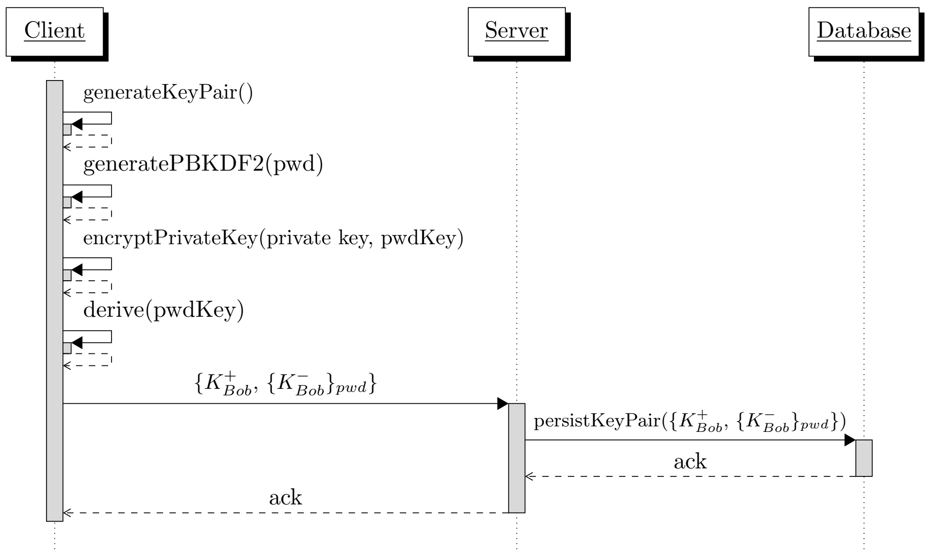

Client
Server
Database
generateKeyPair()
generatePBKDF2(pwd)
encryptPrivateKey(private key, pwdKey)
derive(pwdKey)
$\{K^+_{Bob}, \{K^-_{Bob}\}_{pwd}\}$
persistKeyPair($\{K^+_{Bob}, \{K^-_{Bob}\}_{pwd}\}$)
ack
ack

Diagram 3.2: Persisting a key pair. After the private key encryption, the key pairs are persisted in the database.

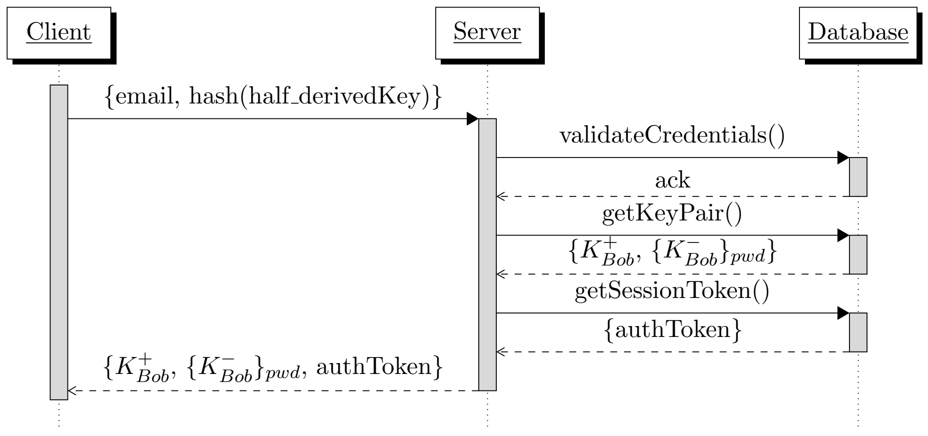

Client
Server
Database
{email, hash(half_derivedKey)}
validateCredentials()
ack
getKeyPair()
$\{K^+_{Bob}, \{K^-_{Bob}\}_{pwd}\}$
getSessionToken()
{authToken}
$\{K^+_{Bob}, \{K^-_{Bob}\}_{pwd}$, authToken}

Diagram 3.3: Fetching a persisted key pair. First, the *Client* authenticates and the *Server* validates the credentials. Subsquently, the *Server* fetches the key pair and the session token by requesting the *Database*. Finally, the *Server* returns the key pair and the session token. When the *Client* receives the key pair, it decrypts the private key using the *User* password and imports the key pair.

credentials and returns the key pair as well as the session token. Finally, the private key is decrypted and both keys become available in the node.

To sum up, the main advantage of these decisions is the fact that the information can be encrypted, decrypted and signed always in the client side preventing leakage and tampering attacks. In order to meet integrity and access control, before persisting to the database, the *Group* state is always encrypted with a *Group* key, which, in turn, is encrypted with all *Group* members' public key and signed by the *Group* founder.

Group Encryption Model

Achieving secure group communication requires a adequate spcification of the *Group* encryption model. The following paragraphs detail the required keys and how the model works in order to prevent attacks such as leakage, tampering or Man In The Middle (MITM).

The *Group* state is always encrypted with a *Group* key, as shown in Equation 3.1. K_s represents the *Group* key and is used to encrypt and decrypt the *Group* state.

$$GroupStateEncryption = \Big(\{M\}_{K_s}\Big) \tag{3.1}$$

When the *User* creates a *Group*, the *Group* key is generated and signed with the founder's private key. In addition, when someone is invited, the *User* who invited them must encrypt the *Group* key with the *Group* members' public keys in order to spread the *Group* key to all members, as shown in Equation 3.2. For instance, this equation presents a *Group* key signed by Bob's private key and shared with all *Group* members, namely Bob and Alice.

$$\begin{aligned} GroupKeyEncryption = &\Big([K_s]_{K^-_{Bob}}, \\ &\{K_s\}_{K^+_{Bob}}, \{K_s\}_{K^+_{Alice}}\Big) \end{aligned} \tag{3.2}$$

When a member leaves the *Group*, the *Group* key has to be updated in order to protect future information from unauthorised entities. This mechanism is always initiated by the *User* who removed someone from the *Group*. Therefore, a new *Group* key is generated, signed and persisted by following the same process of creating a *Group*, as specified in the Equation 3.2.

The *Group* state structure is a dictionary and a list of dictionaries, as in Equation 3.3. The dictionary has the data encrypted with the *Group* key, while the list contains multiple dictionaries, which represent the *Group* key encrypted for each *Group* member.

$$\begin{aligned} GroupState = &\Big(\{data : \{M\}_{K_s}\}, \\ &\big(\{key : \{K_s\}_{K^+_{Alice}}, email : \text{"}alice@...\text{"}\}, ...\big), \\ &\{signed : [K_s]_{K^-_{Bob}}\}\Big) \end{aligned} \tag{3.3}$$

To prevent tampering, the *Group* key is signed with the founder's private key. At any time the *User* is able to check the integrity of the symmetric key.

Threat Model

To evaluate the protocol security, it is required to define what kind of attacks are expected. Obviously, the main goal is to limit the attacker to access and manipulate private information.

We consider the *Server honest but curious* which means that the *Server* will not deliberately attack the protocol by tampering or replaying messages. However, the *Server* might read and infer from plain text information. For instance, the *Server* manages the group and, in order to prevent confidentiality and integrity, the protocol shifts the key distribution problem to the client side. As a consequence, the *Server* knows what users belong to each group, but the *Server* would never manipulate information in order to attack the protocol and exploit private information.

In terms of classic attacks, masquerading attacks may be attempted when the Attacker uses other identity to send and to receive messages from the *Groups*. Tampering attacks can occur when Attacker intercepts *Group* states and manipulates them. Eavesdropping attacks occur when the Attacker copies the states of the *Groups* without authorisation. Denial of Service (DOS) attacks occur when multiple Attackers requests the *Clients* and the *Server*. Section 3.4 details the security evaluation of these attacks.

3.3 Features

Authentication mechanisms, group communication and data persistence are the most required features of any system. Thus, *Sharelock* provides these requirements through three cross-cutting features, namely *User, Group and Data Management*, detailed in the next subsections.

3.3.1 User Management

User Management includes *Login*, *Account Registration*, *Login with Session Token*, *Account Deletion*, *Account Information* procedures and *Management of the Trust List.*

User Registration

In most systems, *User Registration* is a must-have feature usually implemented through a form. Thus, according to Diagram 3.4, the following steps are required to register an account:

1. The *User* presents the credentials, namely the email and the password.

2. The key pair is generated using the RSA-OEAP algorithm.

3. The password is imported into a new key through the PBKDF2 algorithm.

4. This key is derived and the first half is used for authentication proposes.

5. The private key is encrypted with the second half of the previous derived key by using the AES-GCM algorithm.

6. The request is made, including the email, a hash of the first half of the derived password key, the public key and the encrypted private key.

7. The *User* is created and the email, the derived password key and the key pair are associated to their account.

8. When the registration process is complete, the *User* is notified with a success message.

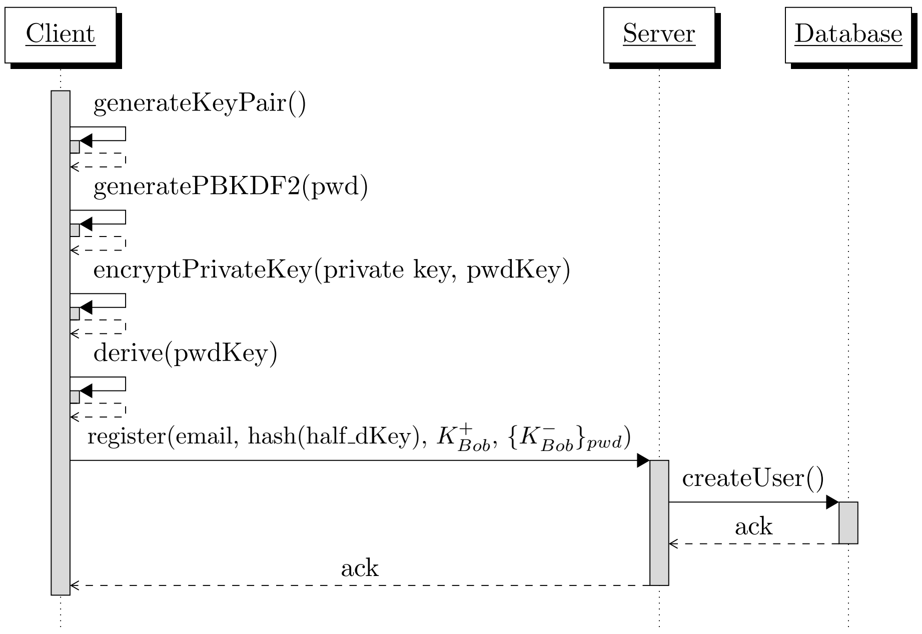

Client
Server
Database
generateKeyPair()
generatePBKDF2(pwd)
encryptPrivateKey(private key, pwdKey)
derive(pwdKey)
register(email, hash(half_dKey), K^{+}_{Bob}, $\{K^{-}_{Bob}\}_{pwd}$)
createUser()
ack
ack

Diagram 3.4: User Registration. The key pair is generated, the private key is encrypted with a derived password key, the password is derived and converted to a base 64 string. Afterwards, the request is made in order to create a new account.

User Login

User Login has to authenticate a *User* by validating their credentials. This feature is implemented with an adequate level of security through salt functions, encrypted passwords and hashes. Similarly to Diagram 3.5, the following steps detail the *User* authentication procedure.

1. The *User* presents the credentials, namely the email and the password.
2. The password is imported into a new key through the PBKDF2 algorithm.
3. The previous key is derived and the first half is used for authentication proposes.
4. The request is made with their email and a hash of the first half of the derived password key.
5. The *Server* validates the password with the associated salt. In case of correctness, the *Server* returns a new session token, the public key and the encrypted private key.
6. When the *Client* receives the information, it initiates the process to decrypt the private key and to import the key pair.
7. The *Client* is able to consider the *User* as authenticated.

