Developing Mobile Applications on Top of a Self-destruction System with Data Recovery

Meng Li

(Master's Program in Department of 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
at the
University of Tsukuba

Month 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 paper presents Grouper, a framework for developing mobile applications without trusted central servers. 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 and a Web service. We have implemented 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 have implemented two applications using Grouper: an iOS application named Account Book, and a benchmark application named Test. We have evaluated the development efforts of applications using Grouper 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

1 Introduction **1**

2 Related Work **3**

2.1 Self-destruction Systems . . . 3

2.1.1 Vanish . . . 3

2.1.2 SafeVanish and SeDas . . . 3

2.1.3 CloudSky . . . 3

2.2 Data Encryption Systems . . . 4

2.2.1 Mylar . . . 4

2.2.2 Sweets . . . 5

2.2.3 DepSky . . . 5

2.2.4 SPORC . . . 5

3 Threat Model **6**

3.1 Target Application . . . 6

3.2 Assumptions . . . 6

4 Design **8**

4.1 Overview . . . 8

4.2 Architecture . . . 9

4.3 Grouper API . . . 10

4.4 Reliable Message Delivery . . . 11

4.4.1 Basic Reliable Multicasting . . . 11

4.4.2 Data Recovery on Self-destruction System . . . 12

4.5 Data Protection . . . 13

4.6 Grouper Message Protocol . . . 15

4.7 Group Management . . . 16

4.7.1 Group Creation . . . 16

4.7.2 Member Invitation . . . 17

5 Implementation **19**

5.1 Client Framework . . . 19

5.1.1 Dependencies of Client Framework . . . 19

5.1.2 Sender Manager . . . 20

5.1.3 Release and Installation 21
5.2 Web Service 21
5.3 Application 22
5.3.1 Initialization of Grouper 22
5.3.2 Object Creation and Updating 23
5.3.3 Object Deletion 24
5.3.4 Synchronization from Untrusted Servers 24
5.3.5 Sending Confirm Message 25

6 Evaluation **27**
6.1 Development Efforts 27
6.2 Performance 28
6.3 Discussion on the Number of Messages 29
6.4 Scalability of Grouper 32

7 Conclusion **33**

Acknowledgements **34**

Bibliography **35**

List of Figures

2.1 Architecture of DepSky. 4

4.1 Architecture of Grouper. 9
4.2 Basic reliable multicasting. 12
4.3 Implementing reliable messaging with continuous sequence numbers. 13
4.4 Message transportation using the extended secret sharing scheme $f(2, 4, 3)$. 14
4.5 Member invitation. 17

5.1 Member invitation. 20
5.2 Instance methods of $SenderManager$. 21
5.3 Initialize Grouper in the Account Book application. 22
5.4 Add and delete a record in the Account Book demo application. 23
5.5 Create a new record and send an update message. 24
5.6 Delete a record and send a delete message. 24
5.7 Receive messages from untrusted server and send a confirm message. 25
5.8 Receive messages and send a confirm message in the Account Book demo application. 26

6.1 Processing time of uploading and downloading multiple messages. 29
6.2 Processing time of uploading and downloading a single message with a constant k or n. 30

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 CouldSky[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 are going to develop such mobile applications on top of a self-destruction system with data recovery. Concretely, we are developing Grouper, a framework for developing mobile applications using such a self-destruction system. 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[5] 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 works in Chapter 2. In Chapter 3, we describe the threat model and some assumption in our framework Grouper. Then, we describe the design of Grouper by introducing its layers, and the way to send or receive among devices using our Grouper Message protocol in Chapter 4. Next, we describe the three parts of implementation of Grouper: client framework, Web service and application detailedly in Chapter 5. In Chapter 6, we give the evaluation of Grouper from developer efforts and performance. Finally, we conclude the paper and introduce the future works in Chapter 7.

Chapter 2

Related Work

In this chapter, we introduce the following related works. 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

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[6] or VuzeDHT[7] which is controlled by a single maintainer. Thus, it is not very secure against some Sybil attacks[8]. In addition, the surviving time of the key in Vanish cannot be controlled by the applications.

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 by extending the length range of key shares while SeDas exploits active storage networks instead of P2P nodes to maintain a divided secret key.

2.1.3 CloudSky

By extending SeDas, Zeng's group proposed CloudSky[4], a controllable data self-destruction system for untrusted cloud storage. In CloudSky, a user can control the surviving time of a message. CloudSky uses Attribute Bases Encryption (ABE) and allows users to define access control policies.

However, these systems are not suitable for developing information sharing applications.

