Grouper: A Framework for Developing Mobile Applications using a Secret Sharing Scheme and Untrusted Servers

Meng Li
University of Tsukuba
limeng@softlab.cs.tsukuba.ac.jp

Yasushi Shinjo
University of Tsukuba
yas@cs.tsukuba.ac.jp

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 associated with implementing Grouper as well as its and 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.

KEYWORDS

Mobile application security; secret sharing; untrusted server

1 INTRODUCTION

People use mobile applications every day. Conventional mobile applications are built based on a client-server mode and require central servers for storing shared data and processing confidential information. When users use mobile applications, 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[7], SafeVanish[24], SeDas[23] and CouldSky[25] constructed a data self-destruction system as their cloud 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 frameworks for developing mobile applications.

To address these problems, we are developing Grouper, a framework for developing mobile applications. 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 a Sync[15] 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.

2 THREAT MODEL

In this section, we introduce 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. The owner invites other members in a face-to-face manner.

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 on 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.

3 DESIGN

This section describes the design of the Grouper framework.

3.1 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 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
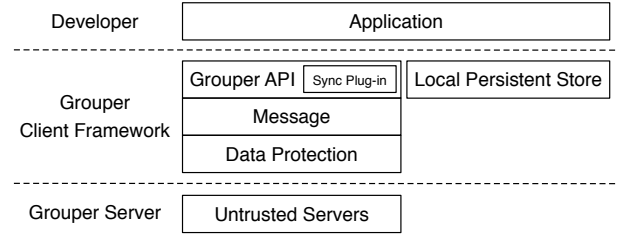

Figure 1: Architecture of Grouper.

using a cryptographic function[20]. At least k or more shares can reconstruct the secret, but $k - 1$ or fewer shares output nothing about the secret[16]. 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, as demonstrated in Section 3.4. 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 3.7.

3.2 Architecture

Figure 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 adds the synchronization function to his application by using the API.
- **Synchronization Plugin.** Grouper uses 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 a 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

Table 1: Client API of Grouper.

Method	Description
$grouper.\textbf{setup}(appId, dataStack)$	An application invokes this method to initialize Grouper with appId and dataStack. AppId is the unique ID of an application. Datastack is used by the synchronization plugin.
$grouper.sender.\textbf{update}(object)$	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.
$grouper.sender.\textbf{delete}(object)$	An application invokes this method when it wants to delete an existing object. Grouper will delete the object and remove it from the persistent store of a device automatically, after sending a message to other devices.
$grouper.receiver.\textbf{receive}(callback)$	An applications 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.
$grouper.sender.\textbf{confirm}()$	An application needs to invoke this method periodically or occasionally to send a confirm message to other devices.

Table 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 in the byte array b of object IDs.

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 subsections describe the details of these layers from the top layer to the bottom layer.

3.3 Grouper API

The Grouper framework provides object synchronization among mobile devices through a simple client API. Table 1 shows the client APIs of Grouper that have been used to develop our target applications. An application initializes the framework by invoking the method $grouper.setup()$. When the application needs to update an object in all devices, it invokes the method $grouper.sender.update()$. When the application needs to delete an object in all devices, it invokes the method $grouper.sender.delete()$. When the application needs to synchronize data, it invokes the method $grouper.receiver.receive()$ 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 $grouper.confirm()$ 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 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 and save unmarshalled data 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[15] 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.

3.4 Reliable Message Delivery

Grouper implements reliable message delivery in the Grouper API layer over the self-destruction system.

To design this, we use a reliable multicasting technique in distributed systems[21]. 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 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 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 the member who request it using a unicast address.

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.

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.
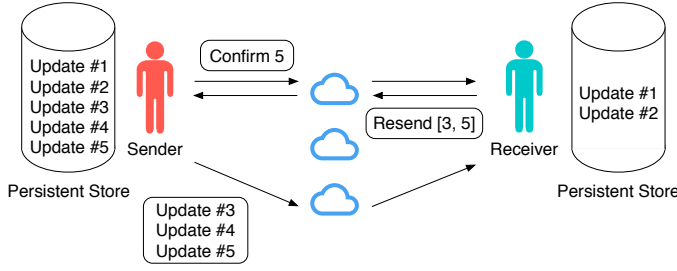
Figure 2: Implementing reliable messaging with continuous sequence numbers.

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

Figure 2 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. Again, 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 2. The receiver receives this sequence number and compares it with the newest sequence number in the persistent store. In Figure 2, 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[2]. 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.

