University of Waterloo

Faculty of Engineering

Department of Electrical and Computer Engineering

Distributed Instant Messaging System

Detailed Design and Project Timeline

Group #2015.010

Professor Werner Dietl (Consultant)

Qi Liu (20358515)

Asif Arman (20349964)

SangHoon Lee (20357600)

Danny Yan (20387735)

July 2, 2014

Table of Contents

1. High-Level Project Description

1.1 Motivation

Instant messaging systems of today simply cannot adequately safeguard the privacy of their users.

Users' contacts lists, profile data, and even message histories are always logged and stored on centralized servers fully-controlled by the messaging system's service providers. There is simply no guarantee that users' data won't be searched and abused by these companies.

Even if we put aside the trust issue between customer and service provider, a data breach by malicious third-parties is a dangerous, ever-present possibility on any centralized server exposed to the internet.

Furthermore, recent leaks provided by Edward Snowden on NSA's overreach in its information collection practices has highlighted the fact that governments can easily and legally force service providers to hand over any and all user data available to them.

Some believe that the widely popular instant messaging network, Skype, is can protect the privacy of its users, as it makes use of a peer-to-peer architecture. However, this has not been true of the network since 2012, when Microsoft replaced all of the decentralized supernodes in the Skype network (peers that had enough resources to act as relays for traffic between other peers) with servers under their control [1]. A quick look at the current Skype privacy policy verifies that Microsoft indeed reserves the right to collect "Content of instant messaging communications, Voice messages, and video messages" [2].

As of today, the demand for truly private instant messaging systems has yet to be met.

1.2 Project Objective

We aim to design an instant messaging system that protects the privacy of its users as an utmost priority.

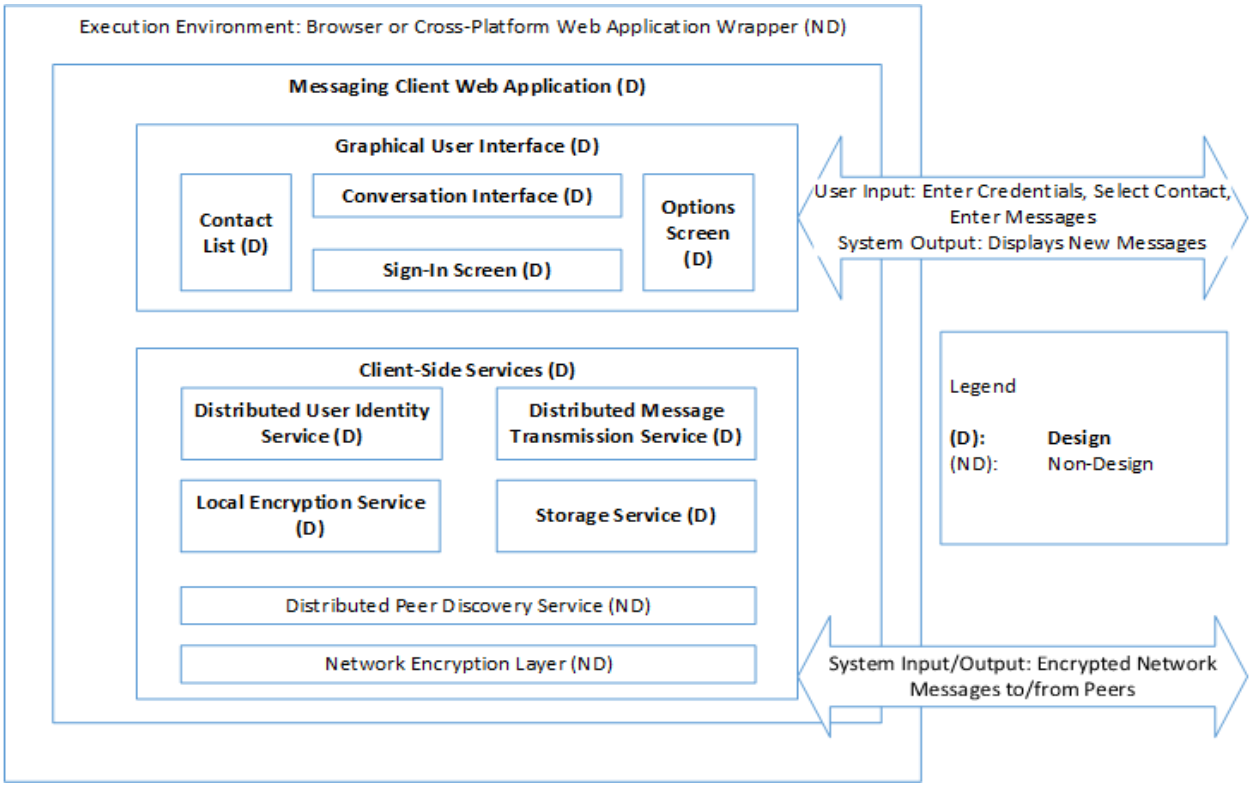
Our system will meet this objective by implementing a truly decentralized peer-to-peer architecture with no centrally controlled servers of any kind.

