Developing mobile applications on top of a self-destruction system with data recovery

Meng Li
(Master's Program in Computer Science)

Advised by Yasushi Shinjo

Submitted to the Graduate School of
Systems and Information Engineering
in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering
at the
University of Tsukuba

March 2018

Abstract

Conventional mobile applications are built based on a client-server modal and require central servers for storing shared data and processing confidential information. If the central servers are accessed by an attacker, a curious administrator or a government, private information will be revealed because data is often stored on the central servers in the form of cleartext. This thesis presents Grouper, a framework for developing mobile applications without trusted central servers. With Grouper, it is easy to develop mobile applications on top of a self-destruction system with data recovery. Grouper provides object synchronization among mobile devices. It also uses a secret sharing scheme to create several shares from a marshalled object and uploads these shares to multiple untrusted servers. These untrusted servers construct a self-destruction system. Uploaded shares will be deleted after a certain period of time. Mobile devices exchange messages via untrusted servers based on the Grouper Message Protocol. Grouper consists of a client framework in Objective-C and a Web service in Java. We have implemented two applications using Grouper and evaluated the development efforts of them as well as the performance. Developing these applications demonstrates that Grouper requires little development effort to convert a standalone application to a data sharing application. Experimental results prove that the performance of Grouper is satisfactory for mobile applications that are used among a small group of people.

Contents

List of Figures

Chapter 1

Introduction

People use mobile applications every day. When users use conventional mobile applications based on a client-server mode, users fully trust the central servers. In fact, they are often unaware of their existence and simply rely on the functionality to be provided. If the central servers are accessed by an attacker, a curious administrator or a government, private information will be revealed because data is often stored on the central servers in the form of cleartext. In addition, users may lose their data when service providers shut their servers down.

To address the problem of using central servers, the proposals including Vanish[1], SafeVanish[2], SeDas[3] and CloudSky[4] constructed a data self-destruction system as their storage. In these approaches, servers store encrypted data temporarily and delete it after a certain period of time.

These existing approaches have several problems. First, they do not support data recovery when some nodes are unable to obtain data from shared storage. Often, application developers have to deal with such cases on their own. Second, these approaches do not provide support for developing mobile applications.

To address these problems, we develop such mobile applications on top of a self-destruction system with data recovery. Concretely, we implement Grouper, a framework for developing mobile applications using a self-destruction system[5]. Grouper provides object synchronization among mobile devices. In Grouper, a sender node translates an updated object into shares using a secret sharing scheme and uploads these shares to untrusted servers. A receiver node downloads some of these shares and reconstructs the object. The untrusted servers construct a self-destruction system, and delete these shares after a period of time. Unlike existing approaches, Grouper supports data recovery when some nodes are unable to obtain shares from untrusted servers. When a receiver node is unable to obtain shares, the Grouper framework automatically asks the sender to upload the missing shares again. This ensures reliable data sharing among the devices of a group. In addition, data can be recovered even if untrusted servers shut down because all devices of a group have the complete data set of this group.

Grouper consists of a client framework and a Web service. We have implemented the client framework for iOS, macOS, tvOS and watchOS in Objective-C. The Web service is implemented in Java and runs on multiple untrusted servers. We used the Sync[6] framework in Grouper to synchronize objects among mobile nodes. We have implemented two applications using Grouper: an iOS application named Account Book, and a benchmark application named Test. These implementations demonstrate that Grouper makes it easy to develop mobile applications with data synchronization. Experimental results prove that the performance of Grouper is satisfactory for mobile applications that are used among a small group of people.

The contributions of this paper are as follows. First, we provide support for data recovery when some nodes are unable to obtain data from untrusted servers. Grouper provides reliable data synchronization among nodes using a reliable multicast technique. Second, we make it easier to develop mobile applications. In fact, a developer can add data synchronization functions to stand alone applications with a few lines of code.

The rest of the paper is organized as follows. We present related work in Chapter 2. In Chapter 3, we describe the threat model and some assumptions in the Grouper framework. In Chapter 4, we describe the layered design of Grouper, and the way to exchange messages among devices using the Grouper Message protocol. In Chapter 5, we describe the three parts of the implementation of Grouper: the client framework, the Web service and applications. In Chapter 6, we give the evaluation of Grouper using developement efforts and performance. In Chapter 7, we conclude the paper and show the future work.

Chapter 2

Related Work

In this chapter, we show related work. Most of them use data encryption that requires private key generation and distribution to protect user data, and do not support data recovery.

2.1 Self-destruction Systems

We show self-destruction systems which keep data on servers temporarily in the following subsections.

2.1.1 Vanish

Vanish[1] is a self-destruction system that uses Distributed Hash Tables(DHTs) as the back-end storage. Vanish encrypts a message with a new random key, threads the key to shares using a secret sharing scheme, stores these shares in a public DHT, and eliminates the key from the local storage. The key in Vanish is removed after a certain period of time, and the encrypted message becomes permanently unreadable. Vanish is implemented with OpenDHT[7] or VuzeDHT[8] which is controlled by a single maintainer. Thus, it is not very secure against some Sybil attacks[9, 10]. In addition, the surviving time of the key in Vanish cannot be controlled by the applications for the user.

However, Vanish is not suitable for developing information sharing applications. Vanish is suitable for exchanging immutable messages, whereas our target applications need to modify objects repeatedly after other devices receive the object.

2.1.2 SafeVanish and SeDas

To address these issues in Vanish system, Zeng et al. proposed SafeVanish[2] and SeDas[3].

SafeVanish prevents "hopping attacks"[10], which is one kind of the Sybil attacks[9, 10] by extending the length range of key shares to increase the attack cost substantially, and do some improvement on the Shamir's secret sharing algorithm implemented in the Vanish system. The proposal of SafeVainish can only increase the cost of hoping attack, but cannot stop it.

Based on active storage framework like Oasis[11], SeDas proposes a distributed object-based storage system with self-destructing data function. Concretely, it exploits active storage networks instead of P2P nodes to maintain a divided secret key. In SeDas, user can specify the key survival time of distribution key and use the settings of expanded interface to export the life cycle of a key, allowing the user to control the subjective life-cycle of private data. The measurement and experimental security analysis of SeDas show the practicability of its approach.

Like Vanishe, SafeVanish and SeDas do not support the developent of information sharing applications.

2.1.3 CloudSky

Although, SeDas allows user to control the survival time of the key, it still cannot handle the cases that require the access control over the encrypted data including user grant and revocation. By extending SeDas, Zeng's group proposed CloudSky[4], a controllable data self-destruction system for untrusted cloud storage. CloudSky uses Attribute Bases Encryption (ABE) to implement the access control over the encrypted data, and allows the data owner to have user grant and revocation capabilities. CloudSky deals with the untrusted cloud storage network in two ways including the data self-destruction and the use of Hash-based Message Authentication Code (HMAC) [12], which ensures the data integrity.

CloudSky solves the problems regarding user controllability that exist in Vanish, but offline devices also cannot download messages after they have been removed from untrusted servers. In Grouper, offline devices can download such messages by using confirm and resend messages. A trusted authority is necessary in CloudSky to manage user profiles, whereas Grouper does not rely on any trusted authority.

2.2 Using Untrusted Servers with Data Encryption

We show systems that use data encryption in this section. These systems encrypt user data and save the encrypted data in untrusted servers.

2.2.1 Mylar

Mylar[13] stores encrypted data on an untrusted server and decrypts this data only in the browsers of users. Application developers of Mylar use its application program interface (API) to encrypt regular (non-encrypted) Web applications. Mylar uses its browser extension to decrypt data on the client-side. Mylar also supports keyword searching in Web applications by computing over encrypted data and JavaScript code verification in Web broswers.

Mylar enables client-side computation on data received from servers. In addition, Mylar still requires a trusted identity provider (IDP). Compared to Mylar, which uses a single untrusted server, Grouper takes advantages of the data redundancy provided by the secret sharing scheme by using multiple untrusted servers and does not require any trusted IDP.

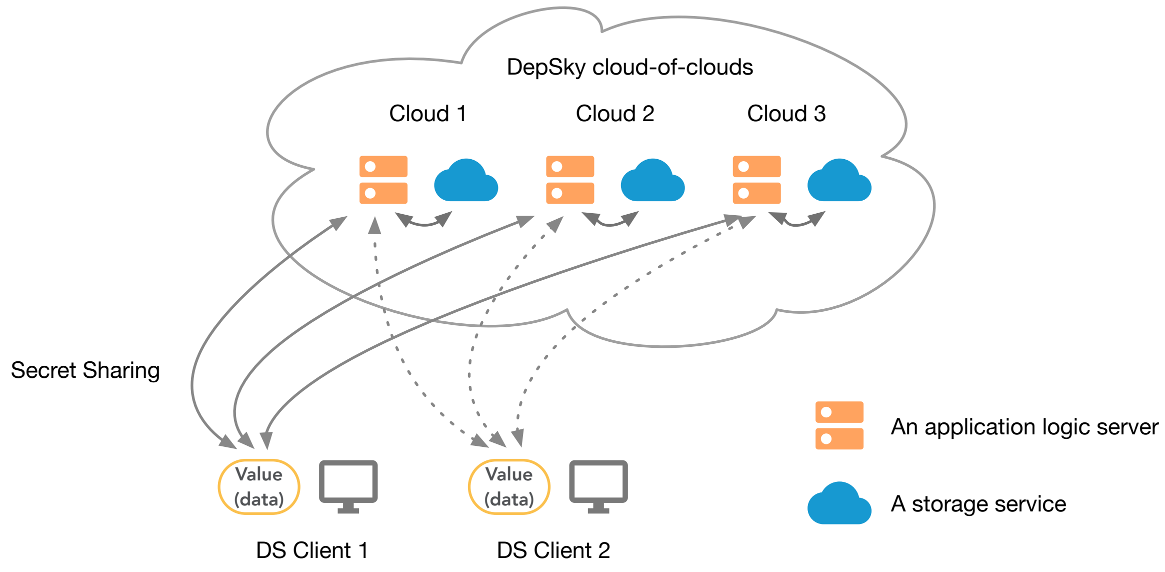


Figure 2.1: Architecture of DepSky.

2.2.2 DepSky

DepSky[14] is a system that keeps encrypted data in commercial storage services and performs application logic in other individual servers. It implements the concept of *Cloud-of-Clouds*. *Cloud-of-Clouds* is the core concept in DepSky. It represents that DepSky is a virtual storage, and its users invoke operations in several individual severs. Figure 2.1 shows the architecture of DepSky. When a client wants to share data with another client, it generates several shares and uploads these shares to the application logic servers on the left side. Application logic servers encrypt and save shares into commercial storage services on the right side via the APIs provided by the storage provider. By using commercial storage services, DepSky improves the availability and confidentiality of private information.

However, commercial storage services may not supports temporary data storage like the self-destruction systems. In Grouper, untrusted servers undertake the responsibility of temporary data storage and message delivery. Mobile devices perform application logic.

2.2.3 SPORC

SPORC[15] is designed for developing group collaboration applications using untrusted cloud resources. In SPORC, a server observers only encrypted data and cannot deviate from correct execution without being detected. From a conceptual distributed systems perspective, SPORC demonstrates the benefit if combing *operational transformation* (OT)[16] and *fork* consistency* protocols[17]. To maintain synchronization among clients, SPORC uses OT, which allows each client to apply local updates optimistically. To solve the problem that the server causes the views of two clients to diverge, SPROC uses the *fork* consistency* protocol, which realizes a consistency model for interacting with an untrusted server.

In SPORC, synchronization should be done among online nodes, while Grouper allows synchronization among offline nodes.

2.3 Peer-to-peer System

In this secton, we show an application Sweets[18] using a peer-to-peer (P2P) technique. Sweets is a decentralized social networking service (SNS) application on Android devices that uses data synchronization with P2P connections among mobile devices. Sweets performs data synchronization not only between two online nodes directly but also via common friend nodes indirectly for offline nodes. This indirect synchronization uses ABE to provide access control. Consequently, Sweets provides data confidentiality and availability without permanently available centralized storage servers. Sweets applys OpenID Connect[19] to provide decentralized user authentication and implements a self-issued OpenID provider[20] that runs on users devices. Sweets implements indirect replication via a hybrid cryptographic mechanism that combines ABE and AES encryption schemes. Taking the advantage of ABE, protected data can be hosted and relayed by users who have no permission to access it. In addition, by combining ABE and AES, Sweets can void expensive and repetitive ABE operations.

Both Grouper and Sweets provide a way to develop mobile application without central servers. However, direct synchronization in Sweets can only be done when two devices are online at the same time. On the other hand, in Grouper, users can synchronize data from multiple untrusted servers, anytime.

Chapter 3

Threat Model

In this chapter, we describe the assumptions and the threat model underlying the Grouper framework.

We target mobile applications that are used among a small group of people. A group consists of a special member called *the owner* and other members. Each member has a mobile device. Only the group owner can invite other members in a face-to-face manner. In this group, each user creates a new object and shares it with other grouper members. When a user receives a new object from another user, he/she can edit the content of the object or delete the object. The modification or deletion also sepreads to the devices of other group members.

Our target groups have sensitive shared data. Those groups must prevent access to shared data by other parties including server managers.

To implement such mobile applications, we assume the following assumptions in Grouper.

