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

Meng Li and Yasushi Shinjo
University of Tsukuba

August 17, 2017

Abstract

This paper presents Grouper, a framework for developing mobile applications, which protects user data by storing data in multiple untrusted servers. Grouper uses a secret sharing scheme to create several shares from a marshalled object and uploads these shares to multiple untrusted servers. Our multiple untrusted servers are a self-destruction system. Uploaded shares will be deleted after a period of time by the Web service running on the multiple untrusted servers. To transfer data among mobile devices, we design our own *Grouper Message*. We have implemented Grouper in the Objective-C language. We evaluate the developer efforts and the performance of Grouper. We have implemented three applications using Grouper: an iOS application Account Book, a macOS application Notes and a benchmark application Test.

Keywords: mobile application security, secret sharing, untrusted server

1 Introduction

Conventional mobile applications are built based on a client-server mode and require the central servers for storing shared data and processing confidential information. The users of such mobile applications must fully trust the central servers. If the central servers can be accessed by an attacker, a curious administrator, or a government, user information will be revealed because data is often stored on the server in cleartext. In addition, users may lose their data when service providers shut down their servers.

To address such a problem, Vanish[1], SafeVanish[2], SeDas[3] and CouldSky[4] use a data self-destruction system as their cloud storage. In these approaches, servers store data temporarily and delete data after period of time. Mylar[5] and Sweets[6] use data encryption to protect user data. These existing approaches have following problems. Firstly, these existing approaches do not support data recovery when some nodes miss getting data from shared storage. Application developers have to deal with such cases by themselves. Further, these approaches do not support developing mobile applications.

To address these problems, we are developing Grouper, a framework for developing mobile applications. Grouper provides objects 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 support 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. Such scheme ensures reliable information sharing among devices of a group. In addition, data can be recovered even untrusted servers shut down because all devices of group members keep a complete data set of this group.

Grouper consists of a client framework and a Web service. We have implemented the Grouper framework for iOS, macOS, tvOS and watchOS in the Objective-C language. We have implemented the Web service running on the multiple untrusted servers in Java. We have embedded the 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 shows that Grouper makes it easy to develop mobile applications with data synchronization. Experimental results show that the performance of Grouper is feasible for mobile applications that are used in a small group of people.

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 an applications with a few lines of code.

2 Assumption and Threat Model

In this section, we introduce assumptions and threat model of Grouper. There are four following basic assumptions underlying the Grouper framework.

Firstly, a server is a passive adversary, and can read all data, but it does not actively attack. The server hosts Web services and performs device authentication. Servers generate access keys for group users. When a device wants to get/put data from/to untrusted servers, the device sends a request with an access key in the request header. A node divides data into shared before it is uploaded to the server, and recover the data after downloading using a secret sharing scheme. In this paper, we do not address other attack types such as user tracking and metadata collection by servers. For example, the server can track users with IP addresses, and Grouper cannot hide social graphs against such tracking.

Secondly, data transportation between a device and an untrusted server is secure. Grouper improves mobile application security by concentrating on data storage in servers rather than data transportation. If someone attacks the HTTP connection and gets some shares, he can try to recover these shares and may get the original data.

Thirdly, in an application, all group members are not malicious and their devices connect to each other in a face-to-face distance. We target applications that are used in a small group, like all members in a small office. Thus, the group members are persons this office know one another and they are not malicious. In members inviting, a group owner authenticates group members by a face-to-face way. Connections for data transportation between two devices only be established in members inviting. Malicious users outside the group can pretend as the member of the group to get secret information.

At last, a server is isolated from one another and managed by independent providers. In fact, we hope each untrusted servers manager does not know the existence of others. To use applications by grouper, the leader of a small company can assign three different employees to deploy the Web service in different servers secretly. This leader must ensure those employees do not collude to crack user data. For example, a group owner picks up servers of Amazon, Google, and Microsoft, which are supposed not to expose users' data to other cloud providers. If anyone has privilege to access more than enough untrusted servers to recover the shares to original data, he can get all shared information of a group.

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 relining on trusted central servers. 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.

To support developing such mobile applications, we provide the Grouper framework. This framework provides the following functions:

- **Data Synchronization.** If an user updates an object in his device, the mirrors of this object in other devices are updated.
- **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 and share these records to others. Anyone can edit and delete existing records.