Token Based User Authentication

Sharelock returns a *authToken* after a successful *User* authentication. It brings some advantages against the session based authentication, namely decreased memory usage, scalability and flexibility. This token is a mandatory parameter in most endpoints of the public interface as it is used as an authentication token. According to Diagram 3.6, when a *User* is authenticated with a session token, the *Server* returns an acknowledgement.

Delete Account

Deleting an Account allows a *User* to delete their own account from the system, as shown in Diagram 3.7. Note that this procedure simply deletes the account information but not the *Groups* states they belong to.

Trust List Management

Trust List Mechanism manages out of band authentications of *User's* identities. Each *User* has a group of trusted people. It means that the *User*

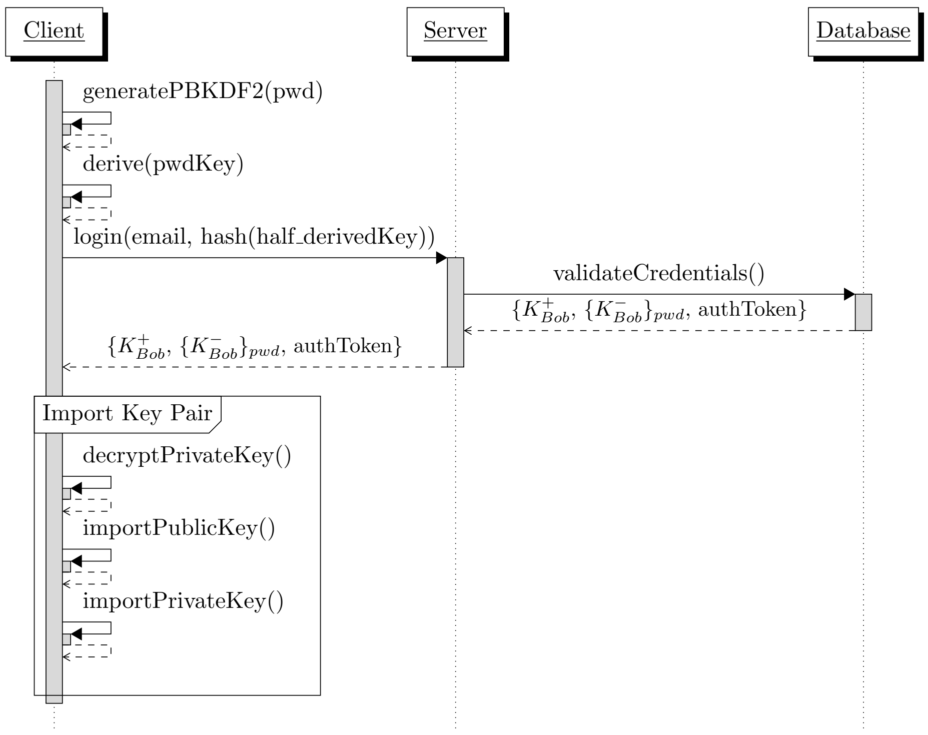


Diagram 3.5: User Login. The password is transformed into a PBKDF2 key which is derived into a set of bits and converted to a base 64 string, after, the login request is made. Then, the *Server* validates the credentials by requesting the Database and, in case of success, it returns the public key, the encrypted private key and the session token. Finally, the *Client* decrypts and imports the keys.

knows the groups' members true identity as the *User* previously authenticated them by out of band mechanisms.

Diagram 3.9 portrays that the *User* is able to fetch their trust list and to authenticate an *User* public key through out of band mechanism that they trust. In case of matching success, they sign the public key and the email, and persist it, as shown in Diagram 3.8.

Fetch Account Information

According to Diagram 3.10, the *User* is able to request their own account information, which includes the email address, the public and the private key, the *authToken* and the *Groups* they belong to. According to Diagram 3.11, if *User* requests account information from another *User*, the procedure simply returns the email address and the public key of this *User*.

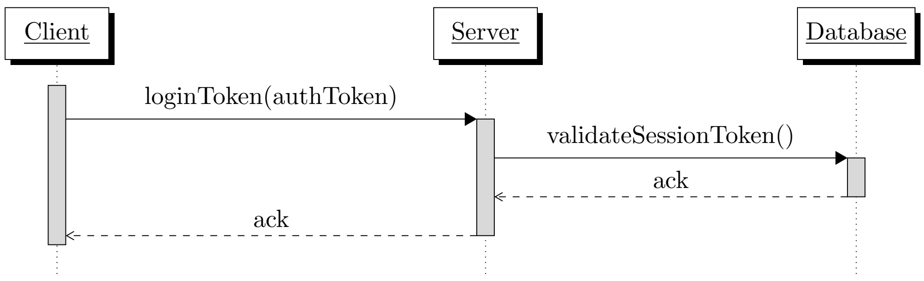


Diagram 3.6: Token Authentication.

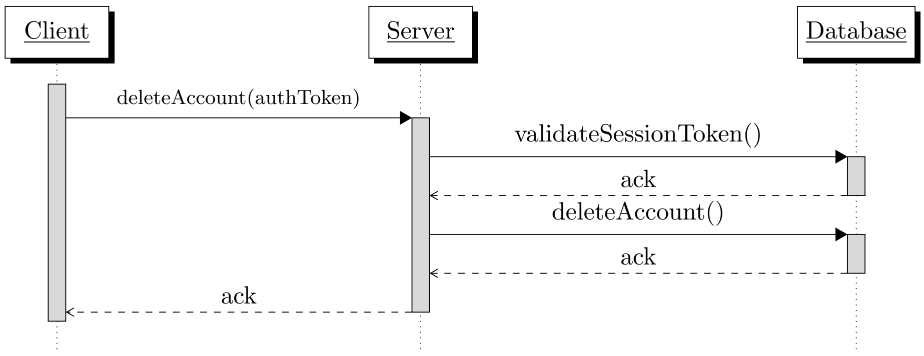


Diagram 3.7: Delete User Account.

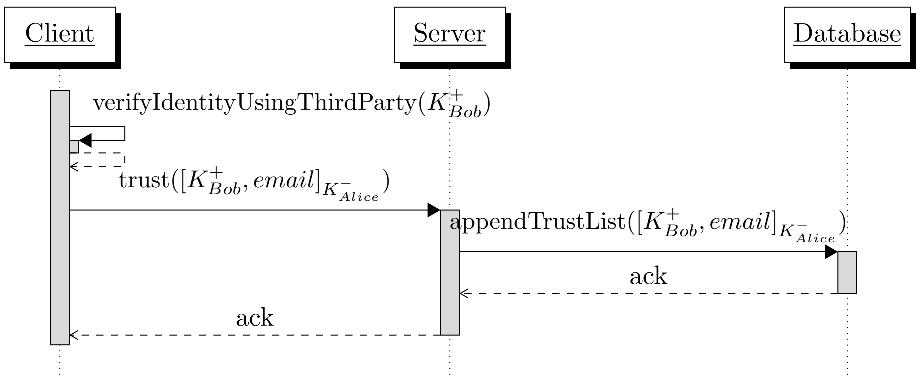

Client
Server
Database
verifyIdentityUsingThirdParty(K^+_{Bob})
trust($[K^+_{Bob}, email]_{K^-_{Alice}}$)
appendTrustList($[K^+_{Bob}, email]_{K^-_{Alice}}$)
ack
ack

Diagram 3.8: Trust User. Bob's identity is verified by Alice through a third party entity. In case of correctness, Alice signs Bob's public key and Bob's email with his private key and appends it into the trust list.

3.3.2 Group Management

The protocol offers a set of primitives to manage a *Group* shifting the management of the *Group* policies to the application layer. Therefore, each member is able to add and remove *Users*, and delete *Groups*. The following subsections detail the primitives *Create Group*, *Delete Group*, *Fetch Group Key*, *Fetch Group Members*, *Add User to Group* and *Remove User from Group*.

Create Group

According to Diagram 3.12, in order to *Create a Group*:

1. The *User* has to name the *Group*.
2. The *Group* state was previously defined by the application developed. Therefore, the *User* initiates a new *Group* state.
3. A new *Group* symmetric key is generated in the client side.
4. The *Group* key is signed with the *User's* private key.
5. The *Group* state is encrypted with the *Group* key.
6. The *Group* key is encrypted with *User's* public key.
7. The *Client* requests the services layer. The request body includes the encrypted *Group* key and the *Group* name.
8. When the *Group* is created, the *User* is able to add new members.

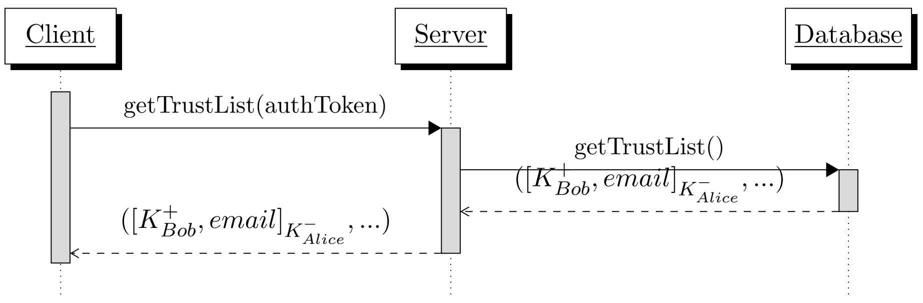

Client
Server
Database
getTrustList(authToken)
getTrustList()
$([K^{+}_{Bob}, email]_{K^{-}_{Alice}}, ...)$
$([K^{+}_{Bob}, email]_{K^{-}_{Alice}}, ...)$

Diagram 3.9: Fetch Trust List. When Alice requests hers trust list, the *Server* returns a list of dictionaries which represent each *User's* identity signed and trusted by her.

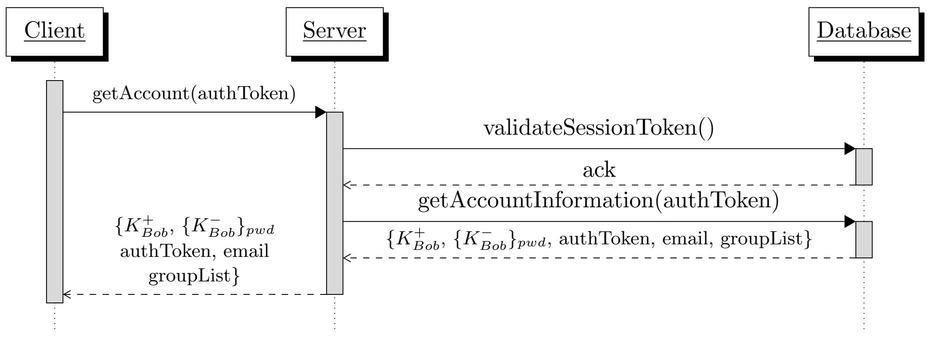

Client
Server
Database
getAccount(authToken)
validateSessionToken()
ack
getAccountInformation(authToken)
$\{K^+_{Bob}, \{K^-_{Bob}\}_{pwd}$, authToken, email, groupList$\}$
$\{K^+_{Bob}, \{K^-_{Bob}\}_{pwd}$ authToken, email groupList$\}$

Diagram 3.10: Get User Account.

Delete Group

Any team member has permission to delete the *Group*. According to Diagram 3.13, when it happens, the *Group* state and the *Group* keys are lost and cannot be recovered.