First, because we use servers, they are passive adversaries and can read all of the data but they do not actively attack. Servers host Web services and perform device authentication. Servers generate access keys for group members. When a device wants to get/put data from/to servers, the device sends a request with an access key. In this paper, we do not address other types of attacks such as user tracking and metadata collection by servers. For example, servers can track users with IP addresses, and Grouper cannot hide social graphs against such tracking.

Second, data transportation between a device and an untrusted server is secure. We can protect it using Transport Layer Security (TLS) or other encryption techniques. Grouper focuses on the privacy of the data storage on servers rather than on data transportation.

Third, in an application, all group members are not malicious and their devices connect to each other securely at a face-to-face distance at the time of user invitation. For example, group members working in an office know one another and are not malicious. When the group owner invites new members, the owner authenticates group members in a face-to-face manner. Note that after user invitation, devices communicate through servers and no secure, direct communication path is required.

Last, servers are isolated from one another and managed by independent providers. We assume that providers of untrusted servers do not expose user data to other providers. For example, a group owner can leverage the servers of Amazon, Google, and Microsoft, which are not supposed to expose user data to other cloud providers.

Chapter 4

Design

This chapter describes the design of the Grouper framework.

4.1 Overview

Our goal is to support the development of mobile applications that do not rely on trusted central servers.

To achieve this goal, we are developing the Grouper framework. This framework provides the following functions:

- **Data Synchronization.** If an user updates or deletes an object in his/her device, the mirrors of this object in other devices are updated or deleted.
- **Group management.** A group owner can create a group and invite other members to his/her group.

For example, *Account Book* is an iOS application developed using Grouper. In this application, a leader of a small company can create a group and invite employees to join the group. Then, the employees can record the income and expenditure of their company. These income and expenditure records are represented as objects and shared among devices. Anyone can edit and delete existing records in his/her device.

Grouper uses untrusted servers to exchange messages among mobile devices. Untrusted servers construct a self-destruction system, and delete messages after a certain period of time. We refer to this as the Time to Live (TTL).

Grouper uses Shamir's secret sharing scheme to protect messages from the providers of untrusted servers. In this scheme, a member securely shares a secret with other members by generating n shares using a cryptographic function[21]. At least k or more shares can reconstruct the secret, but $k-1$ or fewer shares output nothing about the secret[22]. We describe this scheme as a function $f(k,n)$, where n is the number of shares, and k is the threshold to combine shares.

Grouper has the following advantages over conventional approaches using a self-destruction system. First, it is easy for a developer to recover from message losses in untrusted servers,

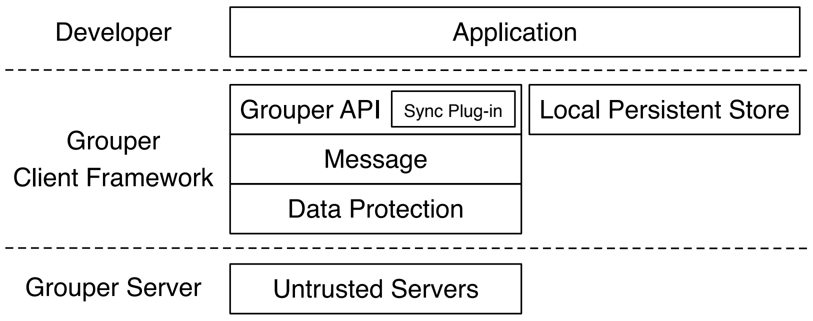


Figure 4.1: Architecture of Grouper.

as demonstrated in Section 4.4. Grouper performs retransmission when some mobile devices miss getting messages from untrusted servers. Developers of mobile applications do not have to specify the lifetimes of messages. Second, it is easy for a group owner to invite other members using a safe communication channel at a face-to-face distance, as discussed in Section 4.7.

4.2 Architecture

Figure 4.1 describes the architecture of the Grouper framework. Grouper consists of the following four layers and a plugin:

- **Grouper API.** A developer develops an application without data synchronization at first. He/she adds the synchronization function to his/her application by using the API.
- **Synchronization Plugin.** Grouper uses a third-party framework for data synchronization. This plugin marshalls an updated object in a local persistent store and the Grouper API layer passes the marshalled message to the lower message layer. When the Grouper API layer receives a message, this layer unmarshalls the message using the plugin, reconstructs the message, and puts the object included in the message into the persistent store.
- **Message.** This layer provides a messaging transportation service with multicasting capability among devices. The destination of a message is not only the node identifier (ID) of a single device, but also "*", which means it is delivered to all the other nodes. This layer does not ensure message delivery to other devices.
- **Data Protection.** Grouper protects user data by a secret sharing scheme in this layer. This layer divides a message into several shares, and uploads these shares to untrusted servers. When this layer downloads shares from untrusted servers, it recovers the original message using the secret sharing scheme.
- **Untrusted servers.** When a mobile device uploads a share to an untrusted server, this server receives it and stores it to a database. When a mobile device downloads a share from an untrusted server, this server retrieves it from the database and sends it into the device. An untrusted server performs device authentication using device keys.

Table 4.1: Client API of Grouper.

Method	**Description**
setup(*appId*, *entities*, *dataStack*)	An application invokes this method to initialize Grouper with *appId*, *entities* and *dataStack*. The parameter *appId* is the unique ID of an application. The parameter *entities* is an array which contains the names of all entities in this application. The developer should pay attention to the order of entities in the array. If the entity A is referred to by the entity B, A should be in front of B. The parameter datastack is used by the synchronization plugin.
sender.**update**(*object*)	An application invokes this method after creating a new object or modifying an existing object. Grouper performs updates asynchronously and sends an update message to other devices.
sender.**delete**(*object*)	An application invokes this method when it wants to delete an existing object. Grouper deletes the object and removes it from the persistent store of a device automatically, after sending a delete message to other devices.
receiver.**receive**(*callback*)	An application invokes this method to register the callback function that is called after Grouper processes the received messages and updates objects according to the messages. The application can use this callback function to update the user screen.
sender.**confirm**()	An application needs to invoke this method periodically or occasionally to send a confirm message to other devices.

The following sections describe the details of these layers from the top layer to the bottom layer.

4.3 Grouper API

The Grouper framework provides object synchronization among mobile devices through the simple client APIs.

4.3.1 Methods of Grouper API

Table 4.1 shows the client API of Grouper that is used to develop mobile applications. An application initializes the framework by invoking the method *setup*(). When the appli-

Table 4.2: API of the synchronization plugin.

Method	**Description**
marshall(o)	Marshalls the object o and returns the marshalled byte array.
updateRemote(b)	Unmarshalls the byte array b to the object and puts the object into the persistent store.
deleteRemote(b)	Deletes the object of the object ID in the byte array b.

cation needs to update an object in all devices, it invokes the method *sender.update*(). When the application needs to delete an object in all devices, it invokes the method *sender.delete*(). When the application needs to receive update notifications, it invokes the method *receiver.receive*() with a callback function. This callback function is called when another node updates an object and its local mirror has been updated. The application can use this callback function to change the values that are shown on the user interface screen. The method *sender.confirm*() is used for realizing reliable messaging. We will describe reliable messaging in Section 4.4.

These methods should be used after initializing the Grouper framework in the application's project. We will show the usage of these methods in Section 5.3.

4.3.2 Synchronization Plugin

The Grouper API layer relies on the synchronization plugin. As described in Table 4.2, the synchronization plugin should provide the functions *marshall*(o), *updateRemote*(b) and *deleteRemote*(b). Grouper invokes these functions to obtain marshalled data from the persistent store, save unmarshalled data in the persistent store, and delete objects in the persistent store.

We have not implemented the synchronization plugin on our own, but we provide it as a pluggable module. This is because there are many such synchronization modules that provide various features. Additionally, application requirements may also vary. Each application developer should choose a suitable module based on a consistency model and other requirements.

Because we implement our client framework in Objective-C, we currently use the Sync framework[6] to implement the data synchronization plugin. The Sync framework marshalls objects into JavaScript Object Notation (JSON) strings and provides a consistency model where the newest version wins.

4.4 Reliable Message Delivery

Grouper implements reliable message delivery in the Grouper API layer over the self-destruction system. In this section, we show an existing technique to solve the reliable message delivery problem called basic reliable multicasting at first. Then we describe our own reliable message delivery protocol based on basic reliable multicasting technique.

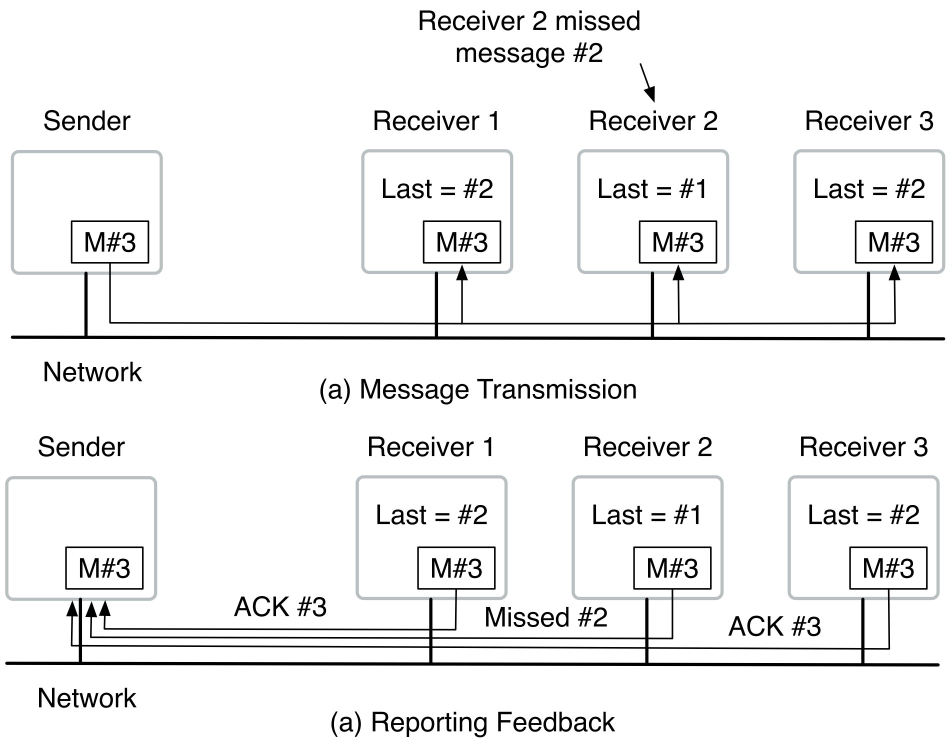


Figure 4.2: Basic reliable multicasting.

4.4.1 Basic Reliable Multicasting

Figure 4.2 shows basic reliable multicasting technique in a text book[23]. In this reliable multicasting technique, each message has a sequence number for each sender as shown in Figure 4.2. Each member keeps the newest sequence numbers for senders and detects missing messages. If a member notices a missing message from a sender, the member asks the sender to resend the message. For example, consider that a sender sends message No.3 to all other members using a multicast address. When a member receives message No.3, the member compares the sequence number 3 with the newest sequence number of the sender. If the newest sequence number of the sender is 1, this means the member missed message No.2. The member asks the sender to resend message No.2 using a control message. The sender will send message No.2 to the member who request it using a unicast address.

4.4.2 Data Recovery on Self-destruction System

This basic reliable multicasting technique works well for continuous media, such as video streaming in Internet communication. However, it does not work well if receivers go offline often or for a long time, as our untrusted servers delete messages within a short time. Therefore, the basic reliable multicasting technique is not suitable for data recovery on our self-destruction system.

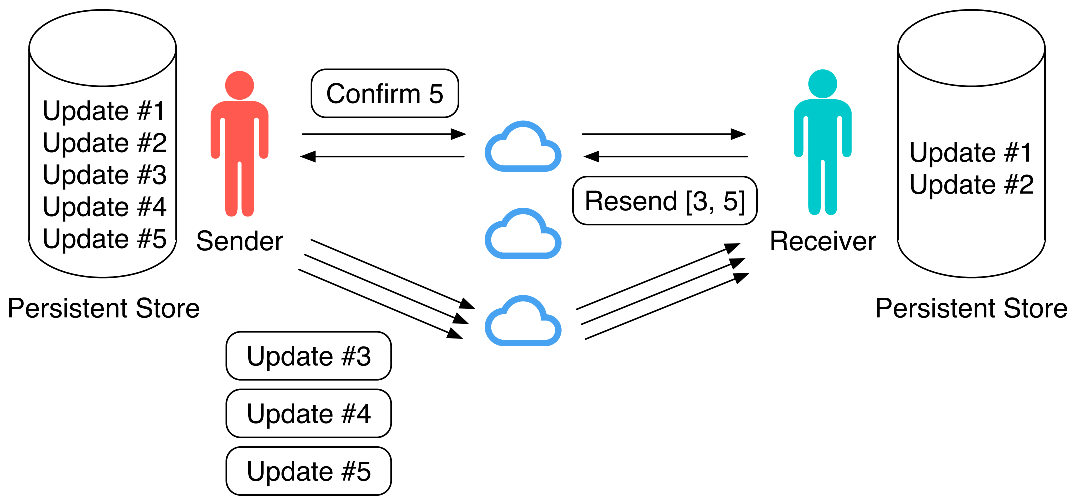


Figure 4.3: Implementing reliable messaging with continuous sequence numbers.

To address this problem, we extend the basic reliable multicasting technique. We use a special type of message that includes active sequence numbers. Using these messages, a receiver can easily identify missing messages. We call these type of messages *confirm messages*.