3.5 Data Protection

The data protection layer uses Shamir's secret sharing scheme $f(k, n)$ as introduced in Section 3.1. 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.
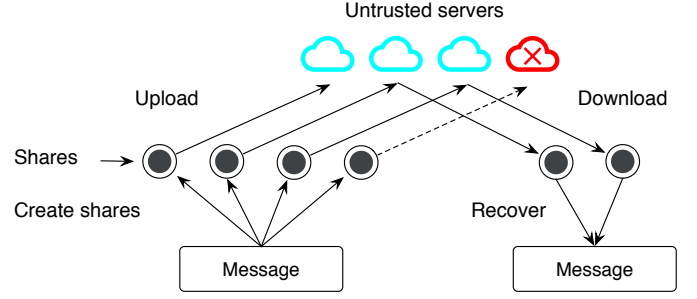
Figure 3 shows the message transportation from a device to another device using the extended secret sharing scheme $f(2, 4, 3)$. In this figure, the message layer in a sender device is sending a message to the message layer in a receiver device. At first, this layer calls the data protection layer and creates four shares, according to the secret sharing scheme. Next, this layer uploads those shares to four untrusted servers. In Figure 3, 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.

3.6 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 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

Table 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.

message contains the identifier of a deleted object. A confirm message contains the sequence numbers of all normal 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. Both the sequence number and the ID of the sender are needed to identify a unique message.

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 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.

Algorithm 1 Handle message 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**

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

Algorithm 1 describes the handle process when the Grouper API layer receives a message. For an update message or a 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.

3.7 Group Management

To manage a group, Grouper provides the following two functions:

- Group Creation.
- Member Invitation.

A user can create a group, and the creator becomes the owner of the group. Before creating a group, the owner prepares his 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 node identifier and the TTL to multiple untrusted servers. The node identifier, which represents his 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 according to the master key.

After creating a group, the owner can invite new members to his group. To join the group, a new member first prepares his 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*[9]. First, the new member sends usethe r information and a node identifier to the owner. The owner saves the user information and node identifier to his device. Secondly, 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.

4 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 4.1, the implementation of the Web service in Section 4.2, and the demo applications in Section 4.3.

4.1 Client Framework

Grouper's client framework is written in Objective-C, and it supports developing applications on iOS, macOS, watchOS and tvOS. It makes use of the following frameworks.

- **Multipeer Connectivity**[9], 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.
- **Core Data**[8], 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**[15], a synchronization framework for Core Data using JSON. Grouper currently uses it in the synchronization plugin.
- **c-SSS**[17], an implementation of the secret sharing scheme.
- **AFNetworking**[1], a networking library in Objective-C. Grouper uses it to invoke the RESTful API provided by our Web services running on multiple untrusted servers.

4.2 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[6] that is an open-source implementation of the Java Servlet, JavaServer Pages, Java Expression Language, and Java WebSocket technologies. We use the Spring MVC[11], a Web model-view-controller framework, to create our RESTful API, and Hibernate[12], 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.

4.3 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.

Figure 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 of a group. When the user click the save button, Account Book invokes the $grouper.sender.update(object)$ method of Grouper to share this record with other group members. Figure 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. When the user click the delete button, Account Book invokes the $grouper.sender.delete(object)$ method of Grouper to delete this record on the devices of other group members.

5 EVALUATION

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

Table 4: Application lines of code.

Application	Platform	Lanaguage	Number of Entities	Standalone Application LoC	Increased LoC
Account Book	iOS	Objective-C	5	8076	190
Test	iOS	Swift	1	621	18
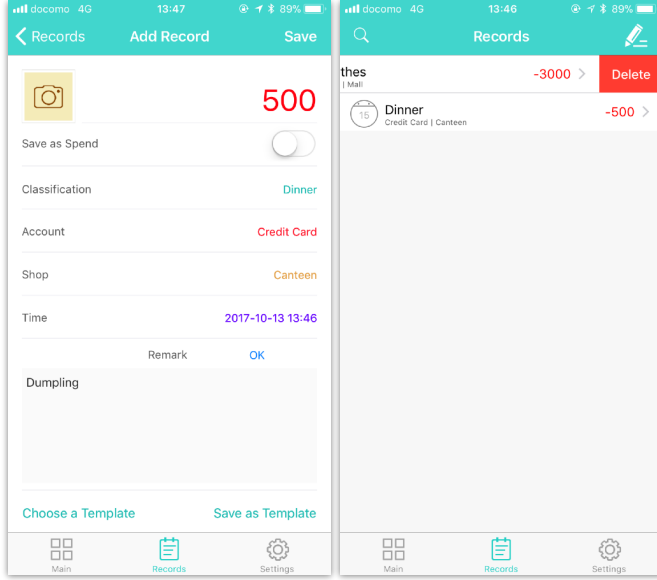
(a) Add a record.   (b) Delete a record.