Grouper uses the secret sharing scheme to protect user data rather than data encryption methods. In a 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 can obtain nothing about the secret[8]. We describe this scheme as a function $f(k,n)(0 \leq k \leq n, k \in N, n \in N)$, where n is the number of all shares, and k is the threshold to combine shares. If one person outside a group can access to k or more untrusted servers, he has enough shares to recover the original user data.

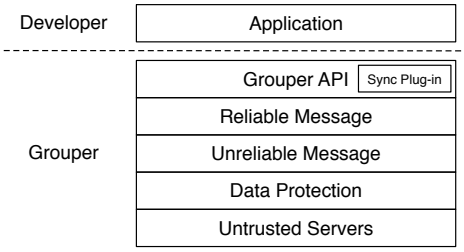


Figure 1: Architecture of Grouper.

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. Grouper protect messages from the providers of untrusted servers using a secret sharing scheme. 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).

3.2 Architecture

Figure 1 describes the architecture of the Grouper framework. An application using Grouper consists of the six following layers:

- **Grouper API** A developer develops an application without data synchronization at first. He can use the API provided in the client framework of Grouper to add the synchronization function to his application.
- **Sync Plug-in.** Grouper uses the third-party framework for data synchronization. This layer marshall an updated object in a persistent store and send it using the lower reliable message layer. When this layer receives a message, this layer unmarshall the message, reconstructs an object, and put the object into the persistent store.
- **Reliable Message.** This layer provides a reliable 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 try to deliver a message to other devices even the message is deleted in untrusted servers before some nodes download the message.
- **Unreliable Message.** This layer provide an unreliable message service multicasting capability among devices. This layer do 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

Table 1: Client APIs of Grouper.

Methods	Semantics
grouper.**setup**(appId, dataStack)	Setup Grouper with appId and dataStack. AppId of an application must be unique. Datastack can be created by invoking the API provided in the Sync framework.
grouper.sender.**update**(object)	Invoke this method after creating a new object or modifying an existing object. Developers must ensure this object has been saved to persistent store before invoking update method.
grouper.sender.**delete**(object)	Invoke this method when a user wants to delete an existing object. Developers need not to delete the object and save to persistent store before invoking delete method. Once you delete it, Grouper cannot create Grouper message from this object. Grouper will delete the object and save it to persistent store automatically after finishing message transportation.
grouper.receiver.**receive**(callback)	Invoke this method if a user wants to synchronize data from untrusted servers. Callback functions is provided for executing UI updating code.
grouper.sender.**confirm**()	Invoke this method to send confirm message to other group members.

Table 2: The API of the synchronization layer

Name	Description
marshall(o)	Marshalls the object o and returns the marshalled byte array.
sync(b)	Unmarshalls the byte array b to the object and puts the object into the persistent store.

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 a server, the server receives it and stores it into a database. When a mobile device downloads a share from a server, the server retrieves it from the database and send 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 throughput a simple API. A developer can add object synchronization functions to a standalone application with a few lines of code. Table 1 shows the 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 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 rely on two lower layers: the sync plug-in layer and the reliable messaging layer. When an object is updated in a local device, the sync plug-in layer marshall an updated object and obtains a message. Next, the Grouper API layer sends this messages using the reliable messaging layer. When an object is updated in an other device, the Grouper API layer receives a message using the reliable messaging layer. Next, the Grouper API layer updated the object by passing the received message to the sync plug-in layer.

We have not implemented the sync plug-in layer by ourselves, but we make this layer as a pluggable module. 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 sync plug-in layer should provide the $marshall(o)$ and $sync(b)$ functions. Grouper invokes them to get marshalled data from the persistent store and save unmarshalled data into the persistent store.

Currently, we use the Sync framework [12] as a data synchronization module. The Sync framework marshalls objects into JavaScript Object Notation (JSON) strings and provides a consistency model where the newest edition wins Application developers can use other data synchronization modules, such as Ensemblesm[9], TICoreDataSync[10], etc.

Because we implement our client framework in Objective-C at first, we use the Sync framework[12] for data synchronization. To assist the Sync framework, we design our own messaging function called Grouper Message Protocol (Section 3.6). Using the Sync framework, we get a JSON string from an updated object, send the JSON to other devices, and update the mirrors of the object in these devices.