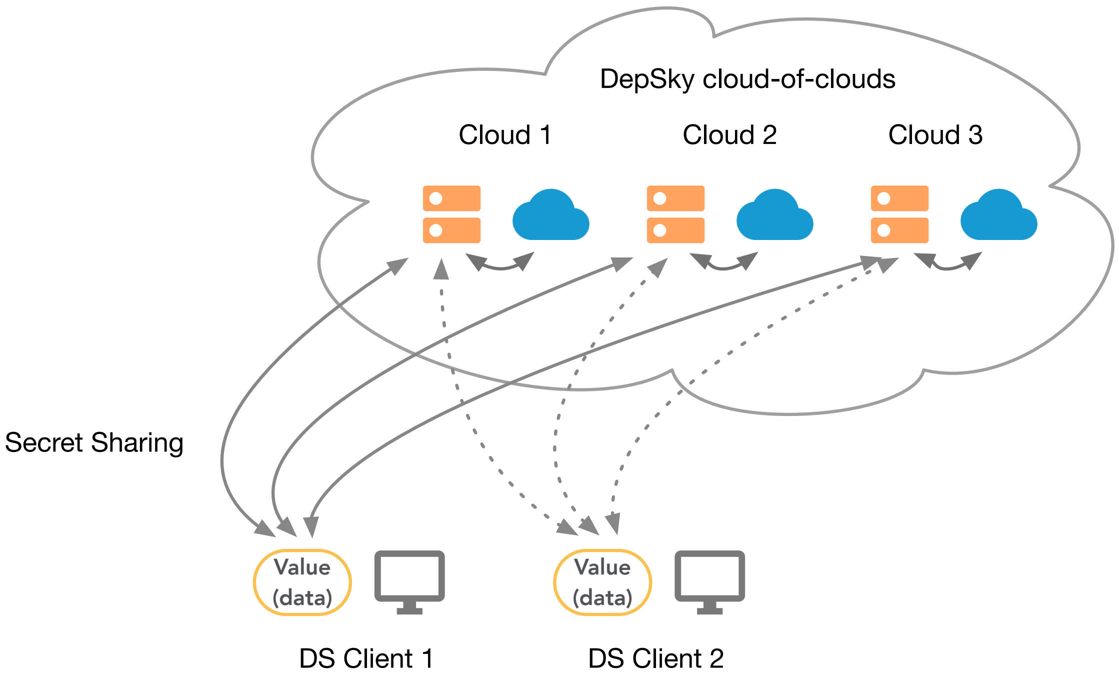


Figure 2.1: Architecture of DepSky.

They are suitable for exchanging immutable messages, whereas our target applications need to modify objects repeatedly after other devices receive the object. Although CloudSky solves the problems regarding user controllability that exist in Vanish, offline devices 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 Data Encryption Systems

2.2.1 Mylar

Mylar[9] stores encrypted data on an untrusted server and decrypts this data only in the browsers of users. Application developers of Mylar use its API to encrypt regular (non-encrypted) Web applications.

Mylar uses its browser extension to decrypt data on the client-side. 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.

2.2.2 Sweets

Sweets[10] is a decentralized social networking service (SNS) application 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. Direct synchronization 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.

2.2.3 DepSky

DepSky[11] implements the concept of *Cloud-of-Clouds*. DepSky keeps encrypted data in commercial storage services and performs application logic in other individual servers. In Grouper, untrusted servers undertake the responsibility of temporary data storage and message delivery. Mobile devices perform application logic.

2.2.4 SPORC

SPORC[12] is designed for developing group collaboration applications using untrusted cloud resources. To maintain synchronization among clients, SPORC uses operational transformation (OT), which allows each client to apply local updates optimistically. In SPORC, synchronization should be done among online nodes, while Grouper allows synchronization among offline nodes.

Chapter 3

Threat Model

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

3.1 Target Application

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 create a new object and share it with other grouper members. If an other user synchronized the new object, he / she can edit the content of the object or delete the object. The modification or deletion by this user is able to synchronized to the devices of other group members.

Our target applications are suitable for those groups which are sensitive for their shared data. Those groups requires their shared data cannot be accessed by the server manager.

3.2 Assumptions

To implement our target mobile applications, there are following assumptions which is necessary 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 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 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[13]. At least k or more shares can reconstruct the secret, but $k-1$ or fewer shares output nothing about the secret[14]. 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,

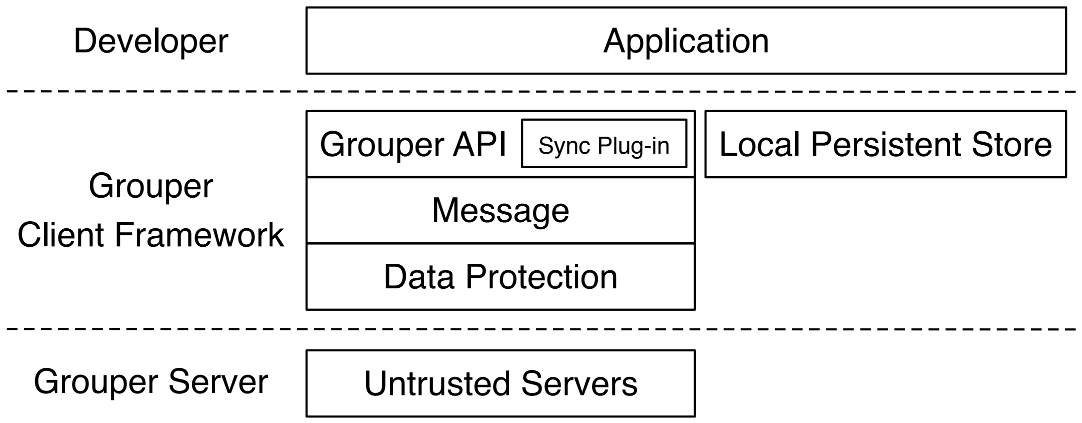


Figure 4.1: Architecture of Grouper.

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

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 used in this application. The developer should pay attention to the order of entity name in the array. If entity A is reference by an 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.

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.

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 a simple client 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 application needs to update an object in all devices, it invokes the method

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.

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 3.4.

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 user the Sync framework[5] 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 introduce an existing technique to solve the reliable message delivery problem called reliable multicasting at first. Then we reference the reliable multicasting technique to design our own solution which is more suitable for Grouper.

