Musubi: Middleware for Secure Mobile Collaborative Applications

Master’s Thesis

Willem Bult
Musubi: Middleware for Secure Mobile Collaborative Applications

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Willem Bult
born in Groningen, The Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

MobiSocial Computing Laboratory
Computer Science Department
Stanford University
Stanford, CA, United States
mobisocial.stanford.edu
Musubi: Middleware for Secure Mobile Collaborative Applications

Author: Willem Bult
Student id: 1179268
Email: w.bult@student.tudelft.nl

Abstract

As we work more and more while on the go, mobile collaborative applications will play an increasingly important role in our lives. We note that the conventional client/server model for these applications poses several shortcomings. The model is unintuitive and unnecessarily complex. Centralized servers further introduce cost, inflexibility, and concerns about privacy.

Musubi is a middleware for mobile collaborative applications based on a top-level peer-to-peer architecture and secure group communication. We demonstrate how a secure messenger application can be built on top of our system. Three diverse case studies in citizen journalism, cross-generational interaction, and computer science education are used to evaluate the middleware. We show that Musubi simplifies the development of collaborative applications. The created applications are transport agnostic and can provide a good user experience while using fully end-to-end encrypted communication.

Thesis Committee:

Chair: Dr Eelco Visser, Faculty EEMCS, TU Delft
University supervisor: Dr Alexandru Iosup, Faculty EEMCS, TU Delft
External supervisor: Prof. Monica S. Lam, CS Department, Stanford University
Committee Member: Prof. Geert-Jan P.M. Houben, Faculty EEMCS, TU Delft

The work in this thesis has been carried out at Stanford University. The research was financially supported by US National Science Foundation (NSF) Programmable Open Mobile Internet (POMI) 2020 Expedition Grant 0832820 and the Stanford MobiSocial Computing Laboratory.
Preface

Nearly ten years ago I decided to enroll at TU Delft as an undergrad. And today, this thesis marks not only the end of a masters program, but also of those many great years I had as a student at Delft. I am grateful for the opportunities I’ve had during this time to explore not only computer science, but also entrepreneurship, and indeed many exciting places around the world. I’ve been able to do so both within the context of TU Delft, for example with a bachelor project in the Philippines, and outside, as on several occasions I interrupted my academic journey to pursue one or another entrepreneurial adventure. Because of these intermezzos, many, not excluding myself, have often doubted whether this thesis would ever see the light of day. Nevertheless, dear reader, here we are. And I couldn’t have hoped for a better finale than the year that constitutes the work for this thesis.

Stanford University, the MobiSocial Computing Laboratory, and Silicon Valley in general have been, and continue to be, a tremendous source of inspiration. I remember what Monica Lam, who later became my supervisor, told me when I first met with her: “it’s all about impact.” Therefore, this thesis is not just about an interesting new system (although it is that, too). More importantly, it is about real-world problems that the system can solve. None of the projects described in this thesis were solo efforts. I want to express my sincerest thanks to Monica for giving me the opportunity to be part of the MobiSocial team. She was always full of enthusiasm and countless grand ideas. Ph.D. candidates Ben Dodson, T.J. Purtell, and Ian Vo have all been mentors in the world of grad school and it was an absolute pleasure to work with them on Musubi. Thanks to Daniela Steinsapir, the Dispatch team, particularly Kanak Biscuitwala and Mathias Lécuyer, for the opportunity to work together.

At TU Delft, I would like to thank the members of my thesis committee. First, to Eelco Visser for agreeing to supervise my graduation. He must have thought it would never end, because I was working on something new every few months. However unscripted all this work was, there was a common thread, demonstrated by this thesis. Thanks also go out to Alexandru Iosup for his feedback, particularly in the early stages. It was a challenge at times, but also an invaluable experience, to balance input from Monica, Eelco, and Alex, each with a different focus and domain of expertise. Finally, I would like to thank Geert-Jan Houben for agreeing to be part of my thesis committee.
To conclude I would like to thank the people who are further removed from my master’s thesis, but less so from me. The many great friends I have met, Henk Pijper, René Crapts, Jeremy Raes, and many others, have made my time in Delft priceless. Thijs de Vries and Eline van Beest, who not only are terrific cofounders, but also have always encouraged me to complete this degree. My friends and roommates in San Francisco who helped evaluate Musubi. My parents, for their trust and support during all of this. And last, but certainly not least, my girlfriend Agata, who has been a great support and a great help in reviewing preliminary versions of this thesis. Without all of you, this work would not exist.

Willem Bult
San Francisco, California
December 15, 2012
# Contents

**Preface**  iii  
**Contents**  v  
**List of Figures**  vii  

## 1 Introduction  
1.1 Conventional model  .................................................. 1  
1.2 Proposed model  ......................................................... 2  
1.3 Research goal and questions  ............................................ 2  
1.4 Contributions  ........................................................... 3  
1.5 Outline  ................................................................. 3  

## 2 Background  
2.1 Mobile devices  ........................................................... 5  
2.2 Mobile wireless networks  ............................................... 6  
2.3 Mobile collaborative applications  ...................................... 7  
2.4 Challenges in MoCoA development  ..................................... 8  
2.5 Client/server architecture  .............................................. 11  
2.6 Top-level peer-to-peer architecture  ................................... 12  

## 3 System overview  
3.1 Developer API  .......................................................... 14  
3.2 Identity based routing  .................................................... 17  
3.3 Identity based encryption  ............................................... 18  
3.4 Message format  .......................................................... 20  
3.5 Architecture  ............................................................ 21  
3.6 Sample use case: secure Messenger for iPhone  ...................... 27  
3.7 Evaluation  .............................................................. 31  
3.8 Discussion  .............................................................. 31
# CONTENTS

4 Dispatch 33
  4.1 Citizen journalism .................................................. 33
  4.2 Pocket switched networks .......................................... 35
  4.3 The Dispatch application ......................................... 36
  4.4 U.S. presidential election day .................................... 38
  4.5 Discussion .......................................................... 39

5 Migo and SocialKit-JS 41
  5.1 Computer science 101 ................................................. 41
  5.2 SocialKit-JS .......................................................... 42
  5.3 Migo study ........................................................... 45
  5.4 Discussion ........................................................... 47

6 Grandma's Trunk 49
  6.1 Cross-generational communication ................................ 49
  6.2 Application design .................................................. 50
  6.3 Architecture .......................................................... 53
  6.4 User study ............................................................. 55
  6.5 Discussion ............................................................. 56

7 Related work 57
  7.1 Serverless APIs ....................................................... 57
  7.2 Group communication ............................................... 58
  7.3 Social networking ..................................................... 59
  7.4 Routing ................................................................. 59
  7.5 Privacy protection .................................................... 60

8 Discussion and future work 61
  8.1 Intuitive API .......................................................... 61
  8.2 Frictionless security .................................................. 62
  8.3 Flexible routing ....................................................... 63
  8.4 Commodity infrastructure ......................................... 63
  8.5 Computer science education ...................................... 63

9 Conclusions 65
  9.1 Answers to research questions ..................................... 65
  9.2 Summary of contributions ......................................... 67

Bibliography 69

A Glossary 75

B Migo student applications 77
List of Figures

3.1 Musubi data model ............................................. 14
3.2 Musubi message format ........................................ 20
3.3 Musubi architecture ........................................... 22
3.4 Messenger welcome and settings screens ...................... 28
3.5 Messenger conversation start and list screens ................ 29
3.6 Messenger conversation screen ................................ 29
3.7 Messenger class diagram ....................................... 30
4.1 Dispatch system architecture .................................... 36
4.2 Passing messages locally with Dispatch ......................... 37
4.3 Published election reports with Dispatch ......................... 38
5.1 SocialKit-JS architecture ....................................... 43
6.1 Grandma’s Trunk photo sharing activity ......................... 51
6.2 Grandma’s Trunk story telling activity ......................... 51
6.3 Grandma’s Trunk drawing activity .............................. 51
6.4 Grandma’s Trunk architecture ................................. 53
Chapter 1

Introduction

Applications for mobile devices are becoming increasingly popular. Wireless connectivity to the Internet using technologies such as EDGE and 3G are now easily available to use with these devices in many places worldwide. An extensive ecosystem of mobile applications, commonly referred to as ‘apps’, for this environment has evolved. In this thesis we focus on a particular type of application that supports collaboration. Mobile collaborative applications (MoCoAs) allow users, often geographically distributed, to work together towards a common goal. One can think, for example, of a shared whiteboard application that allows users in different places in the world to brainstorm using their smartphones. Another example is an application that allows rescue workers to share information while on the scene in an emergency response situation [14]. Motivated by the quick growth of this ecosystem we want to assist the development of such collaborative mobile applications. We thus take an interest in these applications from a software engineering perspective.

1.1 Conventional model

MoCoAs can be classified as distributed systems whose development suffers from many of the classical challenges in distributed system design. The de facto architecture for mobile applications on the Internet today is the client/server model. Indeed, cloud computing is often used to solve many of the challenges facing the mobile environment, such as limited resources and reliability [18]. The introduction of a central server provides solutions for many of the challenges in communication, synchronization, and persistency. We realize, however, that this architecture also introduces a number of problems of its own.

First, we note that for application developers unfamiliar with distributed systems, the concept of a centralized party for collaborative applications seems rather unintuitive. A model where messages are exchanged directly between the collaborating parties is a more natural representation of the real world, and therefore more intuitive.

Secondly, the dependency on an application server makes the model inflexible as it imposes a dependency on connectivity to a central infrastructure that may not always be accessible. Many situations can be imagined, for example in deprived areas or crisis situations, where this infrastructure is indeed not available.
1. Introduction

Thirdly, centralized application servers are expensive to operate, and for many applications maintaining such a server may not be economically feasible. Applications can be imagined that can not be easily monetized, and thus operational costs should ideally be kept to a minimal.

Finally, there is a concern for privacy and data ownership with the use of central servers. If sensitive data is operated on by application servers, measures need to be taken to protect access to and ownership over that data. This is a non-trivial task for developers, and also potentially requires costly legal measures.

1.2 Proposed model

We propose a different application model that offers a solution to these issues. We combine the use of centralized infrastructure with a peer-to-peer (P2P) programming model in what we describe as a top-level P2P architecture. In our model, there is no application code running on a central server and, as such, applications are logically decentralized. We use messaging to allow the applications to communicate. We further realize that, because of intermittent connectivity in mobile environments, the communication should be asynchronous and failure tolerant. Centralized infrastructure may still be used for communication purposes, e.g. in the form of a message broker.

The proposed architecture can be used to provide solutions to the aforementioned problems. It provides an easier programming model, because the developer has to be concerned only with developing client-side code and can, seemingly, pass messages directly between the collaborating parties. The model is also more flexible, because it allows for the development of applications that run on any transport, whether it is a centralized infrastructure or ad-hoc network. Clearly, since there is no application server, the dependency on operating servers is eliminated, thereby reducing cost. In the case of using centralized infrastructure like the Internet there is still cost of data transport, but this transport can be managed more efficiently by network operators. Finally, when the transport layer merely passes data around without introspection, i.e. as a dumb pipe, our model allows for end-to-end encryption to effectively provide access control and privacy protection.

1.3 Research goal and questions

We realize that the proposed model inhibits many of the classical distributed system challenges – in communication, coordination, and collaboration – that centralized servers would otherwise solve. To help in these struggles of distributed systems, there is a need for middleware to provide easy abstractions. Middleware is used in software engineering to facilitate application development by abstracting concerns away from the application layer [1]. This leads to our main research goal:

*Develop a middleware to support the development of mobile collaborative applications based on secure group communication.*

To approach this goal we stated the following research questions.
1. What are the functional requirements of MoCoAs?

2. How can we provide frictionless secure group communication that is independent of any particular transport?

3. How can we ease the development of MoCoAs using an API that supports group communication?

4. Can our middleware meet the non-functional requirements of MoCoAs?

1.4 Contributions

We present the following contributions in this work:

- A list of functional requirements of MoCoAs
- The Musubi middleware architecture that addresses the functional requirements, along with a reference implementation and a sample case application: a secure Messenger application.
- SocialKit-JS, a Javascript library that exposes a Musubi API to web applications.
- Dispatch, an application to support citizen journalism in crisis areas.
- Migo, a project that uses Musubi as a tool to introduce students to computer science.
- Grandma’s Trunk, an application that enables cross-generational interaction.

We evaluate Musubi using three separate case studies that address the problems of MoCoA development. We focus on qualitative evaluation using real-world applications, rather than simulations and performance evaluations. We note that the success of a middleware is dependent on the success of its applications. We therefore want to verify that it is possible for applications built on Musubi to be successful in real-world situations.

1.5 Outline

This thesis continues to provide a background of the domain of mobile collaborative applications in chapter 2. We introduce the Musubi system and describe it in detail in chapter 3, along with a sample case of building a secure iPhone messenger application. We follow up with the three evaluation case studies Dispatch, Migo, and Grandma’s Trunk in chapters 4, 5, and 6. In chapter 7, we present an overview of related work. We discuss our research along with perspectives for future work in chapter 8. Finally, we present our conclusions in chapter 9.
Chapter 2

Background

Over the past years we have seen a vast increase in the use of mobile devices. The sales numbers of smartphones and tablets are forecasted to reach 821 million in 2012 [65]. More recently, Internet usage through mobile devices has also significantly increased. Gartner, world’s leading information technology research company, predicts that by 2013 mobile phones will overtake PCs as the most common device used to access the web worldwide [50].

As mobile devices are used increasingly to access the web, they are also used more often to communicate and collaborate with co-workers and friends. Collaboration happens using traditional e-mail and calendar applications, but the new ecosystem also creates new possibilities for mobile collaboration. In this chapter we will discuss the background of mobile devices and the wireless networks they operate on. We will continue to discuss the mobile collaborative applications and the challenges developers have to face. This provides a background against which the related work and our work should be seen.

2.1 Mobile devices

When designing middleware for mobile applications, we must take into consideration the constraints given by the mobile devices those applications run on. In general, we must consider that the spectrum of available mobile devices is diverse and the specifications of these devices are indeed largely heterogeneous. We consider four different constraints: processing power, storage, battery life, and sandboxing.

First, mobile phones are typically equipped with more limited processors and memory units than desktop computers. We can thus not run the same type of computation on mobile devices that we can on desktop or server computers. However, that gap is closing and it will not be long before the energy efficient processors that go into mobile phone are on a par with desktop computers [58].

Secondly, the storage space on mobile devices is limited. Typical smartphones these days are equipped with storage capacity of several tens of gigabytes, which is enough to hold ephemeral data like work in progress, pictures, and movies. However, they are not well suited to store large amounts of data. Given the usage pattern of mobility, the high
Potential for loss must also be considered, which make mobile phones unsuitable to serve as a single storage for important data.

Thirdly, mobile devices operate on battery power. The battery life differs per device. Tablets typically have longer lasting power supply than mobile phones. As a result, the devices become unavailable once in a while as power runs out. Another consequence to notice is that consuming resources limits the lifetime of the device. Running complex computations, for example, should ideally be limited. Care for computational complexity must thus be taken in the design of our middleware and our applications.