3.4 Reliable Messaging

The reliable message layer provides a reliable messaging service over the unreliable message layer. The upper layer rely on this reliable communication. The reliable message layer supports not only unicasting but also multicasting.

To address this problem, we use a reliable multicasting technique in distribute systems[11]. In this reliable multicasting technique, each message has a sequence number for each sender. Each member keeps the newest sequence num-

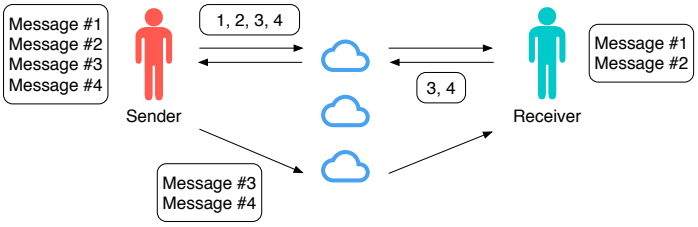


Figure 2: Implementing reliable messaging with sequence numbers.

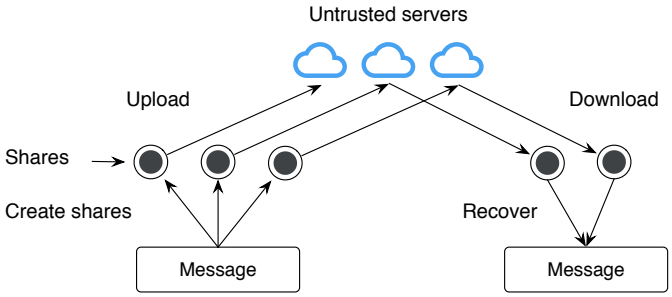


Figure 3: Implementing unreliable messaging with untrusted servers and a secret sharing scheme.

bers for senders and detects missing messages. If a member notices missing a message from a sender, the member asks the sender to resend the message. For example, consider that a sender sends Message No.25 to all other members using a multicast address. When a member receives this Message No.25, the member compares the sequence number 25 with the newest sequence number of the sender. If the the newest sequence number of the sender is 23, this means the member missed Message No.24. The member asks the sender to resend Message No.24. The sender will send Message No.24 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 design our own reliable messaging by extending the basic reliable multicasting technique. We add a special message type that includes active sequence numbers. A receiver can easily know missing messages. This idea is inspired from the checkgroups message of Usenet[12]. In Usenet, the list of active newsgroups is maintained with two basic massages: 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 message 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.

Figure 2 shows that the sender is sending four messages, No.1 to No.4, 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 two messages, No.3 and No.4 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 special message that includes active sequence numbers, No.1 to No.4. The receiver receives these sequence numbers and compares them with the newest sequence number in the persistent store. In Figure 3, the receiver notices that Messages No.3 and No.4 are missing. The receiver asks the sender to resend Messages No.3 and No.4. The sender will send Messages No.3 and No.4 to the receiver again using a unicast address.

3.5 Unreliable Messaging

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

In these layers, we use untrusted servers and a secret sharing scheme as shown in Figure 2. In this figure, the unreliable message layer in the sender is sending a message to that in the receiver.

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.

In Figure 3, the receiver is online, and its the unreliable message layer downloads two shares from two servers. Next, this layer calls the the data protection layer and recovers the message from these two shares using the secret sharing scheme. Finally, the unreliable message layer in the receiver puts the message to the upper layer, the reliable message layer.

The data protection layer uses Shamir's secret sharing scheme $f(k, n)$ introduced in Section 3.1 to protect a message from untrusted servers. In this scheme, a message is divided into n shares using a cryptographic function. A receiver gets more than k shares to combine them to original message. In Grouper, we use our new scheme $f(k, n, s)$ by extending the scheme $f(k, n)$. 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 $s \in [k, n] \bigcap N$. Although a receiver is able to recover the original message from at least k shares, we should consider server crashing. When a sender uploads n shares to n untrusted servers, Grouper will try 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 share uploading is successful. Otherwise, Grouper keeps trying to upload these shares.

3.6 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 2 shows the attributes in a Grouper message. There are three important attributes:

- **Type.** This attribute means types of messages. There are four types: update messages, delete messages, con-

Table 3: Attributes of Grouper message.

Attributes	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	Physical 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.
sendtime	Unix timestamp of sendtime.