Figure 4.3 shows that the sender is sending five update messages, from No.1 to No.5, to the receiver. The sender uploads two messages, No.1 and No.2, using a multicast address. The receiver downloads these two messages and goes offline. The servers delete these two messages. The sender uploads the next three update messages, from No.3 to No.5, using a multicast address. Then, the servers delete these three messages again. The receiver comes online, but the receiver does not notice that these three messages are missing.

At this time, the sender sends a confirm message that includes the newest active sequence number, 5, as shown in Figure 4.3. The receiver receives this sequence number and compares it with the newest sequence number in the persistent store. In Figure 4.3, the receiver notices that update messages No.3, No.4, and No.5 are missing. The receiver asks the sender to resend update messages No.3, No.4, and No.5 using a resend message. The sender will again send update messages No.3, No. 4, and No.5 to the receiver using a unicast address.

This idea is inspired by the *checkgroups* message of Usenet[24]. In Usenet, the list of active newsgroups is maintained with two basic messages: *newgroup* and *rmgroup*. When a node receives a newgroup message, the node adds the newsgroup to the list. When a node receives a rmgroup message, the node removes the newsgroup from the list. However, these basic messages can be lost and the list can become obsolete. Checkgroups messages supplement these basic messages. A checkgroups message includes the list of all newsgroups in a newsgroup hierarchy. A checkgroups message is distributed periodically or after some time after the basic messages are distributed.

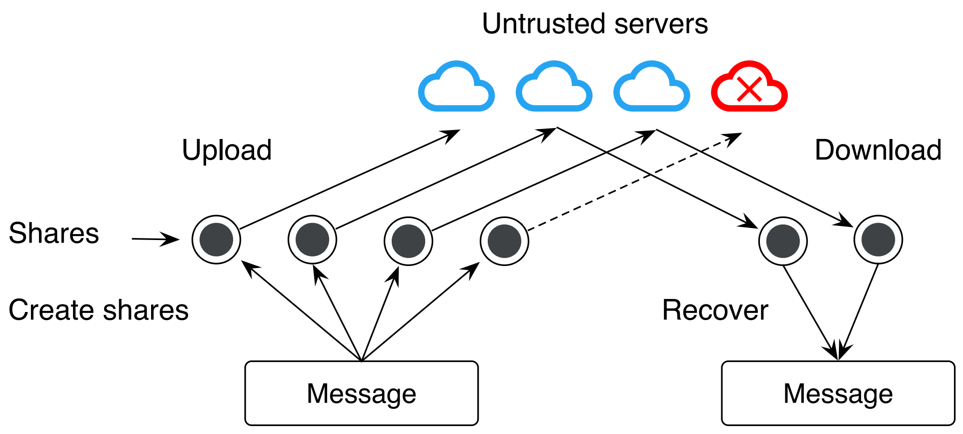


Figure 4.4: Message transportation using the extended secret sharing scheme $f(2,4,3)$.

4.5 Data Protection

We design an extend secret sharing scheme and data transportation flow to protect user data.

The data protection layer uses Shamir's secret sharing scheme $f(k,n)$ as shown in Section 4.1. In Grouper, we extend this scheme and design a new scheme $f(k,n,s)$. In our scheme, the parameters k and n are same as those in the scheme $f(k,n)$. The parameter s represents the minimum number of untrusted servers when a sender uploads shares, where $k \leq s \leq n$. Although a receiver is able to recover the original message from at least k shares, we should also consider the scenario in which the server crashes. When a sender has n shares, Grouper tries to upload these n shares to all n untrusted servers, at first. If the shares are uploaded to s or more untrusted servers, we consider that this upload to be successful. Otherwise, Grouper continues trying to upload these shares.

Figure 4.4 shows the message transportation from a device to another device using the extended secret sharing scheme $f(2,4,3)$. In this figure, the message layer in a sender device is sending a message to the message layer in a receiver device. At first, this layer calls the data protection layer and creates four shares, according to the secret sharing scheme. Next, this layer uploads those shares to four untrusted servers. In Figure 4.4, although only the three untrusted servers on the left receives shares successfully, we regard this upload as successful, because we set $s = 3$ in the $f(k,n,s)$ scheme. Finally, the message layer in the receiver device downloads two shares from the four untrusted servers and recovers the message by calling the data protection layer.

We can choose the parameter s based on the following policies.

- **Sender first.** An application sets s close to k. A sender receives a successful result earlier. When some servers are not available, it is more likely that a receiver will lose a message.
- **Receiver first.** An application sets s close to n. A sender must continue attempting to obtain a successful result. When some servers are not available, a receiver will be

Table 4.3: Attributes of the Grouper message.

Attribute	**Explanation**
type	Type of this message.
content	A marshalled object or sequence numbers.
sequence	Sequence number of the message.
class	Class name of an object.
objectId	ID of an object.
receiverId	Node identifier of the receiver.
senderId	Node identifier of the sender.
email	Email address of the sender.
name	Name of the sender.
sentTime	Time the message is sent.

more likely to receive a message.

Compared to the traditional secret sharing scheme, the extended secret sharing scheme allows developers to tune the performance of uploading.

Compared with data encryption methods, the secret sharing scheme has the following advantages. First, using a secret sharing scheme does not require key management, including generation and distribution. Key generation and distribution usually need a central identity provider, which is against with the fully untrusted concept in Grouper. Second, the secret sharing scheme ensures the data availability when a number of untrusted servers are not accessible. For the $f(k, n, s)$ scheme in Grouper, a sender can complete uploading when at least s untrusted servers are available, and a receiver can recover the object when at least k untrusted servers are available. Third, the secret sharing scheme reduces the risk of an attack because an attacker who can only access only $k-1$ or less untrusted servers cannot obtain any information.

4.6 Grouper Message Protocol

The Grouper API layer sends and receives messages using our own protocol, the Grouper Message Protocol. In this protocol, a message is a JSON string that contains a marshalled object and its attributes.

4.6.1 Attributes of Grouper Message

Table 4.3 shows the attributes in a Grouper message. Figure 4.5 shows an example of an update message. There are three important attributes:

- **Type.** This attribute refers to the message types. There are four types of messages: update messages, delete messages, confirm messages, and resend messages. An update message contains the marshalled objects of an application. A delete message contains

```
{
    "object" : "Test",
    "content" : "{\"content\":\"2017-09-2805:04:23+0000\",\"updator\":\"07DF1068-815C-4C70-B833-A6012B8BE846\",\"creator\":\"07DF1068-815C-4C70-B833-A6012B8BE846\",\"id\":\"A4ABC905-B6CE-4BA1-9224-F5AE7D0F0300\",\"update_at\":\"2017-09-28T14:04:23+09:00\",\"create_at\":\"2017-09-28T14:04:23+09:00\"}",
    "sender_id" : "07DF1068-815C-4C70-B833-A6012B8BE846",
    "receiver_id" : "*",
    "message_id" : "CEA7050B-CD14-4863-A127-A2C3FF74FBEC",
    "sequence" : 3,
    "email" : "test@test",
    "object_id" : "A4ABC905-B6CE-4BA1-9224-F5AE7D0F0300",
    "type" : "update",
    "name" : "test",
    "send_time" : 1506575063
}
```

Figure 4.5: An update message.

the identifier of a deleted object. A confirm message contains the maximun sequence number of all update and delete messages created in a device. A resend message contains the sequence numbers of missing messages. We call update messages and delete messages *normal messages*. Both confirm messages and resend messages contain control information about a reliable multicast. We call these messages *control messages*.

- **Content.** If the message is an update message, the content value contains the JSON string of a marshalled object. If the message is a delete message, the content value contains the objectId of an object. If the message is a confirm message, the content value contains the maximum sequence number of messages created in the device of the sender. If the message is a resend message, the content value contains the range of missing message sequence numbers.
- **Sequence.** This attribute refers to the sequence number of a message. When a sender sends a new normal message, the sender increments the sequence number and includes it in the new message. The sequence number of any control message is 0.

The receiverId attribute contains the addresses of destination devices. An address is either a list of device IDs or ”*”, which signifies multicasting to all devices.

Applications send update, delete and confirm messages through the Grouper API, and send resend messages after receiving confirm messages automatically. When an application invokes the method $grouper.sender.update()$, the Grouper API layer sends an update message that contains the marshalled object to all devices. When an application invokes the method $grouper.sender.delete()$, the Grouper API layer sends a delete message that contains the ID of the deleted object to all devices. When an application invokes the method $grouper.confirm()$, the Grouper API layer sends a confirm message to all devices. The confirm message includes the sequence numbers of objects that were recently created in the device.

The method $grouper.confirm()$ is invoked in the following situations:

- **Periodically.** For example, an application sends a confirm message once during the

TTL.

- **When the device becomes online.** Sometimes, a device is offline and cannot send a confirm message within the TTL. The device therefore sends a confirm message when the device becomes online.

4.6.2 Algorithm to Handle Grouper Message

Algorithm 1 describes the handle process when the Grouper API layer receives a message. For an update or delete message, Grouper invokes the method *sync.updateRemote*() or *sync.deleteRemote*() of the synchronization plugin to update the persistent store. For a confirm message, the Grouper API layer copies the maximun sequence number from the message content. Next, the Grouper API layer creates a resend message that contains the missing sequence numbers and sends it to the sender of the confirm message. For resend messages, the Grouper API layer retrieves the sequence numbers from the resend message, finds the corresponding normal messages, and sends them to the sender of the resend message.

4.7 Group Management

To manage a group, Grouper provides the following two functions: group creation and member invitation.

4.7.1 Group Creation

A user can create a group, and the creator becomes the owner of the group. Before creating a group, the owner prepares his/her own user information (including his/her email address and name), multiple untrusted servers, a group ID, and a group name. Next, the owner initializes the group on all untrusted servers by submitting his/her node identifier and the TTL to multiple untrusted servers. The node identifier, which represents his/her device, is generated by Grouper randomly when the application is launched for the first time. On each untrusted server, the Web service initializes this new group and returns a master key including the highest privilege to the owner. The owner can add other members to an untrusted server using the master key.

4.7.2 Member Invitation

After creating a group, the owner can invite new members to his/her group as shown in Figure 4.6. To join the group, a new member first prepares his/her user information. The owner invites the new member in a face-to-face manner, rather than using central servers. At this time, Grouper establishes a connection between the devices using a local safe communication channel like *Multipeer Connectivity*[25]. First, the new member sends his/her user information and a node identifier to the owner. The owner saves the user information and node identifier to his/her device. Second, the owner registers the new member to the multiple untrusted servers by submitting the node identifier of the new

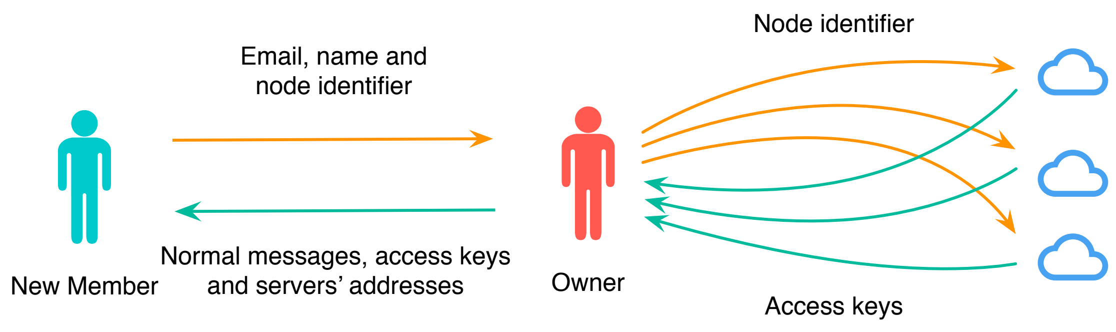

Node identifier
Email, name and node identifier
New Member
Normal messages, access keys and servers' addresses
Owner
Access keys

Figure 4.6: Member invitation.

member. Third, the untrusted servers return access keys for the new member to the owner. Last, the owner sends the access keys, the addresses of the untrusted servers, the list of existing members and all normal messages in his/her device to the new member. After receiving them, the new member holds all data in the devie of group member and is able to access the untrusted servers with the keys.

Algorithm 1 Message handling algorithm

```
1: procedure onMessageReceived($msg$, $sender$)
     ▷ Check duplicate message.
2:     $historyMsgs \leftarrow getMsgsBySender(msg.sender)$
3:     if $msg \in historyMsgs$ then
4:         return
5:     end if
6:     $lastMsg \leftarrow historyMsgs.last()$
7:     $historyMsgs.add(msg)$
     ▷ Basic reliable multicast.
8:     if $msg.seq \neq 0$ && $lastMsg.seq + 1 \neq msg.seq$ then
9:         $resendMsg \leftarrow createResendMsg(lastMsg.seq +$
10:                $1,\ msg.seq)$
                                   ▷ Create a resend message with maximum sequence number.
11:        $sendMsg(resendMsg,\ sender)$
12:    end if
     ▷ Handle the message by its type.
13:    if $msg.type =$ "$update$" then
14:        $sync.updateRemote(msg)$
15:    else if $msg.type =$ "$delete$" then
16:        $sync.deleteRemote(msg)$
17:    else if $msg.type =$ "$confirm$" then
18:        $maxSeq = getMaxSeqFrom(msg.content)$
19:        $resendMsg \leftarrow createResendMsg(lastMsg.seq +$
20:                $1,\ maxSeq)$
21:        if $resendMsg \neq null$ then
22:            $sendMsg(resendMsg,\ sender)$
23:        end if
24:    else if $msg.type =$ "$resend$" then
25:        $seqs \leftarrow getSeqs(msg.content)$
26:        for $seq \in seqs$ do
27:            $missingMsg \leftarrow getMsg(seq)$
28:            $sendMsg(missingMsg,\ sender)$
29:        end for
30:    end if
31: end procedure
```

Chapter 5

Implementation

Grouper consists of a client framework for developing mobile applications and a Web service running on multiple untrusted servers. We describe the implementation of the implementation of the Web service in Section 5.1, the client framework in Section 5.2, and demo applications in Section 5.3.

