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

Meng Li and Yasushi Shinjo
University of Tsukuba

September 26, 2017

Abstract

This paper presents Grouper, a framework for developing mobile applications. Grouper 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 period of time. Mobile devices exchange messages via untrusted servers based on the Grouper Message Protocol. We have implemented the Grouper framework for iOS, macOS, tvOS and watchOS in the Objective-C language. It consists of a client framework and a Web service. We have implemented three applications using Grouper: an iOS application Account Book, a macOS application Notes and a benchmark application Test. We evaluate the development efforts and the performance of Grouper. Experimental results show that the performance of Grouper is feasible for mobile applications that are used in a small group of people.

Keywords: mobile application security, secret sharing, untrusted server

1 Introduction

People use mobile applications everyday. 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 such mobile applications, they must fully trust the central servers. 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 cleartext. In addition, users may lose their data when service providers shut down their servers.

To address this problem about using central servers, Vanish[1], SafeVanish[2], SeDas[3] and CouldSky[4] construct a data self-destruction system as their cloud storage. In these approaches, servers store data temporarily and delete data after a period of time.

These existing approaches have following problems. Firstly, they do not support data recovery when some nodes miss getting data from shared storage. Application developers have to deal with such cases by themselves. Secondly, these approaches do not support developing general 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, although Grouper uses the data self-destruction scheme, it supports data recovery when some nodes miss getting shares from untrusted servers. When a receiver node misses getting shares, the Grouper framework automatically asks the sender to upload missing shares again. This ensures reliable data sharing among devices of a group. In addition, data can be recovered even untrusted servers shut down because all devices of a group keep a complete data set of this group.

Grouper consists of a client framework and a Web service. We have implemented the Web service running on the multiple untrusted servers in Java. We have embedded a Sync framework in Grouper to synchronize objects among mobile nodes. We have implemented three applications using Grouper: an iOS application Account Book, a macOS application Notes and a benchmark application Test. These implementations show that Grouper makes it easy to develop mobile applications with data synchronization.

The contributions of this paper are as follows. Firstly, we provide support for data recovery when some nodes miss getting data from untrusted servers. Grouper realizes reliable data synchronization among nodes using a reliable multicast technique. Secondly, we make it easier to develop mobile applications. 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 threat model underlying the Grouper framework. We target mobile applications that are used in a small group of people. A group consists of an owner and other members. Each member has a mobile device. A owner invites other members in a face-to-face way.

Firstly, server are a passive adversary, and can read all data, but it does 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 untrusted 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 tracks.

Secondly, 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 privacy in the data storage of servers rather than data transportation.

Thirdly, in an application, all group members are not malicious and their devices connect to each other securely in a face-to-face distance. We target applications that are used in a small group at the user invitation time. For example, the group members are working in an office. They know one another and they are not malicious. When a group owner invites new members, the owner authenticates group members by a face-to-face way.

At last, a server is isolated from one another and managed by independent providers. We assume that providers of untrusted servers do not expose users' data to other providers. For example, a group owner can pick up servers of Amazon, Google, and Microsoft, which are supposed not to expose users' data to other cloud providers.

3 Design

This section describes the design of the Grouper framework.

3.1 Overview

Our goal is to support developing mobile applications that are not relying on trusted central servers.

To achieve this goal, we provide 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 deletes.
- **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 creates a group and invites employees to the group. Then, the employees can record the income and expenditure of this company. These income and expenditure records are represented as objects and shared among devices. Anyone can edit and delete existing records.

Grouper uses untrusted servers to exchange messages among mobile devices. Untrusted servers construct a self-destruction system, and delete messages after a period of time. We call it a Time To Live(TTL).

Grouper protect messages from the providers of untrusted servers using a secret sharing scheme. Grouper uses Shamir's secret sharing scheme to protect user data. In Shamir's secret sharing scheme, a member securely shares a secret with other members by generating n shares using a cryptographic function[7]. At least k or more shares can reconstruct the secret, but $k-1$ or fewer shares output nothing about the secret[8]. We describe this scheme as a function $f(k,n)$, where n is the number of all shares, and k is the threshold to combine shares.

Grouper has the following advantages over conventional systems using untrusted servers. Firstly, it is easy for a developer to recover from message losses in untrusted servers (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 lifetimes of messages. Secondly, it is easy for a group owner to invite other members using a safe communication channel in a face-to-face distance (Section 3.7).

