

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

Meng Li

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

Yasushi Shinjo

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

ABSTRACT

Conventional mobile applications are built based on a client-server mode and require central servers for storing shared data and processing confidential information. If the central servers are accessed by an attacker, a curious administrator or a government, private information will be revealed because data is often stored on the central servers in cleartext. This paper presents Grouper, a framework for developing mobile applications without trusted central servers. Grouper provides object synchronization among mobile devices. 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. Grouper consists of a client framework and a Web service. We have implemented the client 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 the Java language. We have implemented two applications using Grouper: an iOS application Account Book and a benchmark application Test. We evaluate the development efforts and the performance of Grouper. Developing these applications shows that Grouper requires little development effort to convert an stand alone application to data sharing application. 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

ACM Reference Format:

Meng Li and Yasushi Shinjo. 2017. Grouper: a Framework for Developing Mobile Applications using a Secret Sharing Scheme and Untrusted Servers. In *Proceedings of 2017 VI International Conference on Network, Communication and Computing (ICNNC)*. ACM, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.475/123_4

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[7], SafeVanish[24], SeDas[23] and CouldSky[25] construct a data self-destruction system as their cloud storage. In these approaches, servers store encrypted 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, Grouper 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 client 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 the Java language. We used a Sync[15] framework in Grouper to synchronize objects among mobile nodes. We have implemented two applications using Grouper: an iOS application Account Book and a benchmark application Test. These implementations show 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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICNNC, 2017, China

© 2017 Copyright held by the owner/author(s).

ACM ISBN 123-4567-24-567/08/06...\$15.00

https://doi.org/10.475/123_4

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 a special member called an owner and other members. Each member has a mobile device. A owner invites other members in a face-to-face way.

Firstly, while we use servers, they are passive adversaries, and can read all data, but they do not actively attack. Servers host Web services and perform device authentication. Servers generate access keys for group members. When a device wants to get/put data from/to servers, the device sends a request with an access key. In this paper, we do not address other types of attacks such as user tracking and metadata collection by servers. For example, servers can track users with IP addresses, and Grouper cannot hide social graphs against such 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 at the time of user invitation. 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. Note that after user imitation, devices communicate through servers and no secure direct communication path is required.

At last, servers are 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 are developing the Grouper framework. This framework provides the following functions:

- **Data Synchronization.** If an user updates or deletes an object in his device, the mirrors of this object in other devices are updated or 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 can create a group and invite employees to the group. Then, the employees can record the income and expenditure of this company. These

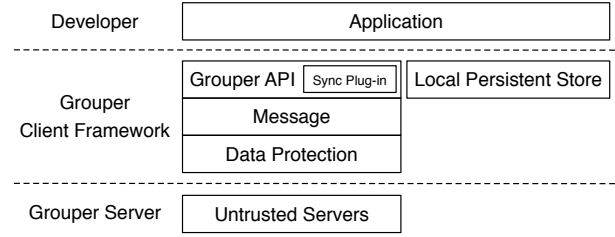


Figure 1: Architecture of Grouper.

income and expenditure records are represented as objects and shared among devices. Anyone can edit and delete existing records in his device.

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

Grouper uses Shamir's secret sharing scheme to protect messages from the providers of untrusted servers. In this scheme, a member securely shares a secret with other members by generating n shares using a cryptographic function[20]. At least k or more shares can reconstruct the secret, but $k-1$ or fewer shares output nothing about the secret[16]. We describe this scheme as a function $f(k, n)$, where n is the number of all shares, and k is the threshold to combine shares.

Grouper has the following advantages over conventional approaches using a self-destruction system. 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. Grouper consists of the following four layers and a plugin:

- **Grouper API.** A developer develops an application without data synchronization at first. He adds the synchronization function to his application by using 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 message layer. When the Grouper API layer receives a message, this layer unmarshalls the message using the plugin, reconstructs an message, and puts object included in the message into the persistent store.
- **Message.** This layer provides a messaging transportation service with multicasting capability among devices. The destination of a message is not only the node identifier (ID) of a single device but also "", which means delivering to all the other nodes. This layer does not ensure the message delivery to other devices.