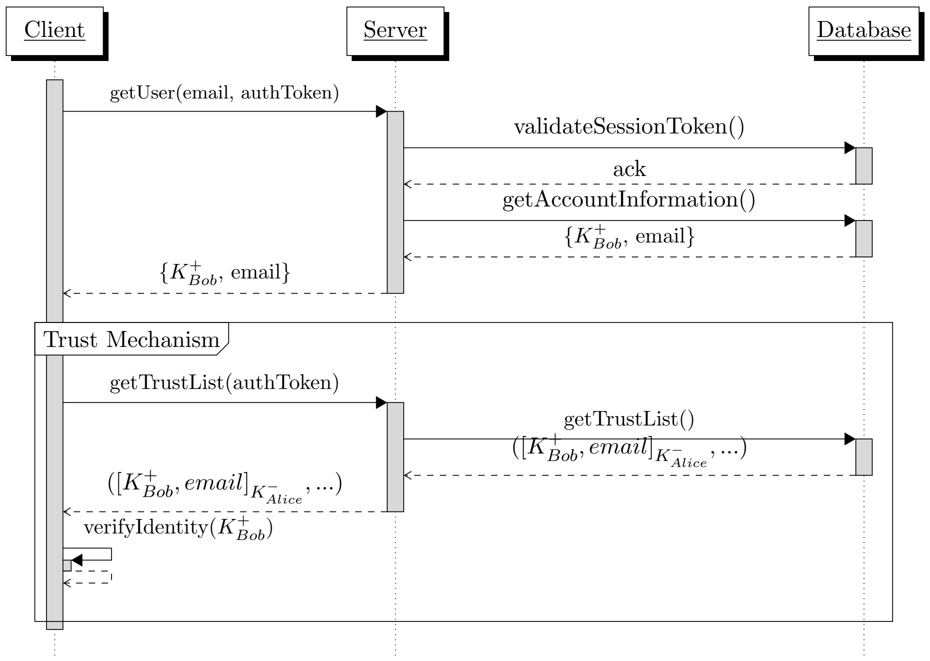


Diagram 3.11: Fetch User Information. When Alice fetches Bob's identity from the *Server*, she has to validate the identity returned by the *Server*. Consequently, she requests for her trust list and validates Bob's identity. If Bob's identity was not in the Alice's public list, she could verify Bob's identity through a third party mechanism as explained in 3.8.

Fetch Group Key

According to Diagram 3.14, in order to *Fetch a Group Key*, the *User* has to mention the group_id for their *Group* key to be returned. When the *Client* fetches the key, it initiates the decryption process to import the key for future cryptography operations.

Fetch Group Members

According to Diagram 3.14, in order to *Fetch Group Members*, the *User* has to mention the group_id in the request payload for the *Group* members to be returned in a list. Each element of the list contains the email and the public key of each *Group* member.

Add User and Remove User From Group

Any *Group* member has permission to add and to remove someone. According to Diagram 3.16, the *User* who added the new member will get their public key and has to encrypt the current *Group* key with the new members' public key, persisting them afterwards. Then, the *Group* key is shared with the new member when they get the *Group* state.

Additionally, according to Diagram 3.17, when a member is removed from the *Group*, the *User* who did it has to regenerate and to sign the new *Group* key, and get all *Group* members' public keys, in order to encrypt the

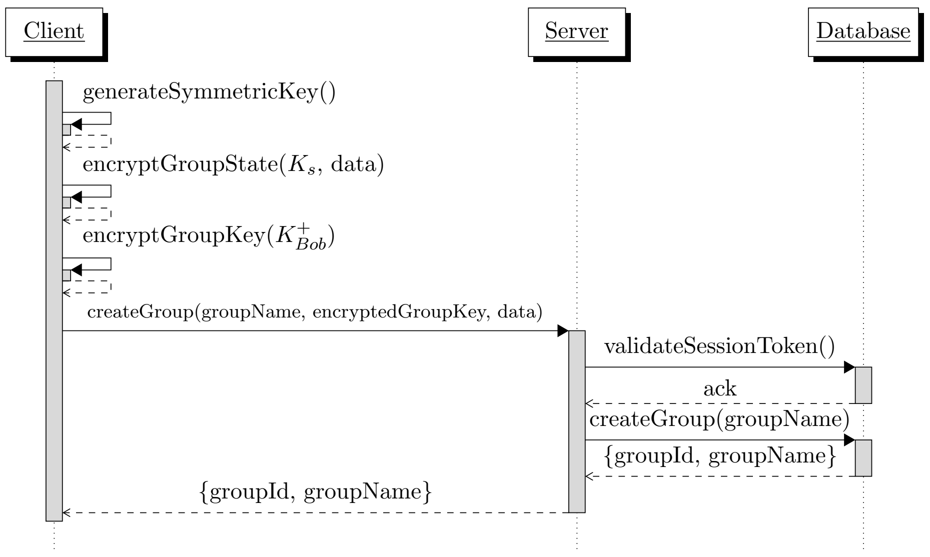


Diagram 3.12: Create Group. The *Group* key is generated, signed and the *Group* state is initialized and encrypted with the *Group* key. Then, the key is encrypted with *User's* public key and the *Group* is created.

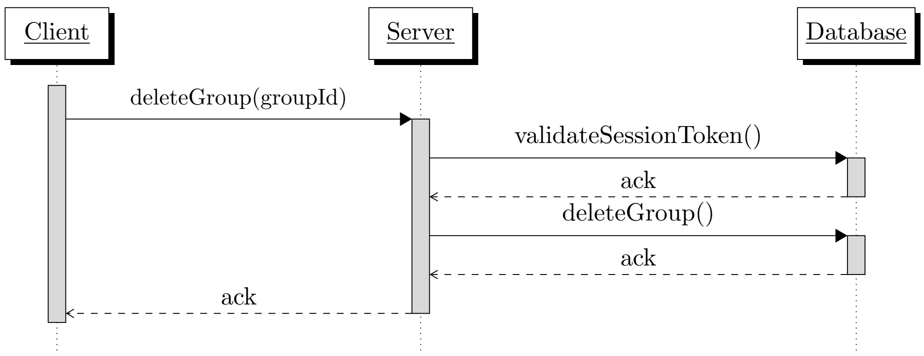


Diagram 3.13: Delete Group.

Group key separately with each *Group* members' public key. Then, the *User* persists the set of encrypted *Group* keys.

3.3.3 Data Management

Each *Group* has multiple states, which are *Objects* (see Section 3.1) converted to strings, encrypted and tagged with labels. When a *Group* member updates the *Group* state and tags it with an existent label, it persists a new stringified *Object* replacing the old one. In addition, the protocol does not provide a group state history. As a consequence, if the application has this requirement, the developer should implement it into *Objects*. In other words, the *bject* will be persisted with multiple *objects* inside. This solution

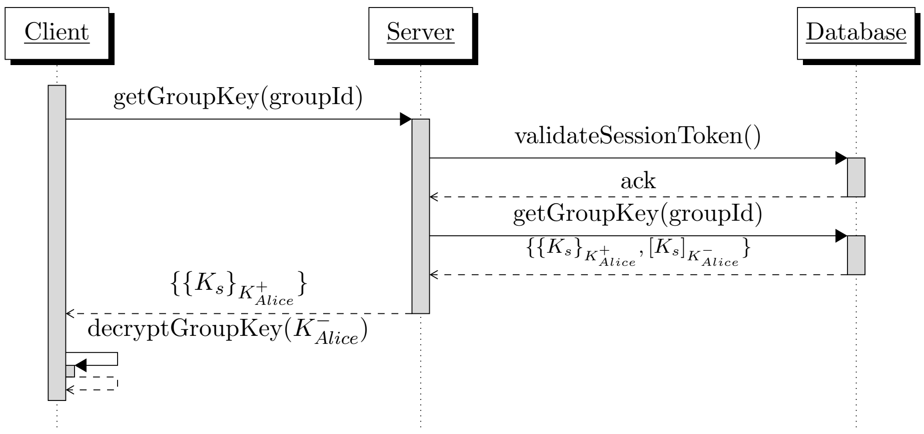

Client
Server
Database
getGroupKey(groupId)
validateSessionToken()
ack
getGroupKey(groupId)
$\{\{K_s\}_{K^+_{Alice}}, [K_s]_{K^-_{Alice}}\}$
$\{\{K_s\}_{K^+_{Alice}}\}$
decryptGroupKey(K^-_{Alice})

Diagram 3.14: Fetch Group Key. Alice requests the *Group* key and decrypts it using her private key. Alice receives the *Group* key signature which was previously signed by the group founder or by a group member (which performed a remove member from group operation).

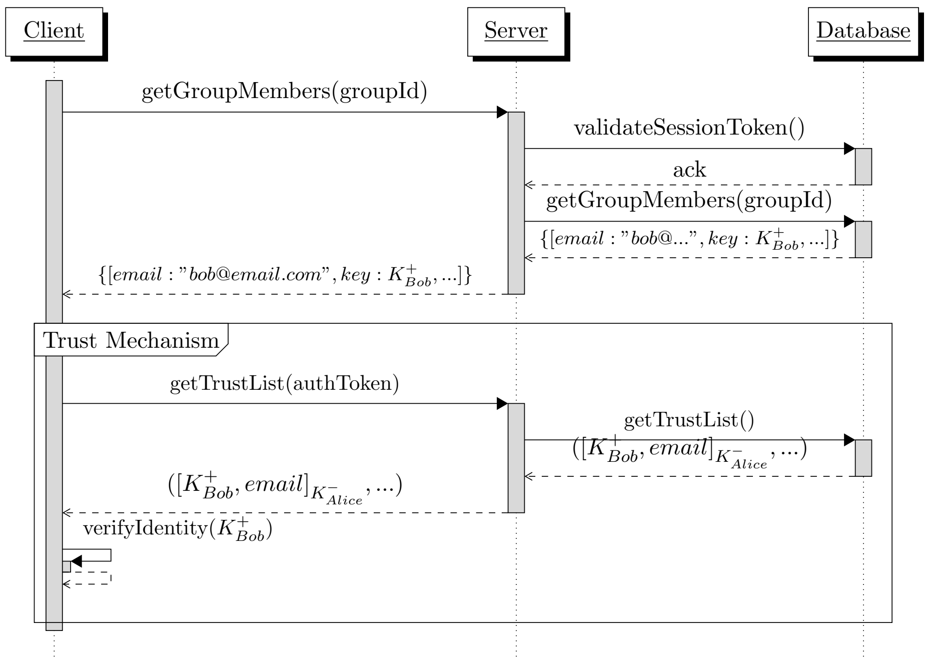

Client
Server
Database
getGroupMembers(groupId)
validateSessionToken()
ack
getGroupMembers(groupId)
{[email : "bob@...", key : K^+_{Bob}, ...]}
{[email : "bob@email.com", key : K^+_{Bob}, ...]}
Trust Mechanism
getTrustList(authToken)
getTrustList()
([K^+_{Bob}, email]$_{K^-_{Alice}}$, ...)
([K^+_{Bob}, email]$_{K^-_{Alice}}$, ...)
verifyIdentity(K^+_{Bob})

Diagram 3.15: Fetch Group Members. When Alice fetches the *Group* members, she has to verify their identities through her trust list. If there is any distrusted member in the list, she can initiate the trust *User* algorithm, as shown in Diagram 3.8.

leads the developers to a version control problem, which is their responsibility. The protocol also provides three primitives to manage the *Group* state, namely *Persist Data*, *Fetch Data* and *Delete Data*.

Persist Data

According to Diagram 3.18, the following steps summarise the *Persist Data* algorithm.

1. The *Group* key is requested.
2. The key is decrypted and imported in the client side.
3. The *Data* object is converted to string, encrypted with the *Group* key and persisted.