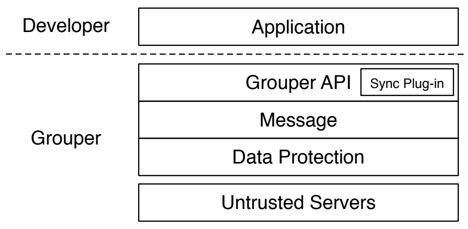


Figure 1: Architecture of Grouper.

3.2 Architecture

Figure 1 describes the architecture of the Grouper framework. An application using Grouper consists of the four following 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 this 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 reliable message layer. When this layer receives a message, this layer unmarshalls the message using the plugin, reconstructs an object, and puts the object into the persistent store.
- **Message.** This layer provides a message service with multicasting capability among devices. The destination of a massage is not only the node identifier (ID) of a single device but also "*", which means delivering to all the other nodes. This layer does not ensure the 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 into a database. When a mobile device downloads a share from an untrusted server, this server retrieves it from the database and sends it into the device. An untrusted server performs device authentication using device keys.

The following subsections describe 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. A developer can add object synchronization functions to a standalone application with a few lines of code. Table 1 shows the client API of Grouper. An application initializes the framework by invoking the method *grouper.setup*(). When the application needs to update an object in all devices, the application invokes the method

Table 1: Client API of Grouper.

Method	Description
grouper.**setup**(appId, dataStack)	An application invokes this method to initialize Grouper with appId and dataStack. AppId is a unique ID of an application. Datastack is used by the synchronization plugin.
grouper.sender.**update**(object)	An application invokes this method after creating a new object or modifying an existing object. Grouper performs updating asynchronously and sends an update message to other devices.
grouper.sender.**delete**(object)	An application invokes this method when the application wants to delete an existing object. Grouper will delete the object and remove it in the persistent store of a device automatically after sending a message to other devices.
grouper.receiver.**receive**(callback)	An applications invokes this method to register the callback function that is called after Grouper processes received messages and update objects according to the messages. The application can use this callback function to update the user screen.
grouper.sender.**confirm**()	An application needs to invoke this method periodically or occasionally to send a confirm message to other devices.

Table 2: The 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 of object IDs.

grouper.sender.update(). When the application needs to delete an object in all devices, the application invokes the method *grouper.sender.delete*(). The the application uses the method *grouper.receiver.receive*() to register 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 in a 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 two modules: the synchronization plugin and the messaging layer. When an object is updated in a local device, the synchronization plugin marshalls an updated object and returns a message. Next, the Grouper API layer sends this messages using the messaging layer. When an object is updated in another device, the Grouper API layer receives a message using the messaging layer. Next, the Grouper API layer updates the object by passing the received message to the synchronization plugin.

We have not implemented the synchronization plugin by ourselves, but we make this module pluggable. This is because there are many such synchronization modules that provide various features, and application requirements also vary. Each application developer should choose a suitable module based on a consistency model and other requirements. As described in Table 2, the synchronization plugin should provide the functions *marshall*(o), *updateRemote*(b) and *deleteRemote*(b). Grouper invokes them to get marshalled data from the persistent store and save unmarshalled data into the persistent store.

Because we implement our client framework in Objective-C at first, we use the Sync framework[9] to implement a data synchronization plugin currently. The Sync framework marshalls objects into JavaScript Object Notation (JSON) strings and provides a consistency model where the newest edition wins.

3.4 Reliable Message Delivery

Grouper realizes 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[10]. 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 missing a 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 this 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. The sender will send update message No.2 to the asked member using a unicast address.

This basic reliable multicasting technique works well for continuous medias, such as video streaming in Internet communication. However, this does not work if receivers become often offline for a long time and servers delete messages soon.

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

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 becomes offline. The servers delete these two messages. The sender uploads next three update messages from No.3 and No.5 using a multicast address. The receiver becomes online, but the receiver does not notice that the receiver has missing messages soon. At this time, the sender sends a confirm

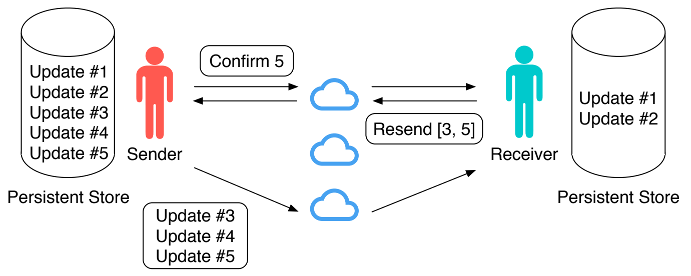


Figure 2: Implementing reliable messaging with continuous sequence numbers.