Messages in this system will thus travel directly from the sender to the recipient, through a completely encrypted channel, giving no opportunity for any third-party to access message contents.

User data is to be stored locally, or optionally on a cloud storage service for remote-access purposes, but always in a securely encrypted form that can only be accessed with the correct user-defined password

1.3 Block Diagram

Figure 1: Block diagram of project components and inputs/outputs



Our high-level application design, outlined in the block diagram in Figure 1, consists of a tiered architecture. The 3 tiers are the Graphical User Interface tier, the Client-Side Services tier, and the Execution Environment tier.

The topmost tier consists of the Graphical User Interface (GUI) module and its various components. This tier is responsible for handling all user input such as logging in, selecting contacts, typing messages, and changing options, as well as displaying output from the system such as new messages received or contact availability status. The GUI tier also communicates to services in the Client-Side Services tier below in order for the system to execute actions based on user input, and display information based on system output.

The middle tier consists of the Client-Side Services module and the various services that it provides to the GUI tier above it. The Distributed User Identity Service maintains the contact list and profile information for the user, the Distributed Message Transmission service sends/receives messages during

a conversation, the Local Encryption Service and Storage Service interact with storage mechanisms on the Execution Environment to persist user data between sessions in a secure manner.

The lowest tier consists of the Execution Environment module, which can be either a Web Browser, because our application is built as a Client-Side Web Application using only web technologies such as HTML and JavaScript, or a Cross-Platform Web Application Wrapper library that allows us to port the Web Application easily onto multiple Mobile platforms.

2. Project Specifications

2.1 Functional Specifications

Table 1: List of functional specifications

Functional Specification #	Essential / Non-Essential	Description
1	Essential	The user can send plain-text messages to other users and receive messages from other users in a reliable and robust manner, with no lost messages and less than 2 second delay under ideal network connectivity.
2	Essential	The application should not require the use of any centralized servers to store user data or send/receive messages to other peers.
3	Essential	Every message sent over the network should be encrypted. Messages should use at least 128 bit key for its encryption scheme.
4	Essential	Users must successfully authenticate themselves before they're able to access any private information such as the contact list or user profile.
5	Non-Essential	Contact list will have an accurate availability indicator for each contact. User can manually set his own availability status to any value when available.
6	Non-Essential	Users are able to participate in up to 10 conversations concurrently.
7	Non-Essential	Users can have access to the same contact list on every device that they authenticate into.
8	Non-Essential	The user can choose to send and display messages in real time. *See below.
9	Non-Essential	Offline message queuing will be supported. If the receiver of the message is offline (unavailable), messages can be queued, and the receiver will receive the message when he/she next becomes available.
10	Non-Essential	Group chat will be supported. Up to 4 users should be able to participate in the same conversation with no noticeable performance

		degradations.
11	Non-Essential	The user can choose to save conversation history. When a user reconnects to their account (on a different device) they will be able to view their previous conversations between the other users and groups.
12	Non-Essential	A user should be able to use the application on more than 1 device concurrently. State between each instance of the application should be synchronized to within 5 seconds of delay under ideal network conditions.
13	Non-Essential	Users should be able to search through their contact list and messages. Searching will be done in real time and filter through new messages as they arrive.
14	Non-Essential	Real time widgets will be supported that allow users to perform more than just sending messages. Examples may include a drawing widget, voice communication widget, video widget, etc.