5.1 Web Service

We have chosen to implement our own Web service rather than using commercial general cloud storage services like Amazon Simple Storage Service (S3), Google Cloud Storage, or Microsoft Azure Storage for the following reasons:

- The Web service must support on the Grouper Message protocol.
- The Web service must delete shares after a prescribed time.

The server side of the Grouper framework consists of the Web application server, database and remote notication pushing server. Figure 5.1 shows the relationship among the clients, Web service, database and remote notification pushing server. A clients sends shares to an untrusted server via the RESTful API provided by the Web service. The Web service saves the share to the MySQL database. Then, it sends a message that contains the notification content and the device tokens of target devices to the remote notification pushing server. Finally, the remote notification pushing server notifies the clients to receive messages from the untrusted servers.

5.1.1 Application Servers, Databases and External Libraries and Frameworks

Our Web service provides a RESTful API to clients. It runs on a Tomcat server[26] that is an open-source implementation of the Java Servlet, JavaServer Pages, Java Expression Language, and Java WebSocket technologies. Our Web service use the MySQL[27] database as the backend storage.

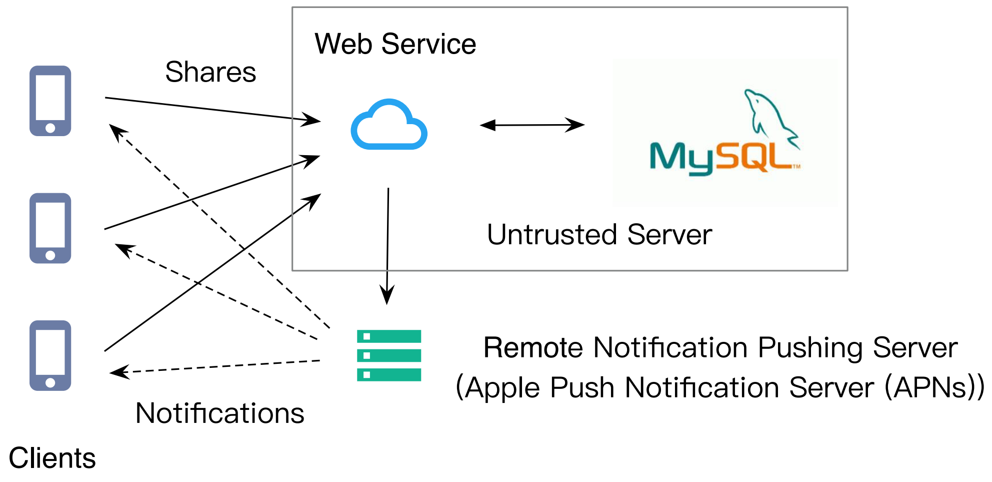

Web Service
Shares
MySQL
Untrusted Server
Remote Notification Pushing Server
(Apple Push Notification Server (APNs))
Notifications
Clients

Figure 5.1: Relationship among the clients, Web service, database and remote notification pushing server.

We use the following frameworks in Java Enterprise Edition (Java EE) to implement our Web Service.

- **Spring**[28], a framework that provides a comprehensive programming and configuration model for modern Java-based enterprise applications. Grouper uses it for dependency injection, a technique whereby one object supplies the dependencies of another object, and managing containers in the Web project.
- **Spring MVC**[28], an original web framework built on the Servlet API and included in the Spring Framework. Spring MVC framework relies on the Spring framework. Grouper uses it to create our RESTful API.
- **Hibernate**[29], an open-source Java Object-Relational Mapping (ORM) framework. Hibernate provides generalized and automated solutions to common tasks associated with object life cycles and object graph management, including persistence. Grouper uses it to save and operate objects in the Web service.

Our Web service has the following four layers.

- **Domain Layer.** This layer is a collection of all entity objects. This layer creates a standardized and federated set of objects that are reused within different layers.
- **Data Access Object (DAO) layer.** This layer is a collection of objects that provide abstract interface and implementation to access the MySQL database. Each entity object corresponds to a DAO object.
- **Service Layer.** This layer is a collection of objects that provide abstract interface and implementation of business logic. This layer invokes the methods of the DAO layer.
- **Controller Layer.** The objects in this layer build the RESTful API for the client framework. This layer is responsible for user authentication with the access keys and invokes the methods of the service layer.

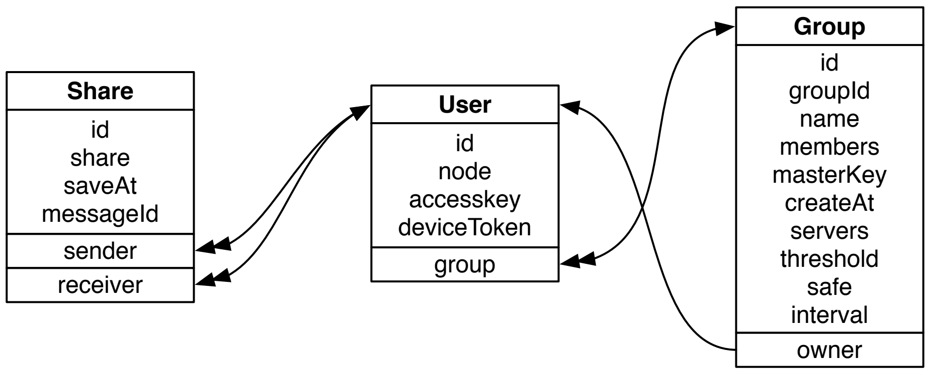


Figure 5.2: Entity relationship in the Web service of Grouper.

5.1.2 Entity Design

Our Web service includes three kinds of entities: the *Group*, *User*, and *Share* entities. Figure 5.2 shows the relationship among these entities. Table 5.1 shows the attributes of these entities.

A Share entity saves the share content generated by a secret sharing scheme and uploaded from a device of a group member. To recover a message from multiple shares, we bind the message ID to the Share entity. When a device downloads several shares from servers, Grouper tries to collect these shares by using the message ID and recover a message from the shares with the same message ID. The sender attributes refers to an User entity, and indicates the uploading device. The receiver attributes refers to an User entity, and indicates the devices that can download the share. If the sender allows all devices in the group to receive the share, the receiver attribute of the entity is a *null* value.

A User entity saves the information of a user's device. Each device has a unique node identifier of the UUID scheme. An untrusted server generates an access key for the user when the group owner registers him/her in this server. To receive a remote notification from the server, he/she should provide a device token to the notification pushing server. The group attribute refers to a Group entity, and indicates the group the user belongs to.

A Group entity saves the information of a group. The group owner generates a unique group identifier and gives a group name to create a new group. The number of group members is one when the owner creates the group. When the group owner invites a new member to the group, the number of group members is incremented. When a group is created, the server generates a master key and returns it to the group owner. The group owner can update the attributes of the Group entity using this master key. In addition, the group owner submits all parameters in the extended secret sharing scheme and the TTL. The owner attribute refers to an User entity, and indicates the group owner.

5.1.3 RESTful API

The Web service of Grouper provides the following RESTful APIs for the client framework.

Table 5.1: Attributes of entities in the Web service of Grouper

Entity	**Attribute**	**Description**
Share	id	ID of the Share entity.
	share	Share content uploaded from a device.
	saveAt	Time the entity was saved.
	messageId	Message ID of the share.
	sender	User entity of the sender of the share.
	receiver	User entity of the receiver of the share.
User	id	ID of the User entity.
	node	Device's unique node identifier of the user.
	accessKey	Access key to upload and download shares of the user.
	deviceToken	Device token for sending a remote notification.
	group	Group of the user.
Group	id	ID of the Group entity.
	groupId	Unique group identifier created by the group owner.
	name	Group name created by the group owner.
	members	Number of group members.
	masterKey	Master key for the group owner.
	createAt	Time the entity was created.
	servers	Number of servers, the parameter n in the $f(k, n, s)$ scheme.
	threshold	Threshold, the parameter k in the $f(k, n, s)$ scheme.
	s	The parameter s in the $f(k, n, s)$ scheme.
	TTL	Time to Live for the shares of this group.
	owner	User entity of the group owner.

- **group/register** (POST)
    - ◇ Register a new group in an untrusted server.
    - ◇ Parameters: group ID and group name.
    - ◇ Return: the master key for the group owner.
- **group/info** (GET)
    - ◇ Get information of a group.
    - ◇ Header: the master key of this group or access key of the group member.
    - ◇ Return: all attributes of a group entity.
- **group/init** (POST)
    - ◇ Initialize a group by submitting the parameters in the extended secret sharing scheme $f(k, n, s)$ and the TTL.
    - ◇ Header: the master key of this group.
    - ◇ Parameters: k, n, s and TTL.

  - ◇ Return: a success flag.
- **user/add** (POST)
  - ◇ Add a new user in the untrusted server by the group owner.
  - ◇ Header: the master key of this group.
  - ◇ Parameters: node identifier and the flag to indicate that this user is the owner or a regular member.
  - ◇ Return: the access key for this new user.
- **user/state** (GET)
  - ◇ Check server state.
  - ◇ Header: the access key of the group member.
  - ◇ Return: server state.
- **user/deviceToken** (POST)
  - ◇ Update the device token of a device.
  - ◇ Header: the access key of the group member.
  - ◇ Parameters: the access token received from the remote notification pushing server.
  - ◇ Return: a success flag.
- **user/notify** (POST)
  - ◇ Send a remote notification to a group member.
  - ◇ Header: the access key of the sender.
  - ◇ Parameters: the content of the remote notification, the node identifier of the notification receiver and the notification category.
  - ◇ Return: a success flag.
- **transfer/put** (POST)
  - ◇ Upload a share to an untrusted server.
  - ◇ Header: the access key of the group member.
  - ◇ Parameters: the JSON array which contains the shares, receivers and the message IDs.
  - ◇ Return: the number of shares which are saved in the server successfully.
- **transfer/list** (GET)
  - ◇ Get the list of share IDs for a group member.
  - ◇ Header: the access key of the group member.
  - ◇ Return: the share IDs that this group member has privilege to download.
- **transfer/get** (GET)
  - ◇ Get the content of shares by the list of share IDs.

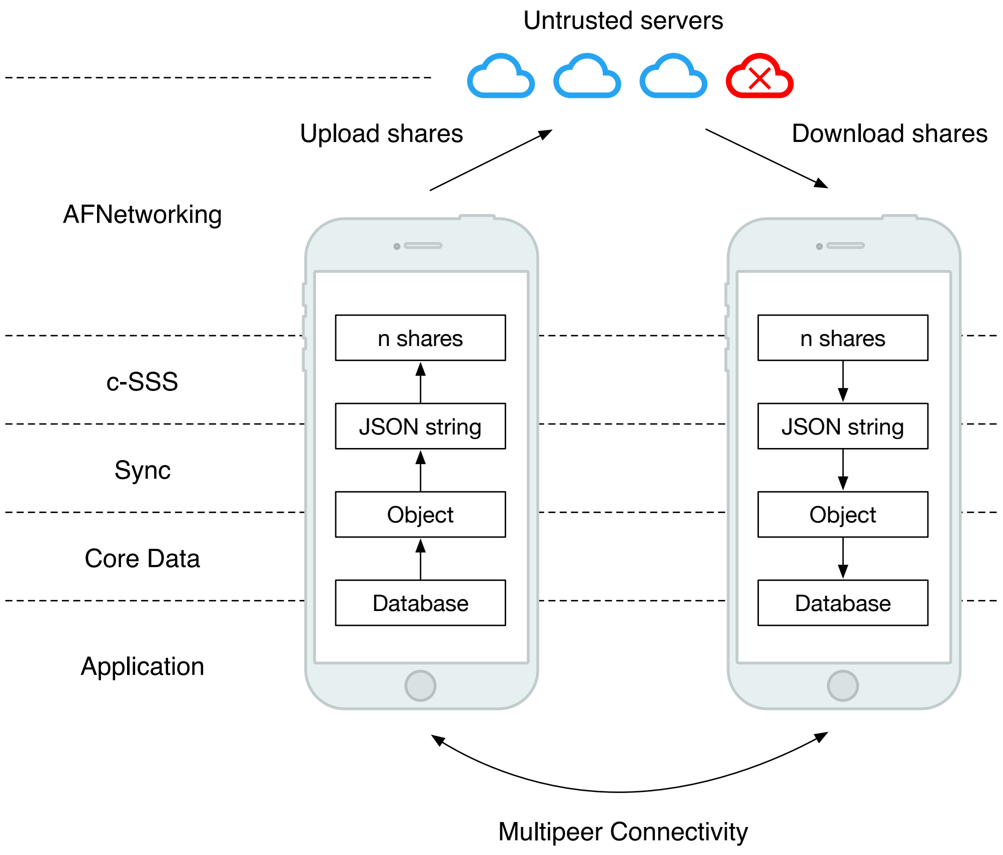


Figure 5.3: External frameworks in the client framework of Grouper.