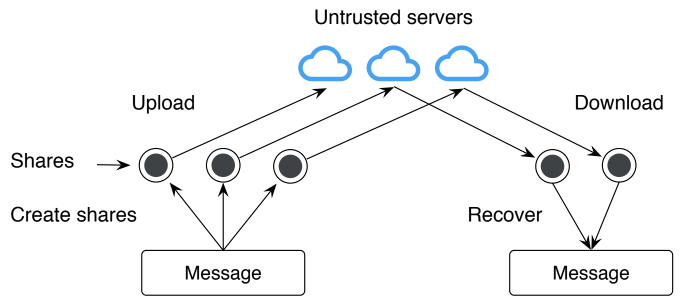


Figure 3: Implementing unreliable messaging with a secret sharing scheme and multiple untrusted servers.

message that includes the newest active sequence number, 5. The receiver receives this sequence number and compares it with the newest sequence number in the persistent store. In Figure 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. The sender will send update messages No.3 and No.4 to the receiver again using a unicast address.

This idea is inspired from the checkgroups message of Usenet[11]. 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. These basic messages can be lost and the list can be obsolete. Ckeckgroups 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 basic messages are distributed.

3.5 Unreliable Messaging Using Untrusted Servers

This subsection describes the design of unreliable messaging in the layers of Messaging, Data Protection, and Untrusted Servers.

In these layers, we use a secret sharing scheme and multiple untrusted servers as shown in Figure 3. In this figure, the messaging layer in the sender is sending a message to that in the receiver.

At first, this layer calls the data protection layer and creates three shares by a secret sharing scheme. Next, this layer uploads those shares to three untrusted servers. Each untrusted server receives one of these shares and stores it into a database.

Table 3: Attributes of Grouper message.

Attribute	Explanation
type	Type of this message.
content	A marshalled object or sequence numbers.
sequence	Sequence number of this 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.

3.6 Parameters of Secret Sharing Scheme

The data protection layer uses the Shamir's secret sharing scheme $f(k, n)$ introduced in Section 3.1. In Grouper, we extend this scheme $f(k, n)$ and design our new scheme $f(k, n, s)$. In our scheme, the parameter k and n are same as these 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 consider server crashing. 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 uploading is successful. Otherwise, Grouper keeps trying to upload these shares.

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

- **Sender first.** An application sets s closed to k. A sender gets a successful result earlier. When some servers that have shares are not available, a receiver more likely loses a message.
- **Receiver first.** An application sets s closed to n. A sender must keep trying longer to get a successful result. When some servers that have shares are not available, a receiver more likely get a message.

3.7 Grouper Message Protocol

The Grouper API layer sends and receives messages using our own protocol, Grouper Message Protocol. In this protocol, a message is a JSON string that contains a marshalled object and attributes. Table 3 shows the attributes in a Grouper message. There are three important attributes:

- **Type.** This attribute means types of messages. 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 physical identifier of a deleted object. We call update messages and delete messages *normal messages*. Both confirm message and resend message contain control information about reliable multicast. We call these messages *control messages*.
- **Content.** If this message is an update message, the content value contains the JSON string of a marshalled object. If this message is a delete message, the content value contains the objectId of an object. If this message is a confirm message, the content value contains

Algorithm 1 Handle message algorithm

```
1:  procedure onMessageReceived($msg$)
      ▷ 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:      $historyMsgs.add(msg)$
      ▷ Begin basic reliable multicast.
8:      if $msg.seq \neq 0$ && $lastMsg.seq{+}1 \neq msg.seq$ then
9:          $resendMsg \leftarrow createResendMsg(lastMsg.seq+1, msg.seq)$
              ▷ Create a resend message by the minimum and maximum sequence number.
10:         $sendMsg(resendMsg)$
11:     end if
      ▷ Handle the message by its type.
12:     if $msg.type =$ "$update$" then
13:         $sync.updateRemote(msg)$
14:     else if $msg.type =$ "$delete$" then
15:         $sync.deleteRemote(msg)$
16:     else if $msg.type =$ "$confirm$" then
17:         $maxSeq = getMaxSeqFrom(msg.content)$
18:         $resendMsg \leftarrow createResendMsg(lastMsg.seq+1, maxSeg)$
19:         if $resendMsg \neq null$ then
20:             $sendMsg(resendMsg)$
21:         end if
22:     else if $msg.type =$ "$resend$" then
23:         $seqs \leftarrow getSeqs(msg.content)$
24:         for $seq \in seqs$ do
25:             $missingMsg \leftarrow getMsg(seq)$
26:             $sendMsg(missingMsg)$
27:         end for
28:     end if
29: end procedure
```

the maximum sequence number of messages created in the device of the sender. If this message is a resend message, the content value contains the range of missing messages' sequence numbers.