Algorithm 1 Handle messages algorithm

```
1: procedure HANDLEMESSAGES(msgs)
2:     for msg in msgs do
3:         if msg.sender not exist then
4:             saveUser(msg)
5:         end if
6:         if msg.type ∈ {"update","delete"} then
7:             sync(message)
8:         else if msg.type ∈ {"confirm"} then
9:             seqs ← getSeqs(msg.content)
10:            seqs ← removeExisted(seqs)
11:            resendMsg(seqs)
12:        else if msg.type ∈ {"resend"} then
13:            if msg.receiver = currentUser then
14:                seqs ← getSeqs(msg.content)
15:                sendExistingMsgs(seqs)
16:            end if
17:        end if
18:    end for
19: end procedure
```

firm messages and resend messages. Both update message and delete message contain the marshalled objects of an application. We call these 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 is a 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 or a resend message, the content value contains sequence numbers.
- **Sequence.** This attribute means the sequence number of the message. When a sender sends a new message, the sender increments the sequence number and includes it to the new message. Using both the sequence number and the sender ID identify a unique message.

The receiver attribute 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 message after receiving messages automatically. When the applications invoke 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 the applications invoke 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 the applications invoke 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. Generally, the method grouper.confirm() is invoked at the following occasions:

- **Periodically.** For example, an application sends a confirm message now, and it will try to 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 tries to send a confirm message when this devices becomes online.

Algorithm 1 describes the handle process when the Grouper API layer receives messages. For an update message or a delete message, Grouper invokes the API provided in third-party synchronization framework to get the object into persistent store. For a confirm message, Grouper gets the sequence numbers from the message content and removes those sequence numbers which is existing in the device. Then Grouper creates a resend message that contains 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, finds the corresponding normal messages and send them to the sender of this resend message.

3.7 Group Management

3.7.1 Creating a Group

A user creates a group, and he 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 server by submitting his node identifier. 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.

3.7.2 Inviting a Member

After creating a group, the owner can invite a new member to his group. To join the group, the 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. Before inviting, Grouper establishes connection between their devices using a local safe communication channel like *Multipeer Connectivity*[14]. Firstly, the new member sends user information and a node identifier to the owner. Owner saves the user information and the node identifier of the new member to his device. Secondly, the owner registers the new member on multiple untrusted servers by submitting the node identifier of the new member. Thirdly,

untrusted servers returns access keys for the new member to the owner. Lastly, the owner sends the access keys, server addresses 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 Web service running on multiple untrusted servers and a client framework for developing applications. We introduce the implementation of the Web service (Section 4.1), the implementation of the client framework (Section 4.2) and demo applications (Section 4.3) in this section.

4.1 Web Service

Grouper needs its own Web service rather than using commercial general cloud services like Amazon S3, Google Cloud for the following reasons:

- The Web service must provide reliable synchronization based on the *Grouper Message* protocol.
- The Web service must ensure that shares are deleted after a prescriptive time.

Our Web service provides RESTful API to transfer data with clients. It runs on the Tomcat server that is an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. We use the Spring MVC, a Web model-view-controller framework, to create our RESTful API, and Hibernate, 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. They are *Group*, *User* and *Transfer* entities. A *Group* entity saves a group 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 *Transfer* entity saves a share generated with a secret sharing scheme, the time when the user uploads the share. For each user, there is a unique access key for him in an untrusted server. For a group, one of a user is its owner who has the highest privilege of this group.

4.2 Client

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

- *Multipeer Connectivity*[14], an official Peer-to-Peer communication framework provided by Apple. Grouper uses it to transfer data between two devices by a face-to-face way.
- *Core Data*[15], 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*[13], a synchronization framework for *Core Data* using JSON. When a user sends messages, Grouper uses it to create JSON strings from objects. When an other user receives messages, Grouper uses it to parse JSON strings and synchronize the recovered objects into *Core Data*.
- *c-SSS*[16], an implementation of the secret sharing scheme.
- *AFNetworking*[17], a delightful networking library in Objective-C. Grouper uses it to invoke the RESTful API provided by our Web services running on multiple untrusted servers.

4.3 Applications

Using Grouper framework, we are developing the following applications.

- *Account Book*, an iOS application in Objective-C, records the income and expenditure of a group.
- *Test*, a benchmark iOS aapplication in Swift, tests the performance of Grouper.
- *Notes*, a macOS application in Swift, takes shared notes for a small group.