Finally, many operating systems on mobile devices execute applications in a security environment called a sandbox. In this environment, the resources the operating system provides to applications are limited. One big limitation to notice in the context of middleware is the restriction on inter-process communication (IPC). A notable example is the Apple iOS ¹ operating system for iPhone and iPad. IPC on Apple iOS is limited to launching other applications through registered URL handlers with an arbitrary amount of encoded data in the URL. An application \( \text{a} \) would register a handle for a specific URL scheme, and afterwards application \( \text{b} \) can open application \( \text{a} \) by opening a URL with that particular scheme, amended with any data that is to be passed to the started process of application \( \text{a} \). As such, the communication is uni-directional, and passing back a result for an operation to process for application \( \text{b} \) is not possible. If we want to enable communication between multiple applications on Apple iOS, they will have to all run within one ‘master’ application that runs within the sandbox. A way to accomplish this is to run HTML applications in a browser embedded in the master application.

### 2.2 Mobile wireless networks

In middleware for applications in wired environments, the assumption is often made that there exists a stable and pervasive connection between two communication end points on a network, or that interruptions are rare. The communication paradigms are therefore commonly synchronous.

In contrast, in mobile wireless networks, such connectivity may not exist at all times and can in fact be classified as intermittent. This problem is mainly caused by the fact that users are mobile and move around in geographical space so that they frequently and unpredictably fall in and out of network connectivity. Situations will certainly exist in which delay tolerant asynchronous traffic is the only form of traffic that could be supported [48]. Disconnections are the rule, rather than the exception. This is not only the case in typically imagined mobile networks with regular connectivity, but can also be extended to situations where there may be no fixed communication infrastructure at all. We might think, for example, of underdeveloped regions or remote places, where perhaps a timeshared satellite connection provides the only available link to the Internet. We can also imagine situations of nomadic networks with periodically isolated clouds of hosts, where traffic passes from source node through other nodes until eventually one of them becomes connected to an end point. We will adopt the term Pocket Switched Network (PSN) here to describe these networks [34].

We need our applications to be able to expect and handle link and device failures gracefully, e.g. without blocking the user interface. To accomplish this we need to adopt a programming model that is based on asynchronous communication methods. We note that a way to achieve this is to structure our applications so that they are based on events and we have a way to asynchronously propagate these events in a failsafe manner to other devices.

Another important practical restriction of mobile networks, such as HDSPA or 4G, is that direct connections between peers are often not allowed. That is, mobile network operators often give out IP addresses that are not routable through the internet, or block incoming connections to those IP addresses. This limits the possibilities in terms of data dissemination between users on a mobile network. It implies that we either have to use alternative technologies, such as local WiFi or Bluetooth networks, or use a central reachable communication facility on the Internet to provide communication.

2.3 Mobile collaborative applications

The applications we are considering are classified as Computer Supported Cooperative Work (CSCW) [26]. The purpose is to allow multiple users to work together towards a shared goal. The notion has been well established in research in the domain of groupware that collaboration is dependent on coordination and communication [21] [69]. Coordination, in turn, is described as the integration and harmonious adjustment of individual work efforts toward the accomplishment of a larger goal [59]. We will further illustrate the domain with three typical, although imaginary, examples of applications we consider mobile collaborative applications (MoCoAs).

CrowdPhoto is an application for smartphones that allows groups of users to instantly share the photos they take at events. Users are paired up and added groups automatically based on geographical proximity. Think for example of a wedding. The guests all start CrowdPhoto on their smartphone and, through the use of Bluetooth or a local WiFi network, the applications discover each other and automatically form a group. Everybody takes photos using their camera and the photos are automatically shared with everyone in the group. This way, the common laborious process of exchanging photos after events or other group activities is eliminated. The individual work efforts here are the pictures each user takes. The coordination is the distribution and collection of the pictures among all users in a group.

ShopShare allows a group of people to maintain a shared shopping list using smartphones or tablets. It can be used, for example, in families to indicate what household items must be restocked. Users organize in groups themselves. One person starts the group and invites the others to it using their e-mail address. Everyone in the group then sees a shared list that everyone can add items to and remove items from. When one of the users goes for the routine shopping trip, they can see at once what is needed by everyone in the household. The individual work efforts are the registration of needed items, and the coordination is in maintaining a shared consistent list, free of duplicates.

DrawingBoard is a whiteboard application for tablet devices that allows distributed teams to engage in brainstorm sessions on a shared digital workspace. Users form groups...
using e-mail addresses like in ShopShare. Afterwards, they will all see a white drawing canvas on their tablet. Users can draw lines or add text to the canvas. The changes are sent to all other users of the group, who will see their canvas updated in near real-time. The changes are merged so that every user sees the same result. The individual efforts are the lines and text that users add, and the coordination is the merging of the changes onto a consistent view of the canvas.

2.4 Challenges in MoCoA development

MoCoAs are true distributed systems that are sufficiently complex for middleware to have a significant role in reducing development effort. We present here a set of challenges that developers face in the development of such applications. This is not an attempt at an exhaustive list of MoCoA development concerns, but it serves as our guideline for the development of MoCoA middleware. Previous work has also addressed the issues in collaborative application development. The notion that collaboration is dependent on coordination and communication has been well established, since early works in the domain of groupware [21].

The challenges in collaboration are identity management, group formation, and shared data consistency. On the other hand, communication has to enable group address expansion, reliable message transport, and security and privacy. It must be noted, though, that the separation of concerns is not always clear. Group formation, for example, is primarily a coordination concern, but also relates to communication as profiles are advertised to other users in some way. Data persistency does not relate to collaboration and is of more individual interest to users.

In [31] Herskovic et al. also propose a list of requirements for mobile collaborative applications. They identify flexibility, protection, communication, heterogeneity, networking, awareness, and information support as the main categories of requirements. Our list is significantly different, although certain parallels can be drawn between them. Our concept of group formation, for example, relates to flexibility as it addresses peer discovery and dynamic groups. We address the requirement of protection under security and privacy, and communication is captured collectively under group address expansion and reliable message transport. Our concern for shared data consistency, finally, relates to the need for information support in [31].

2.4.1 Identity management

Users will typically need to identify themselves with the application, usually by maintaining a user profile. In the first place, this identity information can be used for users to discover each other. The profile could for example contain previously known identities, such as a name or e-mail address. It could also contain interest information, which could be used for users to find each other. The user profile gives people, when collaborating in a group, information about the other people participating in the group.

Middleware can offer functionality for users to manage their user profile, to distribute this profile to other users, and to maintain a list of profiles of other users on the network.
This can be regarded as a manifestation of the shared data consistency challenge, which we discuss below. Managed identities can also be used for identity verification. A form of key exchange and encryption could be used to sign messages, so that recipients can verify the origin of a message.

### 2.4.2 Group formation

For users to work together, they need to organize themselves in groups. Before they can do so, they have to establish contact, i.e. they need to ‘find each other’. This can be done, for example, using previously known identifiers, such as names or e-mail addresses, that users entered into their user profile. The ShopShare and DrawingBoard examples in the previous section adopt this mechanism. Another way to let users discover each other is by geographical proximity, which is the approach taken in the CrowdPhoto example in the previous section. Yet another discovery method is that of shared interest, where users are matched up based on interests they disclosed in their user profile. Such alternative discovery methods can lead to interesting opportunities for serendipitous interaction between strangers.

After users have established contact, they can form groups, or sessions. A distinction can be made between groups and sessions, where groups can start multiple sessions. To simplify the discussion we will consider a group analogous to a session so that every new session forms a new group. Of course, two different groups can have the same set of members. One of the users typically creates the groups and either adds other users to it, or invites them to join, or advertises the group so that others may join on their own accord. Different access levels to groups can thus be distinguished. Groups are further often dynamic, i.e. users may choose to leave a group. We assume that groups are durable. That is, groups survive the disconnection of its members from the network.

### 2.4.3 Group address expansion

To collaborate in a group, the users of the group clearly need to be able to communicate with each other. Hence, we need some form of group communication. The users all have a physical address on the network, but we would prefer our applications not to have to worry about that level of detail about the network. In fact, it can be a privacy violation if this information was exposed to applications. The use of logical addresses for users is preferred, perhaps in the form of an e-mail address or another unique identifier. It is more desirable, even, for the application to be able to address messages directly to the group, using its name or identifier. This challenge is referred to as group address expansion. The group address is expanded by the system into a membership list with a mapping to an address for every user. The membership list is dynamic, because members can join and leave a group.

### 2.4.4 Reliable message transport

Because collaboration depends on communication, we need a reliable communication facility between users. We further note that, due to the failure-prone environment of intermittent connectivity in mobile networks, measures will need to be taken to improve reliability. Situations can occur in which we can not be sure of the successful delivery of a message to
all members of a group. Reliability can be increased by introducing replication, i.e. re-
sending messages when they are not acknowledged. In this case, the receiving end must
eliminate duplicates that may result from unnecessary resends. In the absence of device
failures, exactly-once reliability is desired. However, it must be noted that devices may be-
come permanently unavailable, in which case at-most-once delivery is the best that can be
achieved.

At the application layer, we do not want to be concerned with message send failures
and acknowledgment schemes. It is desirable for middleware to provide the abstraction of
a reliable group communication channel. We also want our transport protocol to eliminate
any duplicates that may arrive due to resending. We realize that it is possible for devices
to disconnect and never regain connectivity. In that case we may want the application to be
notified after a certain time-out that delivery was unsuccessful.

2.4.5 Security and privacy

Applications may have certain security requirements. Information that is exchanged may
only be accessible to a group of people. Measures need to be taken so that the information
exchanged in the collaboration only reaches those who are allowed to see it. Middleware
should thus offer and enforce a form of an access control list (ACL). We also need to
make sure that nobody else can intercept messages from, or send or inject messages to, the
participating users. In other words, we need to protect against man-in-the-middle (MITM)
attacks. Sender verification is also required to effectively ensure access control and protect
against MITM. There is often a trade-off between the level of security and the level of
flexibility and performance a system can offer.

2.4.6 Shared data consistency

We already noted that collaboration also depends on coordination. Consistency of shared
data is a clear occurrence of the coordination problem. As said before, coordination is
described as the integration and harmonious adjustment of individual work efforts toward
the accomplishment of a larger goal [59]. The individual work efforts here are actions
that users perform on shared data. They are communicated as events amongst the users in a
group. Examples are a photo that is taken in the CrowdPhoto example, or a line that is drawn
in the DrawingBoard example application. These individual events need to be integrated in
the shared data towards the shared goal.

Unreliable connectivity in mobile environments poses a challenge here. Processes can
lose and regain connectivity, frequently and unpredictably, leading to situations where con-
flicts can occur. Imagine a situation where two users both send an event, e.g. make a change,
in logical concurrency, i.e. before having seen each others’ changes. There needs to be a
way to reach consensus about which message to handle first, in order to reach a consistent
end result at both parties. That is, all users need to have a consistent view of the shared data.
2.4.7 Data persistency

Persistency relates to the storage of data so that it survives process termination. The common goal that users work towards to in collaborative applications is represented in a shared data structure. This data needs to be persistently stored. Middleware could offer storage facilities or, alternatively, it could offer export functionality so that content can be stored elsewhere. Storage could either be offered on the mobile devices or in a centralized location, i.e. on a server. Persistency also relates to the durability of sessions, i.e. whether or not our groups will survive process termination. A centralized storage for groups, or sessions, could make them even more durable so that they not only survive process termination, but also device loss and replacement.

2.5 Client/server architecture

Typically mobile applications that work over the Internet adopt a client/server model. A web server typically contains business logic and supports persistent storage. In this model, the server enables collaboration by taking the role of a ‘middle-man’ and allowing multiple clients to connect and communicate through the server. The server will typically maintain a list of the users in the system, and offer a service for users to discover each other. It can also offer services to start sessions and invite others to it, thereby solving group formation and group address expansion.

The clients communicate their individual work efforts to the server. The server holds the shared data and is responsible for integrating the clients actions and maintaining its consistency. The order in which messages arrive at the server can be used to reach consensus. The server takes the role of a referee to decide which actions will be accepted. As such, it takes care of persistency and shared data consistency. The integrated actions are communicated to the other users in a group, usually by frequent pulling by the clients, providing a reliable message transport. The server will typically also maintain a list of the active users in the system, and offer a service for users to discover each other, thereby solving the group formation problem. Security can be provided through well established technologies to create secure links, like HTTPS.

The advantage of using a client/server architecture is obviously that it readily offers solutions for many of the challenges we have identified in MoCoA development. However, there are also certain important disadvantages.

First, from a software engineering standpoint, the most important is perhaps an unintuitive programming model. Because of the use of application code on the server, the control requires a model of central coordination. This model does not correspond well to the basic real world scenario of collaboration, where in principle users in the groups are all peers that communicate and coordinate among each other.

Secondly, setting up and running application-specific centralized infrastructure does not come without cost. Applications can be imagined that are hard to monetize, or that, while offering significant benefit to users, offer limited value to an organization to support. In these cases, the operational costs of a centralized server architecture may hinder innovation. In any case, reducing operational cost for these systems in general seems a worthy effort.
Thirdly, the central server also introduces availability concerns. A dependency on a central server introduces a single point of failure, i.e. if the server becomes unavailable the application becomes unusable. The server may also become a bottleneck as usage of the application increases. Although large scale client/server systems exist with millions of daily users, scaling these architectures is a non-trivial task.

Finally, as the server manages the shared data, the server must be controlled by a trusted party to provide adequate privacy protection. When the server is part of a generic middleware that is provided by a party other than the application developer, privacy protection and data ownership become concerns. Note, for example, applications that build on the Facebook platform. Although Facebook does not provide a full application platform and only offers certain social features, user interaction data of the application will indeed be recorded and monetized by Facebook.

2.6 Top-level peer-to-peer architecture

We propose the adoption of an alternative architecture here: that of top-level peer-to-peer (P2P) networking. We describe this as a hybrid form between P2P and client-server architectures. It is characterized by logical decentralization. With that, we mean that there is no application-specific logic placed on centralized components. Centralized infrastructure, i.e. the Internet, is still used in this architecture to offer communication among peers. By using this communication infrastructure the solution overcomes many of the restrictions of pure P2P networks, in terms of reliability and range of communication. This especially makes the architecture applicable to mobile networks, where often direct connections between devices through mobile networks like HDSPA and 4G are not possible.

In our proposed system, applications rely on a communication infrastructure to provide a reliable message transport, group formation and group addressing functionality. Effectively, a centralized server becomes a semi-dumb pipe that distributes messages amongst users in a group. It, however, still provides user discovery features and a mapping between group addresses and user addresses. Coordination, in contrast, happens on the client side because it usually requires specific application knowledge.

The top-level P2P model addresses the disadvantages of the client/server model we pointed out in the previous section. It allows for a more intuitive programming model of direct interaction between peers, which corresponds closely to the real world. It also reduces the cost of running a central server. Centralized infrastructure is still used to support Internet-wide communication, but the servers are generic as opposed to application-specific. These do not have to be maintained by the organization offering the application, but can be run efficiently and economically as a generally available service by network operators. The model is further more flexible, as it does not depend on the central network per se, and also allows applications to use communication through pocket switched networks. Privacy protection is usually a concern in P2P networks, but can be enforced by the client in this model. Because the routing infrastructure does not need insight into the data, end-to-end encryption can be employed to effectively protect data from man-in-the-middle (MITM) attacks.
Chapter 3

System overview

In this chapter we describe the Musubi middleware, specifically in the context of MoCoA development. Musubi is mobile middleware for collaborative applications. The platform enables frictionless, flexible, and trusted communication. First, it is frictionless because no setup is required by users. Secondly, it is flexible because users can adopt any identity they want and it is independent of any particular data transport. Finally, it is trusted because all data are encrypted and privacy is protected.