Table 1: Client API of Grouper.

| Method | Description |
|---|--|
| <i>grouper.setup(appId, dataStack)</i> | 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. |
| <i>grouper.sender.update(object)</i> | 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. |
| <i>grouper.sender.delete(object)</i> | 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. |
| <i>grouper.receiver.receive(callback)</i> | 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. |
| <i>grouper.sender.confirm()</i> | 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 |
|------------------------|--|
| <i>marshall(o)</i> | Marshalls the object <i>o</i> and returns the marshalled byte array. |
| <i>updateRemote(b)</i> | Unmarshalls the byte array <i>b</i> to the object and puts the object into the persistent store. |
| <i>deleteRemote(b)</i> | Deletes the object in the byte array <i>b</i> of object IDs. |

- **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. Table 1 shows the client APIs of Grouper for developing our target applications. An application initializes the framework by invoking the method *grouper.setup()*. When the application needs to update an object in all devices, 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 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 the synchronization plugin. 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.

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.

Because we implement our client framework in Objective-C, we use the Sync framework[15] 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[21]. In this reliable multicasting technique, each message has a sequence number for each sender. Each member keeps the newest sequence numbers for senders and detects missing messages. If a member notices 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

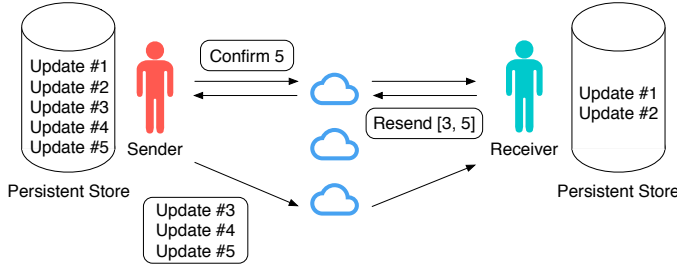


Figure 2: Implementing reliable messaging with continuous sequence numbers.

this update message No.3, the member compares the sequence number 3 with the newest sequence number of the sender. If the newest sequence number of the sender is 1, this means the member missed update message No.2. The member asks the sender to resend update message No.2 using a control message. The sender will send update message No.2 to the 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 servers delete these three messages, again. The receiver becomes online, but the receiver does not notice that the receiver has missing these three messages soon.

At this time, the sender sends a confirm message that includes the newest active sequence number, 5 as shown in Figure 2. The receiver receives this sequence number and compares it with the newest sequence number in the persistent store. In Figure 2, the receiver notices that update messages No.3, No.4 and No.5 are missing. The receiver asks the sender to resend update messages No.3, No.4 and No.5 using a resend message. The sender will send update messages No.3, No. 4 and No.5 to the receiver again using a unicast address.

This idea is inspired from the checkgroups message of Usenet[2]. In Usenet, the list of active newsgroups is maintained with two basic messages: newgroup and rmgroup. When a node receives a newgroup message, the node adds the newsgroup to the list. When a node receives a rmgroup message, the node removes the newsgroup from the list. These basic messages can be lost and the list can be obsolete. Ccheckgroups 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 Data Protection

The data protection layer uses the Shamir's secret sharing scheme $f(k, n)$ as introduced in Section 3.1. In Grouper, we extend this scheme 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 crashes. When a sender has n shares, Grouper tries to upload these n shares to all n untrusted servers at first. If the shares are uploaded to s or more untrusted servers, we consider that this uploading is successful. Otherwise, Grouper keeps trying to upload these shares.

Figure 3 shows the message transportation from a device to another device using the extended secret sharing scheme $f(2, 4, 3)$. In this figure, the message layer in a sender device is sending a message to that in a receiver device. At first, this layer calls the data protection layer and creates four shares by the secret sharing scheme. Next, this layer uploads those shares to four untrusted servers. In Figure 3, although only the left three untrusted servers receives shares successfully, we regard this uploading successful. At last, the message layer in the receiver device downloads two shares from the four untrusted servers and recovers the message by calling the data protection layer.