- ◇ Header: the access key of the group member..
- ◇ Parameters: the array of share IDs.
- ◇ Return: the content of shares.

- **transfer/confirm** (POST)
  - ◇ Confirm whether the shares are existed in an untrusted server.
  - ◇ Header: the access key of the notification sender.
  - ◇ Parameters: the array of message IDs.
  - ◇ Return: the array of message IDs which are not existed in this untrusted server.

All of these RESTful APIs authenticate a regular group member or a group owner with an access key or a master key in the request header.

5.2 Client Framework

This section describes the implementation of the client framework. Grouper's client framework is written in Objective-C, and its core layer supports developing applications on iOS, macOS, watchOS and tvOS. Grouper supports the iOS graphical user interface (GUI) for group management, so that the developers need not to implement it on the iOS platform.

In this section, we show external framework in Subsection 5.2.1. We use singleton pattern to implement manager classes in the client framework. The following subsections describe these manager classes.

5.2.1 External Frameworks in Clients

The Grouper client framework makes use of the following frameworks in Figure 5.3.

- **Multipeer Connectivity**[25], an official Peer-to-Peer communication framework provided by Apple. Grouper uses it to transfer data between two devices in a face-to-face manner using a wireless LAN or Bluetooth network. With Multipeer Connectivity, a private connection can be established between two devices without Internet.
- **Core Data**[30], an official ORM framework provided by Apple. Core Data provides generalized and automated solutions to common tasks associated with object life cycles and object graph management, including persistence. Grouper uses it to manage model layer objects.
- **Sync**[6], a lightweight framework for data synchronization using Core Data. Sync uses JSON strings to synchronize managed objects of Core Data among devices. Sync provides the instance method *export*() in the class *NSManagedObject* to convert a managed object into a JSON string and the asynchronous method *sync*() to covert the JSON string into an object of Core Data. Grouper uses it in the synchronization plugin.
- **c-SSS**[31], an implementation of the secret sharing scheme in the C language. It provides two main functions: generating shares from a string and reconstructing the string from shares. The function *generate_share_strings*() takes a secret string, n and k as arguments, and generates n shares of the secret string with the threshold k. The function *extract_secret_from_share_strings*() takes shares and reconstruct the string from k or more shares.
- **AFNetworking**[32], a networking library in the Objective-C language. Grouper uses it to invoke the RESTful APIs provided by our several Web services running on multiple untrusted servers.

5.2.2 Sender Manager

In our persistent store, Core Data entities of applications are synchronized to other devices. We call an entity in Core Data which is needed to synchronize to other devices a *sync entity*. The class *SenderManager* provides the following methods to create and send the Grouper messages as shown in Figure 5.4.

- The method *update* creates an 6update message that includes a JSON string of a sync

```
// Send an update message for a sync entity.
- (void)update:(NSManagedObject *)entity;

// Delete a sync entity and send a delete message.
- (void)delete:(NSManagedObject *)entity;

// Send confirm message;
- (void)confirm;

// Send resend message with a range to receiver.
- (void)resendWith:(int)min and:(int)max to:(NSString *)receiver;

// Send unsent messages.
- (void)unsent;

// Send existed messages.
- (void)sendExistedMessages:(NSArray *)messages;
```

Figure 5.4: Instance methods of *SenderManager*.

entity. Next, it invokes the instance method *sendMessages* to generate shares from the message and send the shares to multiple untrusted servers.

- The method *delete* deletes a sync entity, creates a delete message which contains the object ID of this entity, and invokes the method *sendMessages*, as same as the method *update* does.
- The method *confirm* finds the maximum sequence number of normal messages in the local persistent store, creates a confirm message which contains the maximum sequence number, and invokes the method *sendMessages*.
- The method *resendWith* is used for sending a resend messages after receiving a confirm control message. It creates a resend message which contains the range of sequence number (the minimum and maximum sequence number), and invokes the method *sendMessages*.
- The method *unsent* resends messages to untrusted servers. Due to some network reasons, a device can not send messages to untrusted servers using the method *sendMessages*. In these situations, the Grouper framework invokes the method *unsent* to send the messages which are sent unsuccessfully. The method *unsent* finds those messages and sends them to untrusted servers later by invoking the method *sendMessages*.
- The method *sendExistedMessages* is used for resending normal messages after receiving a resend control message. It puts the normal message to a mutable array at first. Before sending the normal messages, it confirms with the multiple untrusted servers. If some of them have not been deleted in servers, those messages will be deleted from the mutable array. Finally, it sends the messages in the mutable array to servers by invoking the method *sendMessages*.

All these methods introduced above use a same important instance method *sendMessages* to generate shares and send them to servers. This method converts the Grouper messages to JSON strings at first. Then, it invokes the function *generate_share_strings*() in c-SSS[31] to generate shares for each JSON string. The shares are grouped by the addresses of the multiple untrusted servers. Next, it tries to upload these shares to the servers using the RESTful API *transfer/put*. If the number of servers which are accessed successfully is more than s in the extended secret sharing scheme $f(k, n, s)$, the Grouper framework regards this uploading successful. Otherwise, the object IDs of these Grouper messages will be put into a queue called $unsentMessageIds$. Grouper tries to send the Grouper messages in $unsentMessageIds$ next time an application launches. Finally, if the share uploading is successful, this method sends remote notifications to the receivers of the messages.

Note that there may be too many messages to be sent at the same time. To improve the efficiency of the share uploading, the method $sendMessages$ divides the messages to several parts. For example, if there are 550 messages to be sent at the same time, this method tries to send the first 100 messages and postpones sending the 450 messages.

5.2.3 Receiver Manager

The class $ReceiverManager$ provides the instance method $receiveWithCompletion$ to receive shares, recover messages and synchronize the objects in the messages into the local persistent store.

This method does not download all shares from the multiple untrusted servers. When the application invokes this method, it gets the array of share IDs from the servers using the RESTful API *transfer/list* at first. Because the client framework of Grouper has the old share IDs in the local persistent store, this method compares the new share IDs from the servers with the old share IDs. Then, it removes old share IDs from the new share IDs, and gets the content of new shares using the RESTful API *transfer/get*. If the content of shares are received successfully, the share IDs will be saved into the local persistent store. Next, this method tries to collect the shares by message IDs and recovers the message with a same message ID. Finally, this method uses Algorithm 1 to handle the messages. A callback block is invoked after handling all messages successfully.

5.2.4 GroupManager

The class $GroupManager$ provides methods to create a new group, invite members to an existing group and send device tokens for remote notification to the untrusted servers. Figure 5.5 shows main instance methods in this class.

An application invokes the method $saveCurrentUserWithEmail$ when the user uses the application without creating or joining a group. It saves the name and email address of a user into the local persistent store of his/her device.

An application invokes the method $addNewServer$ when the user adds untrusted servers. It submits the group name and group ID to each new untrusted server using the RESTful API *group/register*. Then, this server generates a master key and returns it to the device. Finally, this method saves the group name, group ID, server addresses and master keys in the local persistent store.

An application invokes the method *initializeGroup* when the user initializes a new group. It submits the parameters of the extended secret sharing scheme $f(k, n, s)$ and the TTL to the multiple untrusted servers using the RESTful API *group/init*.

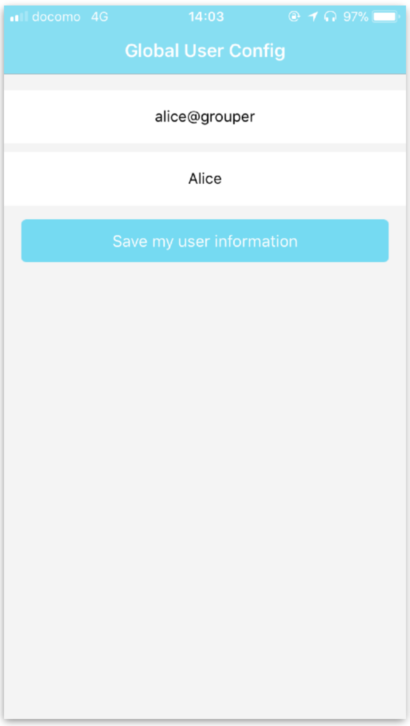
```
// Save global user email and name.
- (void)saveCurrentUserWithEmail:(NSString *)email name:(NSString *)name;

// Add a new untrusted server.
- (void)addNewServer:(NSString *)address
       withGroupName:(NSString *)groupName
          andGroupId:(NSString *)groupId
          completion:(SucessMessageCompletion)completion;

// Initialize group.
- (void)initializeGroup:(int)threshold
       safeServersCount:(int)safeCount
               interval:(int)interval
         withCompletion:(SucessMessageCompletion)completion;

// Setup mutipeer connectivity so that the device can be discoverd by others.
- (void)setupMutipeerConnectivity;

// Open device browser.
- (void)openDeviceBrowserIn:(UIViewController *)controller;

// Send invite message to a peer.
- (void)sendInviteMessageTo:(MCPeerID *)peer;

// Send device token to untrusted servers
- (void)sendDeviceToken:(NSString *)token;
```

Figure 5.5: Instance methods of *GroupManager*.

An application invokes the method *setupMutipeerConnectivity* to initialize the Multipeer Connectivity framework so that the device can be discovered by others. An application invokes the method *openDeviceBroswerIn* to open the device browser of the Multipeer Connectivity framework. In this browser, a user selects another device to connect. In the device of the group owner, the application invokes the method *sendInviteMessageTo* to send an invitation message to a connected device. If the user of the connected device accepts the invitation, he/she will join the group of the owner.

An application invokes the method *sendDeviceToken* to upload the device tokens to the multiple untrusted servers using the RESTful API *user/deviceToken*.

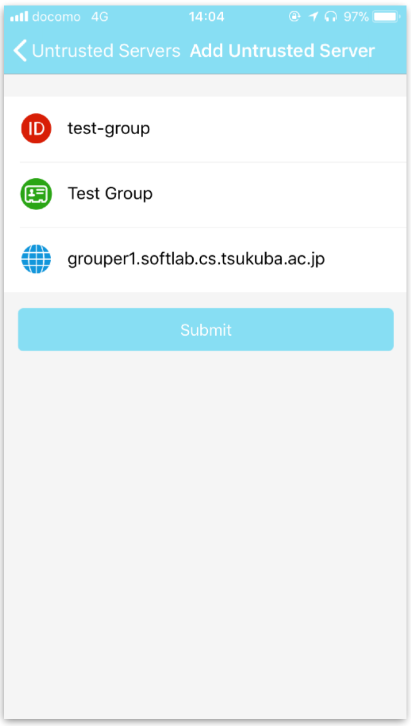
To simplify the group management on the iOS platform, we provide the library on Grouper's client framework. Concretely, we encapsulate the *GroupManager* class and provide the GUI as shown in Figure 5.6. Figure 5.6 (a) shows the screenshot of saving user information using the *saveCurrentUserWithEmail* method. Figure 5.6 (b) shows the screenshot of adding a new untrusted server using the *addNewServer* method. Figure 5.6 (c) shows the screenshot of initializing a new group using the *initializeGroup* method. Figure 5.6 (d) shows the screenshot of the browser to connect other devices provided in the Multipeer Connectivity framework using the *openDeviceBroswerIn* method. With this library for the iOS platform, a developer does not need to implement the group management GUI by themselves.



alice@grouper

Alice

(a) Save user information.

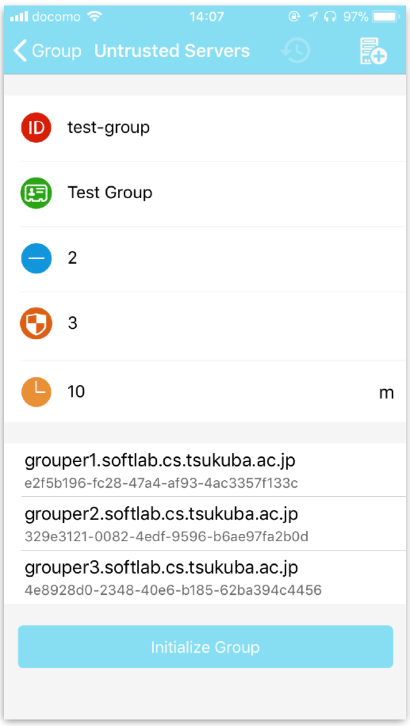


test-group

Test Group

grouper1.softlab.cs.tsukuba.ac.jp

(b) Add a new untrusted server.



test-group

Test Group

2

3

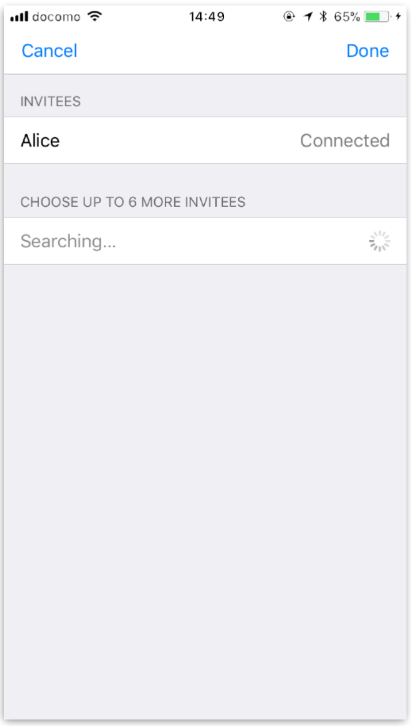
10 m

grouper1.softlab.cs.tsukuba.ac.jp
e2f5b196-fc28-47a4-af93-4ac3357f133c

grouper2.softlab.cs.tsukuba.ac.jp
329e3121-0082-4edf-9596-b6ae97fa2b0d

grouper3.softlab.cs.tsukuba.ac.jp
4e8928d0-2348-40e6-b185-62ba394c4456

(c) Initialize a new group.