At the time we started this work a preliminary version of Musubi for the Android platform was available. That system explored the concept of disintermediated feeds through public key exchange [19]. Here we build on the lessons from that work. Our reference implementation, in contrast, does not use key exchange and is built for the Apple iOS platform. The iOS platform was chosen because it is a more restrictive platform that forces applications to run inside a security sandbox. Testing the model on the more restrictive platform allowed us to verify real-world applicability more effectively. It was also, at the start of this work, the most popular mobile platform in the United States [22], allowing us to more easily conduct user studies on it.

The foundation of the system described here was designed in a collaborative effort between the author and other members of the Stanford MobiSocial Computing Laboratory. The fundamental concepts have also been applied to create an Egocentric Social Platform (ESP) for Android [53]. ESP is compatible with the Musubi middleware described here.

First, we shortly present the application model and explain how it implements communication, collaboration, and security features. Next, we explain in more detail how we provide a secure solution to challenges in MoCoA development, i.e. by identity-based routing and identity-based encryption. We also describe implementation details of the message format. Then, we precisely present the architecture of the Musubi platform and the interaction between different system components. Finally we show a sample use case, for developing a secure messenger for the iPhone.

1http://source.android.com/
3. System overview

3.1 Developer API

We will start the description of our middleware by showing how developers can use it to build MoCoAs using the API. The API consists of a data model and control classes. We will describe both, starting with the primitives provided by the middleware and moving on thereafter to the control classes. This will provide a clear frame of reference for the rest of the more detailed discussion about the functionality exposed by the API. Furthermore, the classes that form our model are not only used in the API exposed to application developers, but are indeed reused throughout the client side of the middleware. The model shown here is a significant simplification of the actual classes used in the reference implementation. For the purpose of our discussion, this simplified model suffices and will make the system more comprehensible to the reader.

3.1.1 Model

Figure 3.1 shows the data model of Musubi. The classes are used throughout the client side of the middleware and exposed through the API to the application layer. Communication in Musubi is centered around a concept of feeds. A Feed is the basic communication channel for a group of users, and it contains a dynamic list of members. The members of a feed in Musubi are instances of Identity. Feeds can be used to represent a session to enable collaboration in a group. A feed for a group of people allows those people to exchange
messages directly and privately. Multiple feeds with the same member list can exist, for instance to support multiple topical communication channels.

The model also contains a `Device` class. This is used to indicate the origin of a message, since multiple devices per identity are supported. The `App` class indicates the application a message originates from. The `Account` class represents an account for an owned identity at the identity provider. Accounts, identities, and identity providers are described in more detail in section 3.2. Accounts, identities and feeds are the primitives that provide the basis for group formation and group addressing.

To collaborate, users of the application communicate their individual work efforts to each other. Data is exchanged between clients using the `Obj` class in Musubi. `Obj` is a special class that Musubi uses to encapsulate data that is communicated in feeds. An `obj` contains a type, a set of an arbitrary number of non-binary key/value pairs, and a binary data field. An obj can be ‘posted’ to a feed, which disseminates it to all the members in the feed. `Obj` also defines how it should be encoded, i.e. serialized, for transport. Certain objects trigger some action in the system upon receipt. Therefore, the `Obj` class also defines how it should be processed when it is received.

Applications extend `Obj` to define their own data types. Worth mentioning is that the Musubi API provides several ready to use types, such as `StatusObj`, `PictureObj`, and `FileObj`. There are also `obj` types for internal use by Musubi: `DeleteObj`, `ProfileObj`, `JoinRequestObj`, and `IntroductionObj`. `ProfileObj` provides support for users to communicate their identity profiles, thereby offering identity management. Custom `obj` types need to be registered with the Musubi middleware, using their type, so that Musubi knows which class to use when it encounters the `obj` type.

One of the most important features of the Musubi architecture is its integrated security mechanism that provides a robust protection of communication. Objects are formatted by the application and encoded by Musubi into encrypted and signed messages. These are represented by the `EncodedMessage` class. To support the encryption chain, private encryption and signature keys are stored in `UserKey`. Secrets are maintained for every other identity in the system to support our encryption mechanism and are cached in `Secret`. The use of these classes is explained in our discussions on encryption and the message pipeline in sections 3.3 and 3.5.4.

All classes in the model extend the `NSManagedObject` class of the Apple iOS Core Data framework. They adapt a form of the ActiveRecord design pattern and, as such, the instances of these classes are all persisted objects.

3.1.2 Control

The API provides a number of classes that offer functionality useful for building Mo-CoAs. Manager classes control the primitives defined in the data model. Most importantly `IdentityManager`, `FeedManager`, and `ObjManager` manage identities, feeds, and objects in the database. Through the managers, an application can create, retrieve, and delete en-
3. **SYSTEM OVERVIEW**

Updates to the data objects are reflected in the database, through the ActiveRecord pattern [25]. Below we describe the important facilities Musubi provides to applications.

### IdentityManager

- `createIdentity(type, id, name)`: Creates an identity (i.e. add a contact).
- `getOwnedIdentities()`: Retrieves the list of our own identities.
- `getIdentities()`: Retrieves the list of all identities (i.e. contacts).
- `getIdentityById(id)`: Retrieves a specific identity.

### FeedManager

- `createFeedWithMembers(identities)`: Creates a feed with the given identities as members.
- `attachIdentityToFeed(identity, feed)`: Adds an identity to a feed.
- `getIdentitiesInFeed(feed)`: Retrieves the members for a feed.
- `getOwnedIdentityForFeed(feed)`: Retrieves our own identity that is a member in a feed.
- `getFeedsWithIdentity(identity)`: Retrieves the feeds that an identity is a member of.
- `deleteFeedAndObjs(feed)`: Deletes a feed and the obis in it.

### ObjManager

- `getRenderableObjsInFeed(feed, beforeDate, afterDate, limit)`: Retrieves a selection of objs for a feed within a given range.
- `getLatestChildForParent(parentObj)`: Retrieves an obj’s latest child obj.
- `getLatestObjOfTypeInFeed(feed, objType, beforeDate, afterDate)`: Retrieves the latest obj of a given type for a feed within a given range.

### ObjHelper

- `sendObjToFeed(obj, feed, fromIdentity, fromApp)`: Sends an obj to a feed.
- `registerObjClass(objClass, type)`: Registers an application-specific Obj with Musubi.
3.2 Identity based routing

To create feeds and send and receive messages to each other, the first step is for users to discover each other. That is, they need to learn about the addresses of the message recipients. In other words, we need to support identity management and peer discovery. We note that there are multiple ways of doing this. One way is based on commonly used unique identifiers, such as a user id, a user name, or an email address. Typical systems allow users to create a profile and assign themselves a unique username. Other users can then find the user by their username. Other ways of peer discovery are, for example, based on geographical proximity, or advertised shared interests. In this section we present Identity Based Routing which enables frictionless interaction by leveraging existing identities and social graphs to route messages.

3.2.1 Frictionless interaction

We notice that, while calling someone on the phone or sending someone a text message does not require the recipient to join any proprietary network, this same frictionless interaction does not exist for most social applications. Typically users will have to sign up for a service, agree to its terms, and create a user profile or at least establish an identity to be able to use an application. We want to leverage existing identities, like phone numbers and email addresses, to enable the same kind of frictionless interaction of text messaging for MoCoAs.

A second observation is that to enable frictionless interaction, it has to be very easy for users to find the people they want to communicate with. In other words, we want quick access to a social graph. To achieve this, at least for previously existing friends, we want to leverage the existing contacts in the address book on the phone. In present day, many people have readily invested a significant amount of time and energy to create profiles on Online Social Networks (OSNs) and to link to their friends to build a social graph. We want to leverage this rich existing source of data as well.

3.2.2 Routing to existing identities

In line with our goal to make interaction flexible and frictionless, we apply Identity Based Routing (IBR). IBR allows us to address messages to any unique identifier, or identity, for a person. An identity can, for example, be a phone number, an email address, or a user id on some third-party OSN, like Facebook ³ or Twitter ⁴. Our system uses a one-way identity hash function over the identity type and a string representation of the identity, id_hash_fn(type, id), for the full address of an identity. We call the organization giving out the identities an identity provider. This way, our system allows users to adopt whichever address they prefer to receive messages on, thereby achieving flexibility. Equally, it also allows users to send messages from whichever identity they wish. The system can thus support multiple personas, e.g. one corporate and one private, on single device. Support

³http://www.facebook.com/
⁴http://www.twitter.com/
for multiple devices for a single identity also becomes possible. Because the messages are routed to an identity, it will automatically be sent to all devices registered for that identity.

IBR further allows us to leverage existing contacts in the address book and on OSNs by reusing the existing identifiers as addresses. Because our design is decentralized, the contact list will only be stored on the device. Though OSNs often have service terms that disallow importing data to other services, we note that a user should have the right to import their own relationships into their own device for their own personal interaction. Note that the social graph in our system is thus distributed between the devices of all users. Group formation now becomes as simple as picking contacts from the consolidated list of contacts on the phone and creating a feed with those contacts as members. There is no need for either party to create a new account. We will discuss the specifics of message routing to identities in more detail when we discuss the Message Router component later.

3.3 Identity based encryption

An important aspect of the system is inherent technological privacy protection. We want to support trusted, private and secure communication between groups of people. Yet, we still want to maintain frictionless interaction. We want to hold the criterion that the first point of contact can be the first message exchanged and no prior setup is required. We note that traditional forms of secure communication, via cryptography based on public key exchange, introduce too much friction. We do not want users to have to exchange keys before they send their first message. We therefore, next to Identity Based Routing, also adopt Identity Based Encryption (IBE). Using IBE, a string representing the identity of the recipient can be used to generate a public key for encryption. That is, IBE allows users to encrypt messages to each other without exchanging public keys beforehand.

Identity Based Cryptography was proposed by Shamir [56] as a concept of cryptography where the public key for an individual is their human readable identifier. Boneh and Franklin [7] discovered a method to perform the proposed techniques using the Weil pairing. Hess showed in [32] how a signature scheme can be enabled that does not require access to the trusted party to validate messages.

3.3.1 Essentials

Asymmetric key cryptography exploits the fact that the intended recipient of a message holds a private key that no other untrusted parties, including the sender, have access to. The encryption technique uses a public key that is mathematically related to the private key, such that decryption using the private key is efficient, but without it is computationally effectively impossible. When this private key is provided by a trusted authority only after a recipient proves their identity, we can rest assured that the message will not be decrypted by anyone other than the intended recipient.

In IBE, a conversation key pair \((\text{ckey}_{\text{raw}}, \text{ckey}_{\text{enc}})\) can be generated for any identity using a well-known mathematical function with as input an identity string \(\text{uid}\), a set of public parameters \(\text{pp}\), and a validity timeframe \(\text{tf}\). Here \(\text{ckey}_{\text{enc}}\) is an encrypted version of private \(\text{ckey}_{\text{raw}}\), such that it can be easily decrypted using a specific private user key.
ukey related to uid. The user key ukey can be generated using a master key mk that is associated with the public parameters pp. The identity provider holds mk and uses it with uid, pp, and tf to create ukey for identity uid. The user key ukey is provided to the legitimate owner of the identity uid to decrypt conversation keys for its identity.

IBE further also enables a signature scheme where similar to the private user key generation a signature key skey is generated by the identity provider. The verification of a signature made using this skey can be done again with just pp, tf, and uid. Communication with the identity provider is thus not necessary.

3.3.2 Identity Key Server

Ideally, the identity provider thus acts as the trusted authority that grants the right to recipients to read messages addressed to them. This, however, requires the cooperation of identity providers, the organizations giving out the identities. That is, they have to support a standard way of obtaining IBE private user and signature keys. This poses a big obstacle for the adoption, and therefore benefit, of our system.

We take a pragmatic approach and supply an intermediate Identity Key Server that supplies IBE keys and delegates authentication of several popular real world identity types to their respective providers. To do this, one of two main methods are used. First, we can use an authentication scheme that is supported by the specific identity provider, such as the exchange of OAuth tokens [29]. Secondly, a secret can be sent over the communication channel that is associated with the identity, such as email or SMS. We will describe the specifics of this flow later when we discuss the Identity Key Server.

We recognize that in many countries laws exist that forces providers of encryption communication to make this communication accessible on request to law enforcement entities. In our case, identity providers can be approached to disclose identity credentials that can then be used to obtain the IBE keys through our Identity Key Server.

3.3.3 Key renewal and revocation

We want to protect users against compromised keys, for example after device loss. A limited validity timeframe is assigned to keys to limit the use of stolen keys. After expiration of the timeframe, a new user key has to be acquired, thereby preventing the use of compromised keys beyond that period. Messages encrypted for a later timeframe can not be decrypted with private keys for an earlier timeframe. It also limits the validity of signatures to the specific timeframe. We choose a phased validity duration of up to 30 days since issuance time. The exact date of expiration can be calculated using a well-known function over the hash of the identity. Clients thus implicitly know what validity timeframe to use for the encryption. It also ensures that all keys in the system expire in an evenly distributed fashion which balances load from key update requests.

To further protect against device loss we want to allow the explicit revocation of keys. Another scenario that requires key revocation is when a corporation wants to revoke access to an identity after an employee leaves the company. We implement two primary revocation
methods, one with high effectivity but longer delay, and one that is more immediate but has limited reach.

First, since the identity provider is the primary authority that verifies identity ownership, we periodically verify with the identity provider if the identity is still valid. We can, for example, verify the validity of a cached OAuth token. A user can revoke the access token at the identity provider, at which point the client would destroy the private key. This method has a delay of the verification interval, but effectively revokes usage of the key.

Secondly, we allow users to request new signature keys with a newer timeframe at any point. When clients receive messages from other identities with a signature key that is newer than the last known one, the previous one is deleted. This way, users can send an invalidation message to all contacts with a new key, at which point the old key is revoked at those contacts. The revocation is effective as soon as the contacts receive the invalidation message, hence limiting delay, but is only effective at the known contacts.

3.4 Message format

All messages in Musubi contain an encrypted payload and a header that contains routing information. The payload is encrypted to each recipient in a feed individually. The current member list of a feed thus functions as the implicit access control list (ACL). When a member is added to a feed, subsequent messages will be encrypted to that identity as well. Prior messages to the feed can not be decrypted by the the new identity. When an identity
Architecture

is removed from a feed, subsequent messages will no longer be encrypted to this identity, and will thus be inaccessible to the removed member.

Figure 3.2 shows the format for messages in the Musubi message. The message header contains hashes of the destination identities, according to the IBR routing scheme, using the one-way id_hash_fn function. Because IBE operations are computationally expensive, we encrypt the payload of the message using a symmetric AES message key mkey that is randomly generated for every message. The message key is subsequently encrypted for each individual recipient using an IBE conversation key, resulting in mkey_enc. The IBE conversation keys have a validity timeframe of 30 days and are cached, again to save computational cost. The encrypted version of the conversation key ckey_enc as well as mkey_enc are included the message header together with the recipient identity hashes. The overhead of the message header is 102 bytes plus 252 bytes per recipient.

We maintain a sequence number for every known identity. A sender includes the sequence number it has for every recipient in the message header. This way, when messages are received we can detect missing messages and we achieve at least ordered delivery per sender. A global message ordering is however not achieved. The message router does not impose a global ordering as the system is designed to be logically decentralized. Although we use a central routing facility (described later in section 3.5.2) to enable global routing between devices on mobile networks, Musubi is designed to be flexible and not depend on centralized infrastructure.