4.4.1 Basic Reliable Multicasting

Figure 4.2 shows the basic reliable multicasting technique introduced in Distributed Systems[15]. In this reliable multicasting technique, each message has a sequence number for each sender. 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

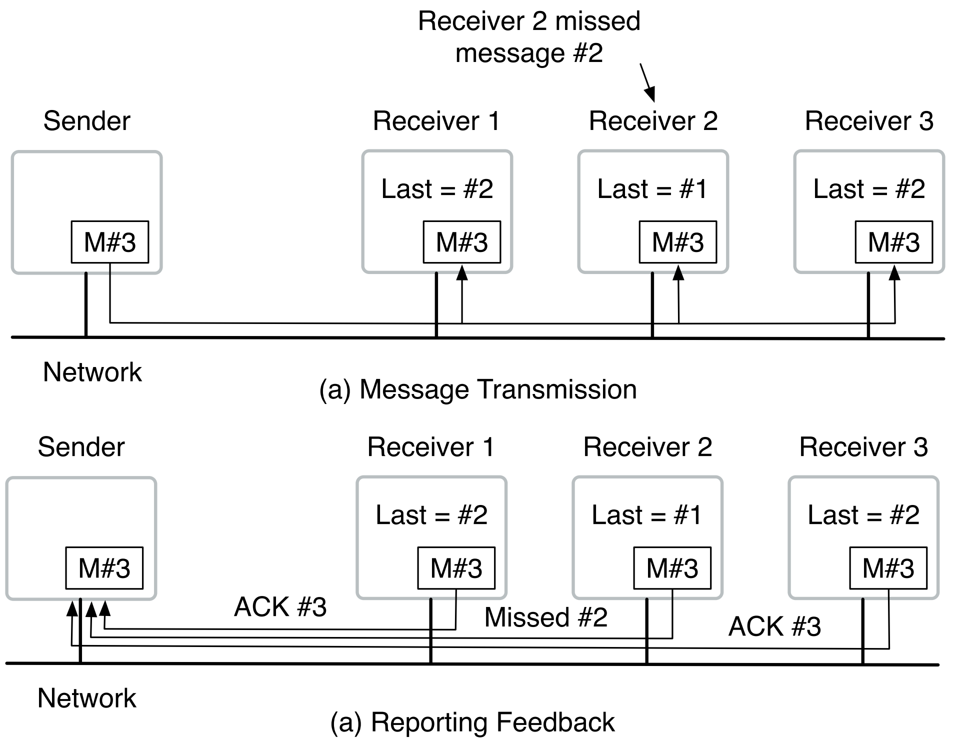


Figure 4.2: Basic reliable multicasting.

the update message. For example, consider that a sender sends update message No.3 to all other members using a multicast address. When a member receives update 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 update message No.2. The member asks the sender to resend update message No.2 using a control message. The sender will send update 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 servers delete messages within a short time. In a word, the basic reliable multicasting technique is not suitable for data recovery on our self-destruction system.

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

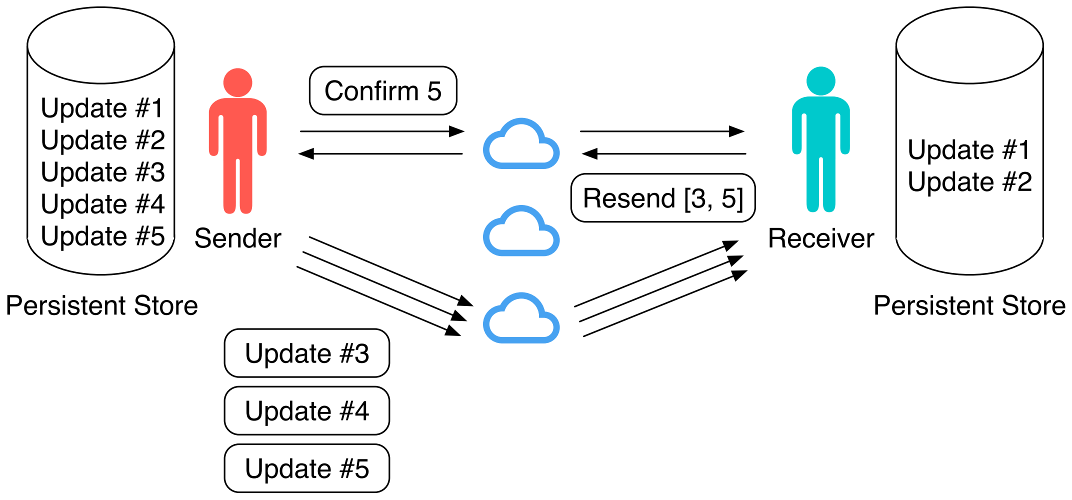


Figure 4.3: Implementing reliable messaging with continuous sequence numbers.

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[16]. 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.

4.5 Data Protection

The data protection layer uses Shamir's secret sharing scheme $f(k,n)$ as introduced 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

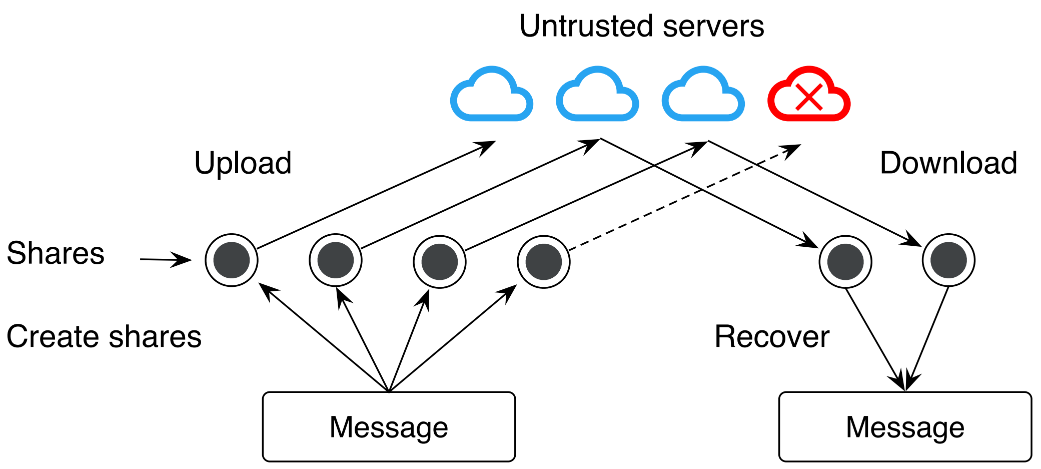


Figure 4.4: Message transportation using the extended secret sharing scheme $f(2, 4, 3)$.