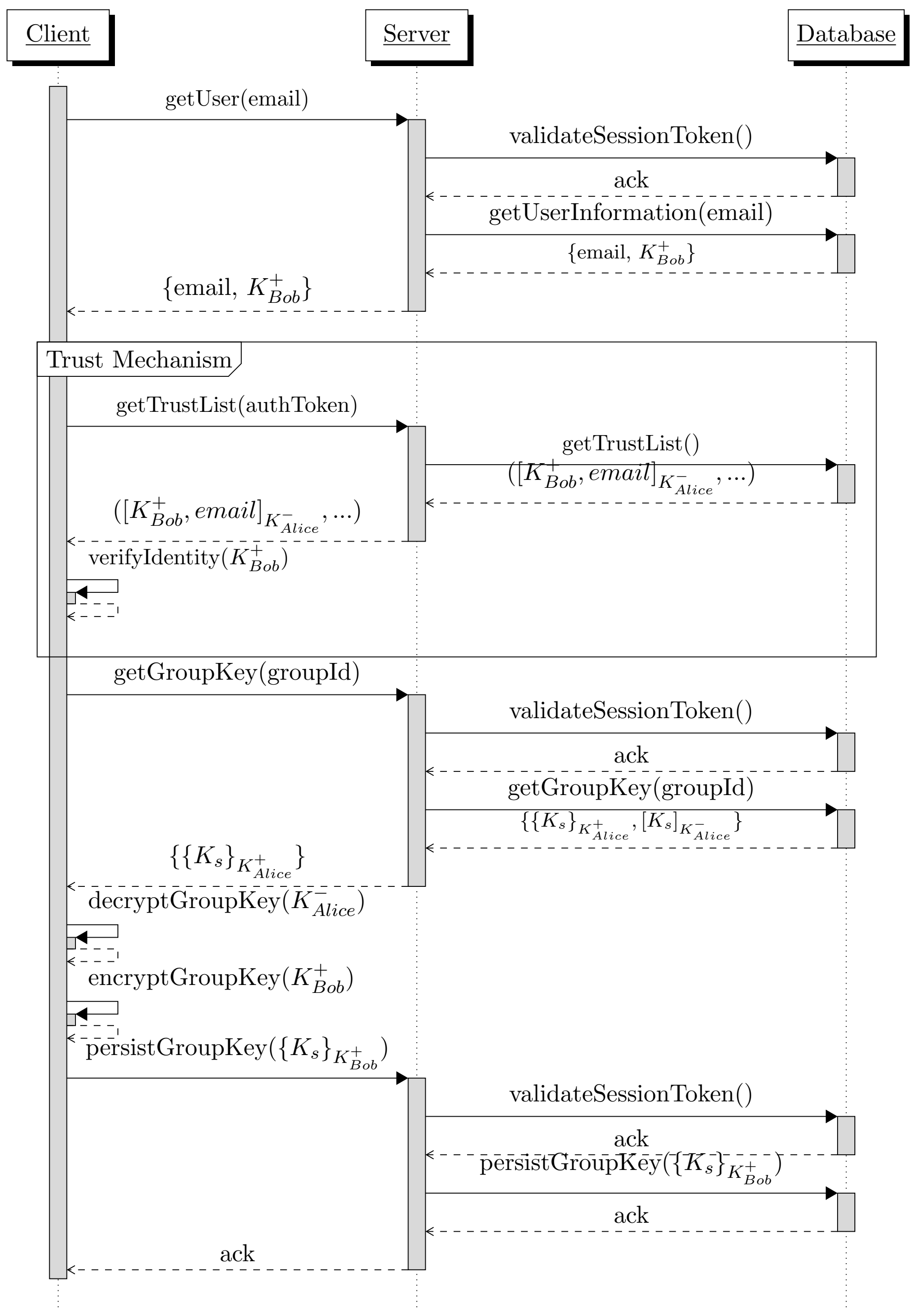

Client
Server
Database
getUser(email)
validateSessionToken()
ack
getUserInformation(email)
{email, K^+_{Bob}}
{email, K^+_{Bob}}
Trust Mechanism
getTrustList(authToken)
getTrustList()
$([K^+_{Bob}, email]_{K^-_{Alice}}, ...)$
$([K^+_{Bob}, email]_{K^-_{Alice}}, ...)$
verifyIdentity(K^+_{Bob})
getGroupKey(groupId)
validateSessionToken()
ack
getGroupKey(groupId)
$\{\{K_s\}_{K^+_{Alice}}, [K_s]_{K^-_{Alice}}\}$
$\{\{K_s\}_{K^+_{Alice}}\}$
decryptGroupKey(K^-_{Alice})
encryptGroupKey(K^+_{Bob})
persistGroupKey($\{K_s\}_{K^+_{Bob}}$)
validateSessionToken()
ack
persistGroupKey($\{K_s\}_{K^+_{Bob}}$)
ack
ack

Diagram 3.16: Add User to Group. In order for a *User* to join the *Group*, the *Group* key must be encrypted with the public key of the *User*.

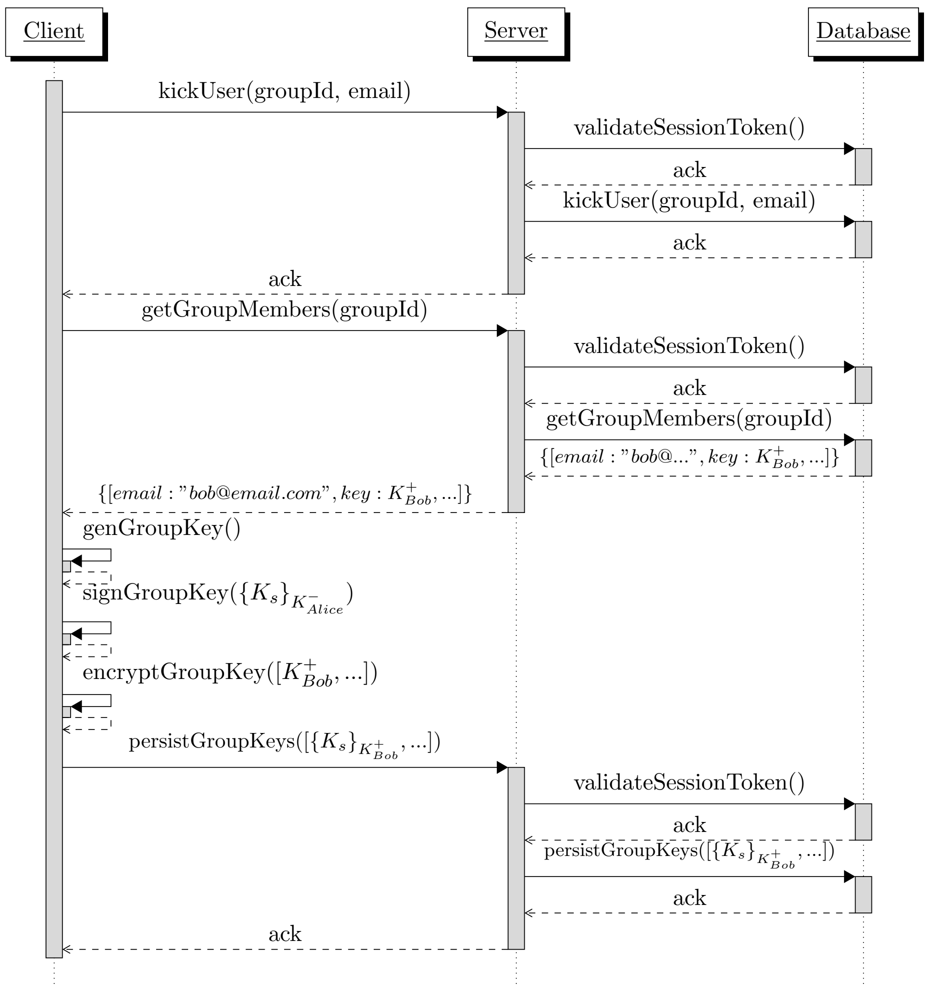

Client
Server
Database
kickUser(groupId, email)
validateSessionToken()
ack
kickUser(groupId, email)
ack
ack
getGroupMembers(groupId)
validateSessionToken()
ack
getGroupMembers(groupId)
$\{[email : "bob@...", key : K^{+}_{Bob}, ...]\}$
$\{[email : "bob@email.com", key : K^{+}_{Bob}, ...]\}$
genGroupKey()
signGroupKey($\{K_s\}_{K^{-}_{Alice}}$)
encryptGroupKey($[K^{+}_{Bob}, ...]$)
persistGroupKeys($[\{K_s\}_{K^{+}_{Bob}}, ...]$)
validateSessionToken()
ack
persistGroupKeys($[\{K_s\}_{K^{+}_{Bob}}, ...]$)
ack
ack

Diagram 3.17: Remove User from Group. Bob was removed from the *Group* by Alice. So, she has to generate a new *Group* key and sign it and to share it with the *Group* members, she has to get all *Group* members' public keys and encrypt the new generated *Group* key with those public keys.

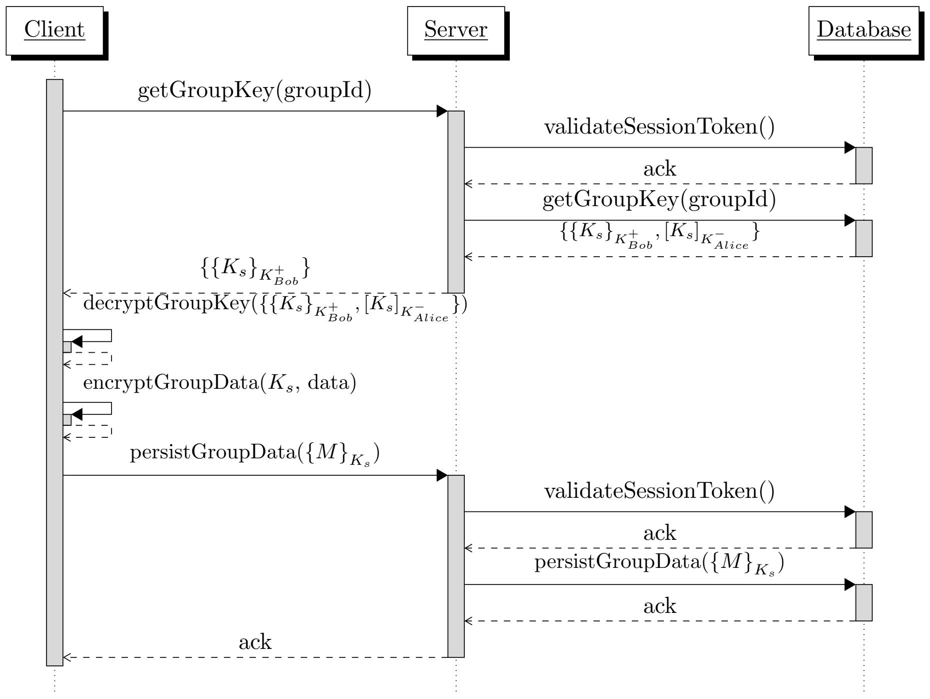

Client
Server
Database
getGroupKey(groupId)
validateSessionToken()
ack
getGroupKey(groupId)
$\{\{K_s\}_{K^+_{Bob}}, [K_s]_{K^-_{Alice}}\}$
$\{\{K_s\}_{K^+_{Bob}}\}$
decryptGroupKey($\{\{K_s\}_{K^+_{Bob}}, [K_s]_{K^-_{Alice}}\}$)
encryptGroupData(K_s, data)
persistGroupData($\{M\}_{K_s}$)
validateSessionToken()
ack
persistGroupData($\{M\}_{K_s}$)
ack
ack

Diagram 3.18: Persist Data to Group. Before persisting *Group Data*, the *Group* key is required to encrypt the *Group Data*.

Fetch Data

As aforementioned, this procedure only returns the current *Group* state as the protocol does not manage the *Group* state history. According to Diagram 3.19, the following steps detail the *Fetch Data* procedure.