The real-time messaging component at #8 in Table 1 was at one point an Essential specification for our application. However, we have investigated the performance characteristics of several distributed networking libraries and none of them could satisfy the performance constraints we deemed necessary for the real-time messaging experience to remain fluid and natural. In the end, we felt that real-time messaging was a relatively insignificant feature when compared to the disruptive privacy benefits that having a distributed architecture would bring. As such, we have changed the main focus of our application from real-time messaging to the distributed architecture itself. We have demoted the real time specification into a non-essential spec for the moment, but will continue to investigate performance characteristics of additional networking libraries to hopefully implement this feature in an acceptable manner in time for our final prototype.

2.2 Non-Functional Specifications

Table 2: List of non-functional specifications

Non-functional specification	Essential / Non-Essential	Description
Efficiency	Essential	Efficiency is one of the most important non-functional specifications that our system requires. The system needs to be as efficient as possible in both local and network resource usage. In terms of network usage, on average it should not surpass 10KB per 1000 characters sent/received. In terms of local resource usage, it should never use more than 50MB of memory under normal operation as a mobile or web application. Steps should be taken to minimize computational resource use on mobile systems whenever possible to conserve power.
Security	Essential	Our system needs to safeguard the privacy of the user as an utmost priority. All messaging traffic needs to be encrypted and resist tampering and eavesdropping. Only the intended recipient should be able to decrypt the messages efficiently with his private key, and be able to validate that it came from the expected sender. All local and storage need to be encrypted with a secret password provided by the user. Communication between clients must be accomplished without the use of a centralized server to store information or to route messages between them.
Dependability		

		operation.
Maintainability	Non-Essential	Our system needs to be easily maintainable so we can update the client and add features without needing to perform architecture overhauls. Our component topology needs to be laid out in a layered architecture with high degrees of separation, with automated tests covering all important features to detect regression issues when changes are made.
Portability	Non-Essential	As our system may need to run on multiple platforms, portability becomes important aspect of our system. It should be able to run on android, iOS mobile platforms in addition to the original web platform, retaining all of functional and non-functional specifications. Once we have completed the core specifications for the web application, this can be accomplished using open source libraries that wrap the web application with platform specific interfaces.

3. Detailed Design

3.1 Graphical User Interface Design

This section of the document will describe the user interactive with our final product. It will describe how the application will display its content and the technologies that will be used to achieve them.

First, our messaging application will have sliding menus. Users can slide in and out the menus, swiping from left to right and right to left. The left menu will have a login form when the user is not logged in otherwise, a user profile display will be displayed. The purpose of the login form is to authenticate the user, which is from the essential specification #6. Also, this menu will have the contact list, a scrollable field that shows all the user's contacts. Regarding to the essential specification #5, each contact on the list will have an indicator of availability. There will be a unique color to indicate the status of that contact. In addition, these categories are implemented as tabs therefore, when selecting a tab, the content of the category will be expanded to display its content. For example, if the user is viewing his contacts list, the profile tab will be collapsed and the contact list tab will be expanded. This can maximize the space usage of the screen and make the application more dynamic and user friendly. Also, within the contact list tab we will have a "search" text input field on the top of the contact list, which will filter the user input in searching for contacts. For example, if the user is searching for the name, "Arthur", the moment the user types "a" it will automatically update the contacts list, showing the contacts that only starts with "a". This can make the search more advanced and will make the application dynamic and more users friendly. The search feature satisfies non essential specification #15. Moreover, there will be another row right below the contact search bar, which consists of two buttons. The first button will be used for sorting the contacts by name. While the second button will be used to sort the contacts by groups. The two buttons allow the user to specify his/her own preferences when searching for contacts. In addition, users will be allowed to group the contacts by right clicking on contacts and clicking on the add group option. Furthermore, invites button will be added in the last row as a static row. When user clicks on it will bring up a dialog with two options: invite by email and invite by ID. Invites by ID will only work if the ID exist so when the invites by ID is pressed we prompt the user for the ID and allow the user to send invitation only when the ID is found.