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

- **Sender first.** An application sets s closed to k . A sender gets a successful result earlier. When some servers 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 are not available, a receiver more likely get a message.

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. Secondly, the secret sharing scheme ensures the data availability when a number of untrusted servers are not accessible. For the $f(k, n, s)$ scheme in Grouper, a sender can complete uploading when at least s untrusted servers are available, and a receiver can recover the object when at least k untrusted servers are available. 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.

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 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

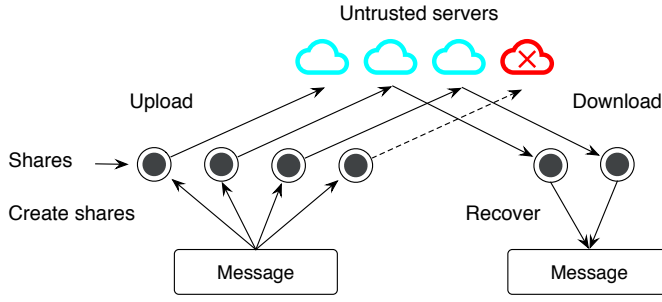


Figure 3: Message transportation using the extended secret sharing scheme $f(2, 4, 3)$.

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. |

contains the marshalled objects of an application. A delete message contains the identifier of a deleted object. A confirm messages contains the sequence numbers of all normal message created in the device. A resend message contains the sequence numbers of missing messages. 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 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.

Algorithm 1 Handle message algorithm

```

1: procedure ONMESSAGERECEIVED(msg, sender)
  ▶ Check duplicate message.
2:   historyMsgs ← getMsgsBySender(msg.sender)
3:   if msg ∈ historyMsgs then
4:     return
5:   end if
6:   lastMsg ← historyMsgs.last()
7:   historyMsgs.add(msg)
  ▶ Basic reliable multicast.
8:   if msg.seq ≠ 0 && lastMsg.seq + 1 ≠ msg.seq then
9:     resendMsg ← createResendMsg(lastMsg.seq+
10:      1, msg.seq)
11:     ▶ Create a resend message with the minimum and
12:       maximum sequence number.
13:     sendMsg(resendMsg, sender)
14:   end if
  ▶ Handle the message by its type.
15:   if msg.type = "update" then
16:     sync.updateRemote(msg)
17:   else if msg.type = "delete" then
18:     sync.deleteRemote(msg)
19:   else if msg.type = "confirm" then
20:     maxSeq = getMaxSeqFrom(msg.content)
21:     resendMsg ← createResendMsg(lastMsg.seq+
22:       1, maxSeq)
23:     if resendMsg ≠ null then
24:       sendMsg(resendMsg, sender)
25:     end if
26:   else if msg.type = "resend" then
27:     seqs ← getSeqs(msg.content)
28:     for seq ∈ seqs do
29:       missingMsg ← getMsg(seq)
30:       sendMsg(missingMsg, sender)
31:     end for
32:   end if
33: end procedure

```

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 once during the TTL.

- **When the device becomes online.** Sometimes, a device is offline and cannot send a confirm message within 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.

3.7 Group Management

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

- Group Creation.
- Member Invitation.

A user can create a group, and the creator becomes the owner of 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 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.

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*[9]. 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*[9], an official Peer-to-Peer communication framework provided by Apple. Grouper uses it to transfer data between two devices in a face-to-face way using a wireless LAN or bluetooth network.
- *Core Data*[8], an official (Object-Relational Mapping) ORM framework provided by Apple. *Core Data* provides generalized and automated solutions to common tasks associated with object life cycles and object graph management, including persistence. Grouper uses it to manage model layer objects.
- *Sync*[15], a synchronization framework for *Core Data* using JSON. Grouper uses it in the synchronization plugin now.
- *c-SSS*[17], an implementation of the secret sharing scheme.
- *AFNetworking*[1], a networking library in Objective-C. Grouper uses it to invoke the RESTful API provided by our Web services running on multiple untrusted servers.