- **Sequence.** This attribute means the sequence number of the message. When a sender sends a new normal message, the sender increments the sequence number and includes it to the new message. The sequence number of any control message is 0. Using both the sequence number and the ID of the sender identify a unique message.

The attribute receiverId contains the addresses of destination devices. An address is either a list of device IDs or "*" that means 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*() of the Grouper API, 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*() of the Grouper API, 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. This confirm message includes the sequence numbers of objects that are recently created in this device.

The method *grouper.confirm*() is invoked at the following occasions:

- **Periodically.** For example, an application sends a confirm message now, and it will send a confirm message after the TTL because the shares of normal messages are deleted in untrusted servers after the TTL.
- **After the device becomes online.** Sometimes, a device is offline and cannot send a confirm message after the TTL. Grouper sends a confirm message when this device becomes 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 this confirm message. For a resend messages, the Grouper API layer gets the sequence numbers from this resend message, finds the corresponding normal messages and sends them to the sender of this resend message.

3.8 Group Management

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

- **Group Creation.** A user can create a group, and the creator becomes the owner of this group. Before creating a group, the owner prepares his own user information including his email and name, multiple untrusted servers, a group ID and a group name. Next, he initializes this group on all untrusted servers by submitting his node identifier, the parameters of scheme $f(k, n, s)$ and the TTL to the multiple untrusted servers. The node identifier, which represents his device, is generated by Grouper randomly when the application is launched at the first time. In 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 by the master key.
- **Member Invitation.** After creating a group, the owner can invite new members to his group. To join the group, a new member prepares his user information at first. The owner invites the new member by a face-to-face way rather than using central servers. At this time, Grouper establishes connection between their devices using a local safe communication channel like *Multipeer Connectivity*[12]. Firstly, the new member sends user information and a node identifier to the owner. The owner saves the user information and the 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.

Thirdly, the untrusted servers return access keys for the new member to the owner. Lastly, 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 these 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 (Section 4.1), the implementation of the Web service (Section 4.2) and demo applications (Section 4.3) in this section.

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*[12], an official Peer-to-Peer communication framework provided by Apple. Grouper uses it to transfer data between two devices in a face-to-face way.
- *Core Data*[16], an official ORM framework provided by Apple. *Core Data* provides generalized and automated solutions to common tasks associated with object life cycles and object graph management, including persistence. Grouper uses it to manage model layer objects.
- *Sync*[9], a synchronization framework for *Core Data* using JSON. This framework marshalls objects into JSON strings and vice versa. Grouper uses it in the synchronization plugin.
- *c-SSS*[17], an implementation of the secret sharing scheme.
- *AFNetworking*[18], a networking library in Objective-C. Grouper uses it to invoke the RESTful API provided by our Web services running on multiple untrusted servers.

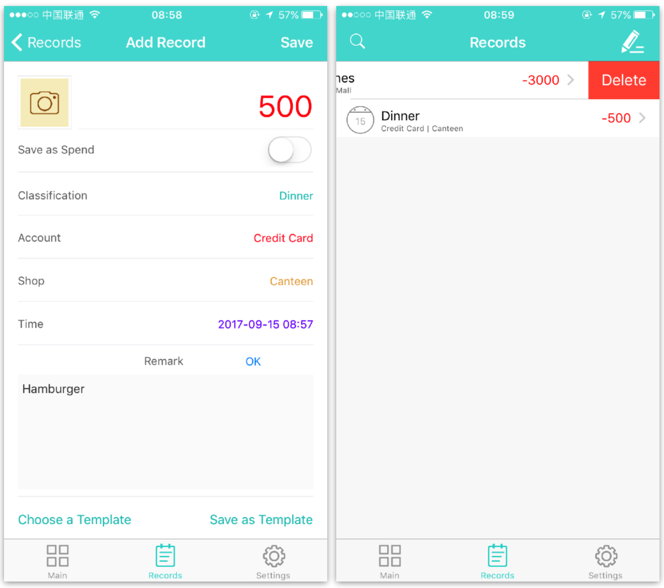
4.2 Web Service

We have chosen implementing own Web service rather than using commercial general cloud services like Amazon Simple Storage Service (S3), Google Cloud Storage, and 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 prescriptive time.