Moreover, on the right sliding menu, it will have options. There will be account setting option, storage switch option, saving chat history option and real-time messaging switch option. The account setting will allow users to change their nickname, password, and other personal information. The storage setting

will allow the user to choose the storage options between cloud and local. Furthermore, the storage setting will satisfy the essential requirement #4 where the user is able to access the same contacts on every device that they sign in to. The saving chat history option allows the user to specific whether or not to save the chat history. This satisfies the non-essential specification #12. The real-time messaging switch will allow the users to choose between real time communication or not.

The sliding menu that will be implemented will use angular snap.js, which is the wrapper around snap.js that helps creating user-friendly shelves with handy styles and angular directives. By using this we can easily make the complex UI and handle the associated events in more organized manner. Furthermore, for the search mechanism, we will use jquery and regular expressions to match the strings and filter out the name of the contacts. This will make the search feature more user interactive by filtering the contacts on every update.

In addition, the main screen, conversation interface, consists of two pages. One of the page displays all the recent chats with different contacts and the other page contains the chat history of the selected contact. Each of the contact on the most recent chat page contains the contact's name and the latest message delivered or sent. Inside the chat history page, the user is able to view the previous messages sent and received by the selected contact. In addition, there will be a text input field at the bottom of the chat screen for the user to communicate with the contact. To achieve the essential specification #3, we can add multiple tabs for different contacts. The multiple tabs will allow the user to participate in multiple conversations concurrently. In the chat screen, the user's messages will be colored in a different color from the contact's messages. The user's messages will be bounded to the right side of the chat screen with a unique color tag. While the contact's messages will be bounded to the left side of the chat screen with a different unique color tag. The color tags are used for the users to easily differentiate which messages are from them and which messages are from the contact. Also, each message will have a time stamp and be grouped by days.

Furthermore, styling of our application will be done using SaSS and compass with twitter bootstrap. SaSS is basically an extension to CSS that allow us to write CSS in organized manner. It will allow us to create variables so that we do not have to memorize or look up the values such as colors or sizes that are being used repeatedly. This can also improve maintainability because when we want to update or change certain colors or sizes we just need to change the value of that variable and it will propagate to the associated components. In addition, SaSS allows us to write advanced style sheet, create nested elements, which improves the readability, and build mixins, which are functions in CSS. By using mixins

we do not need to write redundant codes. For example, there are multiple classes that require the styling attributes shown below.

```
-webkit-border-radius: 10px

Border-radius:10px

Background-clip:padding-box
```

Using SaSS, we can refactor the repeated styling attributes into a mixin and include this mixin wherever we want to apply round edges. This will save us lots of our time and can reduce the huge amount of redundant codes. Moreover, compass is basically a pre-built mixins that runs on top of SaSS and provide us the framework for SaSS. This will improve our workflow and speed up our development. Incorporating compass into our work will reduce unnecessary work since most of the mixins are already provided by compass.

Moreover, twitter bootstrap provides us with a framework that allows us to write very advanced and responsive styles. It will automatically handle the different screen sizes and it changes the layout of the screen with the respect to the screen size. Also, It has a lot of built in styles that looks professional.

3.2 Client-Side Services Design

Client-Side Services in our application will be implemented using the concept of a Service in AngularJS. AngularJS Services allows us to hide away implementation details of each Service module behind a public interface, a concept in object-oriented programming known as abstraction. Other modules (even other Services) can call upon this interface simply by declaring the Service as a dependency. Below is a JavaScript code snippet showing the declaration of the Distributed User Identity Service (named "Identity") that requires a dependency on the Distributed Message Transmission Service (named "Communication") and Storage Service (named "Storage"):

```
angular.module('rtmsgApp')

  .service('Identity', function Identity($rootScope, Storage,
Communication) {

    // Implementation of Service goes here

  });
```

