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

Meng Li and Yasushi Shinjo
University of Tsukuba

August 10, 2017

Abstract

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. 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 JSON string of an object and uploads these shares to different untrusted servers. Our multiple untrusted servers system is a self-destruction system. 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 implement Grouper in the Objective-C language at first and evaluate it from developer efforts and performance. We also describe three applications by Grouper framework: an iOS application *Account Book*, a macOS application *Notes* and a benchmark application *Test*.

Keywords: mobile application security, secret sharing, untrusted server

1 Introduction

Using mobile applications using central trusted servers requires the user to trusted the service provider. Confidential data stored in a central trusted server will be revealed once a malicious person attacked this server. Our research goal is to develop light-weight information sharing applications for small groups with our framework Grouper. How to protect the sensitive data stored in the central servers is the most important problem in our research. However, achieving the goal is a great challenge because servers cannot be controlled by users themselves.

To address such a problem, Vanish[1], SafeVanish[2], SeDas[3] and CouldSky[4] use a data self-destruction system as their cloud storage. Vanish is a system proposed by Geambasu's research group at the University of Washington. Vanish uses Distribute 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[5] or VuzeDHT[6] which are controlled by a single maintainer. Thus, it not strongly secure due to some special P2P oriented attacks[7]. 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 Attribute Based Encryption (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.

Other proposals also use data encryption to protect user data. Mylar[8] 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[9] is a decentralized social networking service (SNS) application using data synchronization with P2P connections among mobile devices. Sweets uses Advanced Encryption Standard (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.

Inspired by such researches, we proposal our own framework Grouper. It ensure both data security and usability. Other advantage of Grouper is data can be recovered even untrusted servers shut down because all devices of group members keep a complete data set of this group.

2 Assumption and Threat Model

In this section, we introduce assumptions and threat model of Grouper.

2.1 Assumption

There are following basic assumptions underlying the Grouper framework:

1. All group members must trust the group owner and they are not malicious.
2. In members inviting, a group owner authenticates group members by a face-to-face way.
3. Nobody has privilege of more than k (the threshold of a secret sharing scheme) untrusted servers.

We consider target users usage scenario of an application by Grouper is a small group like all members in a small office, so all group members of this application are acquaintances and they are not malicious. Due to this reason, their devices can also connect to each other by a face-to-face way.

In Grouper, we use the secret sharing scheme to protect our user data. In a secret sharing scheme, a member securely shares a secret with a group of members by generating n shares using a cryptographic function[10]. At least k or more shares can reconstruct the secret, but $k-1$ or fewer shares can obtain nothing about the secret[11]. 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. Therefore, anyone who has access to k or more shares, he can recover the original data. Although Grouper store shares created by a secret sharing scheme in multiple untrusted servers, these servers are managed by human beings. If one person can access to k or more untrusted servers, he has enough shares to recover the original user data. In fact, we hope each untrusted server's manager does not know the existence of others. For example, 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.

2.2 Threat Model

There are following key properties of our threat model:

- **Server side authentication.** Our untrusted servers must perform 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.
- **Data self-destruction.** Our untrusted servers keep data temporarily. We define a period of time in which data can be kept in a server as an interval time. The data in untrusted servers vanishes after the interval time.
- **Secure data transportation.** We assume that both inter-mobile device communication during user invitation and HTTP connection between a device and an untrusted server is secure.

Due to the assumptions and key properties of threat model, we can design and implement our Grouper framework.

3 Design

In this section, we explain our target applications, basic functions provided by Grouper and advantages in overview (Section 3.1). Then we show the architecture (Section 3.2) of Web service and client of Grouper. At last, we introduce how we design the Grouper framework.

3.1 Overview

We are aiming at developing applications which are not relying on trusted central servers. Concretely, Grouper's design is suitable for applications that:

1. Use in a small groups.
2. Require light-weight informations sharing among members in a small groups.
3. Allow face-to-face connection among members' devices.

Grouper support to exchange object among devices. Binary data can be included in an object, but the isolated binary file is not supported in Grouper. For example, *Account Book* is an iOS application developed by Grouper. In this application, a leader of a small company creates a group and invites employees to the group. Then, everyone can record the income and expenditure of this company and share this record to others. Anyone can edit and delete all records.

To developer such a target application, the Grouper framework should provide 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.

Compared to pute data self-destruction proposals Vanish, SafeVanish, SeDas and CloudSky, or pure data encryption proposals Mylar and Sweets, Grouper is suitable for developing such light-weight informations sharing applications due to such advantages:

1. User invitation requires a face-to-face way between group owner and new member. It ensures group members are not malicious.
2. Unlike CouldSky, users are not required to know the approximate lifetime of a message and synchronize it within the lifetime.
3. Damaged shares from untrusted server do not destroy data synchronization among users completely.