INVITEES

Alice Connected

CHOOSE UP TO 6 MORE INVITEES

Searching...

(d) Initialize a new group.

Figure 5.6: Screenshots of the GUI of the *GroupManger* class

```
use_frameworks!

target 'YourProjectName' do
    pod 'Grouper', '~>2.1'
end
```

Figure 5.7: Podfille of the CocoaPods dependency manager.

```
# Run this script if CocoaPods has not been installed.
sudo gem install cocoapods

# Install Grouper with CocoaPods
cd [YourProject]
pod install
```

Figure 5.8: Script to install Grouper with the CocoaPods dependency manager.

5.2.5 Release and Installation

We use CocoaPods[33], a dependency manager for Swift and Objective-C Cocoa projects, to release the client framework of Grouper. With CocoaPods, the developer can add the dependency of Grouper to his/her applications easily. To install the Grouper framework, a developer needs to add a file, *Podfile*, as shown in Figure 5.7.

The developer sets his/her project name as the target and the version of Grouper. Then, he/she runs the script as shown in Figure 5.8 in the terminal to install the Grouper framework.

5.3 Applications

Using the Grouper framework, we have developed the following applications.

- **Account Book.** An iOS application in Objective-C, that records the income and expenditure of a group.
- **Test.** A benchmark iOS application in Swift, that tests the performance of Grouper.

5.3.1 Account Book

In this subsection, we show how to convert a standalone application into a data sharing application using Grouper. We use Account Book as an example application. We have shown the basic functions of Account Book in Section 4.1. The standalone application of Account Book can record the income and expenditure of a person in a device. Without data sharing, records cannot be shared with other persons. To convert the standalone application into the data sharing one, we need the following five steps.

1. **Initialization of Grouper**

```
1  grouper = [Grouper sharedInstance];
2  UIStoryboard *storyboard = [UIStoryboard storyboardWithName:@"Main" bundle:nil];
3  [grouper setupWithAppId:@"accountbook"
4                 entities:[NSArray arrayWithObjects:@"Classification", @"Account", @"
                          Shop", @"Photo", @"Template", @"Record", nil]
5                dataStack:[self dataStack]
6          mainStoryboard:storyboard];
```

Figure 5.9: Initialize Grouper in the Account Book application.

We invoke the *grouper.setup*() method in Table 4.1 to initialize Grouper in the class *AppDelegate*. Figure 5.9 shows how we initialize our Grouper framework in the Account Book application. We invoke the class method *Grouper.sharedInstance* to get the singleton of the Grouper framework at Line 1. Because our framework supports the GUI in iOS, we get the storyboard instance to use the GUI for creating a new group or joining an existing group at Line 2. At Line 3, we invoke the initialization method to initialize Grouper.
We must emphasize the order of the parameter entities in the method *setup*() of Table 4.1. The reference relationships among entities *Classification*, *Account*, *Shop*, *Photo*, *Template* and *Record* are as follows.

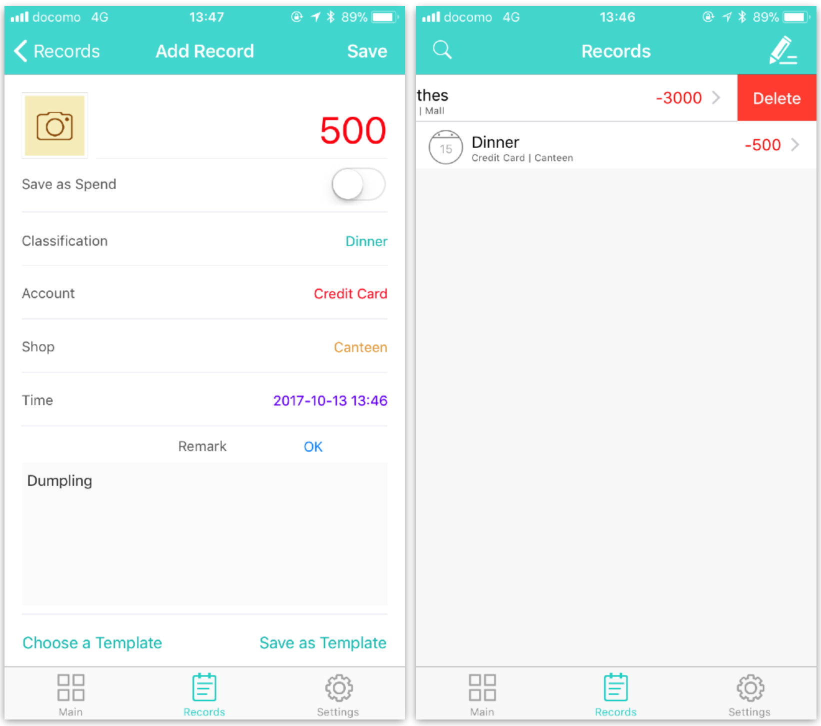
- **Record:** $Classification \leftarrow Record, Account \leftarrow Record, Shop \leftarrow Record, Photo \leftarrow Record$
- **Template:** $Classification \leftarrow Template, Account \leftarrow Template, Shop \leftarrow Template$

The array in the code of Figure 5.9 follows the rule of the method *setup*() of Table 4.1. If an entity refers to another entity, the first entity should appear before the second entity. After initializing the Grouper framework in the Account Book application, we can invoke other methods in Table 4.1.

2. **Object Creation and Updating**

Figure 5.10(a) shows the screenshot of adding a record in Account Book. In this screenshot, a user uses Account Book to add income and expenditure records that include the classifications, accounts, shops, times and remarks. When the user click the save button, Account Book saves the record into the local persistent store at first. The standalone application has the code from Line 2 to Line 8 in Figure 5.11. To share this record with other group members, we invoke the *grouper.sender.update*() method of Grouper at Line 11 after saving the record in the local persistent store.
The method *grouper.sender.updated*() divides a record into several shares and uploads them to multiple untrusted servers. This method is also suitable for updating a record. We can not ensure that other grouper members can synchronize this object successfully, because it has only been uploaded to multiple untrusted server

3. **Object Deletion**



(a) Add a record. (b) Delete a record.

Figure 5.10: Screenshots of adding and deleting a record in the Account Book demo application.

Figure 5.10(b) shows the screenshoot of deleting a record in the Account Book application. A user swipes a cell from right to left to delete a record from the record list as shown in Figure 5.10 (b). In the standalone application, we only delete this record in the local persistent store when the user click the delete button. To delete this record on the devices of other group members, we invoke the $grouper.sender.delete(object)$ method of Grouper at Line 4.
Developers need not to delete the record in the local persistent store using the API of Core Data before invoking the method $grouper.sender.delete()$, like the method $grouper.sender.update()$ does, because Grouper deletes the record in the local persistent store before creating a delete message.

4. **Receiving messages from Untrusted Servers**
Grouper provides the method $grouper.receiver.receive()$ to receive messages from untrusted servers. Developers must decide the policy to receive messages that contains the updated or deleted objects by themselves. For example, the Account Book application receives messages in the following two ways.

   - **Manually.** A user pulls down the view as shown in Figure 5.14(a) to get the

```
// Save a record object to the application local persistent store.
Record *record = [dao.recordDao saveWithMoney:money
                                       remark:_remarkTextView.text
                                         time:_selectedTime
                               classification:_selectedClassification
                                      account:_selectedAccount
                                         shop:_selectedShop
                                        photo:photo];

// Invoke grouper.sender.update() method to send an update message.
[grouper.sender update:record];
```

Figure 5.11: Create a new record and send an update message.

```
Record *record = [records objectAtIndex:indexPath.row];
if (editingStyle == UITableViewCellEditingStyleDelete) {
    // Delete the record in the local persistent store and other devies.
    [grouper.sender delete:record];
    // Update user interface.
    [records removeObjectAtIndex:indexPath.row];
    [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
            withRowAnimation:YES];
}
```

Figure 5.12: Delete a record and send a delete message.

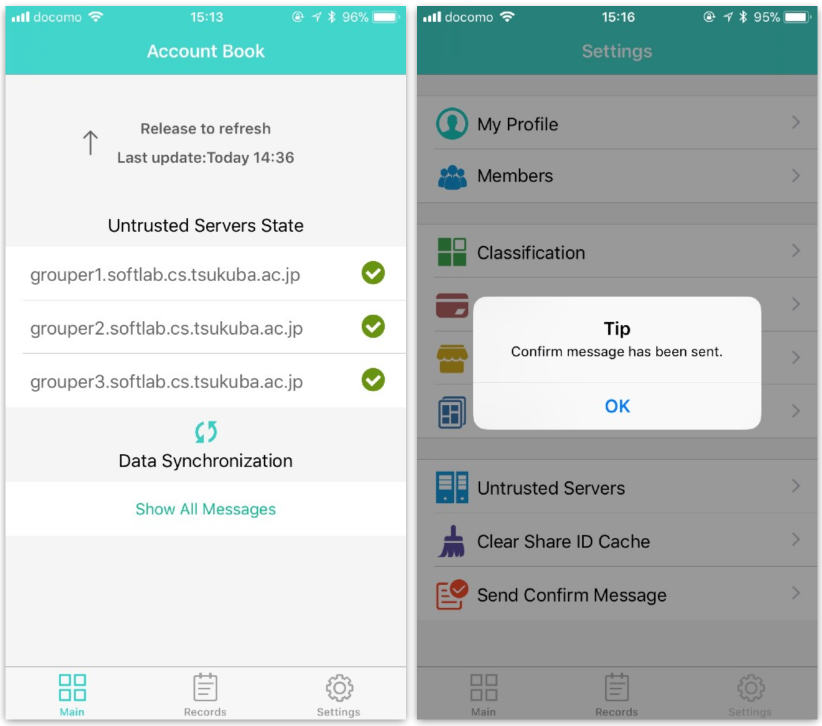
latest records.

- **Automatically when the application is launched.** The Account Book application tries to receive and handle automatically when it is launched.

We can update user interface in the callback block of the method *grouper.receiver.receive*() as shown in Figure 5.13. The parameter success in the block indicates the state of message handling. If the number of servers that a device can access is less than the threshold k in the extend secret sharing scheme $f(k, n, s)$, the message handling will be failed and the parameter success will be false.

```
[grouper.receiver receiveWithCompletion:^(int success, Processing *processing) {
    long now = (long)[[NSDate date] timeIntervalSince1970];
    if (now - grouper.group.defaults.controlMessageSendTime > grouper.group.defaults.
            interval * 60) {
        [grouper.sender confirm];
        // Update the last time to send a confirm message.
        grouper.group.defaults.controlMessageSendTime = now;
    }
}];
```

Figure 5.13: Receive messages from untrusted server and send a confirm message.



grouper1.softlab.cs.tsukuba.ac.jp ✓
grouper2.softlab.cs.tsukuba.ac.jp ✓
grouper3.softlab.cs.tsukuba.ac.jp ✓

Tip
Confirm message has been sent.
OK

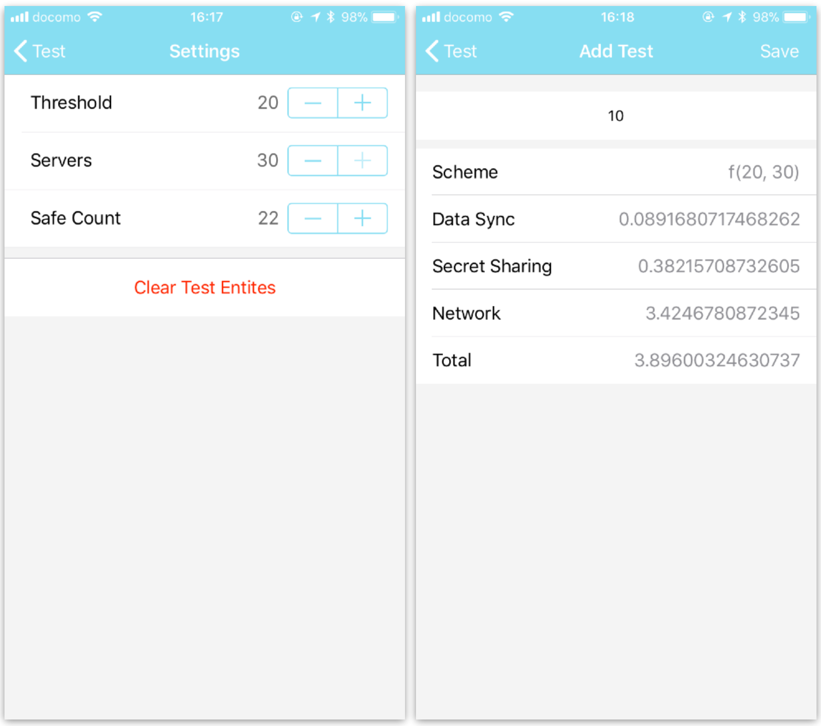
(a) Receive messages. (b) Send a confirm message.

Figure 5.14: Screenshots of receiving messages and sending a confirm message in the Account Book application.