4.2 Web Service

We have chosen implementing own Web service rather than using commercial general cloud storage 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[6] that is an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. We use the Spring MVC[11], a Web model-view-controller framework, to create our RESTful API, and Hibernate[12], an open source Java ORM framework, to save and operate objects in the Web service.

Our Web service includes three kinds of entities: *Group*, *User* and *Share* 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 *Share* entity saves a share generated with the secret sharing scheme, the time when a client uploads the share.

4.3 Application

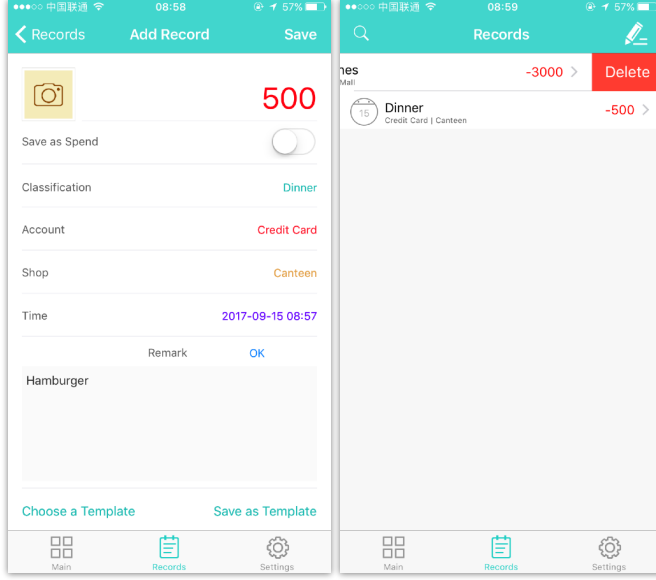
Using the Grouper framework, we have developed the following applications.

- *Account Book*, an iOS application in Objective-C, records the income and expenditure of a group.
- *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

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 |
| Test | iOS | Swift | 1 | 621 | 18 |



(a) Add a record.

(b) Delete a record.

Figure 4: Screenshots of demo application Account Book.

Grouper to share this record to other group members. Figure 4(b) shows the screenshot of deleting a record in Account Book. A user swipes 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 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.

Section 4.3 shows the implementation of two applications. Table 4 shows that 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 mobile applications using the Grouper framework provide sufficient performance for small groups

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 |

of people. Table 5 shows the hardware and software information of our experiment environment. 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.

In our benchmark application, the size of an object was 323 bytes. This object corresponds to a income or expenditure record of the application Account Book. When this object was updated in a node, the Grouper API generated a normal message which size was 656 bytes.

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

Figure 5 shows the processing time of uploading and downloading multiple messages. We divide the processing time into three parts: data synchronization, secret sharing and network. As the number of messages increased, the data synchronization 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 required about two times of processing times. Note that object synchronization is done in a background task asynchronously. While the Grouper framework is performing object synchronization, the user interface of a mobile application never freezes. A user is not aware of data synchronization.

We performed an 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 measured the processing time of uploading and downloading a single message.

Figure 6 shows the relationship between processing time and the parameters, where $s = n$, $0 < k < n$, $k = 3i$, $n = 3j + 3$ and $1 \leq i, j \leq 5$. 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, and sometimes decreased.