3.5 Architecture

Figure 3.3 shows the high level architecture of the Musubi system. Although Musubi adopts a top-level peer-to-peer (P2P) architecture, the system is split up between a client and a server component. From the perspective of an application built on top of Musubi communication occurs directly between devices with equal roles, i.e. peers. However, a centralized server component is used in our implementation to enable communication between devices through the Internet.

The server-side component of our system fulfills two main functions: message routing through the Message Router and IBE private key distribution using the Identity Key Server. Because direct communication between devices on mobile networks is often not possible, the message router delivers messages between users in the system. The key server acts as the trusted party described in section 3.3.2 that delegates authentication to identity providers and issues the required private encryption and signature keys for IBE. We will describe each below in more detail.

The client side of Musubi lives as middleware between the application layer and the operating system, thereby providing social functionality to applications. There are many useful services that the mobile operating system provides. Also, the address book that lives on the phone plays a key role in the identity management of Musubi. The Musubi middleware is roughly divided between a developer API, that offers easy to use constructs to developers of social applications, and support components that offer the required services
and are used by the API. A database offers persistency and is also used to communicate data between the different components of the middleware and the application.

### 3.5.1 Identity Key Server

The Identity Key Server provides the necessary mediation between identity providers and the Musubi system in the absence of a standard way to obtain private IBE keys, as described in section 3.3.2. It verifies the authenticity of identities with third party identity providers and generates the IBE private keys accordingly. Identity verification either occurs through a supported authentication scheme like OAuth [29] or by sending a secret out-of-band over the associated communication medium. Scenarios of these two methods supported by Musubi are described below.
OAuth authentication flow

1. Jim indicates in a Musubi application that he wants to register his Twitter identity with Musubi.

2. The Musubi application presents Jim with the OAuth web page of Twitter in an embedded browser window. Jim is asked here to enter his Twitter username and password.

3. Jim grants Musubi access to his Twitter account information and the OAuth access token is returned to the Musubi application.

4. The Musubi application requests private keys from the Identity Key Server, including the OAuth access token in the request.

5. The Identity Key Server verifies the OAuth token with Twitter and, if valid, returns the private keys to the application.

Out-of-band verification

1. Jim indicates in a Musubi application that he wants to register his corporate email identity: jim@dundermifflin.com.

2. Since DunderMifflin is not known to provide OAuth, the application sends a request to the Identity Key Server to verify the ownership of jim@dundermifflin.com.

3. The Identity Key Server sends an email containing a link with a verification code to jim@dundermifflin.com.

4. Jim opens the email on his phone and clicks the verification link. This opens the Musubi application, passing in the verification code.

5. The Musubi application requests private keys from the Identity Key Server, including the verification code in the request.

6. Upon verification of the code, the Identity Key Server returns the private keys to the Musubi application.

Note that the out-of-band method could work equally well with phone numbers, in which case a SMS text message would be sent.

3.5.2 Message Router

As we pointed out in chapter 2, direct communication between devices on wide-area mobile networks such as HDSPA and 4G is not feasible. There is a need for an alternative facility to provide communication through the Internet. The Message Router offers this functionality and delivers messages between users. It also supports the Identity Based Routing scheme we discussed in section 3.2. It is important to note that although the Message Router is
3. **System overview**

typically used to route messages, Musubi does not depend on this component and this form of routing. In fact, we apply a form of content-based routing (identities are part of the content) and messages can be delivered using any type of routing. The Message Router guarantees delivery where direct connectivity between nodes is not available.

An obvious consequence of the centralization of the message router is that messages will not be sent or received until a connection to the message router can be established. However, it also allows messages to be sent even when a direct connection to the recipient does not exist, i.e., when the recipient is not online. When the recipient connects, the messages will be delivered. In other words, we enable the email model of asynchronous communication for application interaction.

**IBR over RabbitMQ**

The Message Router is an out-of-the-box message queuing server that implements the open standard Advanced Message Queuing Protocol (AMQP) [67]. We opted to use RabbitMQ in our reference implementation 5. A RabbitMQ server allows queues to be created and messages be published to them. Receivers can consume the messages from queues and listen on queues to have new messages pushed to them in near real-time as they are published. RabbitMQ also supports the creation of exchanges. An exchange can be bound to multiple queues. Messages that are published to an exchange are forwarded to all the queues it is bound to.

To support IBR, every identity that has received messages (which can include yet unclaimed identities) has a queue on the message router. The queue name is the result of the one-way hash function `id_hash_fn` over the identity type and the identity string. When a user claims a new identity, it creates a message queue for itself on the message router. After a Musubi application is started, it will continuously listen for new messages on the queues for all its claimed identities.

Exchanges are used to support feeds and are bound to the queues for all identities that are recipients in a group. The name of the exchange is a one-way hash function over the identities that are members of the feed. When the member list changes, a new exchange is created to reflect the new situation. The message router thus holds implicit member lists for groups. This could be considered a privacy concern as parts of the social graph can be extracted from these lists. However, the exchanges only map to one-way hashes over identities which makes any graph information that can be derived anonymous.

3.5.3 **Musubi software development kit**

While the server side glues everything together through the communication facility it provides, most of the novelty of the solution lies on the client side, at the Software Development Kit (SDK) that Musubi provides to applications. The SDK provides a full-fledged MoCoA middleware that meets our requirements, such as identity management, group formation, reliable group messaging, and persitency while also protecting privacy through secure communication. The heart of the system is in the support components and the database shown in

5http://www.rabbitmq.com/
Applications interact with the SDK through the developer API that we discussed in section 3.1.

The database provides persistency to applications by storing the identities, groups, and the data that are exchanged. It is also used to store partially processed incoming and outgoing messages. This makes applications resilient against failures and makes it easy to pass the control over the messages between different processes. The amount of objects in memory can therefore be kept to a minimum. The support components provide the core of the message handling functionality in an object pipeline, consisting of four services running in separate threads: **Obj Processor Service**, **Message Encoder Service**, **Message Decoder Service**, and **Message Transport**. It also provides **Identity Management**, which handles the private encryption keys and aggregates contacts from the phone address book and OSNs into the Musubi database.

The object pipeline takes objects that are offered to the middleware in the API and passes them through all the processing steps to finally send them as messages to the Message Router on the server. Partly processed messages are stored and read/removed by subsequent processes in the pipeline. Control is passed between different the processes through signals, which includes references to the objects. Similarly, the reverse pipeline listens for new messages that arrive from the server and processes them retrieve the objects that are offered to the application through hooks in the API.

Exchanged objects can be placed in hierarchical structures by a parent-child relationship. The parent id of an object is encoded within it. The middleware will on receipt attach the object to its parent. This relationship can be used for a basic level of merge strategies for conflict resolution. Two parties can both make their independent changes and encode them in a child object. They will both be attached to the parent and applications can make a decision on how to merge them. Musubi does not currently offer other means of conflict resolution.

### 3.5.4 Object pipeline

The processes below show in detail how the pipeline system handles the sending of messages through the API and receiving of messages. These processes are significantly simplified for the purpose of illustration, but they accurately demonstrate the flow of information, the role of the important entities, and the general operations performed. The exact format of the messages exchanged over the transport is shown in section 3.4.

**SendMessage: Posting an object to a feed**

1. Jim uses a photo sharing application to post a picture to a feed that has a few of his colleagues as members. He uses the `sendObjToFeed` method in the API, passing in the `PictureObj` that represents the picture.

2. The API stores `obj` in the database and sends a `PlainObjReady` signal with a reference to `obj`.

3. The Message Encoder Service responds to the `PlainObjReady` signal.
3. **System overview**

a) It creates a binary encoding of obj in BSON format, attaching some required metadata, i.e. the id of the originating feed and application and the current timestamp. The result is the message payload p.

b) It generates a random AES message key mkey and uses it to encrypt the payload p. It stores the result in a EncodedMessage object em in the database.

c) For every recipient it generates a conversation key pair (ckey_raw, ckey_enc) and creates a signature sig using the private IBE signature key (see section 3.3 for details). The unencrypted ckey_raw is used to encrypt mkey to mkey_enc. For every recipient the computed values are cached in Secret so that they only have to be computed again once the timeframe changes. The values of {hash, ckey_enc, mkey_enc, sig} are added to em. Here hash is the result of the one-way id_hash_fn function over the recipient identity.

d) It fires a PreparedEncoded signal with a reference to em, and a AppObjReady signal with a reference to obj.

4. The Object Processor responds to the AppObjReady signal.

a) It extracts possible relationship information from the metadata in obj and attaches obj to related entities in the database if needed.

b) It passes obj to the application for application-specific processing through the API. In case it is a platform Obj it is processed by Musubi.

5. The Message Transport responds to the PreparedEncoded signal.

a) It creates a queue for every recipient identity in em on the Message Router using the one-way id_hash_fn function over the identity, if it does not already exist. Note that this may include identities who are not yet using Musubi. When they sign up, the messages will be waiting for them.

b) It creates an exchange for the list of recipients on the Message Router, using the one-way exchange hash function over the identities in the member list, if it does not already exist.

c) It publishes the message data from em to the exchange. A handler is registered to be called when the Message Router acknowledges receipt of the message.

d) On acknowledgement of the message receipt from the server, it marks obj as sent and sends a ObjSent signal to provide feedback to the application through the API.

**ReceiveMessage: An incoming message on a feed**

1. Message Transport starts to listen on the queue for each claimed identity as soon as the application is started. The queue name is again defined using the one-way id_hash_fn function over the identity.

---

http://bsonspec.org/#/specification
2. When a new message is published on a queue, MessageTransport consumes it and stores the data in a new EncodedMessage object \( em \) in the database. It then fires an EncodedMessageReceived signal with a reference to \( em \).

3. The Message Decoder Service responds to the EncodedMessageReceived signal.
   
   a) It checks if the message is a duplicate of an already processed message by comparing the hash of the message payload with stored hashed in the database.
   
   b) It looks up the values of \( \{ \text{hash, ckey\_enc, mkey\_enc, sig} \} \) in \( em \) for which hash is equal to \( \text{id\_hash\_fn} \) over an owned identity. It decrypts the IBE conversation key \( \text{ckey\_enc} \) to \( \text{ckey\_raw} \) using the IBE private user key, verifies the signature \( \text{sig} \), and decrypts the message key \( \text{mkey\_enc} \) to \( \text{mkey} \) using \( \text{ckey\_raw} \). See section 3.3 for details.
   
   c) It decrypts the message data using the AES message key \( \text{mkey} \) to retrieve the payload and metadata in BSON format.
   
   d) It decodes the payload and creates an \( \text{Obj} \) object in the database: \( \text{obj} \), using the metadata from the decrypted message to bind it to the correct sender, feed, and application. It then fires an AppObjReady signal with a reference to \( \text{obj} \).

4. The Object Processor responds to the AppObjReady signal.
   
   a) It extracts possible relationship information from the metadata in \( \text{obj} \) and attaches \( \text{obj} \) to related entities in the database if needed.
   
   b) It passes \( \text{obj} \) to the application for application-specific processing through the API. In case it is a platform \( \text{Obj} \) it is processed by Musubi.

### 3.6 Sample use case: secure Messenger for iPhone

In this chapter we show how the Musubi middleware was used to develop Messenger, a secure messenger application for iPhone. Messenger is an example of a MoCoA that exhibits most of the requirements discussed in chapter 2, but doesn’t have a strong requirement on global consistency. It is therefore a perfect fit for our system. It not only serves as an evaluation case, but was in fact used to drive the development process of Musubi. The successful implementation demonstrates the effectiveness of the middleware.

The goal of the development of Messenger was threefold. In the first place, its goal was to inform the development of the Musubi middleware about functional and non-functional requirements a typical MoCoA imposes on the middleware. Secondly, it also served to evaluate whether applications can be built on Musubi that hold up in the current competitive mobile social application landscape. This means that while providing secure communication, the user experience has to be frictionless. Finally, we believe there should be a secure, data ownership preserving, alternative to the variety of mobile messaging applications currently available. Popular examples include Whatsapp, whose security has been under continued scrutiny [70], and Facebook, who is often criticized for its monetization of user data [63].
3. System Overview

We first describe the motivation for the development of Messenger. Then we show how the application is built on top of Musubi and how it leverages the MoCoA functionality that is offered by the middleware. We then shift our discussion towards the experience of using the Messenger application.

3.6.1 User experience

Even though secure private communication is what sets Messenger apart from other mobile messenger applications currently available, it still has to compete on user experience. Most users are not particularly concerned about privacy and data ownership, while in contrast they will not compromise on user experience. Any application that imposes friction on the user will face a hard time getting adoption or will not be accepted at all. Users do not want to be concerned with setting up privacy profiles or exchanging encryption keys, for example. Messenger fulfills both expectations and provides a high quality user experience while still ensuring secure private messaging through the Musubi middleware.

Musubi offers identity management in the form of a user profile that is disseminated to other users in the network proactively as well as on request. Messenger uses this functionality and defines a user interface component, i.e. a settings screen, where users can set their name and a profile picture. It also allows users to link their OSN accounts, both Facebook and Google, to register those identities in Musubi and prefetch profile information from the OSN. When the user first starts using Messenger, a welcome screen is shown that walks the user through the setup process (i.e. linking an account). Figure 3.4 shows how the settings and welcome screens can be used to manage the user profile. No more steps are required to start using Messenger than for other popular messenger applications, such as Whatsapp 7, or applications using Facebook Login 8.

7http://www.whatsapp.com/
8https://developers.facebook.com/docs/concepts/login/
Sample use case: secure Messenger for iPhone

Figure 3.5: Messenger conversation start and list screens

To start conversations, Musubi provides a FriendPicker screen that shows a list of all known contacts. The user can select people from this list to create a new feed with. Contacts are automatically imported from the address book and linked OSNs by Musubi. Figure 3.5 shows how the FriendPicker is used to start a conversation. A FeedListView, also shown in figure 3.5 shows a list of all the active group conversations, i.e. feeds.

When a user touches one of the feeds in the FeedListView, the FeedView screen is opened which shows all the shared updates in sequential order. It also allows a user to share a status, take a photo, make a voice recording, or share their location instantly. We have kept the number of actions a user has to take to a minimum and the experience highly interactive. Figure 3.6 shows how the FeedView is used to have interactive group conversations.

Figure 3.6: Messenger conversation screen

29
3. SYSTEM OVERVIEW

3.6.2 Architecture

The Messenger system borrows most of its functionality from the Musubi middleware. It defines classes primarily related to the user interface, account registration, and a data model for shared updates in group conversations. Figure 3.7 shows the main classes that were created for Messenger.

Even though Musubi offers primitives for accounts through the API, the actual registration and request for authorization tokens from the third party OSNs needs to happen from the application itself. Therefore, Messenger defines an AuthManager and three classes representing authentication flows for email, Facebook, and Google accounts. Once the account is verified it is saved using the Musubi API. Because Musubi does not support the Facebook and Google networks, Messenger also implements background services that import contacts from those networks. They are then added as identities to Musubi using the API.

Musubi defines a number of default data objects, or Obj classes, that are used by Messenger. StatusObj and PictureObj are particularly important. We further defined obj classes to enable users to share voice recordings and their location. According to the Musubi

![Figure 3.7: Messenger class diagram](image-url)
API, they define methods to pack and unpack their data to and from a raw and JSON field. No special processing is required for these obj types.