5. **Sending Confirm Messages**
   We have the method *grouper.confirm*() in Section 4.6. The Account Book application invokes the method in the following two ways.
   - **Manually.** A user can send a confirm message manually in the *Settings* table view as shown in Figure 5.14(b).
   - **After receiving messages.** As described int the code from Line 2 to Line 7 of Figure 5.13, when the Account Book application receives messages from untrusted servers, it checks the time that the application sent the last confirm message. If it past a longer time than the TTL defined by the user, the Account Book application sends a confirm message.

5.3.2 Test

We have developed an application, Test, in the Swift language. In the Test application, we can change all the parameters in the extended secret sharing scheme as shown in Figure 5.15 (a). To measure the performance, the Test application creates and sends multiple messages at the same time, and shows processing time as shown in Figure 5.15 (b).



Threshold	20
Servers	30
Safe Count	22

10

Scheme	f(20, 30)
Data Sync	0.0891680717468262
Secret Sharing	0.38215708732605
Network	3.4246780872345
Total	3.89600324630737

(a) Changing parameters. (b) Adding multiple test objects.

Figure 5.15: Screenshots of the Test application.

Chapter 6

Evaluation

This chapter discusses the development efforts of applications using Grouper for developing mobile applications, as well as its performance.

6.1 Development Efforts

We view the development efforts through two factors: the usability of the client API and the code size, in terms of the lines of code (LoC) the developer has to add after using Grouper. As described in Table 4.1, Grouper provides simple client APIs for developers. A developer can easily convert a standalone application to a data sharing application with our client APIs.

Section 5.3 demonstrates the implementation of two applications. Table 6.1 shows that developers can add data synchronization to these applications with Grouper by adding a small amount of code.

The Grouper framework provides the synchronization as a pluggable module. We use the Sync[6] framework. It provides a consistency model where currently, the newest object wins in the synchronization plugin. For example, if a record in the Account Book application has been modified by two users, the modification of one user will be lost. Account Book maintains the newest modification and put it into the persistent store.

If an application requires another consistency model, the developer has to implement the related synchronization plugin. For example, if a developer wants to use eventual

Table 6.1: Application lines of code.

Application	**Account Book**	**Test**
Platform	iOS	iOS
Lanaguage	Objective-C	Swift
Number of Entities	5	1
Standalone Application LoC	8076	621
Increased LoC	190	18

Table 6.2: Devices in the performance experiment.

Device	**CPU**	**RAM**	**OS**
iPod touch 5th	A5	512MB	iOS 9.3.5
iPhone 4s	A5	512MB	iOS 9.3.5
Server 1	Core i7-5820K	32GB	Ubuntu 14.04.5
Server 2	Core i7-5820K	32GB	Ubuntu 14.04.5
Server 3	Core i7-5820K	32GB	Ubuntu 14.04.5

consistency, he/she can choose Ensembles[34]. For this reason, if a developer wants to use another consistency model, he/she must add more lines of code to those in Table 6.1.

Note that the synchronization frameworks must provide the interfaces as described in Table 4.2. For this reason, Grouper does not support iOS synchronization frameworks for iCloud or Dropbox. For example, Grouper does not support Core Data with iCloud[30] and TICoreDataSync[35] at this point. These frameworks provide APIs for files whereas Grouper requires APIs for sending multicast messages. If a developer want to use them, he/she has to add an additional layer that provides the methods in Table 4.2.

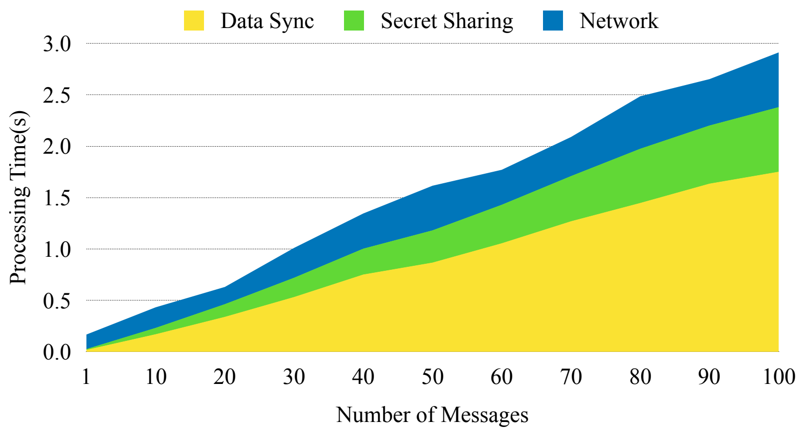
6.2 Performance

In this section, we show that mobile applications using the Grouper framework achieve a satisfactory performance for small groups of people. Table 6.2 lists the hardware and software configuration of our experiment environment. In our performance experiments, we used the benchmark application *Test* to transfer data between the iPhone 4s and iPod touch 5th generation on a wireless LAN network (802.11n). We installed 30 Web services on three different servers. Each server ran a Tomcat server instance which hosted 10 Web services.

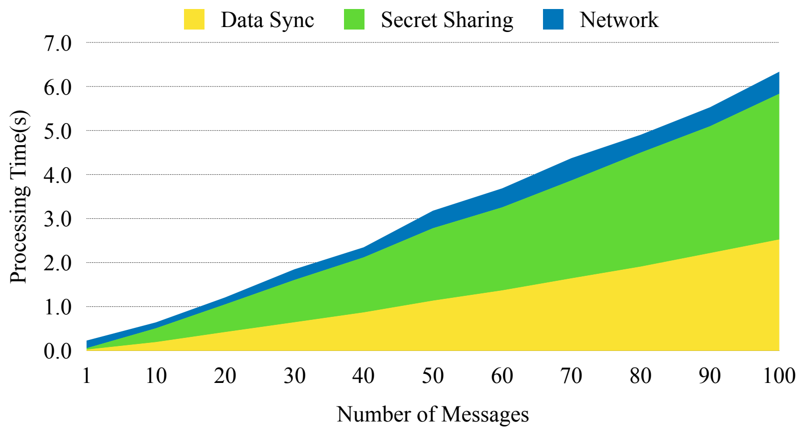
In our benchmark application, the size of an object was 323 bytes. The object corresponded to an income or expenditure record of the Account Book application. When the object was updated in a node, the Grouper API generated a normal message whose size was 656 bytes.

We performed an experiment to measure the processing time of object synchronization according to the number of updated objects. In this experiment, we set the secret sharing scheme to $f(2,3,3)$. We sent multiple messages from a device and received them in another device. Concretely, we sent multiple messages from the iPod to the iPhone for three times, and from the iPhone to the iPod for three times. Then, we obtained the average value of the measured processing times.

Figure 6.1 shows the processing time of uploading and downloading multiple messages. We divide the processing time into three parts: data synchronization, secret sharing, and networking. As the number of messages increased, the data synchronization and secret sharing time increased linearly. The networking time increased slowly and sometimes decreased. On the whole, the total processing time increased linearly. Compared with up-



(a) Uploading multiple messages.



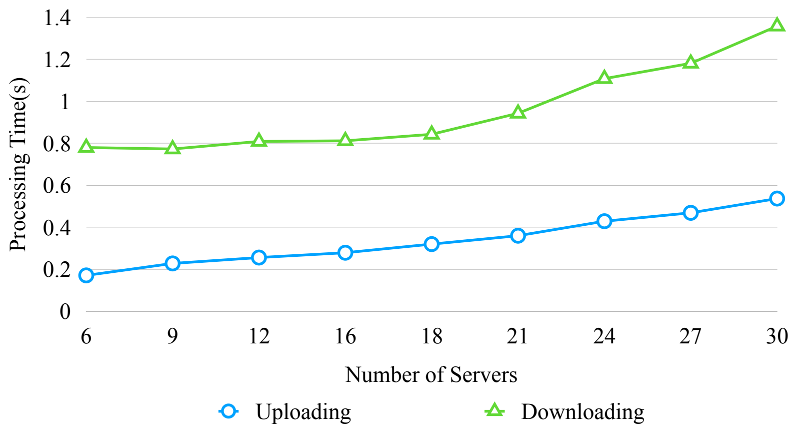
(b) Downloading multiple messages.

Figure 6.1: Processing time of uploading and downloading multiple messages.

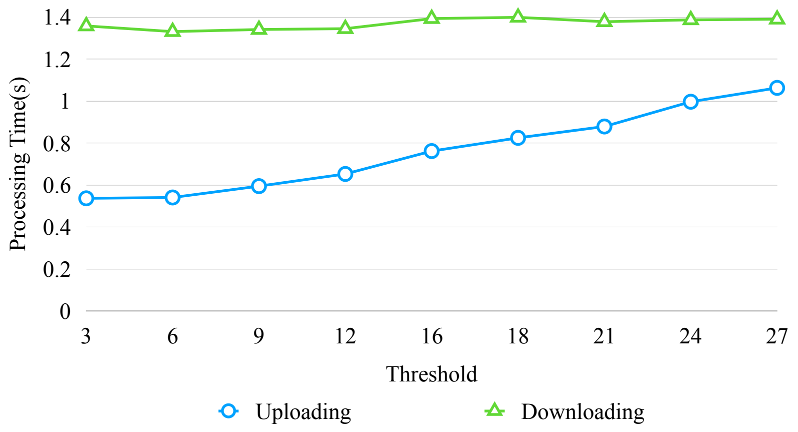
loading messages, downloading messages required more processing time. Note that object synchronization is done as a background task, asynchronously. When the Grouper framework is performing object synchronization, the user interface of a mobile application does not freezes. Therefore, a user is not aware of the activity of data synchronization.

We performed an experiment to measure the processing time of object synchronization according to the parameters of the secret sharing scheme. Specifically, we changed the parameter k, n and s of the secret sharing scheme and measured the processing time of uploading and downloading a single message.

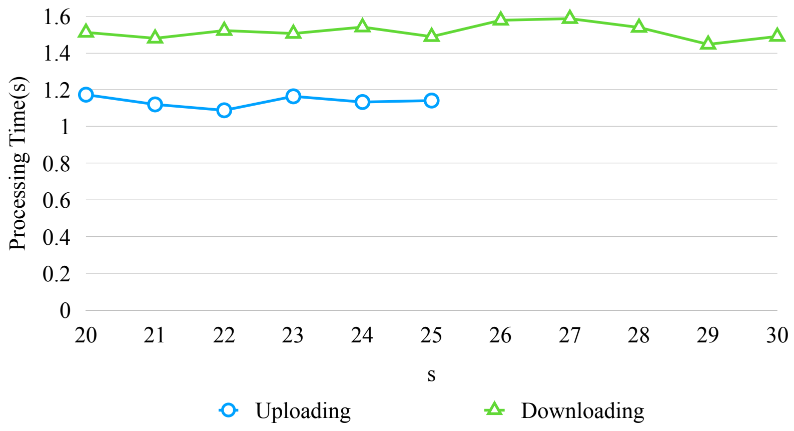
Figure 6.2(a) shows the relationship between processing time and the number of servers n, where $k = 3$, $n = 6, 9, ..., 30$ and $s = n$. As n increased, the processing time of uploading and downloading increased linearly. Figure 6.2(b) shows the relationship between processing time and the threshold k, where $n = s = 30$ and $k = 3, 6, 9, ..., 27$. As k increased, the processing time of uploading increased linearly and that of downloading did not change.



(a) Fixed threshold (k = 3, s = n).



(b) Fixed number of servers and safe servers count (n = 30, s = n).



(c) Fixed threshold and number of servers (k = 20, n = 30).

Figure 6.2: Processing time of uploading and downloading a single message.

Table 6.3: Parameters for discussing the number of messages.

Parameter	**Explanation**
D	Number of devices.
T	Time to live (TTL).
F	Average number of offline devices during the TTL.
L	Average offline time.
Ui	Average number of updated/deleted messages in the device i during the TTL.
URi	Average number of updated/deleted messages by resend messages in the device i during the TTL.
C	Average number of confirm messages in the device i during the TTL.
U	Total number of updated and deleted objects during the TTL.
NUi	Number of uploads for device i during the TTL.
NU	Total number of uploads during the TTL.
NDi	Number of downloads for device i during the TTL.
ND	Total number of downloads during the TTL.

To confirm the effectiveness of the parameter s, we stopped some untrusted servers and measured processing times. In this experiment, we set $k = 20$ and $n = 30$. We shutdown five untrusted servers (No.1, No.7, No.13, No.19 and No.25).

Figure 6.2(c) shows the relationship between processing time and the parameter s, where $k = 20$, $n = 30$ and $s = 20, 21, 22, ..., 30$. Grouper regarded the uploading successful where $20 \leq s \leq 25$ and the uploading unsuccessful where $26 \leq s \leq 30$. The processing times of uploading where $26 \leq s \leq 30$ were not measured. Although the parameter s was changed, the processing times of uploading and downloading did not change. This is because we used virtual severs and the processing times do not include the timeout period. In a real usage scenario, the timeout period can make the processing times longer. Note that these processing times do not influence the user experience of applications.

6.3 Discussion on the Number of Messages

In this section, we discuss the number of messages. We use the parameters in Table 6.3.