Figure 4: Screenshots of the Account Book demo application.

5.1 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 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 4.3 demonstrates the implementation of two applications. Table 4 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[15] 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 can choose Ensembles[4].

Note that the synchronization frameworks must provide the interfaces as described in Table 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[8]

Table 5: 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 LTS
Server 2	Core i7-5820K	32GB	Ubuntu 14.04.5 LTS
Server 3	Core i7-5820K	32GB	Ubuntu 14.04.5 LTS

and TICoreDataSync[14] 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 has to add an additional layer that provides the methods in Table 2. For this reason, if a developer wants to use another consistency model, the increased LoC of the application will be more than which in Table 4.

5.2 Performance

In this subsection, we show that mobile applications using the Grouper framework provide satisfactory performance for small groups of people. Table 5 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.
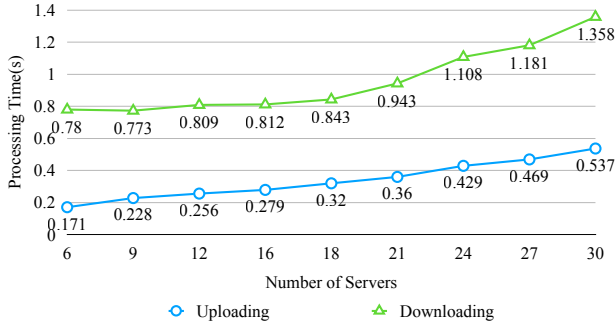
Figure 5 shows the processing time of uploading and downloading multiple messages. We divide the processing time into three parts: data synchronization, secret sharing, and networking. As the number of messages increased, the data synchronization and secret sharing time increased linearly. The networking 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. Furthermore, 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
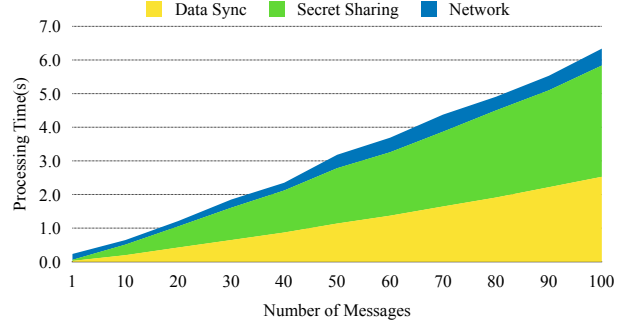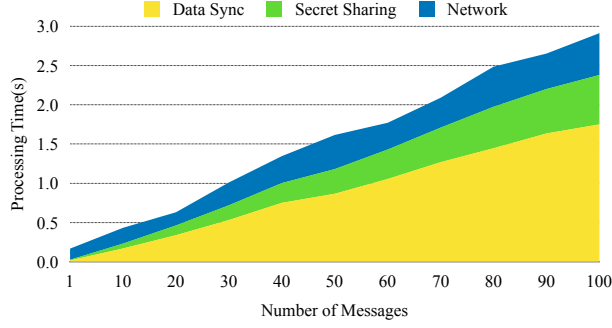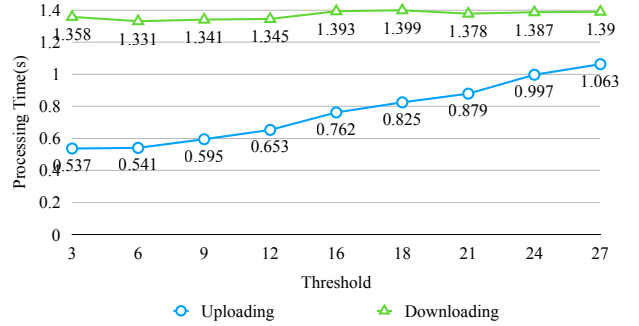
(a) Uploading multiple messages.

(b) Downloading multiple messages.

Figure 5: Processing time of uploading and downloading multiple messages.

(a) Fixed threshold (k = 3).

(b) Fixed number of servers (n = 30).

Figure 6: Processing time of uploading and downloading a single message with a constant k or n.

not freezes. Therefore, a user is not aware of the occurrence 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(a) shows the relationship between processing time and the number of servers n, where $k = 3$ and $n = 2, 4, 6, ..., 10$. As n increased, the processing time of uploading and downloading increased linearly. Figure 6(b) shows the relationship between processing time and the threshold k, where $n = 30$ and $k = 3, 6, 9, ..., 27$. As k increased, the processing time of uploading increased linearly and that of downloading did not change.