### 3.7 Evaluation

Messenger was informally evaluated by assessing adoption after distribution to a group of 47 people, including the author, in different friend groups. Overall, the users were comfortable with the user interface and did not have trouble signing in with their existing identities. During regular usage, the message transmission rate, responsiveness, and impact on battery life were also found to be acceptable to users. Indeed, the application saw good adoption immediately after release.

However, usage declined gradually over the course of a few weeks. The reason identified was that the application has a lead time that is too long after a period without usage. When a queue of messages is waiting, Messenger decodes and displays them one by one. Even though the decryption time for messages is acceptable while they are coming in at a normal rate, it turned out to be too long if a large number of messages is waiting to be processed. This caused people to stop using the application, because it would prevent them from quickly reading and sending messages. A solution is needed to regularly activate the application in the background so that it can decrypt messages.

### 3.8 Discussion

Musubi addresses most of the requirements for MoCoAs listed in chapter 2. IBR allows us to send messages to anyone with a known address, e.g. from the address book, thereby enabling group formation. Identity management is supported through the exchange of ProfileObj messages. Group address expansion is delivered by the concept of a feed, which also serves as an implicit ACL. Reliable transport is offered through the Message Router, which queues messages while recipients are not available. Messages can be delivered to the Message Router either directly or indirectly, through other devices. IBE secures the transport and the implicit ACL in a feed enables privacy protection. The system uses a database for storage on the device, thereby offering data persistency. Centralized storage is not part of the solution. Support for data consistency is also minimal, offering only a tree structure for exchanged objects to facilitate conflict resolution merge schemes.

The Musubi middleware was successfully used to implement a secure messaging application. Due to IBE, the amount of friction introduced by the system was kept to a minimum. The implementation effort for the Messenger related mostly to user interface components. To support third-party OSN accounts extra functionality needed to be implemented for registration and contact imports. However, no functionality pertinent to message transport had to be implemented in the code base of Messenger. Messenger is not a fully adequate evaluation case since it was used to drive the development of Musubi. Several other applications were developed on Musubi to test its effectiveness in abstracting MoCoA concerns.
Chapter 4

Dispatch

In this chapter we introduce the Dispatch project. Dispatch leverages the security and routing features of Musubi to enable and protect citizen journalists under oppressive regimes, specifically in crisis situations. A user study was done with journalists who used a custom built application on top of Musubi for quick and secure broadcasting during the day of the United States presidential election. The project is a collaboration between the Stanford MobiSocial Computing Laboratory, the Columbia University Graduate School of Journalism, and Tumblr.

4.1 Citizen journalism

Over the recent years advances in web and mobile technology have equipped the masses with the tools to tell their news stories to the world. Thanks to the popularity of Internet-connected mobile devices equipped with rich input capabilities such as high definition cameras on the one hand and widespread social media on the other, it is now easier than ever to capture and quickly broadcast news as it occurs. When public citizens, without professional journalism training, engage in such reporting we refer to this as citizen journalism. In fact, citizens can often respond more quickly to breaking news than journalists can when they are immediately involved in the events that take place.

Especially meaningful, newsworthy stories spread quickly due to the viral nature of online social channels like Facebook 1, Twitter 2, and YouTube 3. A particular class of such reports portray distressing incidents, such as criminal offenses, that occur in crisis situations. Citizen journalism has played a large role in the recent wave of public uprising and war against oppressive regimes in the Arab world, commonly referred to as the Arab Spring. In the series of public protests and demonstrations social media was used effectively to organize civil resistance and raise global awareness about political repression by states. Another example where citizen journalism can be very effective is in the case of natural disasters. After the 2011 Tōhoku earthquake and following tsunami in Japan large numbers

1http://www.facebook.com/
2http://www.twitter.com/
3http://www.youtube.com/
of photos, videos, and other reports by nonprofessional journalists were distributed on the Web by citizens.

4.1.1 Problems in oppressive regimes

Although technology is effectively used by citizens to get stories out, several social issues exist with citizen journalism in oppressive regimes today. Two important problems are identity exposure and Internet censorship.

First, in situations where governments oppress political opposition, citizen journalists fear possible persecution. When a repressive government exposes the identity behind reports against the regime this could pose a serious danger to the citizen journalist’s life. Thus, reports and traffic must be anonymous. The device that is used to capture stories must also leave no detectable trace since the device may be lost or confiscated.

Secondly, governments may try to censor traffic sent over the Internet. The most obstructive form is when states attempt to terminate connectivity to the Internet. This makes it hard for reporters to disseminate their content. Internet connectivity is likely not to be available when reports are made. Connectivity may only be available through individual satellite connections, or outside of the country. It is both dangerous and less likely that citizen journalists manage to reach a connection so that packets can be routed.

4.1.2 Pocket reporting Using Musubi

We note that the risk of identity exposure can be addressed using encrypted communication. When intercepted messages can only be read by trusted parties, there is no way for government to detect reports against it. Of course, encrypted content can still be suspicious to the government. Therefore it is also desirable for the identity of the sender not to be visible without decrypting the content.

In Musubi messages are encrypted and the only visible sender information is a one-way hash over an email address. If needed, an anonymous email address can be used as the sender so that there is no way to link intercepted messages back to a person. The sender must however still be careful to use an Internet connection that can not be linked to their identity.

Regarding the problem of Internet isolation, we note that this is not limited just to situations where political forces break connectivity. After the occurrence of a natural disaster, for example, infrastructure is typically severely damaged and connectivity to the Internet sparse. To increase the chance of content dissemination in areas isolated from the Internet, delay-tolerant networking and human mobility can be utilized. The nature of the problem is such that only the sender of the messages, i.e. the citizen journalists, are isolated. We can assume that the trusted recipients of messages are connected to the Internet. The recipients can be people or services outside the crisis area that will help spread the reports globally. As such, we do not need P2P routing all the way to the recipient. We only need to make sure that the message reaches an Internet connected host, a gateway, that can reach the message router.
We can not rely on contemporaneous end-to-end connectivity between the sender and the gateway. Such connected P2P networks are unlikely to exist given the circumstances. On the contrary, connectivity between devices can be expected to be highly intermittent. We want to store and pass messages from device to device, when opportunities arise. When one of them eventually connects to the internet, it becomes the gateway and delivers the message to the message router.

4.2 Pocket switched networks

Currently, the majority of mobile Internet applications are set up such that they require a connection to the Internet in order to function. This can perhaps be attributed to their evolution out of web applications that rely heavily on a pervasive connection to backend web services. Often, however, situations arise where such pervasive connections are not available. In contrast, Internet connectivity is only available in a limited number of connectivity islands. The scenarios of oppressive political forces limiting connectivity to the Web and natural disasters that disable infrastructure are two extreme examples of such situations.

4.2.1 Human mobility

Pocket Switched Networks (PSNs) are a specific class of Delay Tolerant Networks (DTNs). They use local and global connectivity as well as human mobility to transfer data between mobile users’ devices [34]. They are particularly useful in situations where there is a limited availability of connectivity islands. Outside these islands end-to-end routing often becomes effectively impossible.

Internet routing assumes a pervasive path between sender and recipient. Traditional routing becomes unavailable when such a path does not exist. PSNs use opportunistic networking to improve routing capability in these scenarios from zero to best-effort. PSNs essentially use human mobility of users combined with storage on their devices as a form of data transport. This concept is also known as data muling [55] and store-and-haul forwarding [61].

4.2.2 Forwarding protocol

The routing strategy in a PSN is not to try to build end-to-end paths. Packets are instead forwarded opportunistically from device to device, when local connectivity between them occurs. Such opportunities arise as a result of device users’ mobility. When two devices connect, either through discovery or through intentional pairing by the users, data is transferred between the devices.

In the context of our citizen journalism scenario, we want to route traffic towards the Internet. We can achieve this by forwarding messages locally towards nodes with global connectivity. A problem is that it is often unpredictable to the network which nodes will at some point reach a global Internet connection. The likelihood of this depends on contextual information that is not necessarily known to the network. It is also not necessarily known to the network if users can be trusted to carry content.
4. **Dispatch**

![Dispatch System Architecture](image)

**Figure 4.1:** Dispatch system architecture

### 4.3 The Dispatch application

To aid citizen journalists and address the problems we pointed out before the *Dispatch* application was developed using Musubi. It is essentially a modified version of the Messenger application, with a modified user interface, added support for Tumblr authentication, and support for device-to-device message forwarding over an ad-hoc connection using Bluetooth or WiFi. Figure 4.1 shows the architecture of the full system. *Dispatch* allows users to easily create reports, and securely publish them on the Web while possibly in a situation of disconnection from the Internet. The reports are securely forwarded using Musubi through a PSN of *Dispatch* users to a Musubi-connected *web publishing service*.

In order to be able to sign messages using Musubi, the sender must retrieve their IBE signature key from the Identity Key Server. Since this is a centralized facility, Internet connectivity is required to do so. However, the keys only have to be retrieved once per validity timeframe, which is a month in our case. Furthermore, keys for future timeframes can be requested from the server. The client can be programmed to always maintain a reserve of keys, to extend the lifetime outside of global connectivity. Reporters should thus at least connect to the Internet once, to allow the application to retrieve keys, before it can be used offline.
In the context of our scenario of citizen journalists using Musubi, a path between the sender and the message router may not be available. We create a PSN to introduce disconnection tolerance and be able to route messages between connectivity islands. Messages are opportunistically routed from device to device, until eventually one reaches global connectivity and can deliver the message to the message router. We call the device that eventually reaches the Internet the gateway device. It is important to note that although other devices help to route the message, they will not be able to decrypt it unless they are also a recipient of the message. Typically, this is not the case, as messages are addressed only to the well-known identity of the publishing service.

Direct forwarding of messages between devices in Dispatch happens by establishing a local connection from the application. A PSNController was added to Messenger which manages device pairing using Bluetooth or local WiFi and passes over unsent messages. We leave it up to the user to actively choose to forward messages from their device to that of another user. Figure 4.2 shows how the user passes unsent messages to another device.

The forwarding is a form of epidemic routing [64]. Aside from duplicate detection, the forwarding is unbounded, i.e. there is no max hop count. We assume the user is capable of selecting devices that are likely to become a gateway. Even though messages are encrypted, senders could eventually become suspected and endangered if encrypted messages get in the hands of malicious people. Trust is thus also an important consideration in the forwarding decision. Once a device reaches connectivity to the Internet it sends all queued messages, including ones from other senders, to the message router.
4. Dispatch

4.3.2 Publishing service

The Dispatch application allows reporters to publish created content to a Tumblr “blog”\(^4\). To this extent, a service was set up that uses Musubi to accept reports and the Tumblr API to publish them to the reporter’s Tumblr blog. The service receives messages on a well-known and trusted identity on the Musubi system.

To be able to publish to a Tumblr blog on the reporter’s behalf, the service needs authentication credentials to the Tumblr service. The reporter sends these credentials in the form of an OAuth access token to the service, using a `TumblrAuthObj`. The service is trusted to maintain a list of these access tokens, which have a limited validity timeframes. To obtain the access tokens from Tumblr the user has to be connected to the Internet. This authentication sequence thus has to be done prior to offline usage, similar to the retrieval of signature keys.

Eventually, one could imagine the publishing service being replaced by the news desk of a (online) newspaper, or any other publishing entity. This would allow citizen journalists to securely and anonymously get content out to major news distribution organizations.

4.4 U.S. presidential election day

The viability of the Dispatch application was tested in a small user study. Students of the Columbia University Graduate School of Journalism used Dispatch on the day of the

\(^4\)http://www.tumblr.com/
2012 U.S. presidential election. The version of Dispatch that was tested was specifically designed to publish all reports to a single Tumblr blog at http://dispatch-politics.tumblr.com/. The students were asked to sign up with an anonymous email address and report on the election from New York City. Figure 4.3 shows part of the reports that were made that day using Dispatch.

During election day, 9 users published 95 reports to the blog. However, most of the posts were published by a particularly enthusiastic student who published 74 stories. The students were asked afterwards to complete a survey about their experience of using the application. Although the overall feedback was positive, some students frustratedly reported that not all stories were published. This indicated that there were possible reliability issues with the system. There indeed turned out to be faults in the Tumblr publishing service component. A solution to the reliability issue was found as a result of the user feedback.

4.5 Discussion

Dispatch demonstrates it is possible to use Musubi for secure broadcasting on the Web. It was intended and used here to support citizen journalism, but other broadcasting services can be imagined as well. Dispatch also succeeded in achieving routing over a PSN where local P2P routing over Bluetooth or local WiFi is combined with the Musubi routing to make traffic resilient to situations of scarce Internet connectivity islands. Dispatch proves Musubi offers the required flexibility to support both added services and different transport models.

A user study conducted with journalists using Dispatch to do reporting proved the broadcasting service worked, although some reliability issues had to be addressed. The discovered faults were not in the Musubi middleware, but in the added services, and could easily be resolved. The device to device routing was not yet evaluated in a larger real-world study, but was shown to work in a lab setting.
Chapter 5

Migo and SocialKit-JS

This chapter describes SocialKit-JS and the Migo project. Migo uses Musubi as a tool to engage high school students with computer science. At the same time, it serves as an evaluation of our application model and developer API. SocialKit-JS was designed to expose the Musubi system to web applications and was used in Migo. We describe how SocialKit-JS was used by students without programming experience to create real mobile social applications in a project of five weeks.

5.1 Computer science 101

The goal of Musubi is to create a middleware that eases the development of MoCoAs. Ideally, the provided API makes it possible even for novice developers, without experience in building distributed systems, to build collaborative applications for mobile devices. We realize that this opens up the possibility to use Musubi as a way to introduce students to software development. Instead of an introductory programming language class, students can build actual social applications as a way to explore computer science. We started the Migo project and let students create simple mobile social applications with Musubi. The participants in Migo were high school students and college freshmen, all new to programming.

5.1.1 Sparking interest

We note that students often have a false image of software development. The popular image is of spending hours behind a computer, without social interaction. It neglects to portray the creative, design, and social aspects of software engineering. This false image is unappealing to many students and results in their loss of interest in exploring computer science. Introduction level programming classes often do not help the situation. They focus on teaching programming language constructs like conditional statements and loops. These do not show many of the important aspects of software development and do not speak well to the imagination of the younger students.

We hypothesize that an approach that is oriented towards a more exciting goal for students will be more effective. Our solution is to guide students through the process of build-
5. Migo and SocialKit-JS

Creating simple applications that they can use together with their friends on their phones. We argue that this will speak to the imagination more, be more engaging, motivating, and create more interest.

5.1.2 Instant social

Creating an application to the point where it can be used together with friends is a non-trivial task. It involves many classical distributed system problems that can clearly not be tackled in an introductory programming class. Musubi opens up the possibility to make applications instantly social. Developers do not need to be familiar with the distributed systems or networking specifics and instead can focus on client code. Further, we believe that the programming model in Musubi is intuitive enough to be understood and used at a starting point.

Because the participants in the Migo project are high school students with little or zero prior programming experience, the project also evaluates the facility of our API. We verify whether the API offers the right abstractions and primitives. We also assess whether or not they are easy enough to understand for inexperienced programmers.