Our Web service provides a RESTful API to clients. It runs on the Tomcat server[13] that is an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. We use the Spring MVC[14], a Web model-view-controller framework, to create our RESTful API, and Hibernate[15], an open source Java Object-Relational Mapping (ORM) framework, to save and operate objects in the Web service.

Our Web service includes three kinds of entities: *Group*, *User* and *Share* entities. A *Group* entity saves a group



500
Save as Spend
Classification Dinner
Account Credit Card
Shop Canteen
Time 2017-09-15 08:57
Remark OK
Hamburger
Choose a Template Save as Template

-3000
Dinner -500
Credit Card | Canteen

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

Figure 4: Screenshots of demo application Account Book.

ID, a group name and its owner. A *User* entity saves the node identifier of a user, the access key for this user, and the group entity of this user. A *Share* entity saves a share generated with the secret sharing scheme, the 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, records the income and expenditure of a group.
- *Notes*, a macOS application in Swift, takes shared notes for a small group.
- *Test*, a benchmark iOS application in Swift, 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 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 to other group members. Figure 4(b) shows the screenshoot of deleting a record in Account Book. A user can swipe a cell from right to left to delete a record in 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

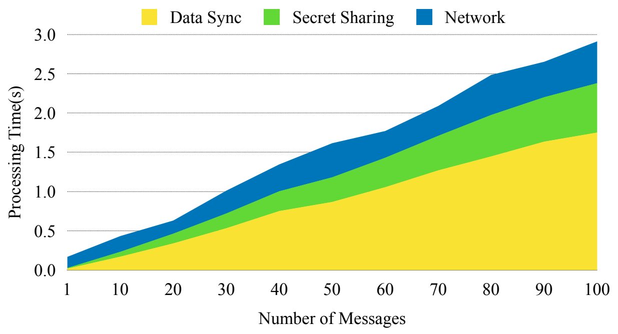
This section shows the development efforts to use Grouper and the performance of Grouper.

5.1 Development Efforts

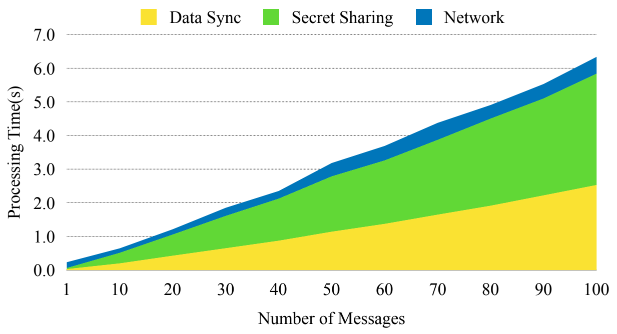
We see development efforts through two factors: the usability of the client API and the code size in the lines of

Table 4: Applications' lines of code.

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



(a) Sending multiple messages.



(b) Receiving multiple messages.

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

Table 5: Devices in the performance experiment.

Device	CPU	RAM	OS
iPod 5	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

code(LoC) the developer has to add after using Grouper. As described in Table 1, Grouper provides the simple client APIs for developers. A developer can convert a standalone application to a data sharing application with our client APIs easily.

To know the LoC developer has to add, we have developed the following applications, Account Book, Notes and Test using the Grouper framework. As described in Table 4, based on the stand alone application without data synchronization, developers can add data synchronization to these applications with Grouper by adding a small number of code.

5.2 Performance

In this subsection, we show that Grouper mobile applications using the Grouper framework provide sufficient performance for small groups of people. In our performance experiments, we used the benchmark application *Test* to transfer data between iPhone 4s and iPod 5 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. Table 5 shows the hardware and software information of our experiment environment. In our benchmark application, the size of an object was about 323 bytes. This object corresponds to a record of benchmark application Test and contains the creation time of this record. When this object was updated in a node, the Grouper API generated a normal message which size was about 656 bytes.

Through experiments, we measured the following processing times of object synchronization (uploading and downloading).