1. The *Client* requests the *Group* state including the *User* session token for authentication needs.
2. When the *User* has the *Group* state and the *Group* key, the decryption mechanism is initiated.
3. The *Group* key is decrypted using their own private key.
4. The *Data* is decrypted using the *Group* key.
5. The *Data* is converted into an object.

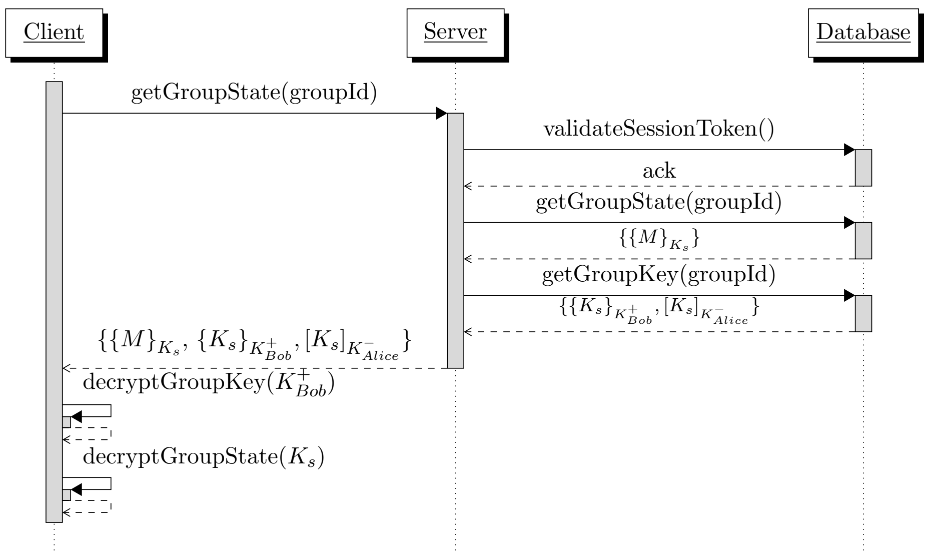


Diagram 3.19: Fetch Data from Group.

Delete Data

This feature follows the *Persist Data* procedure. The *Client* requests the *Group* key, encrypts the empty object and persists it. In this case, the *object* is an empty *Data* model.

3.4 Security Evaluation

Sharelock uses cryptographic algorithms to hash keys and information, to encrypt and decrypt *Group* states, *Group* keys or key pairs as well as to generate key pairs (asymmetric keys) and *Group* keys (symmetric keys).

- *PBKDF2* is a key derivation function and it is used to derive passwords creating a key with more bits in order to prevent potential brute force attacks. Sharelock uses this algorithm to derive *Users'* passwords (see Section 3.3).
- *RSA-OEAP* is mainly used in asymmetric encryption and it is secure against chosen cipher text attacks. It prevents attackers from decrypting only a part of the message inferring knowledge from it. Additionally, the algorithm is used to generate a key pair when the *User* registers in the system.
- *AES-GCM* is used in symmetric encryptions and supports authenticated encryption providing integrity and confidentiality of the encrypted information preventing tampering attacks. Therefore, Sharelock uses this algorithm for cryptographic work with *Group* keys.

Passive Attackers

Passive Attackers do not interact with any system entity hacking the system only by watching the behaviour of the system entities and sniffed information. Sharelock is secure against them as they only see encrypted information (i.e., $\{M\}_{K_s}$).

Let Eve be an eavesdropper capturing the protocol communications from a *Group*. She cannot read the messages without the *Group* key and to access the *Group* key she needs a *User's* private key to decrypt it. Consequently, each private key is encrypted with a derived PBKDF2 key which was previously generated with the *User's* password.

In contrast, if she knew the *User's* password, she would be able to read all *Group* states. Although, to prevent weak passwords in the protocol, we can make some adjustments, namely use temporary passwords or use strong password hashes algorithms.

Active Attackers

By Active Attackers, we mean a hacker who tries to manipulate information, replay identities or flood the servers with requests in order to disrupt their service.

In Sharelock, we identified message tampering, replaying and masquerading as the main active attacks. In spite of the messages and the keys being encrypted by symmetric authenticated encryption or by digital signatures, the protocol presents some weaknesses, namely requests without signed and encrypted payloads. For instance, *Remove User From Group* requires the session token for authentication proposes, the groupId and the email from the User who we want to remove. In this scenario, if the communication channel is encrypted, Mallory cannot manipulate the request and remove any user from any *Group*. We also evaluate another weaknesses, namely replayed attacks and key manipulation which are described in the next paragraphs.

Replayed Attacks occur when the attacker replicates information and resends it to the system. These attacks also arise when the attacker resends delayed information over the network. These attacks are mainly applied to stateless servers as they treat each request as a new one, they cannot retain clients information. However, Sharelock cannot be affected by these attacks as the protocol uses a secure channel between client and server. Otherwise, Mallory could delay the message to change the *Group* keys and reset the *Group* key to a previous value.

Key Manipulation is considered a threat as any *Group* member is able to alter the *Group* key which might compromise the *Group* privacy. Let Alice be a member of two *Groups*. She can change both *Group* keys to the same key compromising the privacy of both *Group* states.

Forward Secrecy

Forward secrecy prevents past encrypted information from being disclosed as long-term secret keys cannot decrypt previous information. Sharelock uses long-term secret keys such as *Group* keys K_s and key pairs K^+_{Bob}, K^-_{Bob}. In a scenario of a *Group* key was leaked, it would compromised all information encrypted with K_s. Therefore, to implement forward secrecy, K_s should be updated temporarily. For scalability reasons, the protocol does not support automatic group key updates as it would increase the client processing time, consequently, it would lead to a overall loss of scalability.

In addition, if a key pair were disclosed by a passive attacker, the result would be terrible as all group keys encrypted with this key would be leaked as well. For this reasons, we can state that Sharelock does not have forward secrecy.

Chapter 4

Implementation

In order to test and evaluate Sharelock, we created a web implementation. The clients are represented by web browsers, while the server is represented by an web API and a database.

The reminder of this chapter details which technologies we used to implement Sharelock, how the client-server architecture was implemented and what architectural components were developed, namely the services layer, the database and the Javascript library.

4.1 Architecture

We chose the *C4 Model* to describe our implementation architecture. As detailed in [7], this model describes any architecture in four levels of detail. The first level, named System Context Diagram, shows the big picture of our system by representing only the direct and indirect systems or people which our system interacts with. The second level, named Containers Diagram, illustrates how many containers (i.e, public interfaces, databases, servers, etc.) we have in our system, what is their job and their responsibilities, and what technologies they use. The third level, named Components Diagram, illustrates how the containers are made and what their responsibilities are. Finally, the fourth level is optional and represents Unified Modelling Language (UML) diagrams.

This implementation could be applied for multiple web applications as it provides a Javascript library with all protocol features implemented. Diagram 4.1 shows that any signed *User* can perform any protocol primitive, namely *User, Group or Data Management.* As a consequence, the web application developers have to deal with the management of *Group* policies limiting the access of *Users* to some protocol primitives. For instance, the *Group* founder is the only *Group* member who has the right to remove someone.

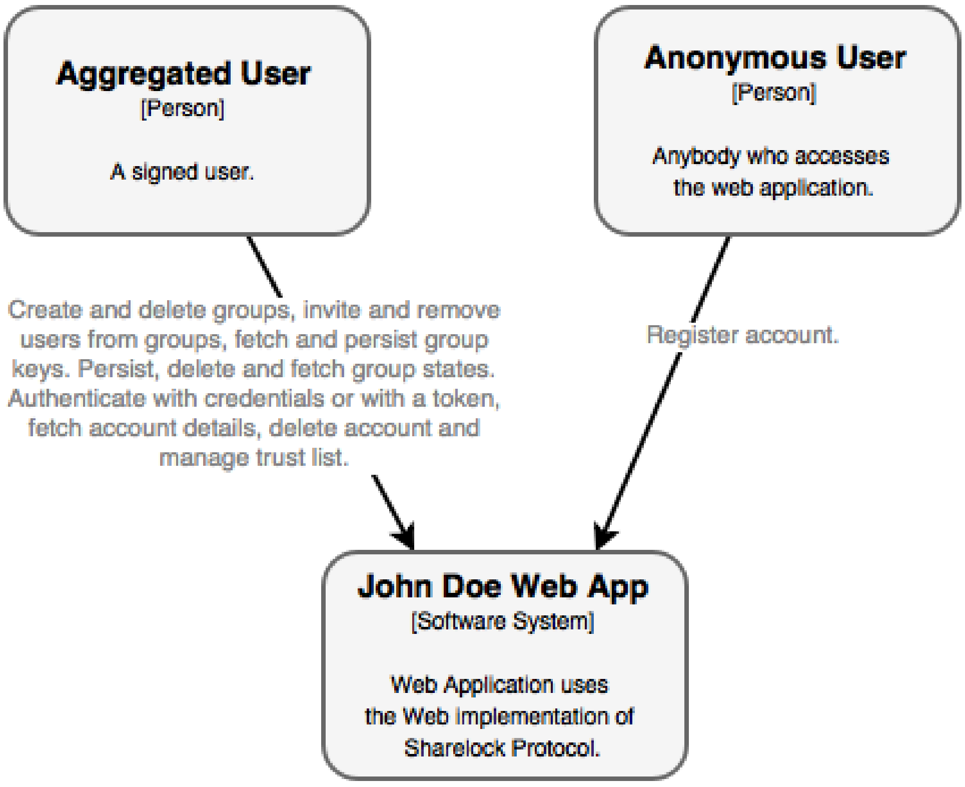

Aggregated User
[Person]
A signed user.
Anonymous User
[Person]
Anybody who accesses the web application.
Create and delete groups, invite and remove users from groups, fetch and persist group keys. Persist, delete and fetch group states. Authenticate with credentials or with a token, fetch account details, delete account and manage trust list.
Register account.
John Doe Web App
[Software System]
Web Application uses the Web implementation of Sharelock Protocol.

Diagram 4.1: System Context Diagram represents an abstract diagram for a web application which integrates the Sharelock Protocol. Basically, the protocol web implementation provides all features namely *User, Group and Data Management* to a signed *User*. It also provides the feature, register new account, to anonymous *Users*.

Diagram 4.2 illustrates that the architecture follows a client-server paradigm where the clients are web browsers and the server is composed by a RESTful API and a *MySQL* database. The RESTful API was developed with Java Platform, Enterprise Edition (J2EE) technologies.