First, we consider the situation in which all devices are online ($F = 0$). In this situation, the number of uploads for the device i are:

$$NUi = Ui + C \tag{6.1}$$

Total number of uploads NU is the sum of NUi.

$$NU = \sum_{i=1}^{D} NUi = U + C \cdot D \tag{6.2}$$

A device does not download the messages that have been uploaded. To simplify these equations, we assume that the device downloads those messages as well. Thus, the number of downloads for device i, NDi, is the equal to the total number of uploads NU.

$$NDi = NU = U + C \cdot D \tag{6.3}$$

The total number of downloads ND is the sum of NDi.

$$ND = \sum_{i=1}^{D} NDi = NU \cdot D = (U + C \cdot D) \cdot D \tag{6.4}$$

Since the current cloud servers scale well according to the number of devices and messages, we discuss NUi and UDi in this section. Equation 6.1 shows that for the device i during the TTL, the order of uploading is $O(U)$. Euqation 6.3 shows that the order of downloading is $O(U+D)$. As the number of updated and deleted objects increases, the total numbers of uploads and downloads increases linearly. As the number of devices increases, the number of confirm messages increases linearly.

Next, we consider the situation in which there are offline devices in a group ($F \neq 0$). In this situation, offline devices send resend messages when they become online and the devices that have the updated objects will resend the update messages to the offline devices. To simplify these equations, we assume that all offline devices receive the updated messages after they send resend messages. For the device i, the number of those update messages by the resend message, URi, is

$$URi = Ui \cdot \frac{L}{T} \cdot F \tag{6.5}$$

Compared to the situation that $F = 0$, in this situation, the number of uploads for the device i includes URi. Thus, we obtain NUi, NU, NDi and ND where $F \neq 0$.

$$NUi = Ui + C + Ui \cdot \frac{L}{T} \cdot F = Ui \cdot (1 + \frac{L}{T} \cdot F) + C \tag{6.6}$$

$$NU = \sum_{i=1}^{D} NUi = U \cdot (1 + \frac{L}{T} \cdot F) + C \cdot D \tag{6.7}$$

$$NDi = NU = U \cdot (1 + \frac{L}{T} \cdot F) + C \cdot D \tag{6.8}$$

Equations 6.6 and 6.8 show that for the device i during the TTL, the order of uploading is $O(U \cdot \frac{L}{T} \cdot F + C)$, and the order of downloading is $O(U \cdot \frac{L}{T} \cdot F + C \cdot D)$. As the numbers of updated and deleted objects, the average number of offline devices during the TTL, and the average online time increase, the number of uploads and downloads for a single device increases linearly.

If we use a longer T, the numbers of messages, NUi and NDi, become smaller. However, this makes it easier for attackers to attack untrusted servers.

Finally, we consider the following two cases with concrete parameters. In the last two cases, there are ten users in a small group and everyone has a device. We set the TTL to one day and a hundred of normal messages are created by the ten users in the group during the TTL. In addition, a device send only one confirm message during the TTL.

- **Case 1:** $D = 10,\ T = 1\ day,\ Ui = 100,\ C = 1,\ F = 0$**.**
  In this case, there are no offline users in the group. Each device uploads 101 messages and downloads 1010 messages during the TTL.
- **Case 2:** $D = 10,\ T = 1\ day,\ L = 1\ day,\ Ui = 100,\ C = 1,\ F = 2$**.**
  In this case, there are two offline users in the group and the average offline time is one day. Each device uploads 301 messages and downloads 3010 messages during the TTL.

6.4 Scalability of Grouper

Section 6.2 and Section 6.3 show that Grouper is able to support a group with a hundred of members, and its data synchronization process does not influence user experience of an application.

Chapter 7

Conclusion

This thesis concentrates on developing mobile applications on top of a self-destruction system with data recovery. For this goal, we have developed Grouper, a framework using a secret sharing scheme and multiple untrusted servers, to implement light-weight information sharing in mobile applications.

Chapter 1 points out the problems of the mobile applications using the central servers to store user data. Although some data self-destruction systems try to solve this problem by encrypting user data and keeping it in the servers temporarily, they do not support data recovery after deleting data. Therefore, we began developing the Grouper framework on top of a self-destruction system and adding data recovery.

Chapter 2 shows related work. First, we show data self-destruction systems including Vanish, SafeVanish, SeDas and CloudSky. They are suitable for developing a Bulletin Board Service (BBS) without data recovery. Compared to conventional self-destruction systems, Grouper provides the reliable synchronization by data recovery and allows mobile devices to synchronize data with other devices after untrusted severs remove the data. Second, we show data encryption systems including Mylar, DepSky and SPORC. These systems usually need key generation and distribution. Compared to pure data encryption approaches, Grouper improves dependability by using multiple untrusted servers and an extended secret sharing scheme $f(k, n, s)$. Third, we show an application Sweets that uses P2P connections to synchronize user data. Compare to Sweets, Grouper does not require that two devices are online at the same time to synchronize data.

Chapter 3 describes the threat model of Grouper. Grouper exchanges messages via multiple untrusted servers. We assume that servers do not attack Grouper actively, two devices can use safe data transportation at the user invitation time, all group members are trusted, and server providers are independent.

Chapter 4 presents the design of Grouper. Grouper has a client framework and a Web service. The Grouper client framework consists of the Grouper API layer, synchronization plugin, messaging layer and data protection layer. The Grouper API layer provides methods for sharing data among devices of a group. Based on the basic reliable multicast technique, the the Grouper API layer ensures reliable messages delivery using control messages that include active sequence numbers. The synchronization plugin helps Grouper to synchronize among devices. The messaging layer provides a message transport service with multicasting capability among devices. The data protection layer uses the extended secret sharing scheme $f(k, n, s)$ to protect the user data. The messaging layer uses the Web service running on untrusted servers to exchange shares among devices. To manage a group, Grouper provides two functions, including group creation and member invitation.

Chapter 5 shows the implementation details of Grouper. We have implemented the Web service in the Java language and Grouper's client framework in the Objective-C language. The Web service running on the Tomcat server provides RESTful APIs for the client framework. The managers including SenderManager, ReceiverManager and GrouperManager in the client framework invoke the RESTful API to send/receive messages and manage groups. Grouper's client framework can be installed with the CocoaPods dependency manager easily. In addition, we have developed two applications, *Account Book* and *Test*, using Grouper. We show how to develop the Account Book application using Grouper including framework initialization, and object creation, updating, deletion, and synchronization. We show the main functions of the Test application.

Chapter 6 evaluates Grouper and discusses its scalability. The development of these applications demonstrates that the Grouper framework requires little development efforts to convert an standalone application into a data sharing application. We also measured the performance of Grouper using our benchmark application and discussed the number of messages. The results prove that the performance of Grouper is satisfactory for mobile applications that are used among a small group of people. In addition, using Grouper in an application does not influence user experience.

Based on the self-destruction system with data recovery, the Grouper framework is able to extend a standalone application to a data sharing application for a small group. To develop Grouper, we take advantage of the secret sharing scheme and propose a new extended secret sharing scheme $f(k, n, s)$. Compared to the existing systems, Grouper has two contributions: data recovery in a self-destruction system and easy client APIs for mobile application development.

In the future, we would like to support additional platforms, including Android. We also have a plan to use file-oriented APIs for exchanging messages among devices.

Acknowledgements

The author would like to express my gratitude to my supervisor Professor Yasushi Shinjo for the useful comments, remarks and engagement through the learning process of this master thesis.

The author would like to thank Elvis Nunẽz for helping us to fix some Objective-C related issues of the Sync framework, and Bernard Paulus for fixing a bug of c-SSS, an implementation of the secret sharing scheme.

The author would like to thank Associate Professor Akira Sato, Associate Professor Hisashi Nakai and other members in our Softlab for giving lots of support and suggestion for this research.

The author would like to thank Professor Au Thien Wan with Unniversiti Teknologi, Brunei, for giving some suggestions about future work, and Associate Professor Dan Li with Northeast Forestry University, China, for giving some suggestions about design of the framework. The author would like to thank Yifan Li for giving some suggestions about English grammar.

Bibliography

[1] Roxana Geambasu, Tadayoshi Kohno, Amit A. Levy and Henry M. Levy. Vanish: Increasing data privacy with self-destructing data. In *18th USENIX Security Symposium*, pages 300–315, 2009.

[2] Lingfang Zeng, Zhan Shi, Shengjie Xu and Dan Feng. Safevanish: An improved data self-destruction for protecting data privacy. In *IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 521–528, 2010.

[3] Lingfang Zeng, Shibin Chen, Qingsong Wei and Dan Feng. SeDas: A self-destructing data system based on active storage framework. In *Asia-Pacific Magnetic Recording Conference(APMRC)*, pages 1–8, 2012.

[4] Lingfang Zeng, Yang Wang and Dan Feng. CloudSky: a controllable data self-destruction system for untrusted cloud storage networks. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 352–361, 2015.

[5] Meng Li and Yasushi Shinjo. Grouper: a framework for developing mobile applications using a secret sharing scheme and untrusted servers. In *Proceedings of the 6th International Conference on Network, Communication and Computing*, 2017.

[6] Elvis Nunẽz. Sync, a modern Swift JSON synchronization to Core Data. Retrieved Oct 3, 2017 from https://github.com/SyncDB/Sync.

[7] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica and Harlan Yu. OpenDHT: a public DHT service and its uses. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '05)*, pages 73–84, 2005.

[8] Azureus Software Inc. Azureus. Retrieved Aug 19, 2017 from http://www.vuze.com.

[9] Cholez Thibault, Chrisment Isabelle and Festor Olivier. Evaluation of sybil attacks protection schemes in KAD. In *Third International Conference on Autonomous Infrastructure, Management and Security*, volume 9, pages 70–82. Springer, 2009.

[10] Scott Wolchok, Owen S. Hofmann, Nadia Heninger, Edward W. Felten, J. Alex Halderman, Christopher J. Rossbach, Brent Waters and Emmett Witchel. Defeating Vanish

with low-cost Sybil attacks against large DHTs. In *17th Proceedings of the Network and Distributed System Security Symposium*, pages 1–15, 2010.

[11] Yulai Xie, Muniswamy Reddy, Kiran Kumar, Dan Feng, Darrell DE Long, Yangwook Kang, Zhongying Niu and Zhipeng Tan. Design and evaluation of Oasis: An active storage framework based on T10 OSD standard. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–12. IEEE, 2011.

[12] Mihir Bellare, Ran Canetti and Hugo Krawczyk. HMAC: Keyed-hashing for message authentication. *Internet Request for Comment RFC 2104*, 1997.

[13] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich and Hari Balakrishnan. Building Web applications on top of encrypted data using Mylar. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 157–172, 2014.

[14] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André and Paulo Sousa. DepSky: Dependable and secure storage in a Cloud-of-Clouds. In *ACM Transactions on Storage (TOS)*, volume 9, pages 12:1–12:33, 2013.

[15] Ariel J. Feldman, William P. Zeller, Michael J. Freedman and Edward W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, volume 10, pages 337–350, 2010.

[16] Clarence A. Ellis and Simon J. Gibbs. Concurrency control in groupware systems. In *ACM Sigmod Record*, volume 18, pages 399–407, 1989.

[17] Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI 07)*, 2007.

[18] Rongchang Lai and Yasushi Shinjo. Sweets: A decentralized social networking service application using data synchronization on mobile devices. In *12th EAI International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 188–198, 2016.

[19] The OpenID Foundation. OpenID Connect. Retrieved Jan 7, 2018 from http://openid.net/connect/.

[20] The OpenID Foundation. OpenID. Retrieved Jan 7, 2018 from http://openid.net/.

[21] Guillaume Smith, Roksana Boreli and Mohamed Ali Kaafar. A layered secret sharing scheme for automated profile sharing in OSN Groups. In *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 487–499, 2013.

[22] Liaojun Pang and Yumin Wang. A new (t, n) multi-secret sharing scheme based on Shamir's secret sharing. *Applied Mathematics and Computation*, 167(2):840–848, 2005.

[23] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[24] Russ Allbery and Charles H. Lindsey. Netnews Architecture and Protocol. RFC 5537, November 2009.

[25] Apple Inc. The Multipeer Connectivity framework. Retrieved Oct 3, 2017 from https://developer.apple.com/documentation/multipeerconnectivity.

[26] The Apache Software Foundation. Apache Tomcat. Retrieved Aug 19, 2017 from http://tomcat.apache.org.

[27] Oracle Corporation. MySQL. Retrieved Dec 25, 2017 from https://www.mysql.com.

[28] Pivotal Software Inc. Spring. Retrieved Aug 19, 2017 from http://spring.io.

[29] Red Hat Software Inc. Hibernate. Retrieved Aug 19, 2017 from http://hibernate.org.

[30] Apple Inc. Core Data Programming Guide, Guides and Sample Code. Retrieved Oct 3, 2017 from https://developer.apple.com/library/content/ documentation/Cocoa/Conceptual/CoreData/.

[31] Fletcher T. Penney. c-SSS, an implementation of Shamir's Secret Sharing. Retrieved Oct 3, 2017 from https://github.com/fletcher/c-sss.

[32] AFNetworking Group. AFNetworking, a delightful networking framework for iOS, OS X, watchOS, and tvOS. Retrieved Oct 3, 2017 from http://afnetworking.com.

[33] The CocoaPods Dev Team. CocoaPods Dependency Manager. Retrieved Dec 26, 2017 from https://cocoapods.org/.

[34] The Mental Faculty. Ensembles. Retrieved Sep 28, 2017 from http://www.ensembles.io.

[35] No Thirst Software Limited Liability Company. TICoreDataSync. Retrieved Sep 28, 2017 from https://github.com/nothirst/TICoreDataSync.