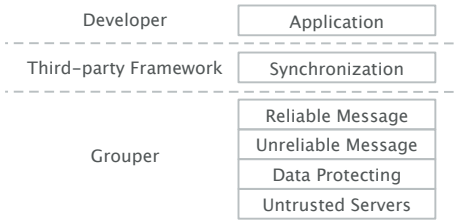


Figure 1: Architecture of Grouper.

3.2 Architecture

Figure 1 describes the architecture of Grouper. An application by Grouper consists of the following layers:

- **Application.** A developer develops a stand alone application without data synchronization at first. He can use the API provided in the client framework of Grouper to extend synchronization function to his application.
- **Synchronization.** Grouper uses the third-party framework for data synchronization. This layer marshall and unmarshall the message from the reliable message layer and get it into persistent store.
- **Reliable Message.** This layer provide a reliable message service. This layer try ensure a message can be received in another device even its shares has been deleted in untrusted servers.
- **Unreliable Message.** This layer provide an unreliable message service to exchange data between two devices in a group. This layer do not ensure the message can be received in another device.
- **Data Protecting.** Grouper protect user data by a secret sharing scheme in this layer. A message is divided to several shares and these shares can be recovered be a part of shares in this layer.
- **Untrusted servers.** Clients upload shares to untrusted servers and download shares from untrusted server by the RESTful API provided in this layer. The untrusted servers install our own Web service of Grouper.

3.3 Data Synchronization

To synchronize data from one device to another device, we consider the following problems:

- **Transportation:** How to transport shares from one device to another device.
- **Synchronization:** How to create several shares from an object in a device. How to recover the shares to an object and get it into persistent store.
- **Reliability:** How to ensure a device can synchronize the messages even he missed in the limited period of time.

We answer these questions and describe data synchronization flow in this section.

3.3.1 Transportation

Figure 2 describes our data transportation flow using multiple untrusted servers. At first, the sender adds an object in

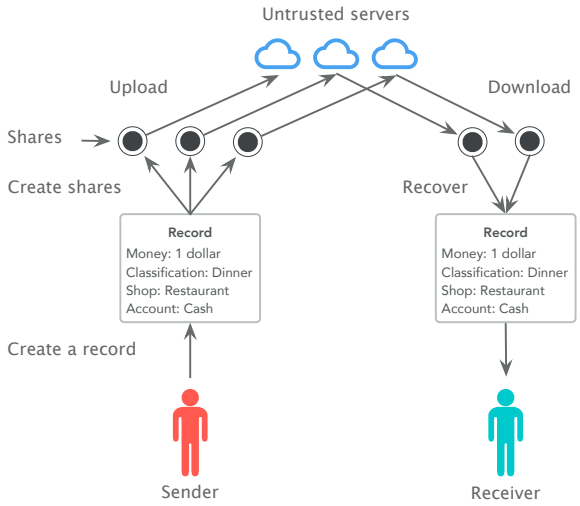


Figure 2: Data transportation flow in Grouper.

his device and Grouper creates three shares by a secret sharing scheme. Next, Grouper uploads those shares to three untrusted servers. In Figure 2, the receiver is online, and he downloads two shares from two servers and recovers the new record. In this process, these servers cannot recover user data because they do not have permission to access other untrusted servers.

To improve the reliability in data transportation, we design our own scheme $f(k, n, r)$ for Grouper based on the basic the Shamir's secret sharing scheme $f(k, n)$. In the $f(k, n, r)$ scheme, the parameter k and n is same as which in the $f(k, n)$ scheme. The parameter r represents the number of necessary untrusted servers when a sender uploads shares. Although the receiver is able to recover the original data from the receiver with more than k shares, we should consider a situation that some shares are broken. Those broken shares may caused by the unstable network connection, the database damage in untrusted servers or malicious attackers. Consider the situation that there are n untrusted servers. When a sender wants to upload shares to multiple untrusted servers, Grouper will try to upload these shares to all untrusted servers at first. If the shares are uploaded to r or more untrusted servers, we consider share uploading transaction is successful. Otherwise, Grouper should try to upload these shares again after a period of time.

3.3.2 Synchronization

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 3.4). 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.

We should consider the confliction problem in data synchronization. Most data sharing applications like Evernote provide a function that user can select an edition by themselves when data is conflicting. However, the Sync framework has not provide such functions by now. The default policy in the Sync framework is using the newest edition. For example, both Alice has modified an object and Bob modified it one hour later. Grouper will receive Bob's modification due to the newest edition policy.