5.3 Discussion on the Number of Messages

In this subsection, we discuss and the number of messages. We use the parameters in Table 6.

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:

$$NUi = Ui + C \qquad (1)$$

Total number of uploads NU is the sum of NUi.

$$NU = \sum_{i=1}^{D} NUi = U + C \cdot D \qquad (2)$$

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.

$$NDi = NU = U + C \cdot D \qquad (3)$$

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

$$ND = \sum_{i=1}^{D} NDi = NU \cdot D = (U + C \cdot D) \cdot D \qquad (4)$$

Since the current cloud servers scale well according to the number of devices and and messages, we discuss NUi and UDi in this subsection. Equation 1 shows that for the device i during the TTL, the order of uploading is $O(U)$. Euqation 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

Table 6: Parameters for discussing the relationship between the TTL and 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.

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

$$URi = Ui \cdot \frac{L}{T} \cdot F \qquad (5)$$

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 $F \neq 0$.

$$NUi = Ui + C + Ui \cdot \frac{L}{T} \cdot F = Ui \cdot (1 + \frac{L}{T} \cdot F) + C \qquad (6)$$

$$NU = \sum_{i=1}^{D} NUi = U \cdot (1 + \frac{L}{T} \cdot F) + C \cdot D \qquad (7)$$

$$NDi = NU = U \cdot (1 + \frac{L}{T} \cdot F) + C \cdot D \qquad (8)$$

Equations 6 to 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 RELATED WORK

Vanish is a self-destruction system[7] 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[19] or VuzeDHT[10] which is controlled by a single maintainer. Thus, it is not very secure against some Sybil attacks[22]. In addition, the surviving time of the key in Vanish cannot be controlled by the applications.

To address these issues in Vanish system, Zeng et al. proposed SafeVanish[24] and SeDas[23]. 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. By extending SeDas, Zeng's group proposed CloudSky[25], 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. 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.

Mylar[18] 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.

Sweets[13] 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.

DepSky[3] 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.

SPORC[5] is designed for developing group collaboration applications using untrusted cloud resources. To maintains synchronization among clients, SPORC uses operational transformation (OT), which allows each client to apply local updates optimistically. In SPORC, synchronization of nodes should be done online, while Grouper allows for offline synchronization of nodes.

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's framework in Objective-C and the Web service in Java. 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.

ACKNOWLEDGMENTS

REFERENCES

[1] AFNetworking. 2016. AFNetworking, a delightful networking framework for iOS, OS X, watchOS, and tvOS. (Mar 2016). Retrieved Oct 3, 2017 from http://afnetworking.com

[2] Russ Allbery and Charles H. Lindsey. 2009. Netnews Architecture and Protocol. (Nov 2009). Retrieved Aug 19, 2017 from https://tools.ietf.org/html/rfc5537

[3] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2013. DepSky: Dependable and secure storage in a Cloud-of-Clouds. In *ACM Transactions on Storage (TOS)*, Vol. 9. Article 12, 33 pages. https://doi.org/10.1145/2535929

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

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

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

[7] Roxana Geambasu, Tadayoshi Kohno, Amit A Levy, and Henry M Levy. 2009. Vanish: Increasing data privacy with self-destructing data. In *18th USENIX Security Symposium*. 300–315. https://www.usenix.org/conference/usenixsecurity09/technical-sessions/presentation/vanish-increasing-data-privacy-self

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

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

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

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

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

[13] Rongchang Lai and Yasushi Shinjo. 2016. 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*. 188–198.

[14] No Thirst Software LLC. 2013. TICoreDataSync. (May 2013). Retrieved Sep 28, 2017 from https://github.com/nothirst/TICoreDataSync

[15] Elvis Nuněz. 2017. Sync, modern Swift JSON synchronization to Core Data. (Sep 2017). Retrieved Oct 3, 2017 from https://github.com/SyncDB/Sync

[16] Liao-Jun Pang and Yu-Min Wang. 2005. A new (t, n) multi-secret sharing scheme based on Shamir's secret sharing. *Appl. Math. Comput.* 167, 2 (2005), 840–848.

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

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

[19] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. 2005. 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)*. 73–84.

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

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

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

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

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

[25] Lingfang Zeng, Yang Wang, and Dan Feng. 2015. 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)*. 352–361.