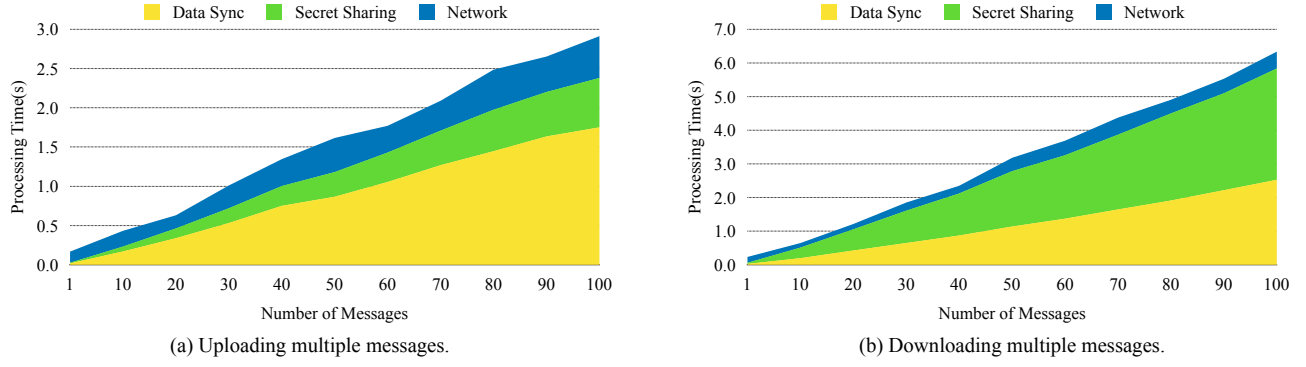


Figure 5: Processing time of uploading and downloading multiple messages.

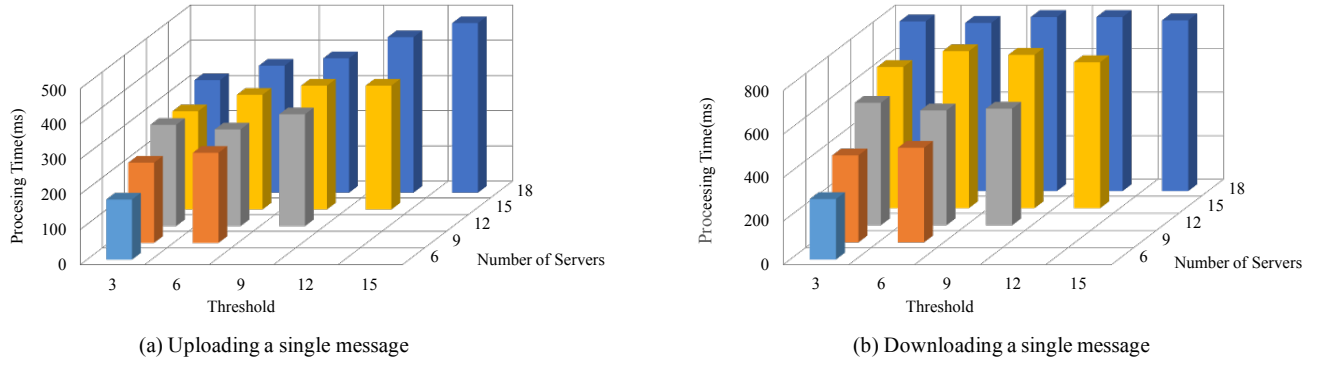


Figure 6: Processing time of uploading and downloading a single message with different scheme.

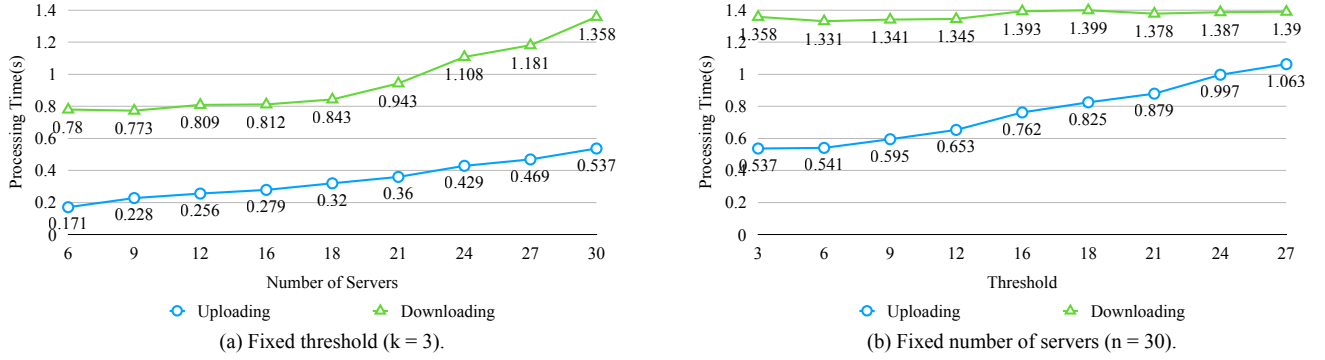


Figure 7: Processing time of uploading and downloading a single message with a constant k or n .