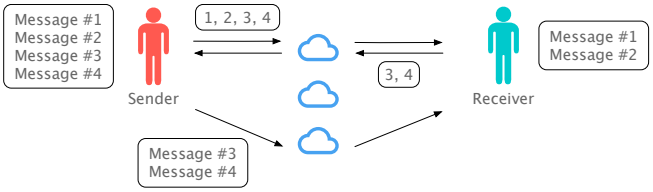


Figure 3: Reliable synchronization.

3.3.3 Reliability

Grouper should provide a reliable synchronization service. A user in a group creates a new record and all of other members in this group should synchronize this record, even if this record is deleted by untrusted servers after an interval time. We call this problem reliable synchronization. A receiver can only download shares from untrusted server with the interval time. If he is offline within the interval time, he will miss the new record.

To solve this problem, we reference the basic reliable multicasting scheme in distribute systems. Messages in this scheme should have sequence number which is added one by one, to indicate the sending order of those messages. For example, the sender of a group send a message No.25 and all of the receivers receive the message No.25 successfully. When the receivers receive new message No.25, they will compare No.25 with the newest sequence number in their persistent store. One of the receivers finds that his newest sequence number is No.23, that means he missed the message No.24. Thus, when they send feedback to the receiver, he will report that he missed the message No.24. At last, the sender will send the missed message to him.

Based on the basic reliable multicasting scheme, we designed our own protocol to ensure the reliability in data synchronization of Grouper. As described in Figure 2, the sender has sent messages from No.1 to No.4. Next, he sends sequence numbers of all messages he sent to the receiver. When the receiver receives the sequence numbers, he checks his local persistent store. In this situation, the receiver finds that he missed the message No.3 and No.4. Thus, he will send a resend request that contains the sequence numbers of messages he missed to the sender. At last, the sender will send the message No.3 and No.4 again to the receiver again.

3.4 Grouper Message Protocol

To transfer data between devices, we design our own protocol, *Grouper Message* Protocol. In this protocol, a message is a JSON string that contains an object of an application and the way to handle it in receivers devices. Table 1 shows the attributes in a Grouper message. We concentrate on three important attributes here.

- **Type.** There are 4 types of Grouper messages: update message, delete message, confirm message and resend message. Both update message and delete message need resending because they contain the objects of an application. We call these messages normal messages. Both confirm message and resend message contain control information about reliable multicast and need not resending. We call these messages control messages.
- **Content.** If this message is an update message, content is a JSON string of an object. If this message is a delete message, content contains the objectId of an object. If this message is a confirm message or a resend, content contains the sequence numbers.
- **Sequence.** The sequence number increases one by one in a device. Combined with the node identifier, it can indicate an unique Grouper message. Thus, it is contained in a confirm message to indicate the all normal messages a device sent or in a resend message to indicate the normal message a device has not been received.

Table 1: Attributes of Grouper message

Attributes	Type	Explanation
type	String	Type of this message.
content	String	JSON string of an object.
messageId	String	Physical ID of this message.
object	String	Object name.
objectId	String	Object physical ID.
receiver	String	Node identifier of the receiver.
sender	String	Node identifier of the sender.
email	String	Email address of the sender.
name	String	Name of the sender.
sendtime	Integer	Unix timestamp of sendtime.
sequence	Integer	Sequence number of this message.

Algorithm 1 Handle messages algorithm

```
1:  procedure HANDLEMESSAGES(msgs)
2:      for msg in msgs do
3:          if msg.sender not exist then
4:              saveUser()
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
```

Developers need to send the update, delete and confirm message using client API of Grouper. When a user creates a new object or modifies some attributes of an existing object, the device sends an update message that contains the JSON string of this object to all group members. When a user deletes an existing object, the device sends a delete message that contains the object ID of this object to all group members. To confirm all devices have received all normal messages (update message and delete message) created by a user, the device sends a confirm message to all devices periodically. In this confirm message, the sequence number of objects recently created in this device are included.

Algorithm 1 describe the handle process when a device get some 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.

Fo ra 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, Grouper gets the sequence numbers, finds the corresponding normal messages and send them to the sender of this resend message.

3.5 Group Management

3.5.1 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.

3.5.2 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 local secure links like *Multipeer Connectivity*[13]. 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*[13], 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*[14], 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*[12], 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*[15], an implementation of the secret sharing scheme.
- *AFNetworking*[16], 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.