5 Evaluation

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

5.1 Developer Efforts

We see developer efforts through two factors: the usability of the client API and the code size in the lines of code(LoC) the developer has to add after using Grouper. As described in Table 2, Grouper provides the simple client APIs for developers. To extend a stand alone application, developers invoke the *grouper.setup()* to set appId and data stack. To synchronize data among devices, developers invoke the *grouper.sender.update()*, *grouper.sender.delete()* and *grouper.receiver.receive()* method. To ensure the data created in a device has been synchronized in other devices, developer invoke the *grouper.confirm* method.

We have developed two applications including *Account Book* and *Test* with Grouper. As described in Table 3, 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.

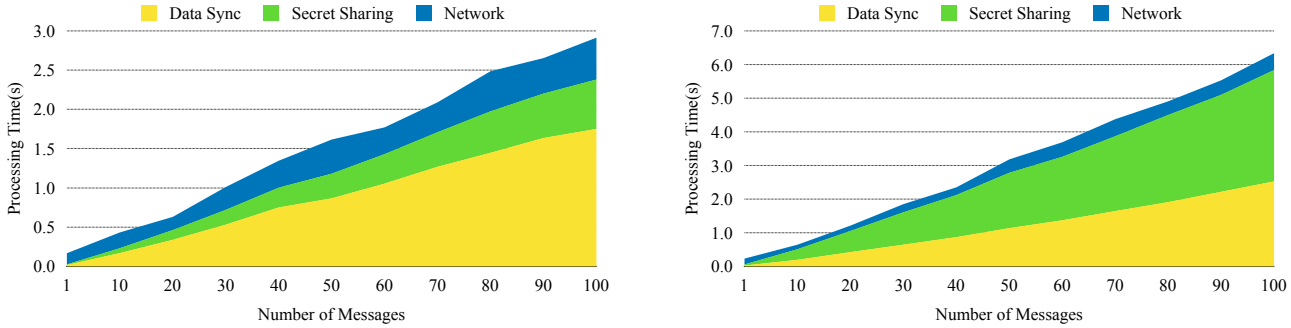
5.2 Performance

The performance goal is to avoid significantly affecting the user experience with the application developed with Grouper. In our performance experiments, we use 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 (azuma1, azuma2 and azuma3). Each server runs a Tomcat server which includes 10 Web services. Table 4 shows the hardware and software information of in our performance experiment. In our benchmark application, the size of a normal message is about 620 bytes.

To evaluate whether Grouper meets this goal, we answer the following questions:

Table 4: Applications' lines of code.

Application	Platform	Lanaguage	Number of Entities	Stand Alone Application LoC	Increased LoC
Test	iOS	Swift	1	621	8760
Account Book	iOS	Objective-C	5	8760	190



(a) Sending multiple messages.

(b) Receiving multiple messages.

Figure 4: Processing time of sending and receiving 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
azuma1	Core i7-5820K	32GB	Ubuntu 14.04.5 LTS
azuma2	Core i7-5820K	32GB	Ubuntu 14.04.5 LTS
azuma3	Core i7-5820K	32GB	Ubuntu 14.04.5 LTS

- How much processing time does Grouper add to the application in data sending and receiving.
- How many users can an application by Grouper support.
- How does the number of servers n and threshold k of the secret sharing scheme influence the processing time.

To answer these questions, we design the following groups of experiments.

5.2.1 Multiple Message

To answer the first and second questions, we design the multiple message transportation experiment. In this experiment, we set the secret sharing scheme to $f(2,3)$. We send multiple messages from a device and receive them in another device. To ensure the veracity, we send multiple message from iPod 5 generation to iPhone 4s for three time and from iPhone 4s to iPod 5 generation for three times. We use the average value of the six groups of data as our experiment results. In the next experiments, we also use this method to statistic the results.

Figure 5 shows the processing time of sending and receiving multiple messages. We divide the processing time into three parts: data sync, secret sharing and network. As the number of messages increased, data sync and secret sharing part increased linearly. The network part increased very slowly and sometimes decreased. On the whole, the total processing time increased linearly. Compared with sending messages, receiving messages cost about two times of processing time. These experimental results show that data synchronization within a hundred messages does not influence the user experience.