5.2 SocialKit-JS

SocialKit-JS provides a slight simplification of the Musubi Developer API to mobile web applications in Javascript. It reduces flexibility in exchange for a more accessible programming environment. Although the model is a simplification of the Musubi API, the way that applications exchange data is the same. The differences are related to group formation, which SocialKit assumes to have been done prior to launch of the application. It must be noted that there exists an implementation of SocialKit in Java for the Android version of Musubi Messenger. When we mention SocialKit here, we refer to the Javascript version SocialKit-JS.

5.2.1 Web development

We note that although the model of our API may be intuitive, it is exposed in a language that is not as easily accessible. The iOS development platform uses the Objective-C programming language and the Cocoa Touch framework. Even experienced developers have difficulty getting familiarized with this environment. To overcome the language and programming environment hurdle we designed SocialKit-JS.

SocialKit-JS is a Javascript library that can be used to create instantly social mobile web applications. HTML and Javascript are friendlier languages for beginning programmers. First of all, there is much more help and documentation available online. The number of topics on Javascript and HTML on Stackoverflow at time of writing are 295,768 and 132,979 respectively compared to 109,307 for iOS. Further, any text editor can be used

---

2 http://stackoverflow.com/tags/
as authoring tool, there is no compiler, and the execution environment is a familiar web browser.

5.2.2 Architecture

SocialKit-JS integrates with the iPhone Messenger to provide applications with a communication channel between a group of users. An integration with SocialKit-JS in the Android version of Musubi Messenger was also created. The Android integration provides the exact same functionality and is interoperable with the iPhone version, thus we will discuss here only the latter. An application communicates between users exclusively in a single feed. Messenger is used to start a new conversation, i.e. feed, with a group of people. Afterwards a SocialKit application can be “launched” in the feed. From the perspective of a SocialKit

![Diagram of SocialKit-JS architecture]

Figure 5.1: SocialKit-JS architecture
application the group formation is already completed upon launch. The feed is immutable and persistent, and the member list is static. The application can not create new feeds and it can not add members to a feed.

Figure 5.1 shows how SocialKit-JS uses Messenger to provide a communication channel in a feed. SocialKit-JS is a Javascript library which provides application primitives and hooks for a Musubi platform. The hooks are used for bidirectional asynchronous communication. We added a HTMLAppController class to Messenger and a Javascript integration component on the browser side that binds to the SocialKit-JS platform hooks. The application primitives are largely the same as for Musubi applications, although the model is simplified. An application only has a single feed, which has a member list and contains Objs. Objs are posted to the feed, identical to Musubi applications.

HTMLAppController runs an embedded web browser in the iOS app and a communication interface between SocialKit-JS and Musubi. The interface calls methods in HTMLAppController by requesting the browser to open specially formatted URLs. The URL encodes the method name and the parameters. HTMLAppController intercepts the request and instead of loading a page returns the call and executes the method asynchronously. Reverse communication is done by executing Javascript code in the browser. This is also used to pass results from method calls back to the SocialKit-JS application. On a Musubi method call in SocialKit-JS a callback function is set in a global variable in the platform interface. Musubi executes this function in the browser with the result as a parameter.

5.2.3 Emulator

Developing and testing on the mobile device is very laborious. To run the application, first the HTML and Javascript code needs to be packaged into the iOS app on compilation. The app then needs to be uploaded and run on the device, or run in a simulator. Debugging is especially hard for social applications, where two devices are needed to test the social functionality. Debugging HTML applications is even harder because the runtime debugger does not work with code running in the embedded web browser. Logging is the only real option, through a Javascript bridge that can be provided. However, this logging is unreliable because once Javascript stops executing after an error is encountered the logging also stops working.

We developed a Musubi emulator that runs in a web browser on desktop computers to reduces the described barriers in the development environment. The emulator consists of a web page containing multiple inner frames. Each frame runs a simulation of a single Musubi instance. All the instances have access to a single feed, that contains all the other frames as members. The main web page simulates the message router and passes messages between the frames. Encryption is not used in this inter-frame communication. The frame instances run a library that provides an integration between the simulation environment and SocialKit using the platform hooks in SocialKit-JS. The applications thus use the same SocialKit library as the applications on top of Musubi do.
Migo study

To evaluate our programming model using SocialKit we performed an informal study with beginning programmers over the summer quarter. The participants of the study were chosen from high school students who had expressed interest in learning more about computer science. The duration of the program was 5 weeks. Participants wrote small social applications on top of Musubi using SocialKit and ran them on actual phones to use with each other.

A total of 7 high school students participated in the Migo study. They were 3 junior and 4 senior students from private high schools in the San Francisco Bay Area. The prior programming experience of the high school students consisted of a introductory high school level programming class. Two first-year undergraduate students who had taken some computer science classes worked on the projects along with the high school students to guide and help them when needed. The author and three other graduate students were also available to provide support.

5.3.1 Process

The goal for the participants was to create a working social mobile application from scratch running on Musubi and SocialKit. The students came up with their own ideas for applications to build and formed pairs to work in. The study was informal and ad-hoc, but the students did follow a structured process which is outlines below.

**Week 1**: During the first week students brainstormed ideas for the applications they could make. They discussed applications that currently existed which did not exactly address their needs, and things they did in real life that mobile apps could help with. The students were given freedom to decide what they wanted to do and what they wanted to get out of the program. At the end of the week the participants had decided on 5 application ideas, and formed teams of two, together with the two undergraduate students. One high school student decided to build an application on his own. The applications are described in section 5.3.2.

**Week 2**: In the second week the students started implementing their applications. The implementation at this point involved purely user interface in HTML, Javascript, and CSS. They essentially started building their applications as if there was only a single user. They designed their user interfaces and started implementing the logic to acquire user input and present results back to the user. Because the languages were new to the students, they needed some help from time to time. Questions typically related to manipulation of the DOM tree or styling attributes based on input events.

**Week 3**: User feedback formed the topic for the third week. Students prepared short presentations with mock screenshots for outside audiences to gather feedback on their application ideas. They first presented their applications to a group of 5 other high school students and engaged in an informal feedback discussion. Later in the week the participants presented their ideas again to a group of 35 high school students in a
computer science summer program to collect more feedback. Students began altering their applications based on suggestions from both feedback sessions.

**Week 4**: In the fourth week the students started to integrate their applications with Musubi through SocialKit. They wrapped the data that needed to be exchanged between phones in `Obj` objects in JSON. Sending the data over to the other phone was made simple by SocialKit-JS. What proved to be difficult was the asynchronous calling. Often a reference to the `Obj` that was posted to a feed was needed later. However, Musubi does not currently return a reference synchronously, but through a callback. The graduate students had to help the participants work around this.

**Week 5**: The last week was spent finishing up the work on the applications. Some of the integration with Musubi still needed work. The issues often related to special cases, e.g. what happens when there is only one member in a two player game. Finally, the students prepared presentations for friends and lab members where they showed off their accomplishments.

5.3.2 Applications

Below we give a short description of each of the applications the students built over the course of the project, to illustrate what they used SocialKit for. The original descriptions of the applications written by the students are included in appendix B.

**Bucket List** is a shared to-do list application centered around the idea of fun activities to do over summer break. Based on feedback from the user session, the students let users create multiple lists. Structuring this collection of lists in a JSON structure in an `Obj` was one of the main challenges for the students.

**Noes Goes** allows users to decide on who will perform an unpleasant task by touching a virtual nose on the screen. The user who touches the nose last loses. The main struggle was to calculate the correct result based on the number of users participating and pressing the nose in the game.

**Game Portal** reads user data from the Steam gaming network and allows users to broadcast what game they are playing. The main challenges were accessing the data through the Steam API, for which the students actually wrote a small PHP script. The application passes the users login data to a server that hosts the PHP script together with a graduate student.

**Truth or Dare** allows users to play a game of challenges. The game occurs in two stages. In the first stage, truth and dare challenges are input by the players and put into a pool. In the second stage, truths and dares are randomly assigned to players. The main challenge was in synchronizing the challenges that were input by multiple users, possibly at the same time. Helped by a graduate student, a merge strategy was applied to solve this coordination problem.
**CrossCheck** presents a crossword puzzle that can be played competitively between two friends. The player who completes the puzzle the fastest wins. The moves are communicated between the players so that they can see each other’s progress.

### 5.4 Discussion

The students all successfully used SocialKit to create working applications that they can use together with their friends. The API proved to be easy enough to use. The students did require some help from more experienced developers in building the applications. However, the help focussed mainly on three areas:

- learning HTML and Javascript,
- defining the data model,
- synchronization of individual user efforts,

The first two are related to programming basics and are not assisted by the middleware. Once the students figured out how to structure the data, they could easily put this format into an `Obj` and communicate between devices. The last issue is, however, related to our middleware. It is a classical distributed systems problem and we have identified it before in chapter 3 as a requirement of MoCoAs. As stated before, our middleware does not currently provide complicated conflict resolution mechanisms. The strategy that was used here was to send multiple child `objs`, which are attached to a single parent `obj`. This effectively constitutes a merge strategy. For more complicated conflict resolution, Musubi has no support. It is desirable for Musubi to contain a conflict resolution layer that abstracts these kinds of difficulties away. This is room for future work.

The emulator was only used sporadically and only in the last week, when the interactions through Musubi became more complicated. Many students chose instead to use SocialKit on their Android phones, because it was more fun for them to use the application on actual devices. We also realized the students did not have experience with debugging using the browser and it was hard for them to adopt the practice. Instead they adopted a time intensive trial-and-error approach using uploads to the Android phones.

Finally, Migo was successful in creating more interest in computer science among the students. The students felt that their prior image of computer science was wrong, and that they now had a better picture. For example, they were surprised to find out that in the course of this project they had to talk to users to gather feedback.
Chapter 6

Grandma’s Trunk

Grandma’s Trunk is a project that was started by a graduate student at the Stanford University School of Education, in the Learning, Design and Technology Masters Program. Grandma’s Trunk aims to improve cross-generational communication through the use of an application on tablet devices. After we presented the capabilities of the Musubi platform to the graduate student, the decision to collaborate and build the application on top of Musubi was readily made.

Grandma’s Trunk serves as an evaluation of using Musubi to build a more substantial application. It is also an excellent example of how Musubi can be applied to quickly construct educational applications. The differentiating effort here is on the Human Computer Interaction (HCI) side. The main challenge is to design an intuitive user interface for users in two completely different age ranges that actually opens up new interaction opportunities not commonly seen before. Developers of such applications cannot be concerned with implementation details of distributed systems. It also serves as an example of an application that may not be easily monetized, and for which specialized application server infrastructure is not feasible. Musubi eliminates that requirement and places the infrastructure cost with the network operators, who can run it more cost-effectively.

6.1 Cross-generational communication

Family members live increasingly far apart from each other. Specifically, the AARP showed that in 2002, over half of the grandparents in North America are separated by more than 200 miles from their grandchildren [16]. Often grandparents even live in a different time zone than their grandchildren. As a result, grandparents are usually not around their grandchildren much during the child’s early developmental stage and do not get to interact much. Cross-generational interactions are not only emotionally highly valued, but also form an important part of a child’s informal learning process. Technology can play a large role in connecting family members who live remotely and can help bridge the gap. Video conferencing technology on phones, computers, and tablets are now commonly used to connect to remote loved ones. Products such as Skype and FaceTime are perceived as more natural, intimate, and satisfying than telephone conversations.
Existing technologies, however, don’t engage users in an emotionally active way [2]. Moreover, they are designed for younger people and are difficult to use for older generations [42]. Grandma’s trunk is designed to be easy to use for both children and older adults. A study on video conferencing in families by Ames et al. [2] reports that children have limited patience and would typically participate in the call for less than a minute. Apart from their difficulty maintaining attention, children may also have a hard time expressing their ideas through words. Action seems to be a better communication medium for children than language [4]. Grandma’s Trunk provides additional opportunities for more meaningful interaction that enables learning by fun.

Families further struggle to coordinate schedules in order to have a video conversation, especially when families are distributed across different time zones [12]. Grandparents are conscious about fitting contact into their schedule and are wary of being intrusive on others [43]. There is an opportunity for asynchronous communication methods to enable families to overcome these difficulties.

6.2 Application design

Grandma’s Trunk is a mobile application for iPads that helps grandparents connect with grandchildren. A user-centered design approach was used for the development of the application. A preliminary field study was previously done, involving 6 distributed families, to inform about the design of Grandma’s Trunk as a collaborative platform and the design of three collaborative activities. We chose to use tablet devices because they are easy to use for seniors and the touch screen and form factor make them particularly suitable for the creative applications. The main goal of Grandma’s Trunk is to strengthen the relationships between children and their grandparents by letting children and grandparents work together on creative online projects. Grandma’s Trunk facilitates affective learning – grandparents and grandchildren are both learning while bonding with each other.

6.2.1 Informal learning

We have gamified three collaborative activities: photo-sharing, story-telling, and drawing. These activities provide various learning opportunities for the children:

- they foster their imagination and creativity,
- through the use of visual images, children develop critical thinking, communication, and visual literacy skills,
- they increase observation skills, evidential reasoning, and speculative abilities.

Grandma’s Trunk is based on a socio-constructivist cooperative learning approach that is suggested to improve higher mental functions such as reasoning, comprehension, and critical thinking [60] [68]. Children can accomplish mental tasks with social support before they can do them alone [72]. Grandma’s Trunk allows grandparents to guide and moderate the content of the gamified exercises. As grandparents start getting involved with this
6.2 Collaborative creative activities

To accomplish relationship strengthening and informal learning we focused on providing a cross-generational collaborative communication platform where seniors and children can create and edit visual media together. Grandma’s Trunk is designed for children between the ages of 4 and 11, and for adults that are 50+. The three collaborative activities in Grandma’s Trunk are:

**Photo Sharing:** Sharing pictures for critical thinking and visual attention. Users share their experiences visually by posting photographs from the camera or gallery. With a
structured interface to comment on images. Figure 6.1 shows an example of how a picture is shared in Grandma’s Trunk. Both participants can comment on the picture.

**Story Writing:** Writing a story for divergent thinking and storytelling. One user starts a story, and subsequent users take turns adding to this potentially never-ending story. The objective is to make it funny, while keeping the story sensible. Figure 6.2 shows an example of a story created in one of our user studies.

**Drawing:** Collaborative drawing. Users can draw together on a canvas and iteratively build upon each other’s creations. Figure 6.3 shows how a grandparent added to a grandchild’s picture.

In addition, Grandma’s Trunk provides support for feedback. There is a heart button for every post that users can press to turn it from gray to red, indicating to the other party that they like the shared content. In addition, a wish list feature allows children to provide a list of things they wish for. Instead of just praises, through the “hearts”, grandparents can reward their grandchildren with real-life presents. The WishList was introduced because interviews with seniors suggested that they often do not know what their grandchildren wants. This facilitates communication, but also gives the grandparents a way of rewarding the children for displayed creativity.

### 6.2.3 Ease of use

Unlike other social network applications, Grandma’s Trunk is designed to facilitate the communication between just two parties. The setup is simple: users simply enter identifying information (e.g. email addresses) of the two parties once, when the program is installed. The users can then interact using any of the three activities. As shown in 6.3, there are only three large buttons, one for each kind of activity. The user is notified when new messages have arrived. Content that is shared is displayed right on the home screen to be easily accessible and hard to miss.