A major practical benefit of using AngularJS Services, and abstraction in general, is that we can change the implementation details of some functionality in a Service without having to change the interface to that Service, so any existing code that calls on the interface will continue working as if nothing has changed at all. This enables better separation of concerns between application modules and allows for better Maintainability of the application as we will inevitably need to make changes to our Services throughout the development process (satisfies Non-Functional Specification "Maintainability" in Table 2). The Storage Service detailed below is a more concrete example of the benefits of abstraction, as it switches out the entire implementation of its interface methods between a Local Storage service and a Cloud Storage service depending on user configuration.

3.2.1 Distributed Message Transmission Service Design

The Distributed Message Transmission Service, henceforth to be referred to as Communication Service, is responsible for the core operation of our application, delivering messages from one user to another in a distributed, peer-to-peer manner without involving any centralized servers.

We have investigated two major designs for the Communication Service, one based on the Tor Anonymity Network and another based on the Telehash Distributed Hash Table (DHT) implementation.

3.2.1.1 Tor Anonymity Network

From the description on their website, the Tor Anonymity Network is a "free and open network that helps you defend against traffic analysis, a form of network surveillance that threatens personal freedom and privacy, confidential business activities and relationships, and state security". [3]

Tor attempts to achieve total anonymity of its users by encoding all source and destination internet addresses into a Tor node address ending in .onion, and routing traffic through a series of randomly selected Tor relay servers run by volunteers across the globe. All packets are encrypted with a 256-bit key in multiple layers (hence the onion on their logo), and at each hop in the series, only a single layer is decrypted as to obtain the destination address of the next hop the packet is to be routed to, and nothing else. This way, no single relay on the path from source to destination can have full information on the original source or intended destination of any packet travelling through it. Packets sent through the Tor network do not need to adhere to any specific format because it allows for the transmission of unaltered TCP/IP packets, but the extra obfuscation and encryption/decryption layers do add a non-negligible overhead to packet size and delivery speed. [3]

Figure 2 below illustrates how this process may look like for a given path through the Tor network between two clients:

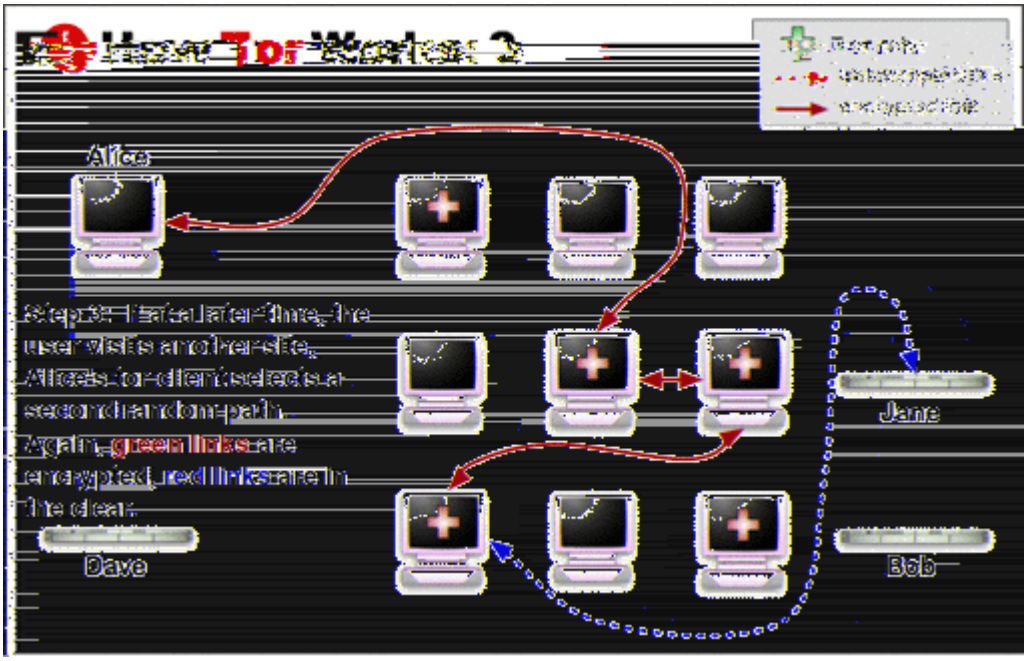


Figure 2 – How Tor Works [3]