$k \leq s \leq n$. 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. 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 more likely to receive a message.

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. 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. Table 4.3 shows the attributes in a Grouper 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 the identifier of a deleted object. A confirm message contains the sequence numbers 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.

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.

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

Developers should send and receive Grouper messages in their applications by invoking the client API of Grouper. 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 sequence numbers from the message content and removes the sequence numbers that exist in the device. 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 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

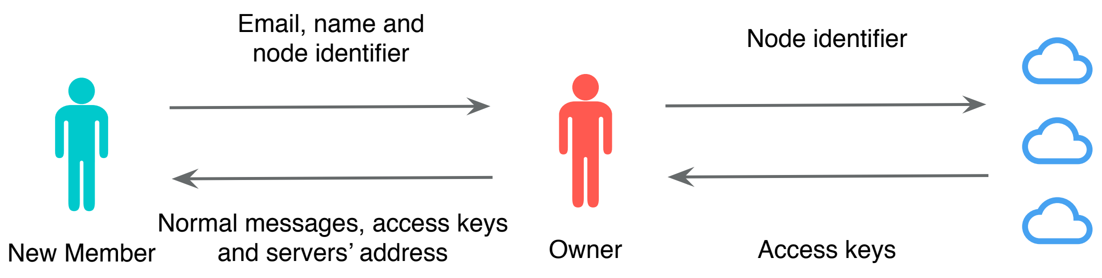

Email, name and node identifier
Node identifier
New Member
Normal messages, access keys and servers' address
Owner
Access keys

Figure 4.5: Member invitation.

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 4.5. To join the group, a new member first prepares his/her user information at first. 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*[17]. 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 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 and the list of existing members to the new member. After receiving them, the new member can 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 the minimum and
11:            maximum sequence number.
12:        $sendMsg(resendMsg,\ sender)$
13:    end if
    ▷ Handle the message by its type.
14:    if $msg.type = "update"$ then
15:        $sync.updateRemote(msg)$
16:    else if $msg.type = "delete"$ then
17:        $sync.deleteRemote(msg)$
18:    else if $msg.type = "confirm"$ then
19:        $maxSeq = getMaxSeqFrom(msg.content)$
20:        $resendMsg \leftarrow createResendMsg(lastMsg.seq +$
21:                $1,\ maxSeq)$
22:        if $resendMsg \neq null$ then
23:            $sendMsg(resendMsg,\ sender)$
24:        end if
25:    else if $msg.type = "resend"$ then
26:        $seqs \leftarrow getSeqs(msg.content)$
27:        for $seq \in seqs$ do
28:            $missingMsg \leftarrow getMsg(seq)$
29:            $sendMsg(missingMsg,\ sender)$
30:        end for
31:    end if
32: 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 client framework in Section 5.1, the implementation of the Web service in Section 5.2, and the demo applications in Section 5.3.

5.1 Client Framework

This section describes how we implement 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 GUI for group management, so that developers need not to implement it on the iOS platform.

In this section, we introduce the dependences of client framework in Subsection 5.1.1. We use singleton pattern to implement the manager classes in the client framework, to avoid multiple instances of a manager class. These manager classes are listed from Subsection

5.1.1 Dependencies of Client Framework

It makes use of the following frameworks in Figure 5.1. The main functions of this frameworks are shown as following.

- **Multipeer Connectivity**[17], 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**[18], an official Object-Relational Mapping (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**[5], a lightweight Swift framework for data synchronization using Core Data. Sync uses the JSON string to synchronize managed objects of Core Data among devices. Sync provides a instance method *export*() for the managed object to convert the it into a

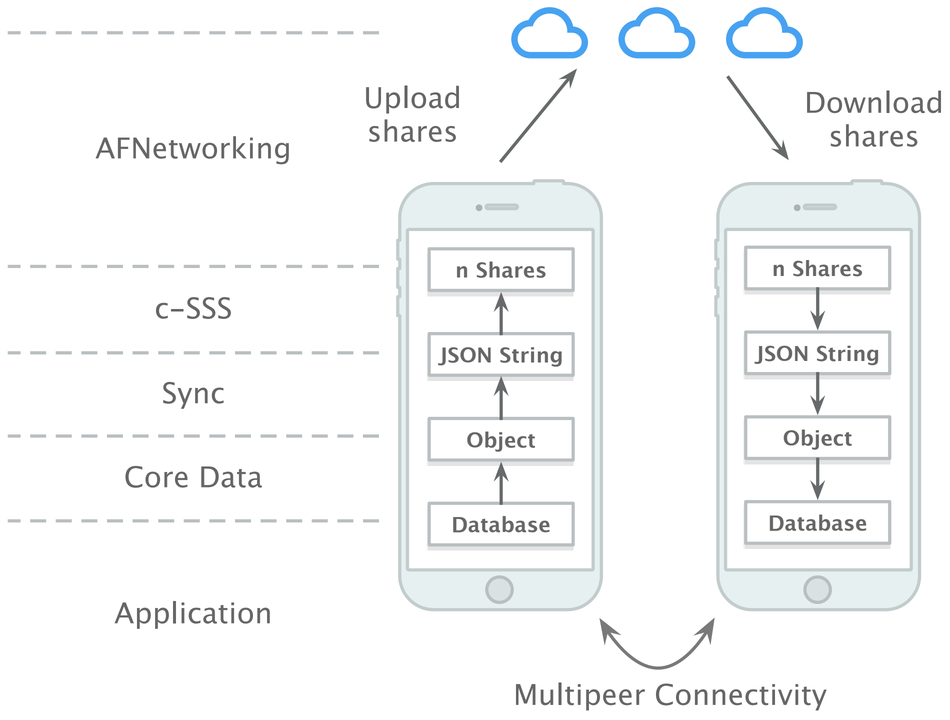


Figure 5.1: Member invitation.