### 6.2.4 Security and privacy

While security and privacy are not the primary focus of this work, they are important considerations in the design of the system. Seniors are often concerned about owning and controlling their data. In addition, as we are working with children under 13, we must comply with the Children’s Online Privacy Protection Act (COPPA) [13]. Commercial online social networks (OSNs) simply demand that users must be over 13 years of age. In practice, kids, often assisted by their parents, lie about their age to join the network. Today’s OSNs typically sustain themselves by monetizing user content. It is desirable to build Grandma’s Trunk using an efficient privacy-preserving architecture that can be commercially viable without monetizing user data. We accomplish this goal by building upon Musubi. IBE means that users do not have to exchange keys before they communicate, allowing the users to enjoy the security without friction.
6.3 Architecture

Grandma’s Trunk builds on Musubi to achieve asynchronous communication. Musubi requires a verified identity to send and receive messages. To that end we enabled email authentication. The large majority of the application concerns the user interface. Figure 6.4 shows the most important classes that were defined for Grandma’s Trunk.

6.3.1 Email authentication

Because Musubi uses verified sender signatures, users need to verify their identity before passing messages through Musubi. In Grandma’s Trunk, users are presented with a welcome screen on first launch that asks the user to enter their email address. The address is passed to the IBE Key Server in the Musubi middleware that sends an email to the address containing a verification link. Upon opening the link, Grandma’s Trunk is reopened and will verify the identity with the IBE Key Server using the verification code from the link. This is handled by the EmailAuth and AuthManager classes.

After verifying the owner email address, Grandma’s Trunk asks the user to enter the email address of the other family member they wish to use the application with. This is then registered by the application as the only other identity it should ever send messages to.
6. Grandma’s Trunk

6.3.2 User interface

The user interface of Grandma’s Trunk makes up the majority of the application code. It is designed to be as easy to use as possible. Every activity that is started is represented as a separate Musubi feed. The main screen shows a list of all the feeds, in descending order of the last received message in that feed. Three buttons on top launch each of the activities.

The Story Writing and Photo Sharing activity have native iOS user interfaces. The Story Writing activity presents the user with a screen to enter the first line of a story. It hints the user to write engaging text with a preset text of “Once upon a time”. The Photo Sharing interface asks the user to select a photo from the gallery or use the device camera to take a photo. Upon doing this, a new feed is created and the content is posted as the first Obj in the feed. Both users can then add comments to the feed using the “Comment” button. They can post a comment either as text or as a voice recording. In the case of a story, there will be a “Add on” button instead of a “Comment” button and it will present users with a dialog to supply the next sentence along with suggestions for connector words (e.g. “then”, “and”, “so”).

The Drawing activity is a special case, as it runs a HTML and Javascript application showing a canvas in an embedded browser. It sends the contents of the canvas back to Grandma’s Trunk when a user hits “send” and is then posted on a newly created feed, as with the other activities. Further, when a user touches a photo or drawing on one of the feeds, it is opened as the starting content in the canvas application. This way, users can add to each others’ drawings and photos.

6.3.3 Data model

In Grandma’s Trunk, pictures and drawings are supported through the PictureObj that is supplied in the Musubi SDK as a default obj type. Pictures and drawings are always added as the first obj in a newly created feed. Similarly, when a story is started, a StoryObj is posted to a new feed. Comments are placed under a photo, either in voice or text, by posting a StatusObj or VoiceObj to the same feed. Added lines to a story are posted as a StatusObj, identical to comments.

6.3.4 Effort distribution

Table 6.1 shows the distribution of the lines of source code between the different parts of the system. It can be seen that the implementation of the user interface required almost 78% of the total lines of code. Much of the actual screen layouts are defined in iOS development in a graphical User Interface Builder, and do not require source code to be written. In contrast, the code for user authentication and the data model together took only 9% of the code lines. Lines of code is not a perfect metric for development effort, but we consider it a good approximation. It is important to note that this only relates to the control code for the graphical components.

The “initialization” category relates to the main application classes that initialize the views and the components in the Musubi SDK. The “utility” category contains some code for common operations, e.g. image transformations.
### User study

We evaluated the application using an extensive study with children in an elementary school and their grandparents. Some of the children were using the application together with their grandparents. Because of logistical challenges, most children could however not be paired up with their grandparents and instead they used the application with each other. We evaluated the effectiveness of messaging, the activities as icebreakers, and the incentive system as a motivational tool for promoting communication between grandparents and grandchildren.

#### 6.4.1 Methodology

Twenty-two third grade students, between 8 and 9 years old, of a private elementary school in San Francisco, participated in the study. Four of the children were paired with participating grandparents, who were between 50 and 70 years old. The other children were paired with each other randomly. The logistical challenge of letting grandparents participate reflects the difficulty of synchronizing schedules for simultaneous engagement in separated families. The study was done during four 30 minute sessions over four consecutive days.

The application was adapted so that it could be put into three different configurations. Configuration 1 supports only text-based messaging. Configuration 2 includes the three activities. Configuration 3 adds the “hearts” and wish list incentive system to configuration 2. The user pairs were divided into 3 even groups that got assigned a different configuration. During the subsequent days, the configurations were shifted.

We performed a series of qualitative studies including a pre-intervention survey, an intervention, daily post-intervention surveys, analysis of the content created, and a debriefing think-out-loud final session with the children. We took the difficulties children experience with surveys into consideration, providing them with a wider opportunity to give feedback. All studies contributed to our understanding of the engagement of the participants.

#### 6.4.2 Results

The pre-intervention survey confirmed the demand for an application such as Grandma’s Trunk. Only 17% of the grandchildren answered that they knew their grandparents well. 29% of the grandchildren communicate once a month with their grandparents, usually over the phone or in person. 54% of the grandchildren have iPads at home and use them regularly.

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>700</td>
</tr>
<tr>
<td>Model</td>
<td>180</td>
</tr>
<tr>
<td>Authentication</td>
<td>640</td>
</tr>
<tr>
<td>User Interface</td>
<td>7366</td>
</tr>
<tr>
<td>Utility</td>
<td>585</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>9471</strong></td>
</tr>
</tbody>
</table>

**Table 6.1:** Source code line count.
6. Grandma’s Trunk

We tested interaction among students while using three different configurations. The addition of the activities in configuration 2 made the application more interesting for the children than simple configuration 1. Finally, configuration 3 received the best rating, with everybody giving it a rating of at least 3. The children offered the following comments on the messaging-only version: “It’s very annoying”, “I don’t like texting”, and “It was boring”. They indicated that they liked the activities much more with comments like “It was fun! I loved it.”. Finally, 8 out of the 11 respondents commented that they liked the hearts and the wish list. Asked which activity they liked the most, 1 answered Story Writing, 5 chose Photo Sharing and 7 preferred Drawing.

All of the grandparents were supportive of the study and were eager to participate. Several made an extra effort to coordinate schedules and one even bought an iPad to join the study. We distributed the application through TestFlight 1, a service that distributed iOS applications outside of the Apple AppStore. Unfortunately, the extra steps required to register with TestFlight turned out to be too difficult for many of the grandparents. Because of this end, only 4 out of the 22 students were able to interact with their grandparents during the sessions in this study. Grandparents were very engaged and felt invigorated by the study.

The communication done through the messaging application was on average just one sentence long. The participants enjoyed the rich communication experience afforded by the activities and the incentive features. It was common for the children to write personal messages such as “I love you grandma” or “Is anybody there?”. All the children that were paired with their grandparents indicated that they strongly liked connecting with their grandparents. Three of them rated Grandma’s Trunk with 5 out of 5, one gave Grandma’s Trunk a rating of 4. They all indicated that they liked the drawing app the most. One student said “Thank you”, and another said “It was great I wish I could do this almost every day”.

6.5 Discussion

Grandma’s Trunk demonstrates that building an application with challenging HCI requirements for cross-generational interaction using Musubi is possible. It also shows that Musubi allows the focus of the development effort to almost entirely be on the user interface. Analysis of the code base shows that nearly 80 percent of the work focussed on the user interface and that the total lines of code was kept under 10,000.

The success of the application itself was evaluated in a user study. The study showed that both grandchildren and grandparents enjoyed interacting through creative activities. Although based on a small study group, the feedback from the children and grandparents that were paired up was especially positive in that regard. Our results clearly show that the creative activities add a lot of value over simple text exchange. Children are much more motivated and more likely to interact using the activities. The application provides a space for self expression and provides a rich bonding experience through users doing something concrete together. It also provides the space to get to know each other and to communicate in a way other than usual. The use of activities can overcome the social issues that synchronous technologies such as video conferencing pose.

---

1 http://www.testflightapp.com/
Chapter 7

Related work

Our work can be seen in many different lights and touches on a broad range of research topics. Collaboration using mobile devices is an active area of research. We point out other middleware solutions here that our system can be compared to. They provide hybrid P2P application models, group communication, or social networking functionality. We also discuss literature in the area of adaptive routing and transport-agnostic applications. Finally, we show related work in the domain of privacy-protected communication, specifically private social networking.

7.1 Serverless APIs

Our work is based on lessons learned from a preliminary version of Musubi, which provided secure group communication using public key exchange and RSA encryption [19]. The foundation of the system described here was designed in a collaborative effort between the author and other members of the Stanford MobiSocial Computing Laboratory. The fundamental concepts have also been applied to create an Egocentric Social Platform (ESP) [53]. ESP is compatible with the Musubi middleware described here. ESP provides background services on the Android platform for applications to hook into, allowing them to communicate through a central messenger application. In contrast, we provide a Software Development Kit (SDK) for the Apple iOS platform with an easy API to build complete and independent applications with.

Several commercial APIs exist that eliminate the need for developers to be concerned with implementing and maintaining a central application server. Products like Firebase 1 and Parse 2 provide a simplified application model. These companies, however, provide fully centralized solutions and the server is aware of the application model. The systems also do not provide the same privacy guarantees we do. In the case of Parse, real-time communication is not supported and applications have to actively poll data from the server.

Lass et al. [40] and Dean et al. [17] take a reverse approach with GUMP and enable developers to write applications in a client/server model, while deployed in a serverless

---

1http://www.firebase.com/
2http://www.parse.com/
environment. The rationale is that a lot of systems have been developed for centralized architectures, which should be made easily adaptable to peer-to-peer (P2P) networks.

7.2 Group communication

Group communication is an effective model that has the potential to more than double the usage of communication systems [15]. Traditional group communication systems (GCS) allow applications to organize processes in a distributed system in groups and provide them with fault-tolerant communication. Users in our system could be seen as distributed processes. These systems also offer consensus and consistency through atomic broadcast. Examples are Isis [5], Phoenix [46], and later Horus [66] and Ensemble [30]. Targeted at wired networks, they incur a high cost on transport failures, assuming them to be scarce. This makes these systems poorly applicable to wireless networks. An application model, on top of the communication facilities, is also not provided. Set in the light of the MoCoA requirements, they only provide group address expansion.

Other systems focus on group formation, but do not provide communication or provide only insecure communication. Notable examples are SocialFlows group extraction from email data [45], community detection from photos on Facebook in [52], and bootstrapping group communication using OSNs with Cluestr [27]. Mobiclique also leverages existing social graphs to form groups and does enable interaction, but only in ad-hoc networks [51].

Publish/subscribe systems offer asynchronous communication and transport decoupling to applications. They do not typically offer solutions to the other identified challenges of MoCoA development. Publishers send messages to queues, which typically reside on a messaging server. Subscribers indicate interests in events, and are notified asynchronously when those events are generated by publishers [23]. These systems can be used for group communication by creating a queue for each group, with the group members as subscribers.

A large body of work is concerned with applying publish/subscribe systems in mobile environments. JMS is an API specification for message routing in the publish/subscribe model for Java applications [47]. It uses a centralized server that handles delivery and persistency. Pronto is an implementation that adds a serverless mode for ad-hoc networks [73]. Musolesi et al. describe a similar system that applies epidemic routing in MANETs and designates hosts to hold certain queues [48].

JXTA is an open source P2P framework in Java for developing peer-to-peer applications [38]. It does not depend on any specific network transport mechanism, and assumes that connections are intermittent. JXTA offers primitives for group formation and group communication. JXME provides a modification for mobile devices but employs proxy servers that run on non-mobile hardware [37]. JMobiPeer allows group communication interoperable with JXTA, but is built for purely ad-hoc networking [6].

MOBILE MOM is a more complete application framework that offers communication and conflict resolution between peers [74]. It allows peers to communicate either directly or through a server, which also supports persistency.
7.3 Social networking

Inspired by the rise of social networks on the Internet, middleware solutions have been proposed which provide applications with social functionality, such as social graph maintenance, community management, and context awareness. Recently, specifically, geographical location awareness has become an important focus of these works.

MobiSoC is an example of a social computing middleware that allows applications to asynchronously pass events between users [28]. It addresses identity management and group formation as well. However, part of the application runs on a server which maintains global state, and thus not offer the same decentralized application model we do.

Proem is a middleware for social P2P applications [39]. It addresses requirements of face-to-face collaboration by offering identity management, access control, persistency, and event-based communication. The basic communication offered is an unreliable and connectionless transport protocol based on asynchronous messaging. Higher level services offer identity management, group formation, and synchronized event disintermediation. SOMU is an architecture for MoCoAs developed to address a set of challenges similar to the ones we identified in chapter 2 [49]. It offers applications with communication, coordination, and collaboration services in three corresponding layers. Compared to our work, both of these systems offer more consistency guarantees. However, they are fully distributed systems, do not address security, and do not provide the same privacy guarantees that we do.

SAMOA offers social and location awareness to applications in ad-hoc networks [8]. It allows people to discover each other and form groups. Its primary focus is on matching users to nearby services, although it also offers group messaging. Messages are however addressed to physical addresses on the network, and are thus not resilient to disconnection.

MobiClique enables formation of communities to enable interaction between people that meet opportunistically [51]. It offers a social ad-hoc networking API that offers social profile and graph management, as well as social graph based communication. Profiles and friendship relations are bootstrapped by downloading information from Facebook. Users discover and communicate with each other through Bluetooth connections, based on friendship or shared interests. Unlike ours, their work focuses exclusively on nearby communication and group communication is limited to small groups. The work also only offers unreliable best-effort messaging. It explores the concept of groups for implicit access control, but offers no additional security measures to protect against MITM attacks.

Rodriguez-Covili et al. present the HLMP API which allows nomad users to detect and interact with each other [54]. The focus of their work is on protocols for P2P group communication. Applications can implement their message types and custom protocols. It does not address the other architectural difficulties of MoCoA development.

7.4 Routing

Our solution applies routing to identities through a server in a way similar to email, and thereby allows applications to be transport agnostic. Content based routing in overlay networks addresses similar issues [41]. We employ a type of content based routing, but our system is more similar to intermediated instant messaging.
7. RELATED WORK

Other work has been done that allows applications to route traffic to high level addresses and separate application logic from networking mechanisms. The Unmanaged Internet Architecture (UIA) is a P2P connectivity architecture that lets users connect their mobile personal devices via personal names [24]. Devices can be paired and automatically connect when possible, both in ad hoc networks and using global infrastructure when available.

Haggle allows applications to be communication agnostic by providing a late-binding transport interface [35] [62]. Applications specify their requirements through an API and the system provides networking that automatically adapts to the best connectivity mode. For example, an application built on Haggle switches between Bluetooth and WiFi connections automatically. The work proposes a name graph that maps user-level names to lower-level protocol names. While Haggle focuses on the lower networking level, it does provide a structured data model for applications to use, supporting persistency. It does not address group communication.

7.5 Privacy protection

To provide privacy protection and data ownership preservation, encryption and decentralization techniques are applied.