As shown in Diagram 4.3, it contains stateless beans, namely Account-Bean, GroupBean and UserBean. Firstly, AccountBean has functions to create and delete accounts, to fetch account information and to login with or without session tokens. Secondly, GroupBean has functions to create and delete groups, to invite and remove users from groups, to persist, fetch and remove group states, to persist and fetch group keys and to fetch all group members. Lastly, UserBean has functions to manage the trust list and to get users information. It also implements an Object-Relational Mapping (ORM) layer using Java Persistence API (JPA) technology. This layer maps Java classes and their relationships into database tables. This container also

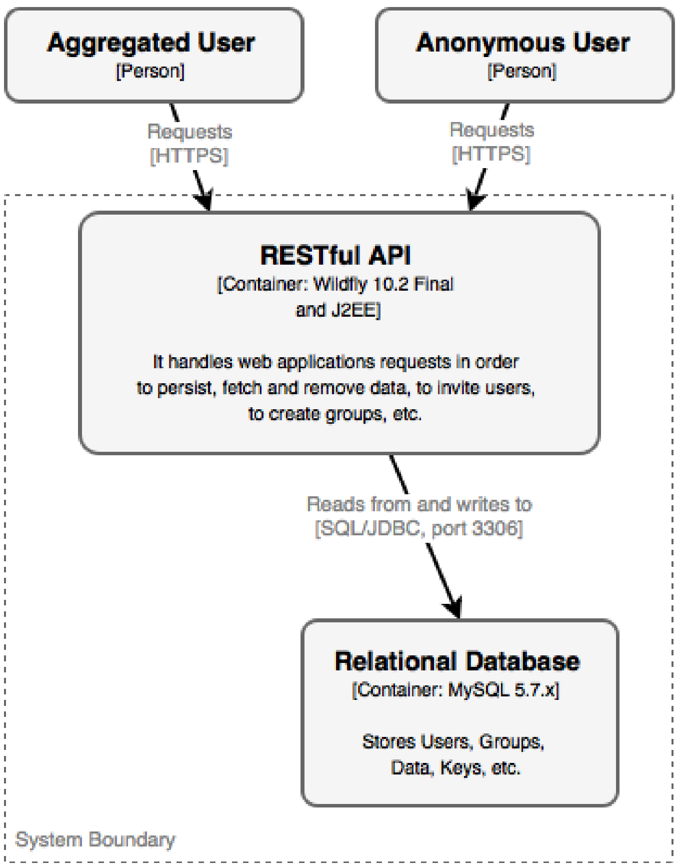

Aggregated User
[Person]
Anonymous User
[Person]
Requests
[HTTPS]
Requests
[HTTPS]
RESTful API
[Container: Wildfly 10.2 Final and J2EE]
It handles web applications requests in order to persist, fetch and remove data, to invite users, to create groups, etc.
Reads from and writes to
[SQL/JDBC, port 3306]
Relational Database
[Container: MySQL 5.7.x]
Stores Users, Groups, Data, Keys, etc.
System Boundary

Diagram 4.2: Containers Diagram shows up that each *User* requests the RESTful API through an HTTPS channel. The services layer was implemented through J2EE technologies and was deployed on WildFly application server. It also uses a *MySQL* database to persist all information shared in the protocol, namely Users, Groups, Data and Keys.

implements a Representational State Transfer (REST) public interface using *J2EE/JAX-RS* technologies. In sum, the web API uses stateless beans which are connected to the database through the MySQL java connector.

Data Model

Diagram 4.4 portrays the tables which were automatically created by the ORM layer, as shown in Diagram 3.1. However, it has more tables, namely *group_members*, *_group_data*, *_group_secret* and *user_trustedIdentity* because these tables represent auxiliary tables created in many to many relationships.

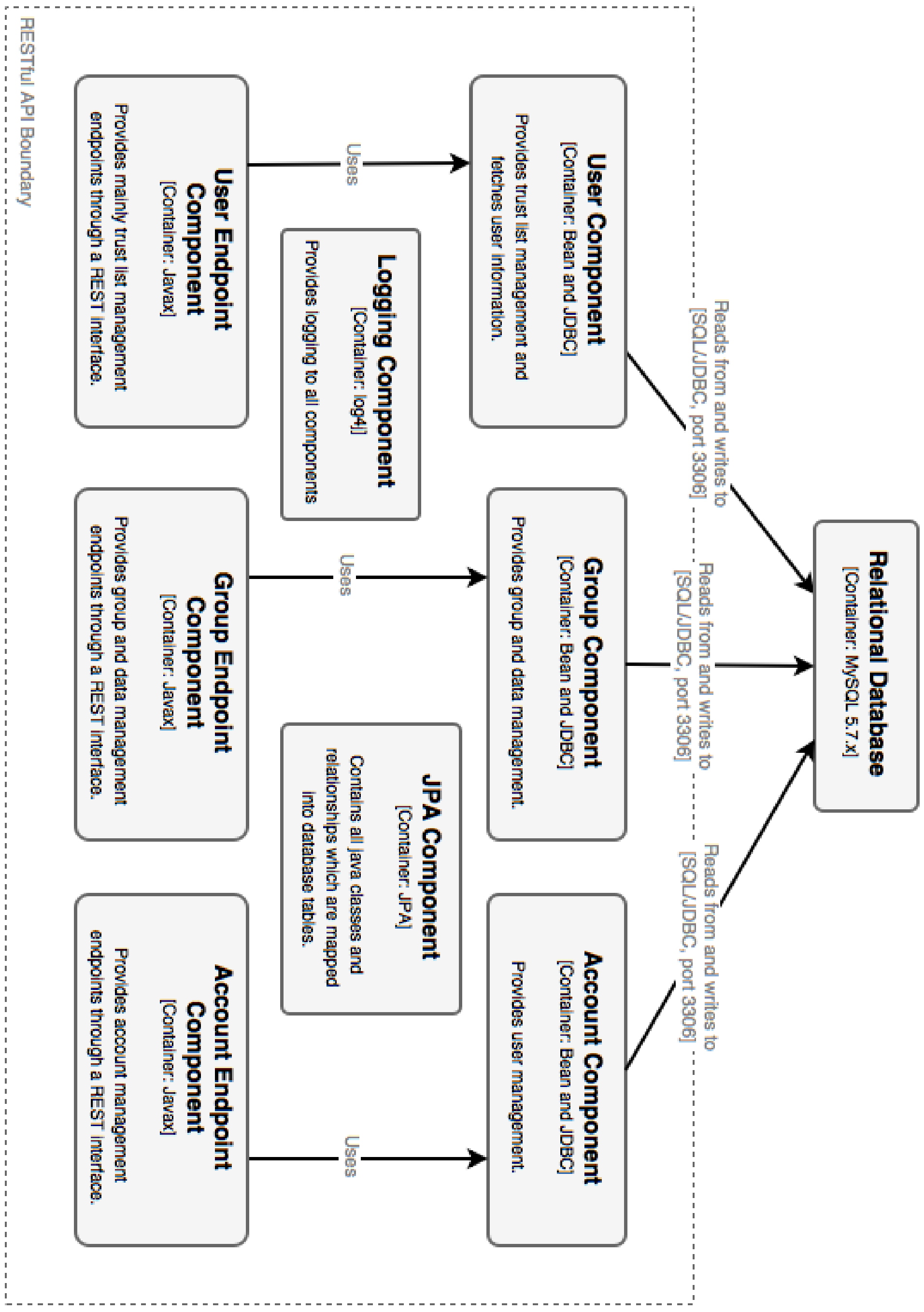

RESTful API Boundary
User Endpoint Component
[Container: Javax]
Provides mainly trust list management endpoints through a REST interface.
Uses
User Component
[Container: Bean and JDBC]
Provides trust list management and fetches user information.
Logging Component
[Container: log4j]
Provides logging to all components
Group Endpoint Component
[Container: Javax]
Provides group and data management endpoints through a REST interface.
Uses
Group Component
[Container: Bean and JDBC]
Provides group and data management.
JPA Component
[Container: JPA]
Contains all java classes and relationships which are mapped into database tables.
Account Endpoint Component
[Container: Javax]
Provides account management endpoints through a REST interface.
Uses
Account Component
[Container: Bean and JDBC]
Provides user management.
Reads from and writes to [SQL/JDBC, port 3306]
Reads from and writes to [SQL/JDBC, port 3306]
Reads from and writes to [SQL/JDBC, port 3306]
Relational Database
[Container: MySQL 5.7.x]

Diagram 4.3: Components Diagram shows up the RESTful API structure which has stateless beans, REST endpoints, an ORM layer and a logging component.

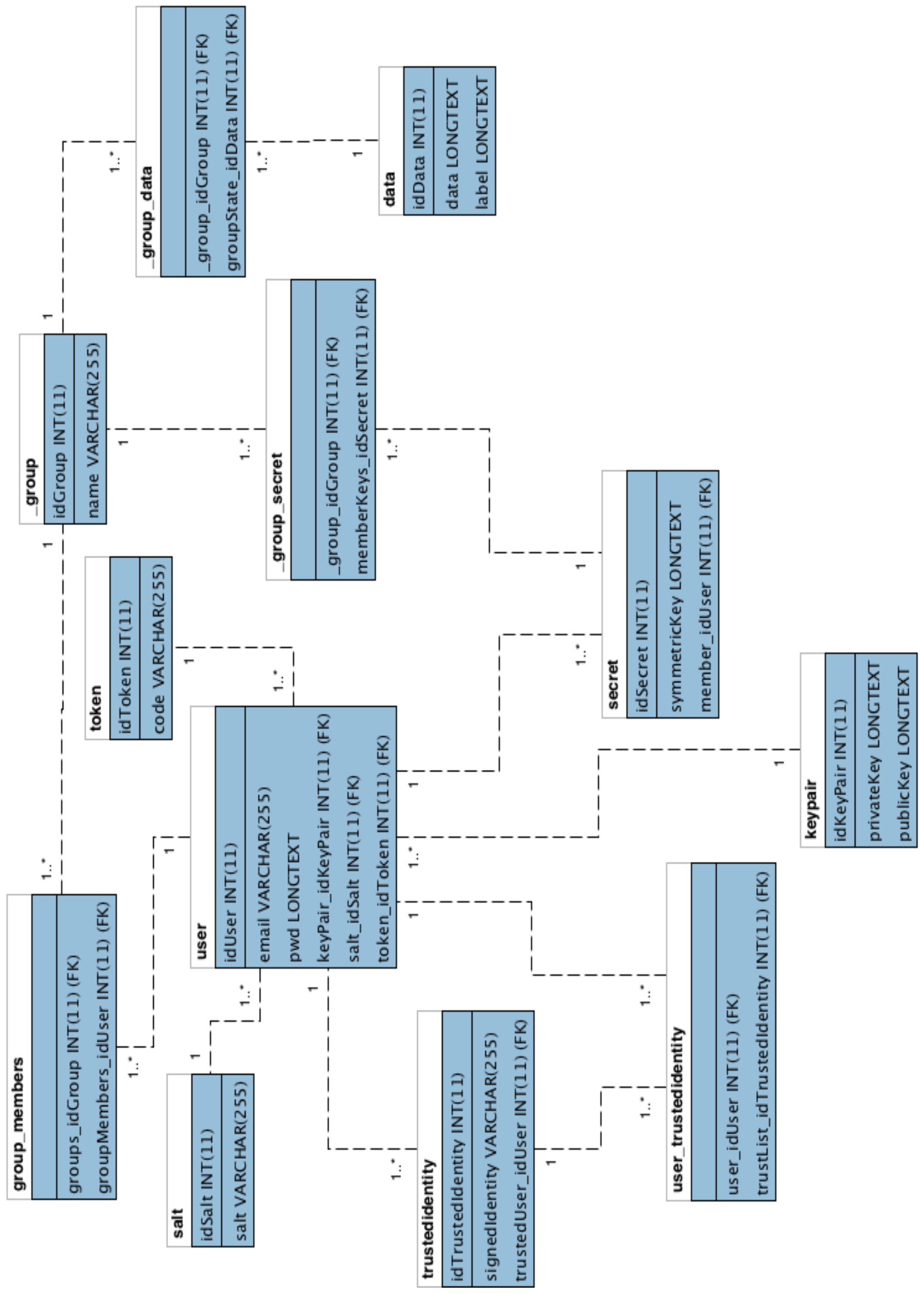


Diagram 4.4: Database Entity-Relationship (ER) Diagram.

Table 4.1: Group and Data Management Endpoints

	Verb	Endpoint	Header	Body
Create Group	POST	/v1/group	X-Auth-Token	name
Delete Group	DELETE	/v1/group	X-Auth-Token	idGroup
Get Group Key	POST	/v1/group/key	X-Auth-Token	idGroup
Update Group Key	PUT	/v1/group/key	X-Auth-Token	members, idGroup
Invite User	POST	/v1/group/member	X-Auth-Token	idGroup, email, groupKey
Kick User	DELETE	/v1/group/member	X-Auth-Token	idGroup, email
Get Members	POST	/v1/group/members	X-Auth-Token	idGroup
Persist Data	POST	/v1/group/data/add	X-Auth-Token	idGroup, label, data
Get Data	POST	/v1/group/data/get	X-Auth-Token	idGroup, label, data
Delete Data	DELETE	/v1/group/data/remove	X-Auth-Token	idGroup, label

Public Interface

Tables 4.2 and 4.1 describe all endpoints available in the public interface. They implement a *Representational State Transfer (REST) API* which allows *HTTP* requests through the four verbs of *HTTP* namely *get*, *post*, *put* and *delete*.