3.2.1.2 Telehash DHT

Telehash claims to be "a secure wire protocol powering a decentralized overlay network for apps and devices". [4]

Telehash's decentralized nature is based on of a concept in distributed computing known as Distributed Hash Tables (DHTs). Similarly to a regular hash table in computer science, the DHT also provides an efficient lookup service for key-value pairs, but unlike a traditional hash table, DHTs store their key-value pairs across multiple nodes in a distributed manner, and each node is responsible for maintaining a certain portion of the key-value pairs in the table. When initiating a lookup, a node will query the DHT network for the peer responsible for the key it's searching for. [4]

Telehash provides on top of a regular DHT, an end-to-end network encryption framework using length 256 bit keys, a uniform message format, and an automatic peer-discovery and packet routing system. Each user is provided with a public/secret keypair on initialization, and an id is then generated based on this keypair that is used to identify and communicate to the user on the Telehash DHT network. Packets sent through Telehash must adhere to the JavaScript Object Notation (JSON) format, which encodes messages in a string format similar to the syntax of JavaScript object declarations. [4]

3.2.1.3 Solutions Analysis

At first glance, both Telehash and Tor are able to meet functional specifications 1, 2 and 3 in Table 1, and the Security non-functional specification in Table 2. They are both completely decentralized communication solutions that are able to carry messaging traffic as payloads from one peer to another, securely encrypted with a 256 bit key without the use of any centralized servers. However they do have some very key differences in their performance characteristics and implementation details that ultimately led us to choose Telehash as the base library to implement our Communication Service.

After setting up some initial prototypes for field testing, we have verified that Telehash generally adds less than 200ms latency on average on top of the base round-trip latency (obtained through the ping command in windows command line) for any given traffic, whereas Tor's latency overhead tends to be much higher and much more unpredictable, ranging from 200ms to 500ms. This can likely be attributed to the higher degree of origin and destination obfuscation that the Tor network performs with its multi-layered encryption setup and random routing path. No packet loss was detected for the duration of the test where network connectivity was uninterrupted, so both solutions adequately satisfy the Dependability non-functional specification in Table 2.

For the purpose of our project, we aim to provide the maximum possible privacy protection to our users, so Tor's extra protections against communication metadata monitoring is definitely desirable despite its high latency overhead. However, after some additional research, we discovered that there was an extension available for Telehash called Opaque Routing Tree (ORT) that enables similar routing obfuscation and layered encryption as Tor [4]. The latency characteristics of Telehash with the ORT extension enabled was discovered to be similarly unpredictable, and slightly worse than Tor, ranging from 300ms to 700ms. However, we believe the extra flexibility in being able to enable/disable the feature as an extension to achieve higher performance on demand is definitely a valuable feature to have as well for possible timing-sensitive extensions to our application.

In summary, the tradeoff we have to make here is between Tor's marginally better absolute performance when routing obfuscation is enabled (always, in Tor's case) versus the higher degree of flexibility that Telehash provides with its optional ORT extension. We believe the latter is more valuable to our application because it the higher performance characteristics when ORT is disabled makes real-time communication much more feasible for our system, paving the way for the successful implementation of functional specs 8, 12, and 14 in Table 1, and also potentially improving the Usability non-functional specification in Table 2.