- **The processing time according to the number of updated objects (the number of update messages).**

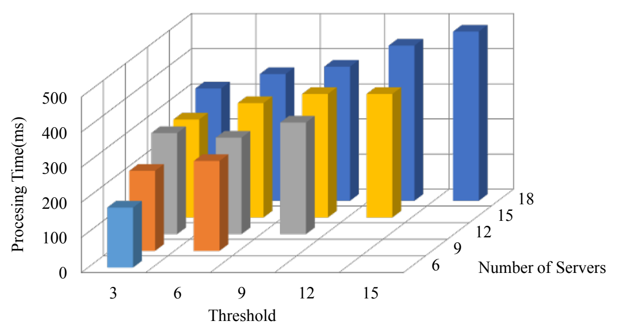
We performed a experiment to measure the processing time of object synchronization according to the number of updated objects. In this experiment, we set the secret sharing scheme to $f(2, 3)$. We sent multiple messages from a device and received them in another device. Concretely, We sent multiple message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

Figure 5 shows the processing time of uploading and downloading multiple messages. We divide the processing time into three parts: data sync, secret sharing and network. As the number of messages increased, the data sync and secret sharing part increased linearly. The network part increased slowly and sometimes decreased. On the whole, the total processing time increased linearly. Compared with uploading messages, downloading messages cost about two times of processing time. Note that object synchronization is done in a background task. Note that object synchronization is done asynchronously. While the Grouper framework is performing object synchronization, user interface of a mobile application never freezes. A user is not aware of data synchronization.

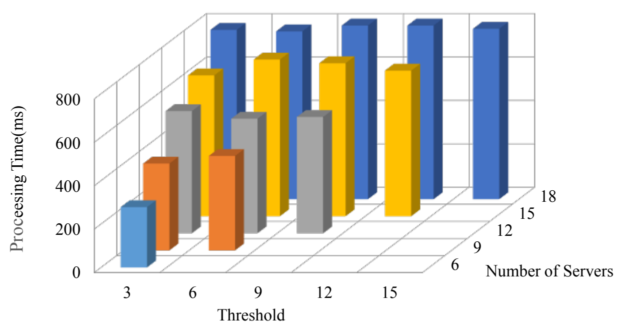
The number of messages a device has to handle is independent of the number of devices in a group. For example, consider that a group consists of 100 members and their 100 devices. When an object is updated in a device, the device has to send a single update message. The device divides the message into multiple shares, and uploads them to multiple untrusted servers. This process is independent of the number of devices in a group.

- **The processing time according to the parameters of the secret sharing scheme.**

We performed a experiment to measure the processing time of object synchronization according the parameters of the secret sharing scheme. Specifically, we changed the parameter k and n of the secret sharing scheme and

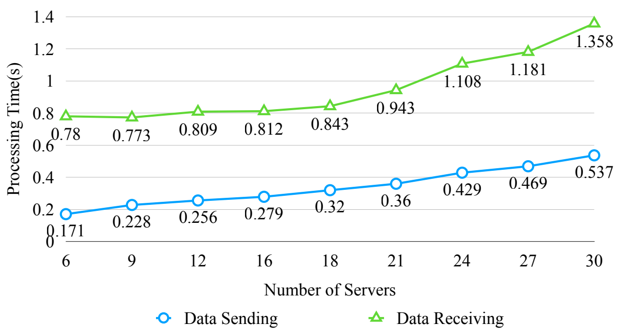


(a) Sending a single message

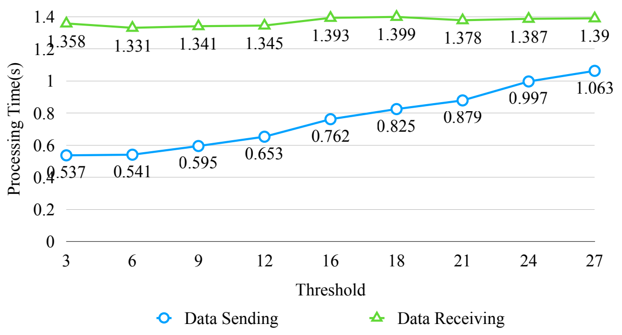


(b) Receiving a single message

Figure 6: Processing time of uploading and downloading a single message with different scheme.



(a) Constant threshold.



(b) Constant number of servers.

Figure 7: Processing time of uploading and downloading a single message with a constant k or n.

measured the processing time of uploading and downloading a single message. Figure 6 shows the relationship between processing time and the parameters, where $0 < k < n, k = 3i, n = 3j + 3, 1 \leq i, j \leq 5$. For uploading a single message, as the parameter k or n increased, processing increased linearly. However, for downloading a single message, as the parameter k increased, processing time changed a little, sometimes decreased.