JSON string and an asynchronous method $sync(json, objectName, callback)$ for data stack to synchronize the JSON string into Core Data. Grouper currently uses it in the synchronization plugin.

- **c-SSS**[19], an implementation in the original C code of the secret sharing scheme. It provide two main functions for generate shares and reconstructing them. The function $generate\_share\_strings(char*secret, int\ n, int\ k)$ can generate n shares from the string secret with the threshold k. The function $extract_secret_from_share_strings(constchar * shares)$ can recover the string after accessing to more than k shares.
- **AFNetworking**[20], a networking library in the Objective-C language. Grouper uses it to invoke the RESTful API provided by our several Web services running on multiple untrusted servers.

5.1.2 Sender Manager

The class $SenderManager$ provides the methods to create and send the Grouper message as shown in Figure 5.2.

```
// 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 update messages for multiple sync entities.
- (void)updateAll:(NSArray *)entities;

// Send update messages for multiple sync entities with callback function.
- (void)updateAll:(NSArray *)entities withCompletion:(SendCompletion)completion;

// Delete multiple sync entities and send delete messages.
- (void)deleteAll:(NSArray *)entities;

// 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 5.2: Instance methods of $SenderManager$.

5.1.3 Release and Installation

5.2 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.

Our Web service provides a RESTful API to clients. It runs on a Tomcat server[21] that is an open-source implementation of the Java Servlet, JavaServer Pages, Java Expression Language, and Java WebSocket technologies. We use the Spring MVC[22], a Web model-view-controller framework, to create our RESTful API, and Hibernate[23], an open-source Java ORM framework, to save and operate objects in the Web service.

Our Web service includes three kinds of entities: the *Group*, *User*, and *Share* entities. The *Group* entity saves the group ID, group name, and its owner. The *User* entity saves the node identifier of a user, the access key for this user, and the group entity of this user. The *Share* entity saves a share generated with the secret sharing scheme, at time when a client uploads the share.

```
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 5.3: Initialize Grouper in the Account Book application.

5.3 Application

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.

We will introduce how to extend a standalone application into a data sharing application using Grouper. In this section, we will show how we use Grouper to develop the demo application Account Book. We have introduced the basic functions of Account Book in Section 4.1. The standalone application of Account Book can record the income and expenditure of a small group by group members. Without data sharing, records by a group member cannot be shared with other members. To extend the stand alone application, we need follow the five steps as the following.