Another aspect of our testing involved packet size overheads in addition to latency. As mentioned before, Telehash enforces the use of JSON for messages transmitted over its network, while Tor has no similar restrictions, and thus can be used with more compact binary message formats. We tested the size overhead of the Telehash JSON packets over a custom encoded binary message holding the same data (approximately of approximate length 100 characters). The overhead proved to be over 100%, with approxima1[n0l

initializing a new local user profile, which will then be populated with other information such as the user's display name and local encryption password.

```
Communication.connect(user)
```

This interface will be used to connect a given user to the Telehash network, similar to the Sign In paradigm of existing messaging systems. The user object is a JavaScript object of the user type, the structure of which is detailed in the Identity Service below. Once connected, a user will be able to send and receive messages using the Telehash network.

```
Communication.send(contact, message)
```

This interface will be used to send messages to another user on the Telehash network. The contact object is a JavaScript of the contact type. The contact object will contain among other things, the user id for that particular contact, which Telehash will use to identify peers and route messages. The message object is a JSON object containing the contents of the message. In the implementation of this service, we will use Telehash to encrypt the message and send it to the intended contact over the network.

```
Communication.receivedMessage(contact, message)
```

This interface is an event handler for handling messages received by other peers on the Telehash network. It will be called whenever a message is received, and will trigger an event in our app to be handled by the GUI layer to display a new message to the user. The user object will contain the Telehash id of the user that sent the message, and the display name of the current user. This id will be passed to the Identity service to perform a lookup against the user's contact list to fetch additional information for the contact, such as the display name. The message object is a JSON object containing contents of the message received.

3.2.2 Distributed User Identity Service Design

The Distributed User Identity Service, henceforth to be referred to as the Identity Service, is responsible for keeping track of information about the current user, as well as managing the user's list of contacts. It will require a dependency on both the Storage Service for accessing and storing persistent data as well as the Communication Service for creating a new user on the Telehash network.

We will implement following internal data structures for the Identity Service:

```
Identity.currentUser = {

     id: '...', //telehash user id for current user

     name: '...', //user's chosen display name

     password: '...', //user's chosen local encryption password

     keypair: '...' //telehash secret/public keypair used for network
traffic encryption

}
```

The currentUser data structure is an object of the user type, containing the 4 properties shown above. This object is used to store profile information regarding the current user of the application. This is passed into the communication service when calling Communication.connect().

```
Identity.contacts = {

     id1 : {  // telehash user id for contact1

           id: '...', //telehash user id for contact1

           name: '...' //display name for contact1

     },

     id2 : { // telehash user id for contact2

           id: '...', //telehash user id for contact2

           name: '...' //display name for contact2

     },

     ...

}
```

The contacts data structure is a JavaScript object with properties containing the id of the contacts and with values of the contact type, with the two properties shown above. This object stores a list id-contact pairs for the user that is used to populate the Contact List GUI module, and allow our application to efficiently map ids fetched from the storage service to the actual contact objects that the application

can process, within a constant time regardless of the length of the contact list, satisfying the Efficiency non-functional spec in Table 2.

Any contact in the list can be accessed in the following extremely computationally efficient manner:

```
Identity.contacts['id1'];
```

Whereby id1 is a Telehash user id fetched directly from the Storage Service. More details regarding the implementation of the Storage infrastructure is available below.

In addition to these internal data structures, we will expose the following interface methods for the Identity Service:

```
Identity.updateUser = new function(user) {

     this.currentUser = user;

     return Storage.save('user', user);

}
```

This method will be used to update the current user stored in the service with the new user object passed in, as well as save the user to the persistent Storage Service under the 'user' key.

```
Identity.createUser = function(user) {

      return Communication.initialize().then(function (newUser) {

        this.updateUser({

          id: newUser.hashname,

          keypair: newUser.id,

          name: user.name

        });

      }.bind(this));

    };
```

This method will be used to generate a new Telehash user id and keypair using the Communication.initialize() service method and then create a new user object using the user profile

information from the user object passed into the method. It then persists this object in the Identity Service and Storage service using the previously discussed Identity.updateUser method.

```
Identity.getContact(contact)
```

This method will be used to fetch an existing contact in the user's contact list by the contact's Telehash id.

```
Identity.inviteContact(contact)
```

This method will send an invite to the Telehash Id of the contact object, along with a specially formatted JSON message, to be handled by Identity.receivedInvite(contact) shown below. It will also add a temporary contact to the contact list that will be promoted to a full contact once it receives a similar specially formatted JSON message (acting as acknowledgement) from the invited contact.

```
Identity.receivedInvite(contact)
```

This method is an event handler for the special JSON message sent from the Identity.inviteContact() method of another client. When the special JSON message is detected, the Communication.receivedMessage() method will pass the message to this event handler instead of triggering the message received event to notify the GUI layer. This method will then trigger an invite received event to notify the GUI layer of an invite from a new contact, and allow the user to accept or decline. If accepted, this method will then proceed to call Identity.inviteContact(contact), passing in the sender of the invite as the parameter. This will complete the handshake processes and establish the two clients as contacts for each other in their respective instances of the application.