Figure 7(a) shows the relationship between processing time and the number of servers n, where $k = 3$ and $n = 2, 4, 6, ..., 10$. Figure 7(b) shows the relationship between processing time and the threshold k, where $n = 30$ and $k = 3, 6, 9, ..., 27$. Here, we can answer the third question. As n increased, the processing time of uploading and downloading increase linearly. As k increased, the processing time of uploading increased linearly and that of downloading did not change.

5.3 Discussion About the Numbers of Messages

In this subsection, we discuss the relationship between TTL and the number of messages. As describe in Table 6, we define the following parameters.

At first, we consider the situation that all devices are online ($F = 0$). In this situation, the number of uploading for the device i is

$$NUi = Ui + C \tag{1}$$

Total number of uploading is the sum NU of NUi.

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

A device should not download those messages uploaded by itself. To simplify the question, we suppose the device downloads those messages uploaded by itself. Thus, the number of downloading for device i NDi is equal to total number of uploading NU. Total number of downloading ND is the sum of NDi.

$$ND = \sum_{i=1}^{D} NDi = NU \cdot D = (U + C \cdot D) \cdot D \tag{3}$$

Equation 1 to 3 shows that for the device i during the TTL, the order of uploading is $O(U)$, and the order of downloading is $O(U+D)$. As the number of updated and deleted objects increased, total number of uploading and downloading increased linearly.

Next, we consider the situation that there are offline devices in a group ($F \neq 0$). In this situation, offline devices send the resend messages when they are online and those devices which saved the updated objects will send update messages to offline devices. To simplify the question, we suppose that all offline devices should receive the updated messages after they sent the resend message. It means that any device will not send a same updated object for more than two times. For the device i, the number of those update messages which are sent after downloading the resend message URi is

$$URi = Ui \cdot \frac{L}{T} \cdot F \tag{4}$$

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

Table 6: The parameters for discussing the relationship between TTL and the number of messages.

Parameter	Explanation
D	Number of devices.
T	Time to live (TTL).
F	Average number of offline devices during TTL.
L	Average online time.
Ui	Number of updated/deleted objects created in the device i during the TTL.
URi	Number of updated/deleted objects sent from the device i after downloading resend messages during the TTL.
U	Total number of updated and deleted objects during the TTL.
C	Total number of confirm messages during the TTL.
NUi	Number of uploading for device i.
NU	Total number of uploading.
NDi	Number of downloading for device i.
ND	Total number of downloading.

$$NUi = Ui + C + Ui \cdot \frac{L}{T} \cdot F = Ui \cdot (1 + Ui \cdot \frac{L}{T} \cdot F) + C \quad (5)$$

$$NU = \sum_{i=1}^{D} NUi = U \cdot (1 + \frac{L}{T} \cdot F) + C \cdot D \quad (6)$$

$$ND = NU \cdot D = \sum_{i=1}^{D} NUi = U \cdot D \cdot (1 + \frac{L}{T} \cdot F) + C \cdot D^2 \quad (7)$$

Equation 4 to 7 shows that for the device i during the TTL, the order of uploading is $O(U \cdot \frac{L}{T} \cdot F)$, and the order of downloading is $O(U \cdot \frac{L}{T} \cdot F + C)$. As the number of updated and deleted objects, the average number of offline devices during the TTL, and the average online time increased, total number of uploading and downloading increased linearly.

From the discussion above, we get the conclusion that as the TTL increased, the number of resend and update messages decreased, it becomes more convenient for group members to sharing information.

6 Related Work

Vanish is a self-destruction system[1]. Vanish uses Distribute 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 to a public DHT and eliminate the key from the local storage. The key in Vanish is removed after a period of time, and the encrypted message is permanently unreadable. Vanish is implemented with OpenDHT[19] or VuzeDHT[20] which is controlled by a single maintainer. Thus, it not strongly secure due to some Sybil attacks[21]. In addition, the surviving time of the key in Vanish cannot be controlled by applications.

To address these issues in Vanish, Zeng et al., propose SafeVanish[2] and SeDas[3]. SafeVanish is designed to prevent hopping attacks by extending the length range of the key shares while SeDas extends the idea of Vanish by exploiting the potentials of active storage networks, instead of the nodes in P2P, to maintain the divided secret key. By extending SeDas, Zeng's group proposes CloudSky[4], a controllable data self-destruction system for untrusted cloud storage. In CloudSky, a user can control the surviving time of a message. Taking advantage of ABE, a user can also define the access control policy.

However, these proposals are not suitable for developing information sharing applications introduced above. They are suitable for exchanging immutable messages, while our target applications need to modify a object repeatedly after other devices received the object. Although, CloudSky solves the problems about user controllability 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, while Grouper does not rely on any trusted authority.