With the increase of the group scale, a device must be able to handle many messages at the same time. For example, in 100 devices in a group, each device sends an update message at the same time and then tries to synchronize messages created by others. In this situation, each device has to receive and handle 99 messages at the same time. Thus, a group of an application by Grouper is able to expand to 100 members.

5.2.2 Single Message with Different Schemes

To answer the third question, we design the single message transportation experiment with different the secret sharing scheme. Specifically, we change the parameter k and n of the secret sharing scheme and test the processing time of sending and receiving a single message. Figure 5 shows the relationship between processing time and the data set $\{(k,n) \mid 0 < k < n, k = 3i, n = 3j + 3, i, j \in [1,5] \bigcap N\}$. For sending a single message, as the parameter k or n increased, processing increased linearly. However, for receiving a single message, as the parameter k increased, processing time changed a little, sometimes decreased.

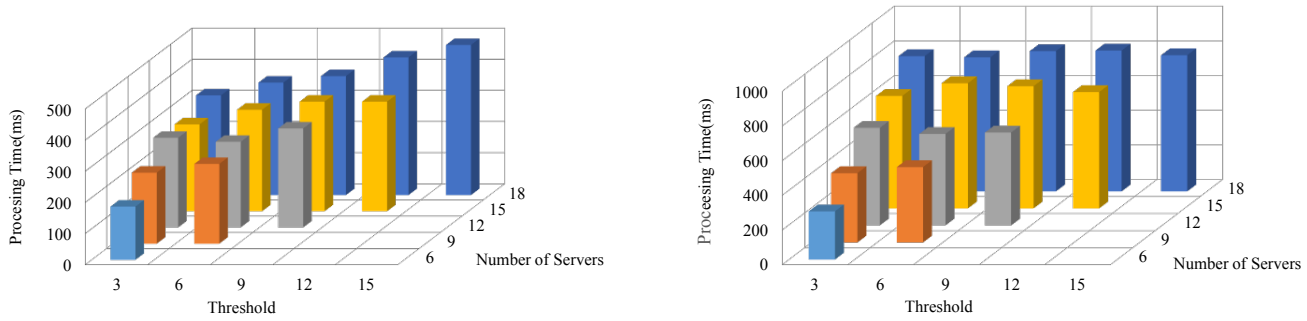
5.2.3 Control Variable

We need more data to verify the result introduced above further. We use control variate method to design this experiment. Figure 6a shows the relationship between processing time and n, here $k = 3$ and $n \in \{x \mid x = 3i, i \in [2,10] \bigcap N\}$. Figure 6b shows the relationship between processing time and k, here $n = 30$ and $k \in \{x \mid x = 3i, i \in [1,9] \bigcap N\}$. Here, we can answer the third question. With the increase of n, processing time of sending and receiving increase linearly. With the increase of k, processing time of sending increase linearly and processing of receiving does not change. We find the reason is that the time of recovering shares by the secret sharing scheme depends only on the parameter n.

From these performance experiment, we can conclude that Grouper is able to support at least 100 members' group and at least 30 untrusted servers.

6 Related Work

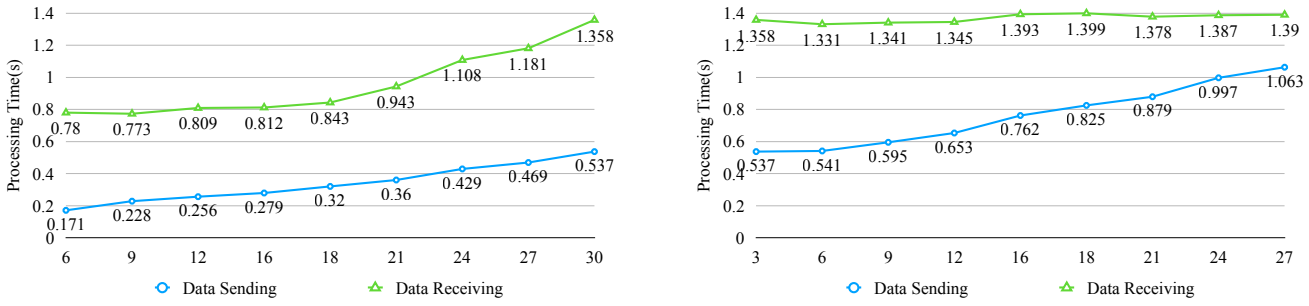
Vanish is a system proposed by Geambasu's research group at the University of Washington. Vanish uses Distribute



(a) Sending a single message

(b) Receiving a single message

Figure 5: Processing time of sending and receiving a single message with different scheme.