Figure 7(a) shows the relationship between processing time and the number of servers n , where $k = 3$ and $n = 2, 4, 6, \dots, 10$. As n increased, the processing time of uploading and downloading increase linearly. Figure 7(b) shows the relationship between processing time and the threshold k , where $n = 30$ and $k = 3, 6, 9, \dots, 27$. 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 number of messages. We use the parameters in Table 6.

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

$$NU_i = U_i + C \quad (1)$$

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

| Parameter | Explanation |
|-----------|---|
| D | Number of devices. |
| T | Time to live (TTL). |
| F | Average number of offline devices during the TTL. |
| L | Average offline time. |
| U_i | Average number of updated/deleted messages in the device i during the TTL. |
| UR_i | Average number of updated/deleted messages by resend messages in the device i during the TTL. |
| C | Average number of confirm messages in the device i during the TTL. |
| U | Total number of updated and deleted objects during the TTL. |
| NU_i | Number of uploading for device i during the TTL. |
| NU | Total number of uploading during the TTL. |
| ND_i | Number of downloading for device i during the TTL. |
| ND | Total number of downloading during the TTL. |

Total number of uploading is the sum NU of NU_i .

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

A device does not download the messages that have been uploaded by itself. To simplify equations, we suppose the device downloads those messages. Thus, the number of downloading for device i , ND_i , is the equal to total number of uploading NU .

$$ND_i = NU = U + C \cdot D \quad (3)$$

The total number of downloading ND is the sum of ND_i .

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

Because current cloud servers scale well according to the number of devices and messages, we discuss NU_i and UD_i in this subsection. Equation 1 shows that for the device i during the TTL, the order of uploading is $O(U)$. Equation 3 shows that the order of downloading is $O(U + D)$. As the number of updated and deleted objects increases, the total numbers of uploading and downloading increase linearly. As the number of devices increases, the number of confirm messages increases 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 become online and the devices that have the updated objects will resend update messages to the offline devices. To simplify equations, we suppose that all offline devices receive the updated messages after they sent resend messages. For the device i , the number of those update messages by the resend message, UR_i , is

$$UR_i = U_i \cdot \frac{L}{T} \cdot F \quad (5)$$

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

$$NU_i = U_i + C + U_i \cdot \frac{L}{T} \cdot F = U_i \cdot (1 + \frac{L}{T} \cdot F) + C \quad (6)$$

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

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

Equations 6 to 8 show that for the device i during the TTL, the order of uploading is $O(U \cdot \frac{L}{T} \cdot F + C)$, and the order of downloading is $O(U \cdot \frac{L}{T} \cdot F + C \cdot D)$. As the numbers of updated and deleted objects, the average number of offline devices during the TTL, and the average online time increase, the number of uploading and downloading for a single device increases linearly.

If we use a longer T , the numbers of messages, NU_i and ND_i , become smaller. However, this makes it easier for attackers to crack untrusted servers.

At last, we consider the following two cases with concrete parameters. In the two cases, there are ten users in a small group and everyone has a device. We set the TTL to one day and there are one hundred of normal messages created by the ten users in the group during the TTL. In addition, a device send only one confirm message during the TTL.

- **Case 1:** $D = 10$, $T = 1 \text{ day}$, $U_i = 100$, $C = 1$, $F = 0$.

In the case, there is no offline users in the group. Each device uploads 101 messages and downloads 1010 messages during the TTL.

- **Case 2:** $D = 10$, $T = 1 \text{ day}$, $L = 1 \text{ day}$, $U_i = 100$, $C = 1$, $F = 2$.

In this case, there are two offline users in the group and the average offline time is one days. Each device uploads 301 messages and downloads 3010 messages during the TTL.