Mylar[5] stores encrypted data on servers, and decrypts this data only in the browsers of users. Developers of Mylar use its API to encrypt a regular (non-encrypted) Web application. Mylar uses its browser extension to decrypt data on clients. Compared to Mylar which is using a single server, Grouper takes advantages of data redundancy provided by the secret sharing scheme.

Sweets[6] is a decentralized social networking service (SNS) application using data synchronization with P2P connections among mobile devices. Sweets performs data synchronization not only between two online nodes directly but also via common friends' nodes indirectly for offline nodes. This indirect synchronization uses ABE to realize access control. Direct synchronization can only be finished during two devices are online at the same time. In Grouper, on the other hand, users can synchronize data from multiple untrusted servers anytime.

DepSky[22] is a system that stores encrypted data on servers and runs application logic on clients. *Cloud-of-Clouds* is the core concept in DepSky. It represents that DepSky is a virtual storage, and its users invoke operations in several individual severs. DepSky keeps encrypted data in commercial storage services and do application logic in individual servers. In Grouper, untrusted servers undertake responsibility of temporarily data storage and message delivery with server-side computation.

Compared with data encryption methods, the secret sharing scheme has following advantages. Firstly, using a secret sharing scheme does not require key management including generation and distribution. Data encryption systems like CloudSky always use trusted authorities for key management. In Grouper, we require all cloud services are untrusted. Secondly, the secret sharing scheme ensures the data availability in the situation that a small number of untrusted servers are not accessible. For the $f(k,n,s)$ scheme in Grouper, the original object can be recovered after accessing at least k untrusted servers. Thirdly, the secret sharing scheme reduces the risk of attack, because the attacker who can access only $k-1$ or less untrusted servers cannot get any information.

7 Conclusion

This paper describes Grouper, a framework using a secret sharing scheme and multiple untrusted servers, to develop light-weight information sharing for mobile applications. In such an application, users can create a group and exchange the information safely via multiple untrusted servers. Grouper provides two main functions: reliable data synchronization and group management for developing such applications. Compared to conventional systems, Grouper provides the reliable synchronization and ensures that a member of a group can synchronize data from others after untrusted severs remove data. Compared to pure data encryption approaches, Grouper improves dependability by using multiple untrusted servers and our extended scheme $f(k,n,s)$.

We implement Grouper's Web service in Java EE and client's framework in Objective-C. To evaluate Grouper's design, we have developed applications, *Account Book*, *Notes* and *Test* on the top of Grouper. Developing these applications shows that Grouper requires little development efforts to convert an stand alone application to data sharing application. We also evaluated the performance of Groper using our benchmark application. The results shows that using Grouper in an application does not influence the user experience.

In the future, we would like to support more platforms including Android.

References

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

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

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

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

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

[6] 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.

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

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

[9] Elvis Nunẽz. Sync, modern Swift JSON synchronization to Core Data. `https://github.com/SyncDB/Sync`, Jun 26, 2017.

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

[11] Netnews Architecture and Protocol. `https://tools.ietf.org/html/rfc5537`, Aug 19, 2017.

[12] Apple Inc. MultipeerConnectivity. `https://developer.apple.com/documentation/multipeerconnectivity`, Apr 24, 2016.

[13] Tomcat. `http://tomcat.apache.org`, Aug 19, 2017.

[14] Spring. `http://spring.io`, Aug 19, 2017.

[15] Hibernate. `http://hibernate.org`, Aug 19, 2017.

[16] Apple Inc. Core Data Programming Guide, Guides and Sample Code. `https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CoreData/`, Nov 10, 2015.

[17] Fletcher T. Penney. c-SSS, an implementation of Shamir's Secret Sharing. `https://github.com/fletcher/c-sss`, Jan 30, 2016.

[18] AFNetworking, a delightful networking framework for iOS, OS X, watchOS, and tvOS. `http://afnetworking.com`, Mar 31, 2016.

[19] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: a public DHT service and its uses. In *ACM SIGCOMM Computer Communication Review*, pages 73–84, 2005.

[20] Azureus. `http://www.vzue.com`, Aug 19, 2017.

[21] Scott Wolchok, Owen S Hofmann, Nadia Heninger, Edward W Felten, J Alex Halderman, Christopher J Rossbach, Brent Waters, and Emmett Witchel. Defeating Vanish with low-cost Sybil attacks against large DHTs. In *NDSS*, 2010.

[22] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, 2013.