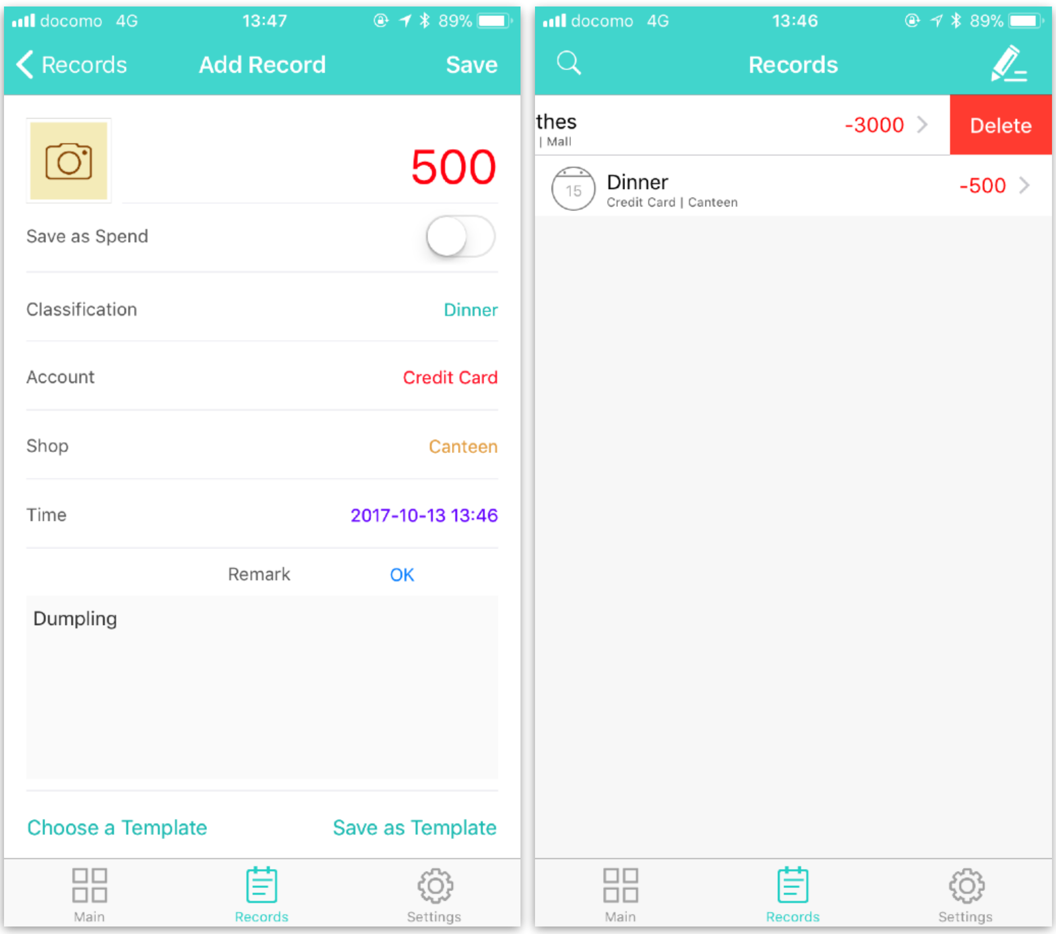
5.3.1 Initialization of Grouper

At first, we invoke the *grouper.setup*() method introduced in Table 4.1 to initialize Grouper in the method $-(BOOL)application : didFinishLaunchingWithOptions :$ of the AppDelegate at first. Figure 5.3 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 in line 1. Because our framework supports the GUI if iOS, we can provide the storyboard instance to Grouper to use the GUI for creating a new group or joining an existing group provided by Grouper. For this reason, we get the storyboard instance in line 2. In line 3, we invoke the initialization method to initialize Grouper.

We have emphasized the order of the parameter entities in Table 4.1. The reference relationships among entities Classification, Account, Shop, Photo, Template and Record are as following.

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

We can see the array in the code above following the principle introduced in Table 4.1. For each entity, if it is referenced by an other entity, it is in front of the entity. After



500

Save as Spend

Classification Dinner

Account Credit Card

Shop Canteen

Time 2017-10-13 13:46

Remark OK

Dumpling

Choose a Template Save as Template

thes -3000 Delete
| Mall

15 Dinner -500
Credit Card | Canteen

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

Figure 5.4: Add and delete a record in the Account Book demo application.

initializing the Grouper framework in the Account Book application, we can invoke other methods in Table 4.1.

5.3.2 Object Creation and Updating

Figure 5.4(a) shows the screenshot of adding a record in Account Book. 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 stand alone application contains the line 2 to line 8. To share this record with other group members, we invokes the $grouper.sender.update()$ method of Grouper to share this record with other group members after saving in the local persistent store in line 11.

After invoking the method $grouper.sender.updated()$, this record will be divided to

```
 1  // Save record to app local persistent store.
 2  Record *record = [dao.recordDao saveWithMoney:money
 3                           remark:_remarkTextView.text
 4                           time:_selectedTime
 5                           classification:_selectedClassification
 6                           account:_selectedAccount
 7                           shop:_selectedShop
 8                           photo:photo];
 9
10  // Invoke update method in sender to send an update message.
11  [grouper.sender update:record];
```

Figure 5.5: Create a new record and send an update message.

```
1  Record *record = [records objectAtIndex:indexPath.row];
2  if (editingStyle == UITableViewCellEditingStyleDelete) {
3      // Delete the record in the local persistent store and other devies.
4      [grouper.sender delete:record];
5      // Update user interface.
6      [records removeObjectAtIndex:indexPath.row];
7      [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
              withRowAnimation:YES];
8  }
```

Figure 5.6: Delete a record and send a delete message.

several shares and uploaded to multiple untrusted servers. This method is also suitable for record updating. Note that after invoking this method, data of the object has only been uploaded to multiple untrusted servers. Grouper does not ensure that other grouper members can synchronize this object successfully.

5.3.3 Object Deletion

Figure 5.4(b) shows the screenshoot of deleting a record in Account Book. A user swipes a cell from right to left to delete a record from the record list. In the stand alone 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 in line 4.

By invoking the method *sender.delete*(), the record in the local persistent store will be deleted at first, so we need not to delete it on the local device before invoking this method like the method *sender.update*().

5.3.4 Synchronization from Untrusted Servers

Grouper provide the method *grouper.receiver.receive*() to synchronize messages from untrusted servers. Developers can decide when and how to synchronize messages that contains the updated or deleted objects by themselves. For example, we provide the following two ways in Account Book for users to synchronize messages.

```
1 [grouper.receiver receiveWithCompletion:^(int success, Processing *processing) {
2     long now = (long)[[NSDate date] timeIntervalSince1970];
3     if (now - grouper.group.defaults.controlMessageSendTime > grouper.group.defaults.
          interval * 60) {
4         [grouper.sender confirm];
5         // Update the last time to send a confirm message.
6         grouper.group.defaults.controlMessageSendTime = now;
7     }
8 }];
```

Figure 5.7: Receive messages from untrusted server and send a confirm message.