Both tables represent an API endpoint detailing the *HTTP* verb, the header and the body parameters. Furthermore, each endpoint is composed by a set of words, namely the first word for the API version, the second word for the scope and usually the last word for the action.

For instance, in Table 4.2, the register account procedure has the following route: */v1/account/register*. It uses the API version one, it is related with the account management scope, while the action is to register a *User*.

In addition, some endpoints have the X-Auth-Token as a mandatory header. This usually happens when the endpoint needs authentication. For instance, when it is strictly necessary the X-Auth-Token to proceed to the authentication of the *User*. Moreover, X-Auth-Token can be used to implement a session mechanism in the web applications.

To sum up, Table 4.2 details the Account Management Endpoints namely Register User, Login User, Delete User, User Info and Token Authentication. All endpoints are RESTful and in some cases is mandatory the X-Auth-Token for authentication proposes. The interface also supports group management and is mandatory to provide the header with the X-Auth-Token in all requests. Additionally, the scope of each endpoint is the group and it allows to Create and to Delete a Group, to Add a User to the group, to Remove a User from a group, and to Persist, Fetch or Delete Group Data.

In order to speed up the protocol integration with web applications, we developed an event based Javascript library which provides, through an interface, a set of primitives that are essential to develop and integrate the protocol. Technically, this library was implemented with *promises* along side *jQuery* which allows for asynchronous requests to the API.

In addition, the library is supported by *WebCrypto API* and *JQuery*. The former, *webCrypto API* implements the World Wide Web Consortium

Table 4.2: Account Management Endpoints

	Verb	Endpoint	Header	Body
Register User	POST	/v1/account/register	-	email, derived_pwdKey public key encrypted private key
Login User	POST	/v1/account/login	-	email, derived_pwdKey
Token Authentication	POST	/v1/account/token	X-Auth-Token	-
Account Information	GET	/v1/account	X-Auth-Token	-
User Information	GET	/v1/account/user	X-Auth-Token	email
Delete User	DELETE	/v1/account	X-Auth-Token	-
Trust User	POST	/v1/user/trust	X-Auth-Token	signature, email
Untrust User	POST	/v1/user/untrust	X-Auth-Token	email
Get Trust List	POST	/v1/user/trustlist	X-Auth-Token	-

(W3C) standard for in-browser cryptography, includes a set of cryptographic primitives and is a default library in most contemporary browsers. The latter, *JQuery* is an event based library which provides an API for HTTPS requests.

Regarding the architecture, the library is composed by multiple files, namely scope, functions, requests and *WebCrypto API* implementations. Firstly, the library implements its own runtime scope where it temporarily persists the user information, namely password, email, public and private keys. Secondly, the functions files include all procedures which implement all the features requests detailed in interaction diagrams in Chapter 3. Thirdly, the requests files import *JQuery API* and implement all requests to the *RESTful* API detailed in Section 4.1. Lastly, the library has multiple implementations of the cryptographic algorithms defined in Chapter 3 using *WebCrypto API.*

4.2 Proof of Concept

Nowadays, web chats are the most popular and used applications in the web, for instance WhatsApp, Facebook Messenger, Snapchat or Telegram, which connect millions of people through web and mobile applications. Consequently, we decided to implement an end-to-end web chat, named Hedwig, which employs the protocol implementation described in Chapter 4.

The requirements of this project are, mainly, to guarantee integrity, availability and confidentiality of the shared messages in group communications through the usage of the aforementioned protocol implementation.

Table 4.3 details the web chat features. ID stands for the numeric identification of the feature, Description for a proper feature description, and Priority for rating the feature as must-have or nice-to-have.

In addition, following the same headers definitions, Table 4.4 details the security and architectural concerns that the application should implement.

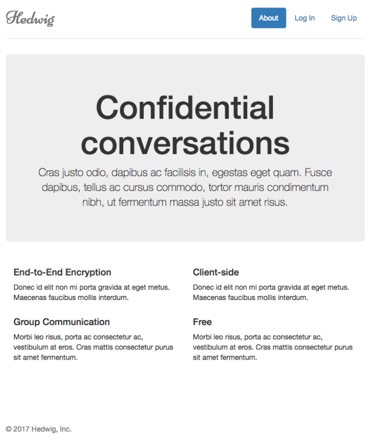
Table 4.3: web chat functional requirements

ID	Description	Priority
1	Allow a chatroom for each group communication.	MUST
2	Allow User to create an account by requesting the email address and the password.	MUST
3	Allow User authentication by requesting the email address and the password.	MUST
4	Allow User to send and receive messages from multiple groups.	MUST
5	Allow User identity verification.	MUST
6	Allow User to trust and to untrust any group member.	MUST
7	Allow User to create a group.	MUST
8	Allow User to delete a group where they are a member of and even if they are not the group founder.	MUST
9	Allow User to invite other members to a group where they are a member of.	MUST
10	Allow User to invite other members to a group where they are a member of.	MUST
11	Allow User to view their own key pair.	NICE

Technically, this web chat was implemented with *React-js* and *Bootstrap*. The former, *React-js*, is a Javascript library to build web interfaces and was developed by Facebook. The latter, *Boostrap*, is a Hyper Text Markup Language (HTML), Cascading Style Sheets (CSS) and Javascript (JS) framework to develop responsive web applications. Fig. 4.1 shows some user interfaces of Hedwig, namely the homepage and the group chat interface.

Table 4.4: web chat non-functional requirements

ID	Description	Priority
1	Information should be always encrypted and decrypted in client-side.	MUST
2	The application must provide confidentiality, integrity and availability of the share messages and keys.	MUST
3	Users should never receive any modified group messages.	MUST
4	All cryptographic mechanisms should run in background.	MUST
5	All API requests should run in background.	MUST
6	The group messages should always be delivered in casual order.	NICE



Confidential conversations

Cras justo odio, dapibus ac facilisis in, egestas eget quam. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus.

End-to-End Encryption

Donec id elit non mi porta gravida at eget metus. Maecenas faucibus mollis interdum.

Client-side

Donec id elit non mi porta gravida at eget metus. Maecenas faucibus mollis interdum.

Group Communication

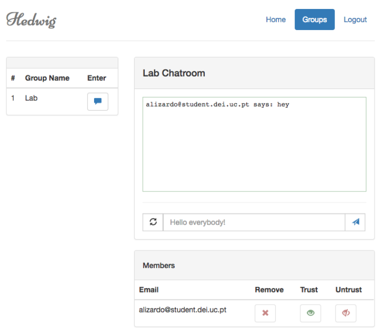
Morbi leo risus, porta ac consectetur ac, vestibulum at eros. Cras mattis consectetur purus sit amet fermentum.

Free

Morbi leo risus, porta ac consectetur ac, vestibulum at eros. Cras mattis consectetur purus sit amet fermentum.

© 2017 Hedwig, Inc.

(a) Homepage



#	Group Name	Enter
1	Lab	

```
alizardo@student.dei.uc.pt says: hey
```

Hello everybody!

Email	Remove	Trust	Untrust
alizardo@student.dei.uc.pt	✗		

(b) Chatroom

Figure 4.1: Hedwig web Application

Chapter 5

Performance Evaluation

"A distributed system is scalable if the cost of adding a user is a constant amount in terms of the resources that must be added."
[**pp. 33**, 10]

For this reason, we made design decisions in order to increase the protocol scalability. We focused the analysis on the most used and complex operations, namely *Persist and Fetch Data* and *Remove User From Group.* The former, *Persist and Fetch Data*, refers to the two most frequent operations. In the both cases, the overhead is getting the group state, $\{M\}_{K_s}$, and decrypting or encrypting it with the group key, Ks. The latter, *Remove User From Group*, is a complex operation as it runs the key distribution algorithm when someone leaves the group.

In order to test the scalability of the protocol, we planned two different experiments. On the one hand, we measure the execution time of the aforementioned primitives requests with payload encrypted with Sharelock. On the other hand, we measure the execution time of the same primitives with payload in plain text comparing both results.

The outliers were removed from the measured results by applying Equation (5.1).

$$Outlier = average(data) \pm 2 * standardDeviation(data) \tag{5.1}$$

The setup of this performance evaluation was a virtual machine which ran Wildfly (v10) and MySQL (v5.6), and a Macbook Pro. The former, virtual machine had 2 Intel(R) Xeon(R) CPU E5620 2.40GHz, 4 gigabyte of RAM and CentOS (v6.7) as the operative system. The latter, Macbook Pro specifications were retina 13-inch display, early 2015, processor 2,7 GHz Intel Core i5, 8 GB of DDR3 RAM with 1867 MHz and Intel Iris 6100 graphics card with 1536 MB of memory. In addition, the HTTPS requests were made through Chrome browser (v59) using jQuery and WebCryptoAPI. These components also communicated through a public network with 64ms of latency. The application server (Wildfly) was reset in each test in order to provide the same initial state. Finally, the times were measure in client-side through the High Resolution Time Level 2 public interface which provides time measurements in milliseconds since 01 January, 1970 UTC.

Persist Data

Persist Data is one of the most used operations, it encrypts the group states in client side and requests the server to persist them. We implemented six different tests where we tested separately the primitive performance using plain text payloads and payloads encrypted with Sharelock. Each test was executed one thousand times with different group state sizes, namely 8000, 80000 and 800000 bytes.

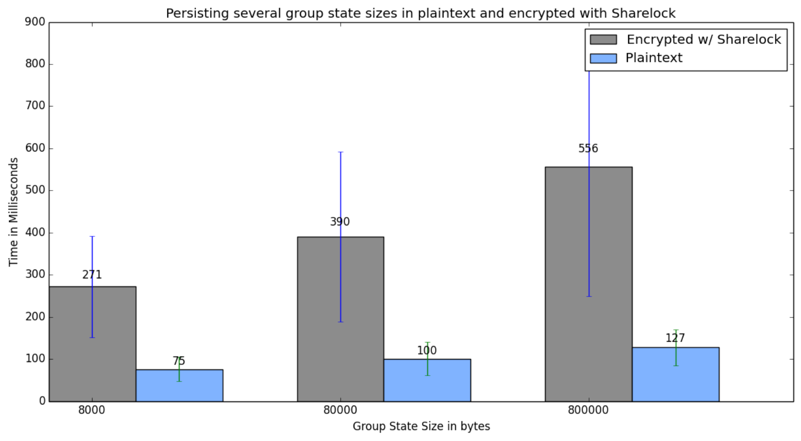


Figure 5.1: Persisting several group state sizes in plain text and encrypted with Sharelock.

Fig. 5.1 illustrates the average execution time for persisting several group state sizes in plain text and encrypted with Sharelock. It is clear that, there are two types of blocks on the figure. The requests with group state sizes encrypted with Sharelock is represented by the grey blocks, while the blue blocks represent requests with group state sizes in plain text. The figure also demonstrates that the requests with payloads encrypted through Sharelock cost more execution time than requests with payloads in plain text.

In addition, the figure shows that, in requests with payloads encrypted through Sharelock primitives, when the group state size grows, the execution time increases as well. Additionally, the execution time of requests with group states encrypted with Sharelock is over treble of the requests with group states in plaintext.

Although, the group key is a symmetric key (which enables faster cryptographic operations than asymmetric keys), the overhead of this operation is on encrypting the whole group state with the group key. As a consequence, this operation could be costly for the clients for large group states. The client will need more resources to encrypt the group size leading to a significant increase in client processing time.