5.4 Data Synchronization Libraries

The Grouper framework makes the synchronization module pluggable. We use Sync[15] framework, which provides a consistency model that the newest object wins in the synchronization plugin now. For example, if a record in the Account Book application has been modified by two users, the modification of one user will be lost. The Account Book application keeps the newest modification and put it into persistent store.

If an application needs other consistency model, the developer has to implement such synchronization plugin. For example, if a developer wants to use eventual consistency, he can choose Ensembles[4].

Note that the synchronization frameworks must provide the interfaces as described in Table 2. For this reason, Grouper does not support iOS synchronization frameworks for iCloud or Dropbox.

For example, Grouper does not support Core Data with iCloud[8] and CoreDataSync[14] now. These frameworks provide APIs for files while Grouper needs APIs for sending multicast messages. If a developer want to use them, he has to add an additional layer that provides the methods in Table 2.

6 RELATED WORK

Vanish is a self-destruction system[7] and 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 eliminates the key from the local storage. The key in Vanish is removed after a period of time, and the encrypted message becomes permanently unreadable. Vanish is implemented with OpenDHT[19] or VuzeDHT[10] which is controlled by a single maintainer. Thus, it is not strongly secure against some Sybil attacks[22]. 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[24] and SeDas[23]. SafeVanish prevents hopping attacks by extending the length range of key shares while SeDas exploits active storage networks instead of P2P nodes to maintain divided secret key. By extending SeDas, Zeng's group proposes CloudSky[25], a controllable data self-destruction system for untrusted cloud storage. In CloudSky, a user can control the surviving time of a message. CloudSky uses Attribute Bases Encryption (ABE) and allows user to define access control policies.

However, these proposals are not suitable for developing information sharing applications. They are suitable for exchanging immutable messages, while our target applications need to modify objects repeatedly after other devices receive 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[18] stores encrypted data on an untrusted server and decrypts this data only in the browsers of users. Application developers of Mylar use its API to encrypt regular (non-encrypted) Web applications. Mylar uses its browser extension to decrypt data on clients. Compared to Mylar using a single untrusted server, Grouper takes advantages of data redundancy provided by the secret sharing scheme by using multiple untrusted servers.

Sweets[13] 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 done 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[3] realizes the concept of *Cloud-of-Clouds*. DepSky keeps encrypted data in commercial storage services and do application logic in other individual servers. In Grouper, untrusted servers

undertake responsibility of temporarily data storage and message delivery. Mobile devices do application logic.

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

7 CONCLUSION

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

We have implemented Grouper's client's framework in Objective-C and Web service in Java. We have developed applications, *Account Book* and *Test* on the top of Grouper. Developing these applications shows that Grouper requires little development effort 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. We have a plan to use file-oriented APIs for exchanging messages among devices.

ACKNOWLEDGMENTS

The authors would like to thank Elvis Nunẽz for helping us to fix some Objective-C related issues of the Sync framework, and Bernard Paulus for fixing a bug of the implementation of the secret sharing scheme, c-SSS.

The authors would also like to thank University of Tsukuba for providing registration fee.

REFERENCES

- [1] AFNetworking. 2016. AFNetworking, a delightful networking framework for iOS, OS X, watchOS, and tvOS. (Mar 2016). Retrieved Oct 3, 2017 from <http://afnetworking.com>
- [2] Russ Allbery and Charles H. Lindsey. 2009. Netnews Architecture and Protocol. (Nov 2009). Retrieved Aug 19, 2017 from <https://tools.ietf.org/html/rfc5537>
- [3] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando Andr , and Paulo Sousa. 2013. DepSky: Dependable and secure storage in a Cloud-of-Clouds. In *ACM Transactions on Storage (TOS)*, Vol. 9. Article 12, 33 pages. <https://doi.org/10.1145/2535929>
- [4] The Mental Faculty B.V. 2017. Ensembles. (2017). Retrieved Sep 28, 2017 from <http://www.ensembles.io>
- [5] Ariel J Feldman, William P Zeller, Michael J Freedman, and Edward W Felten. 2010. SPORC: Group collaboration using untrusted cloud resources.. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, Vol. 10. 337–350.
- [6] The Apache Software Foundation. 2017. Tomcat. (2017). Retrieved Aug 19, 2017 from <http://tomcat.apache.org>