(a) Constant threshold.

(b) Constant number of servers.

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

Hash Tables(DHTs) as the back-end storage. Concretely, to protect a message, Vanish encrypts it with a random encryption key not known to the user, destroys the local copy of the key, and store shares created by a secret sharing scheme of the key in a large, public DHT. The key in Vanish is permanently after a period of time, and the encrypted message is permanently unreadable. Vanish is implemented with OpenDHT[18] or VuzeDHT[19] which are controlled by a single maintainer. Thus, it not strongly secure due to some special P2P oriented attacks[20]. In addition, the surviving time of the key in Vanish cannot be controlled by user.

To address such issues in Vanish, Zeng et al. at Huazhong University of Science and Technology, propose SafeVanish and SeDas. 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 propose CloudSky, a controllable data self-destruction system for untrusted cloud storage. In CloudSky, user can control the surviving time of a message.Taking advantage of ABE, user can also define the access control policy by themselves.

However, both proposals from Geambasu's group and Zeng's group are not suitable for developing a light-weight information sharing application for following reasons. Vanish is suitable for a mail system, because it is designed without needing to modify any of the stored or archived copies of a message and without user controllability, while messages in our target applications should be modified even if it has been sent to multiple untrusted servers. Although, CloudSky solves the problems about user controllability in Vanish, the encrypted message are only valuable to the user for a limited period of time. Our target applications require data usability even user try to synchronize data after the period of time. A trusted authority is necessary in CloudSky to manage user profile, while we do not hope any trusted authority in our target application.

Mylar 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 in the secret sharing scheme.

Sweets is a decentralized social networking service (SNS) application using data synchronization with P2P connections among mobile devices. Sweets uses AES to encrypt user data and ABE to encrypt the keys of AES. However, there is an obvious problem in such a P2P approach. Data transfer can only be finished during two devices are online at the same time. Therefore, it is very troublesome for a user of our a target application if nobody is online when he want to synchronize data. The user can synchronize data from multiple untrusted servers anytime if the application uses the proposal of Grouper.

DepSky[21] is a system that stores encrypted data on servers and runs application logic. DepSky provides a storage service that improves the availability and confidentiality by using commercial storage services. *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 fact, DepSky is suitable for such data storage

applications. 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 features. Firstly, like data encryption method, using the secret sharing scheme is also secure because a single shares created by it is unreadable for a server manager. However, using data encryption requires 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 more than k untrusted servers. Thirdly, the secret sharing scheme improves the anti-attack ability, because the attacker who can access only $k-1$ or less untrusted servers cannot get any readable informations. At last, the performance of the secret sharing scheme we used in Grouper is faster than Attribute Based Encryption (ABE) and slower than Advanced Encryption Standard (AES). For these reasons, using a secret sharing scheme is more suitable for the Grouper framework than using a data encryption method.

7 Conclusion

This paper describes Grouper, a framework using a secret sharing scheme and multiple untrusted servers, to develop light-weight information sharing 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 self-destruction proposals introduced in related works, Grouper solves the reliable synchronization problem and ensures a member of a group can synchronize data from others even Grouper includes a self-destruction scheme. Compared to pure data encryption proposals introduction in related works, Grouper improves dependability by using multiple untrusted servers and our new enhanced secret sharing scheme $f(k, n, r)$.

We implement Grouper's Web service in Java EE and clients in Objective-C. To evaluate Grouper's design, we develop applications including *Account Book*, *Notes* and *Test* on the top of Grouper. These applications shows that Grouper requires little developer effort to extend an stand alone application to data sharing application with synchronization. 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 will try improve the performance for data synchronizations and support more platforms .

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*, volume 316, 2009.

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

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

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

[10] TICoreDataSync. `https://github.com/nothirst/TICoreDataSync`.

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

[12] Usenet. `https://tools.ietf.org/html/rfc5537`.

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

[14] Apple Inc. MultipeerConnectivity. `https://developer.apple.com/documentation/multipeerconnectivity`.

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

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

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

[18] 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*, volume 35, pages 73–84. ACM, 2005.

[19] Azureus. `http://www.vzue.com`.

[20] 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.

[21] 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.