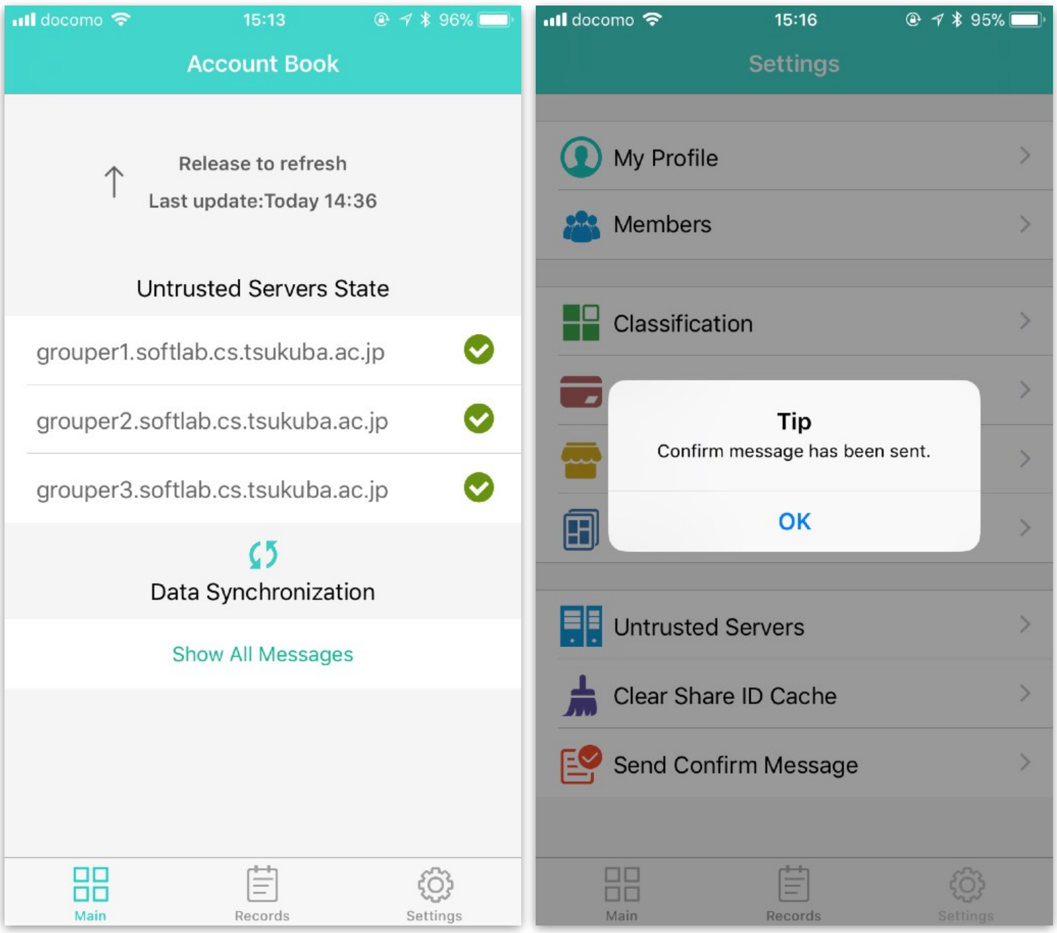
- **Manually.** User can pull down the view as shown in Figure 5.8(a) to synchronize messages.
- **Automatically when the application is launched.** Account Book tries to synchronize messages automatically when it is launched.

We invoke the method $grouper.receiver.receive()$ to receive and synchronize messages, and update user interface or do other things in the callback block. The parameter success in the block indicates the state of synchronization. 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 synchronization will be failed and parameter success will be false.

5.3.5 Sending Confirm Message

We have introduced that method $grouper.confirm()$ is invoked periodically or when the device becomes online in Section 4.6. In the Account Book application, we provide the following two ways to send a confirm message.

- **Manually.** User can send a confirm message manually in the Setting table view as shown in Figure 5.8(b).
- **After receiving messages.** As described from line 2 to line 7 in Figure **??**, if Account Book receives messages from untrusted servers successfully, Account Book will check the last time to send the confirm message. If the time difference between now and the last time is longer than the TTL defined by user, Account Book will send a confirm message.



Release to refresh

Last update:Today 14:36

grouper1.softlab.cs.tsukuba.ac.jp

grouper2.softlab.cs.tsukuba.ac.jp

grouper3.softlab.cs.tsukuba.ac.jp

Tip

Confirm message has been sent.

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

Figure 5.8: Receive messages and send a confirm message in the Account Book demo 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 a 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[5] 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 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 consistency, he/she can choose Ensembles[24]. 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[18] and TICoreDataSync[25] 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.

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

6.2 Performance

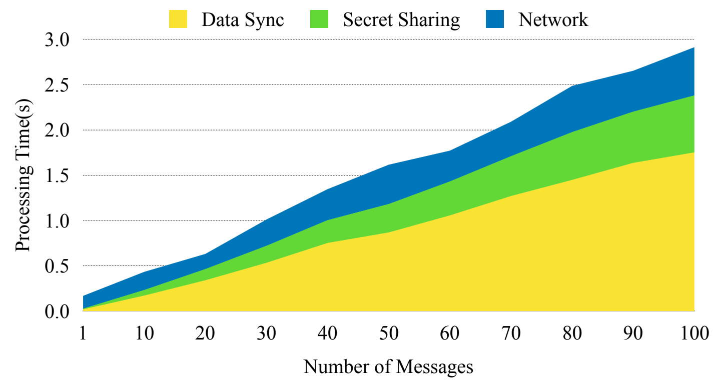
In this section, we show that mobile applications using the Grouper framework provide satisfactory performance for small groups of people. Table 6.2 lists the hardware and software configuration of our experiment environment. In our performance experiments, we used the benchmark application *Test* to 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.

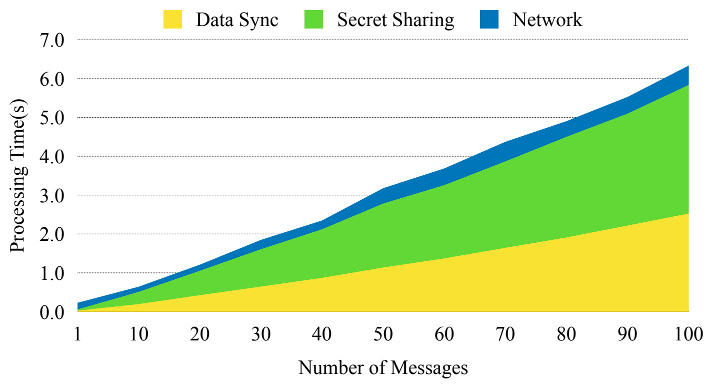
We performed an experiment to measure the processing time of object synchronization according to the number of updated objects. In this experiment, we set the secret sharing scheme to $f(2, 3, 3)$. We sent multiple messages from a device and received them in another device. Concretely, we sent multiple 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 uploading 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 mo-



(a) Uploading multiple messages.