The figure also illustrates that the average execution time for persisting group states in plain text varies around 25ms, while the average execution time in persisting encrypted group states with Sharelock varies around 150ms.

The highest variance of average execution time for persisting group states in plain text occurs when the group state grows from 80,000 to 800,000 bytes reporting an increase of 27%, while when the group state grows from 8,000 to 80,000 bytes, the average execution time is approximately 19%.

In addition, the highest variance of average execution time for persisting encrypted group state with Sharelock occurs when the group state grows from 8000 to 80000 bytes reporting an increase of approximately 44%, while when the group state grows from 80000 to 800000 bytes, the average execution time is approximately 43%.

Fetch Data

Fetch Data is one of the most used operations. It fetches and decrypts the group states in client-side using the group key. The experiment is the same as the *Persist Data*, we implemented six different tests where we tested separately the primitive performance using plain text payloads and payloads encrypted with Sharelock. Each test was executed one thousand times with different group state sizes, namely 8,000, 80,000 and 800,000 bytes.

Fig. 5.2 portrays the average execution time for fetching several group state sizes in plain text and encrypted with Sharelock. The figure also presents two different types of blocks, namely the blue block deals with requests made with plain text payloads and the grey block represents requests made with payloads encrypted through Sharelock primitives.

As shown in Fig. 5.1, the average execution time is bigger in requests with payloads encrypted with Sharelock than with payloads in plain text. However, the difference is under 20ms.

Notice that when the group state size grows, the client-side processing increases. Despite of the group key being a symmetric key, the processing time will increase with the size of the group state.

The figure also illustrates that the average execution time for persisting group states in plain text varies around 15ms, while the average execution time in persisting encrypted group states with Sharelock varies around 27ms.

The highest variance of average execution time for fetching group states in plain text occurs when the group state grows from 80,000 to 800,000 bytes

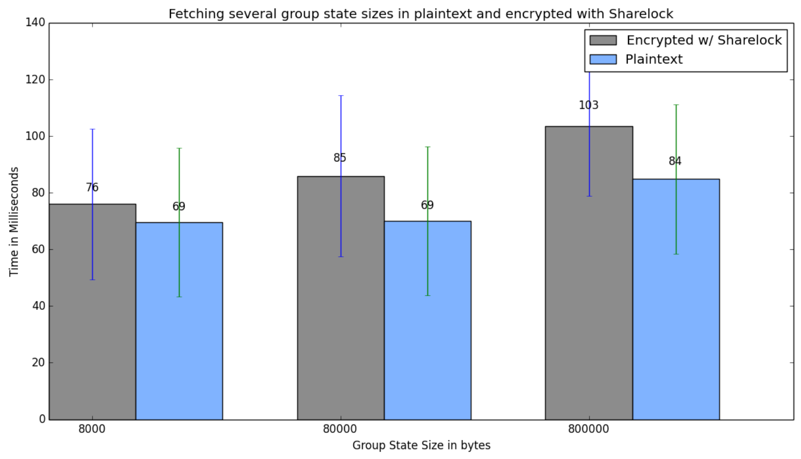


Figure 5.2: Fetching several group state sizes in plain text and encrypted with Sharelock.

reporting an increase of approximately 28%, while when the group state grows from 8,000 to 80,000 bytes, the average execution time is constant.

In addition, the highest variance of average execution time for fetching encrypted group state with Sharelock occurs when the group state grows from 80,000 to 800,000 bytes reporting an increase of 21%, while when the group state grows from 8,000 to 80,000 bytes, the average execution time is approximately 12%.

Remove User From Group

Remove User From Group is an extremely important feature as it represents the key distribution algorithm. Namely, when a *User* is removed from a group, the *User* who performed the action has to get all public keys of the group members, generate a new group key and spread the key to all group members by encrypting it with their public keys. Therefore, key distribution is extremely complex as it depends on the group size.

In terms of the experiment, we implemented six different tests where we tested separately the primitive performance using plain text requests and requests using Sharelock encryption mechanisms. The first test removed 1 group members, in a group size of 6 members. The second test removed 1 group members, in a group size of 36 members. The last test removed 1 group members, in a group size of 216 members. Each test was executed 250 times. Additionally, in each request, the same group member was removed separately and the group key was regenerated, encrypted and spread every time.

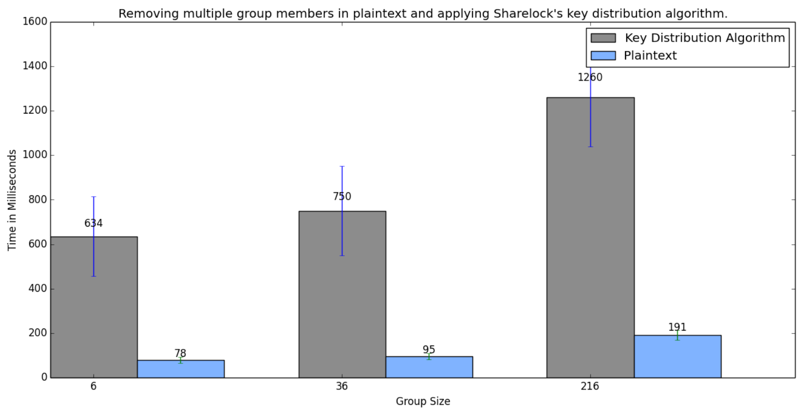


Figure 5.3: Removing several group members in plain text and applying Sharelock key distribution algorithm

Fig. 5.3 illustrates the average execution time for removing several group members in plain text and encrypted with Sharelock. The figure has two different types of blocks, namely the grey blocks represent the removal of group members using Sharelock key distribution algorithm and the blue blocks represent the removal of group members by sending plain text requests.

The figure also demonstrates that there is a dramatically growth of almost 800% of average execution time between requests with payloads in plain text and requests which applied the key distribution algorithm of Sharelock.

Notice that the average execution time for removing group members encrypted with Sharelock varies around 600ms. However, the highest variance occurs when the group size grows from 36 members to 216 members reporting an increase of almost 68% of average execution time, while when the group size grows from 6 members to 36 members, the average execution time rises 18%.

The figure also illustrates that the average execution time for removing group members in plain text varies around 113ms. However, the highest variance occurs when the group size grows from 36 members to 216 members reporting an increase of 101%, while when the group size grows from 6 members to 36 members, the average execution time rises approximately 22%.

Chapter 6

Conclusions

This work describes a security protocol, named Sharelock, which provides secure group communication through untrusted servers. In order to guarantee end-to-end encryption, the protocol shifts all cryptographic tasks to client side using symmetric encryption algorithms to encrypt and decrypt group states as it provides more performance than an arbitrary asymmetric encryption algorithm. Therefore, this design choice is undoubtedly the mechanism of choice to be used on the most frequent primitives. As a consequence, the key distribution problem is one of the main focus of the protocol.

Regarding the key distribution, before persisting a group state, the clients encrypt the group state with the group key, $\{M\}_{K_s}$. The group key was previously encrypted with their public keys, $[\{K_s\}_{K^+_{Bob}}, \{K_s\}_{K^+_{Alice}}, ...]$, and was signed, $[K_s]_{K^+_{Bob}}$, by the group founder or by the user who performed a remove user from group operation. Thus, the protocol meets the defined thesis goals, namely end-to-end encryption and access control.

In order to test the performance and the scalability, we developed a web implementation of the protocol. Therefore, we analysed the most used primitives (Persist and Fetch Data) and the most complex primitives (Remove User from Group).

In terms of the Persist and Fetch Data primitives, the client processing time increases with the size of the group state as they have to encrypt or decrypt it in each operation. Thus, it is expected that these primitives present a linear complexity. However, for large group states, the application developer can split the group state by using multiple scopes in the same group. For instance, the application developer can create a scope for each client, for blocks of time or create public and private scopes. By fetching and persisting smaller states, the user could speed up the entire process, reduce the network latency and the client-side processing. The results show that these primitives apparently follow a logarithm complexity. However, to prove this, it would be necessary to make more tests increasing the group state sizes.

In addition, it is expected that K_s remains stable for a long time as the remove user from group primitive is not very used. Therefore, the results suggest that the protocol could achieve linear scalability.

The cryptographic choices were made in order to achieve linear scalability as well. Therefore, Sharelock uses SHA-256 for hash functions, RSA-OEAP for asymmetric key generation, AES-GCM for symmetric key generation and PBKDF2 for cryptographic operations with passwords. In addition, these design and cryptographic choices also allow Sharelock to be adapted to other web systems (i.e, IoT) as they have low computational costs.

Sharelock is agnostic in relation to the information shared between the users by representing the group state by abstract objects which are converted to strings and persisted in the database. Therefore, the application developer is able to develop the desired data model for each group through the usage of these objects (which are similar to JSON objects).

After analysing the defined requirements and the final results, we can conclude that all requirements were met. Sharelock demonstrated that it is possible to have secure group communications through untrusted servers without compromising the system scalability. We believe that this definition may help future secure applications.

References

[1] Chris Alexander and Ian Goldberg. “Improved user authentication in off-the-record messaging”. In: *Proceedings of the 2007 ACM workshop on Privacy in electronic society*. ACM. 2007, pp. 41–47.

[2] Algirdas Avizienis et al. “Basic concepts and taxonomy of dependable and secure computing”. In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33.

[3] Alessandro Barenghi et al. “A privacy-preserving encrypted OSN with stateless server interaction: The Snake design”. In: *Computers & Security* 63 (2016), pp. 67–84.

[4] Alessandro Barenghi et al. “Snake: An end-to-end encrypted online social network”. In: *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on*. IEEE. 2014, pp. 763–770.

[5] Elaine Barker and Quynh Dang. *NIST Special Publication 800–57 Part 1, Revision 4*. 2016.

[6] Nikita Borisov, Ian Goldberg, and Eric Brewer. “Off-the-record communication, or, why not to use PGP”. In: *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*. ACM. 2004, pp. 77–84.

[7] Simon Brown. “Software Architecture for Developers”. In: *Coding the Architecture* (2013).

[8] J Callas et al. “RFC 4880 - OpenPGP Message Format”. In: *Internet: http://tools. ietf. org/html/rfc4880* (2007).

[9] Katriel Cohn-Gordon et al. “A Formal Security Analysis of the Signal Messaging Protocol.” In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 1013.

[10] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[11] John Cowley. *Communications and Networking: An Introduction*. 2012. ISBN: 978-1-84628-645-2.

[12] Tim Dierks. “The Transport Layer Security (TLS) Protocol version 1.2”. In: (2008).

[13] Tim Dierks and Eric Rescorla. “RFC 5246 - The Transport Layer Security (TLS) Protocol”. In: *The Internet Engineering Task Force* (2008).

[14] Roy Fielding et al. *RFC 2616 - Hypertext transfer protocol–HTTP/1.1.* Tech. rep. 1999.

[15] Simson Garfinkel. *PGP: pretty good privacy.* O’Reilly Media, Inc., 1995.

[16] Jakob Jakobsen and Claudio Orlandi. “On the CCA (in) security of MTProto”. In: *ePrint - International Association for Cryptologic Research* (2015).

[17] Calvin Li, Daniel Sanchez, and Sean Hua. *WhatsApp Security Paper Analysis.* Tech. rep. Massachusetts Institute of Technology, 2016.

[18] Eric Rescorla. “RFC 2818 - Http over tls”. In: (2000).

[19] Eric Rescorla and A Schiffman. “RFC 2660 - The Secure HyperText Transfer Protocol”. In: (1999).

[20] Gustavus J Simmons. “Symmetric and asymmetric encryption”. In: *ACM Computing Surveys (CSUR)* 11.4 (1979), pp. 305–330.