- [7] Roxana Geambasu, Tadayoshi Kohno, Amit A Levy, and Henry M Levy. 2009. Vanish: Increasing data privacy with self-destructing data. In *18th USENIX Security Symposium*. 300–315. <https://www.usenix.org/conference/usenixsecurity09/technical-sessions/presentation/vanish-increasing-data-privacy-self>
- [8] Apple Inc. 2017. Core Data Programming Guide, Guides and Sample Code. (2017). Retrieved Oct 3, 2017 from <https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CoreData/>
- [9] Apple Inc. 2017. MultipeerConnectivity. (2017). Retrieved Oct 3, 2017 from <https://developer.apple.com/documentation/multipeerconnectivity>
- [10] Azureus Software Inc. 2017. Azureus. (2017). Retrieved Aug 19, 2017 from <http://www.vuze.com>
- [11] Pivotal Software Inc. 2017. Spring. (2017). Retrieved Aug 19, 2017 from <http://spring.io>
- [12] Red Hat Software Inc. 2017. Hibernate. (2017). Retrieved Aug 19, 2017 from <http://hibernate.org>
- [13] Rongchang Lai and Yasushi Shinjo. 2016. Sweets: A decentralized social networking service application using data synchronization on mobile devices. In *12th EAI International Conference on Collaborative Computing: Networking, Applications and Worksharing*. 188–198.
- [14] No Thirst Software LLC. 2013. TICoreDataSync. (May 2013). Retrieved Sep 28, 2017 from <https://github.com/nothirst/TICoreDataSync>
- [15] Elvis Nunēz. 2017. Sync, modern Swift JSON synchronization to Core Data. (Sep 2017). Retrieved Oct 3, 2017 from <https://github.com/SyncDB/Sync>
- [16] Liao-Jun Pang and Yu-Min Wang. 2005. A new (t, n) multi-secret sharing scheme based on Shamir’s secret sharing. *Appl. Math. Comput.* 167, 2 (2005), 840–848.
- [17] Fletcher T. Penney. 2017. c-SSS, an implementation of Shamir’s Secret Sharing. (Jan 2017). Retrieved Oct 3, 2017 from <https://github.com/fletcher/c-sss>
- [18] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nikolai Zeldovich, and Hari Balakrishnan. 2014. Building Web applications on top of encrypted data using Mylar. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 157–172.
- [19] Sean Rhea, Brighton Godfrey, Brad Karp, John Kubiawicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. 2005. OpenDHT: a public DHT service and its uses. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '05)*. 73–84.
- [20] Guillaume Smith, Roksana Boreli, and Mohamed Ali Kaafar. 2013. A layered secret sharing scheme for automated profile sharing in OSN Groups. In *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*. 487–499.
- [21] Andrew S Tanenbaum and Maarten Van Steen. 2007. *Distributed systems: principles and paradigms*. Prentice-Hall.
- [22] Scott Wolchok, Owen S Hofmann, Nadia Heninger, Edward W Felten, J Alex Halderman, Christopher J Rossbach, Brent Waters, and Emmett Witchel. 2010. Defeating Vanish with low-cost Sybil attacks against large DHTs. In *17th Proceedings of the Network and Distributed System Security Symposium*.
- [23] Lingfang Zeng, Shibin Chen, Qingsong Wei, and Dan Feng. 2012. SeDas: A self-destructing data system based on active storage framework. In *Asia-Pacific Magnetic Recording Conference (APMRC)*. 1–8.
- [24] Lingfang Zeng, Zhan Shi, Shengjie Xu, and Dan Feng. 2010. Safevanish: An improved data self-destruction for protecting data privacy. In *IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*. 521–528.
- [25] Lingfang Zeng, Yang Wang, and Dan Feng. 2015. CloudSky: a controllable data self-destruction system for untrusted cloud storage networks. In *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 352–361.