(b) Downloading multiple messages.

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

bile 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 and n 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.

6.3 Discussion on the Number of Messages

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

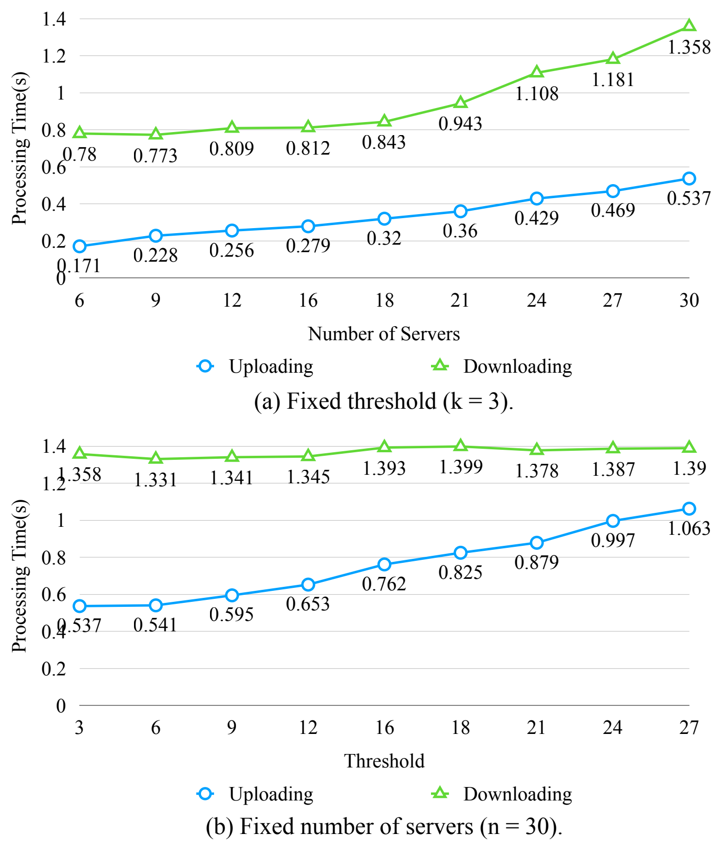


Figure 6.2: Processing time of uploading and downloading a single message with a constant k or n.

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 is has 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.

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.

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

Since the current cloud servers scale well according to the number of devices and 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

From section 6.1 to 6.3, we can conclude that Grouper is able to support a group with a hundred of members, and does not influence user experience of an application.

Chapter 7

Conclusion

This paper describes Grouper, a framework using a secret sharing scheme and multiple untrusted servers, to implement light-weight information sharing in mobile applications. In an application using Grouper, users can create a group and share objects safely via multiple untrusted servers. Grouper provides two main functions: reliable data synchronization and group management. Compared to conventional self-destruction systems, Grouper provides the reliable synchronization and allows mobile devices to synchronize data with other devices after untrusted severs remove the data. Compared to pure data encryption approaches, Grouper improves dependability by using multiple untrusted servers and an extended secret sharing scheme $f(k, n, s)$.

We have implemented Grouper's client framework in the Objective-C language and the Web service in the Java language. In addition, we have developed applications, *Account Book* and *Test*, using Grouper. The development of these applications demonstrate that Grouper requires little development effort to convert an standalone application into a data sharing application. We also evaluated the performance of Grouper using our benchmark application. The results shows that using Grouper in an application does not influence user experience.

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 authors 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.

This work was partially supported by JSPS KAKENHI Grant Number 16H02805.

Bibliography

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

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

[4] Zeng, Lingfang and Wang, Yang and Feng, Dan. 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] Elvis Nunẽz. Sync, a modern Swift JSON synchronization to Core Data. Retrieved Oct 3, 2017 from https://github.com/SyncDB/Sync.

[6] Rhea, Sean and Godfrey, Brighten and Karp, Brad and Kubiatowicz, John and Ratnasamy, Sylvia and Shenker, Scott and Stoica, Ion and Yu, Harlan. 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.

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

[8] Wolchok, Scott and Hofmann, Owen S and Heninger, Nadia and Felten, Edward W and Halderman, J Alex and Rossbach, Christopher J and Waters, Brent and Witchel, Emmett. Defeating Vanish with low-cost Sybil attacks against large DHTs. In *17th Proceedings of the Network and Distributed System Security Symposium*, 2010.

[9] Popa, Raluca Ada and Stark, Emily and Valdez, Steven and Helfer, Jonas and Zeldovich, Nickolai and Balakrishnan, Hari. 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.

[10] 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.

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

[12] Feldman, Ariel J and Zeller, William P and Freedman, Michael J and Felten, Edward W. 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.

[13] Smith, Guillaume and Boreli, Roksana and Kaafar, Mohamed Ali. 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.

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

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

[16] Russ Allbery and Charles H. Lindsey. Netnews Architecture and Protocol. Retrieved Aug 19, 2017 from https://tools.ietf.org/html/rfc5537.

[17] Apple Inc. MultipeerConnectivity. Retrieved Oct 3, 2017 from https://developer.apple.com/ documentation/multipeerconnectivity.

[18] 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/.

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

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

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

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

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

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

[25] No Thirst Software LLC. TICoreDataSync. Retrieved Sep 28, 2017 from https://github.com/nothirst/TICoreDataSync.