Secure group communication through has been explored in many different fields, such as mesh networks [20], vehicular ad-hoc networks [33], and secure multicast using proxy cryptography [57]. FlyByNight leverages the social graph from existing online social networks, while storing encrypted content on a central server [44]. Users of the system are still required to sign up for a single proprietary network, invading privacy by exposing the social graph. Anderson et al. also propose a mechanism to enable private communication over untrusted social networks [3]. ISMANET also applies identity based cryptography to provide secure routing in mobile ad-hoc networks [71].

Decentralization eliminates the need of a single party to gain control over personal data. Diaspora 3 is an open source, community run, distributed social networking system. Peer-SoN applies both decentralization and encryption to provide secure social networking [10]. SocialVPN is a peer-to-peer overlay architecture that securely and autonomously creates VPN tunnels connecting social peers [36].

3https://github.com/diaspora/diaspora
Chapter 8

Discussion and future work

We set out to develop and evaluate a middleware system to enable the easy development of mobile collaborative applications. While the system we designed has proven successful in taking a large portion of the difficulty out of developers hands, the most interesting result of this work is perhaps the projects it spawned off. It turns out that when secure communication is made easy, there is a large demand for it in a broad selection of applications. When we started out with this research we did not expect to apply our work to such an interesting selection of case studies as the one we presented in this thesis. We have used Musubi across a variety of domains, to support citizen journalism, cross-generational cooperative learning, and computer science education. The fact that we did not intentionally select these projects, but rather matched them with Musubi as they came on our path, strengthens our belief that our solution has a wide range applicability.

8.1 Intuitive API

As our lives become more mobile, and we do ever more collaborative work through our mobile devices, the development of MoCoAs becomes an increasingly important activity. Currently the task of creating such an application is unnecessarily complicated and characterized by redundant solving of many challenges related to communication, coordination, and collaboration. The API in Musubi allows for the easy creation of MoCoAs by applying a top-level P2P model that reflects the real world more closely. We have demonstrated this by testing the model with beginning programmers in the Dispatch project. The novelty in the solution is that it is device-centric, while the world is largely moving towards a cloud-centric model to provide solutions to the distributed system issues.

The main limitation of our API is that it does not provide adequate support for coordination, i.e. data consistency among peers. Currently there is even no support for message ordering. The Migo project exposed this limitation clearly as it was a main area where the participants needed assistance. In group communication systems a coordination layer provides consistency on top of reliable communication [11]. A promising direction for future work would be to apply this concept to our solution.
8. DISCUSSION AND FUTURE WORK

Even though Musubi does not address many of the consistency requirements that mobile applications have, it allows applications to use the hierarchical object model to merge work efforts. Musubi presents an API that sufficiently addresses most of the typical challenges developers of MoCoAs face in identity management, group communication, and persistency.

8.2 Frictionless security

Our system routes messages over end-to-end encrypted channels to any previously existing identity, without introducing friction for users. We found that there is a large demand for easy to use secure communication. End users showed interest in the privacy protecting features of Messenger. The Grandma’s Trunk and Dispatch project show that there is a need for secure communication across a diversity of other scenarios as well. Applications can be imagined in the financial and health care sectors for example, where security is a primary concern.

There is a trade-off between security and ease of use, but we have presented a solution that introduces the same level of security that public-key exchange solutions offer while introducing minimal friction. One instance where the trade-off between user experience and security did surface, was in our informal evaluation of Messenger. The need for decryption introduces a long delay when a queue of messages is waiting when the application is opened. In fact, this was the primary reason why users abandoned the application. More work is needed to provide a solution to this problem. The lesson learned here is that, while people care about privacy, user experience is often more important.

The use of Identity Based Encryption (IBE) [7] is not new. However, we have shown how it can be used to support applications based on group communication. One of the caveats of our system is that the identity provider must be a trusted entity, since they can potentially issue access tokens to get access to private keys. Future work is needed to explore the use of federated key distribution, where multiple independent parties each hold a portion of the key. A particularly interesting direction is to explore the implications for user experience in that case.

Another avenue that should be explored concerns the persistency and dissemination of encrypted data that can be offered by the central component of our system, the message router. An open question is how we can more efficiently store and route data without insight into its properties. How can Quality of Service (QoS) be provided for high priority traffic for example, without breaching privacy? Another question is how we can differentiate between ephemeral and permanent data. Semantic tagging of data might be a worthwhile approach here. This raises questions about whether certain metadata should be publicly exposed in the message header and, if so, how they impact our security model.

Consumers forgo privacy protection en masse by using products like Facebook and Whatsapp [63] [70], because the value they offer is currently not matched by secure alternatives. Our work shows that an opportunity exists to introduce such alternatives.
8.3 Flexible routing

Messages in Musubi can be routed through any means. Because the destination is encapsulated in the message contents, regardless of how it arrives at the message router or directly at the destination, it can be delivered. This allows the system to be used in a large range of different scenarios, where connectivity is limited. One example is that of crisis or deprived areas like in the Dispatch project, but we might also think of transit (trains, airplanes, cars), for example.

We have demonstrated the ability to route over pocket switched networks with the Dispatch project. However, it must be noted that our evaluation was very minimal. We have only applied a flooding on demand forwarding model and we have not tested it at scale. More work is needed to evaluate the consequences and constraints of this routing technique in real world scenarios. Another interesting direction for future research is to apply Musubi in mobile or vehicular ad-hoc network (MANET or VANET) settings. Questions regarding routing protocols will need to be addressed. For example, how can anonymized and opaque messages be disseminated in an ad-hoc network other than by using flooding? Providing coordination, i.e. shared data consistency, in a system with decentralized routing is another challenge that will need to be addressed.

While the use of P2P and hybrid routing in these ad-hoc scenarios is not new, we have exposed this capability in a easy-to-use MoCoA middleware for a popular phone platform.

8.4 Commodity infrastructure

Musubi does not use centralized application servers, relying instead on generic infrastructure to route messages globally through the Internet. This saves application developers the cost of operating application servers, shifting the responsibility to maintain the message router to infrastructure companies like ISPs or telephony carrier companies. Especially when it becomes easier to develop MoCoAs we can imagine many non-professional applications coming into existence that can not rely on costly proprietary infrastructure, but that can be supported using such commodity infrastructure. The challenge is to create incentives for companies to provide this commodity infrastructure. Carriers would have to change their business model and charge, instead of for services like SMS, for data traffic or added services on top of this infrastructure. One could imagine the carriers for example offering cloud persistency solutions integrated with the Musubi message router.

8.5 Computer science education

An unexpected result of our work is that Musubi and SocialKit-JS proved to be useful tools to raise interest for computer science among high school students. Computer science as a field has a false negative image and it is difficult to get high school students interested in pursuing it. Introductory programming classes often don’t help, because they focus on teaching language constructs and are generally not “fun” for students in this age group. In
the Migo project we let students experience the process of designing and building a fully functional social application, on top of Musubi and SocialKit.

The main lesson learned from the study are that documentation and tooling is needed that more closely fits to the prior experience of the students. The browser emulator, for example, turned out not to be used much, because students were unfamiliar with the basic concept of debugging. Although no formal or structured study was done and the evaluation was very minimal, we are confident to say this is an interesting avenue to continue to explore within computer science education. Because students were bringing their own ideas to life and, moreover, could use their creations together with their friends, they were very engaged. The students reported that their image of computer science and computer scientists had changed and that they were now much more interested in pursuing a career in the field.
Chapter 9

Conclusions

The goal of this research is to “develop a middleware to support the development of mobile collaborative applications based on secure group communication.” We have designed and implemented Musubi, a middleware system that allows MoCoAs to be created easily. We took advantage of top-level decentralization to offer a communication model that more closely represents real-world scenarios. At the same time, the Musubi architecture allows applications to be transport agnostic. End-to-end encrypted communication was provided in a way that imposes little friction on the user. Finally, it allows applications to run on commodity infrastructure, reducing operational cost. We evaluated the suggested benefits of the system in three case studies.

9.1 Answers to research questions

Research question 1

What are the functional requirements of MoCoAs?

In chapter 2 we have listed functional requirements that characterize mobile collaborative applications. It must be noted that this list is generic and certain classes of collaborative applications will have many additional requirements. For example, real-time communication is not a requirement for all MoCoAs by far, but still characterizes a significant section of these applications. We have focussed in this work on providing generic functionality for identity management, group formation and communication, security, and persistency. We intentionally left data consistency out of the scope of this work, because of its application specificity.

Research question 2

How can we provide frictionless secure group communication that is independent of any particular transport?

In chapter 3 we presented our protocol and message format that enable group communication. We discussed how we applied Identity Based Routing (IBR) and Identity Based
9. CONCLUSIONS

Encryption (IBE) to introduce flexibility and security. We also presented a message router that enables group communication on mobile networks through the Internet. At the same time, the use of IBR makes our system independent of a particular transport. Because messages contain routing information in their content, they can be routed using a variety of methods. We successfully demonstrated with the Dispatch application in chapter 4 that ad-hoc networking and human mobility can be used in a pocket switched network to route messages from Internet-isolated areas to the Web using Musubi.

We noted as well that frictionless communication is necessary to provide a good user experience. However, there is a trade-off between security and ease of use. We demonstrated using the Messenger and the Grandma’s Trunk applications that our solution introduces no extra friction over traditional email signup, while offering fully end-to-end encrypted communication. Using OAuth it allows the same frictionless user experience that Facebook Login\textsuperscript{1} offers. However, the provider of the used identity must be a trusted party. We discussed in chapter 8 how additional levels of security can be introduced, mainly to remove the need for a trusted identity provider, at the cost of more friction.

Research question 3

*How can we ease the development of MoCoAs using an API that supports group communication?*

We presented the API and object model in Musubi in chapter 3 that exposes functionality related to group messaging, but also identity management and persistency. Its design was informed by the requirements laid out in chapter 2 and the parallel development of the Messenger application that we discussed in 3 as well. The API was further evaluated with the development of Grandma’s Trunk, shown in chapter 6, which showed that it successfully directs 80% of the development effort towards application specific user interface code.

SocialKit-JS exposes a minimally simplified version of the API to HTML and Javascript applications running in an embedded web browser. The Migo project that was discussed in chapter 5 evaluates the API. Results show that it is easy enough for high school level beginner programmers to develop fully function social applications, provided with some minimal assistance.

Research question 4

*Can our middleware meet the non-functional requirements of MoCoAs?*

We do not have an all-encompassing answer to this question. To be able to provide a full answer a formal understanding of the non-functional requirements of MoCoAs is needed, along with results from evaluations regarding these requirements. We chose here instead to pursue a qualitative answer to the question. Musubi was tested with three real-world applications: Messenger, Dispatch, and Grandma’s Trunk. The latter two were successfully used to conduct experiments in their own regard. Messenger was informally evaluated by

\textsuperscript{1}https://developers.facebook.com/docs/concepts/login/
Summary of contributions

assessing adoption after distribution to a group of 47 people. The ease of use, message transmission rate, responsiveness, and battery life turned out not to be issues. However, the need for decryption of a long queue of messages after a period without usage caused people to stop using the application. Without solutions to regularly activate the application to allow decryption of messages, this can be a barrier to adoption in certain applications.

9.2 Summary of contributions

The contributions of our work can be roughly divided between the proposed system and three case studies. We summarize the contributions as follows:

- A list of functional requirements for mobile collaborative applications (Chapter 2).
- Musubi, a middleware for top-level P2P mobile collaborative applications on top of fully end-to-end encrypted message exchange. We presented the architecture, a reference implementation, and a sample case application (Chapter 3).
- SocialKit-JS, a Javascript library and emulator that allows the construction of mobile web applications on top of Musubi (Chapter 5).

We evaluated Musubi using three separate case studies that address the problems of Mo-CoA development mentioned before. We focused on qualitative evaluation using real-world applications, rather than simulations and performance evaluations.

- Dispatch enables citizen journalism in crisis areas. It demonstrates the support for alternative routing options in Musubi. We showed how to use ad-hoc networks and human mobility to route messages from Internet-isolated areas to the Web. We also discussed how the security features of Musubi can protect journalists (Chapter 4).
- Migo showed that SocialKit-JS can be used to raise interest for computer science with high school students (Chapter 5).
- Grandma’s Trunk enabled cross-generational communication between grandchildren and grandparents, showing how larger educational applications can be built on top of Musubi (Chapter 6).

This work showed that middleware for mobile collaborative applications knows high demand and has broad applicability. The easier development of these inherently complex systems not only allows to create more applications faster, it also enables engineers to focus on the main application goal while guaranteeing the required level of quality and security.
Bibliography


Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

ACL: Access control list
AES: Advanced encryption standard
AMQP: Advanced message queuing protocol
API: Application programming interface
IBE: Identity based encryption
IBR: Identity based routing
JSON: Javascript object notation
HCI: Human-computer interaction
HTML: Hypertext markup language
MANET: Mobile Ad-Hoc Network
MITM: Man in the middle
MoCoA: Mobile collaborative application
OSN: Online social network
P2P: Peer to peer
PSN: Pocket switched network
SDK: Software development kit
SMS: Short message service
URL: Uniform resource locator
Appendix B

Migo student applications

Bucket List

Its summertime, and even though you were so excited to be out of school a couple weeks ago, you’re starting to get bored. Like really bored. Like Ohh em gee! if I watch another season of How I Met Your Mother, my brain might actually explode. We’ve all hit this low at some point in our summer, and while some of us just went right ahead into Season 7, others decided to do something about the interminable boredom. We created a list.

The Bucket List is essentially a glorified to-do list. Open a Musubi feed with some friends and let the fun begin!

Nose Goes

Have a job you don’t want to do? Use Nose Goes! Using Musubi, access the Nose Goes application via the pin of apps to begin a simple nose goes. Then press the nose and you’ll be safe. Once you start, your friends will get a notification on the Musubi feed for them to press the nose, and the feed will record each person that presses the nose. The last person to press the nose loses, and your group of friends will be notified of the results.

Game Portal

Game Portal is a simple web-app designed for Musubi that aggregates all the user data across the three gaming networks, Xbox, Playstation, and Steam, (and soon Wii-U to make four) allowing a user to Check-in or tell their friends what they are playing and on what console, at the click of a button. There is a one time set-up where the user provides his/her usernames for each network, but the application remembers the usernames thereafter to provide the most convenience for the user. Game Portal aims to bridge the gap between the networks by allowing visibility of a friends online/offline state between the networks simply through a check-in button.
B. MIGO STUDENT APPLICATIONS

Truth or Dare

Truth or Dare on Musubi introduces a new way of playing games asynchronously with your friends that utilizes the potential of social apps running on mobile devices. With the feeds available in Musubi, the Truth or Dare app allows you to begin a game with those available in the feed where truths and dares are submitted by each player and gathered into a pool. Once the game begins, all players receive a random truth or dare from the pool and they complete their challenge by typing a response or taking a picture with their phones. The progress of the game is displayed in real time on a dashboard containing all tasks completed by the players.

CrossCheck

CrossCheck is an educational web application meant for Musubi that seeks to do the impossible: make studying fun. Given a piece of text, this application will automatically generate a crossword covering the content in the piece of text. The clues for the crossword are in the form of fill-in-the-blanks, with the sentences lifted directly from the text. The clues are also filtered to ensure that only key terms are tested. CrossCheck also allows you to play the crossword against a friend. Both you and your friend compete to see who can solve the crossword faster. The twist is that players can see each others progress, a feature that adds competition and fun!