Table 2: 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 3: 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

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: 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

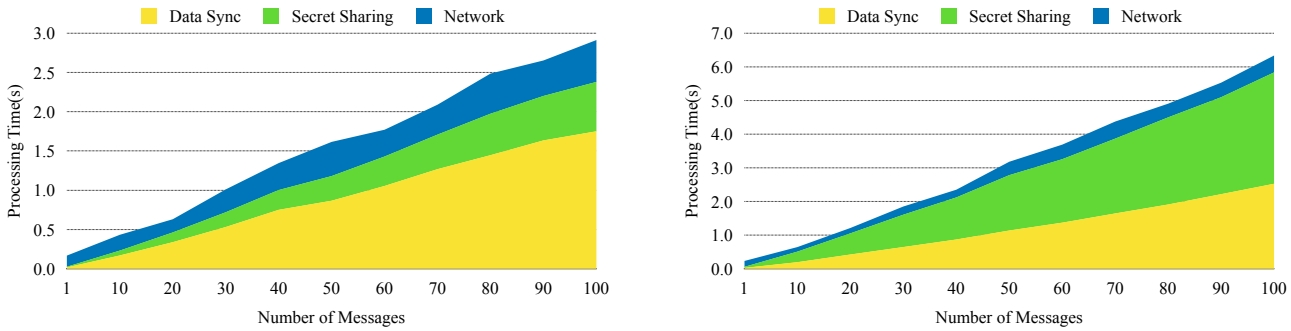
- How much processing time does Grouper add to the application in data sending and receiving.
- How many users can an application by Grouper support.
- How does the number of servers n and threshold k of the secret sharing scheme influence the processing time.

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.

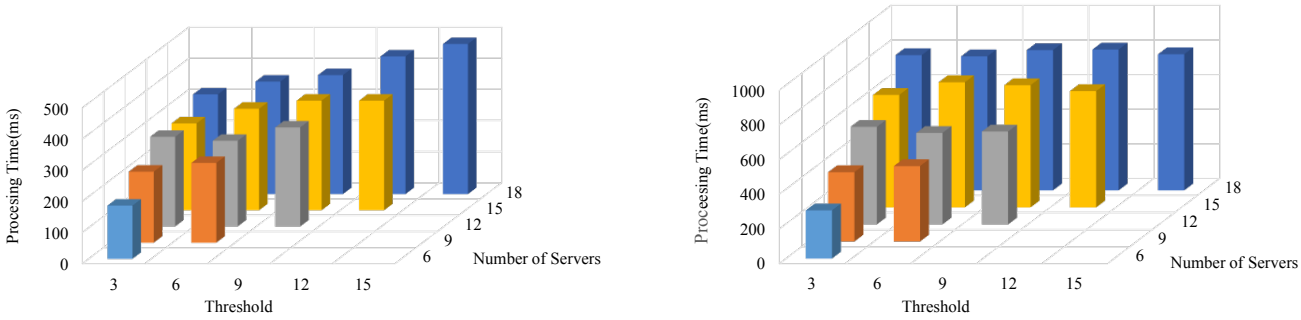
Figure 5 shows the processing time of sending and receiving multiple messages. We divide the processing time into three parts: data sync, secret sharing and network. As the number of messages increased, data sync and secret sharing part increased linearly. The network part increased very slowly and sometimes 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.



(a) Sending multiple messages.

(b) Receiving multiple messages.

Figure 4: Processing time of sending and receiving multiple messages.



(a) Sending a single message

(b) Receiving a single message

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

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.

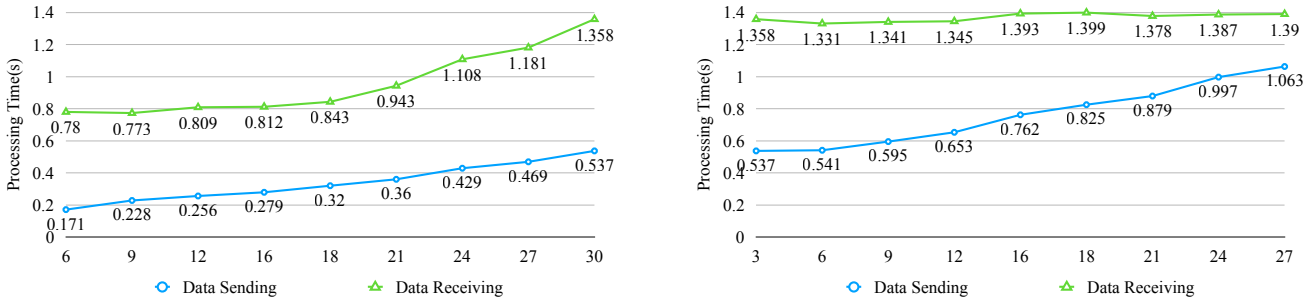
From these performance experiment, we can conclude that Grouper is able to support at least 100 members' group and at least 30 untrusted servers.

6 Related Work

We have introduced a large amount of related works in Section 1 throughout the text. Their proposals protect user data by self-destruction and encryption. As additional related work, DepSky[17] 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.

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



(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.

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] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

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

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

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

[17] 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.