3.2.3 Storage Service Design

The storage component will be used to store the user profile for an individual user. This will handle the essential feature that allows user to access and save their contact list. The Storage component will be implemented as an Angular Module and will depend on 2 other Angular Module. These 2 services are the LocalStorage service and the CloudStorage service. The user will be able to configure whether to use LocalStorage or CloudStorage. If the user chooses LocalStorage, the users profile data will be saved locally on the browser, if the user chooses CloudStorage, the user profile will be saved server sided which allows the use of this application on multiple devices. The Storage service will support the following 4 functions as a minimum:

```
Storage.save(key, value);
Storage.read(key, value);
Storage.update(key, value);
Storage.delete(key, value);
```

This means the Storage service provides the application with key value storage. When one of these function is called, the Storage service will check to see the user configuration, if the user configuration is set to local storage, it will use the LocalStorage Service to save the data onto the local browser. If Cloud Storage is set, the CloudStorage service will be used instead to save the data server side.

To make the Storage service more flexible and easier to use, tables may also be implemented. This will make it easier when we want to save more than just contact list, such as saving messages sent and received (a non-essential feature). So there would be a Contact table, and a Messages table. The function would be:

```
Storage.save(table, key, value)
```

Also the Storage service will use the Local Encryption Service to encrypt the data before inserting it into the Local/Cloud storage. The detail of this is can be found in the Local Encryption Service section.

3.2.3.1 Local Storage Service

The local storage service is used to save the data onto the local browser. To save data locally on the users disk, HTML5 Web Storage will be used. These are similar to cookies but unlike cookies which are meant to be read on the server-side, web storage is meant to be read only on client side. Unlike cookies they do not take part in HTTP requests and have a much greater limit of 5 MB. The HTML5 Web Storage is currently available on all major web browsers.

In terms of the 5 MB size limit, if each Contact takes no more than 50 Characters as their user name, and their unique telehash Hashname is 64 characters, then each Contact requires 114 B of space. This means almost 9000 contacts can be saved if we only use 1 MB of space. There is clearly more than enough space if only contact user name and connectivity information is saved. If other data is to be saved such as message history, we can limit the size of the Contact list to 150 contacts, and use the rest of the space to store other information. For storing message history we will only keep track of the most recent messages and if the storage space is full, the new oldest message will be replaced with the newest message. Note that the final data that is actually stored will be encrypted meaning that size will actually be greater than what is specified here. We performed a variety of quick test with the encryption service

by encrypting different files with different sizes and found that the size does not increase more than 30% which makes the storage system still viable.

LocalStorage is based on key value pairs, so it is simple to build this under the Storage service. The code to set a key item using HTML5 Web Storage is simple:

```
localStorage.setItem(key, value);
```

So When Storage.save(key, value) is called, if LocalStorage service is enabled, the above line of code will be used to set the Storage service. To create the illusion of tables using web storage, the table will be saved as the key, and the key value pair will be saved as an object under the value as shown below:

```
localStorage.setItem( table,  { key : value } )
```

3.2.3.2 Cloud Storage Service

The use of local storage is limited to a single browser on a single device. To allow users to save their profile information along with the ability to synchronize profile information across devices, this data must be stored online. An alternative that was considered is to store user profile information into a database such as MongoDB or MySQL in our own hosted server. Due to large overhead and costs associated with this solution, we would like to avoid it.

The solution we ended up committing to, is to use a free cloud storage services that offers space on a per user basis. These services include Dropbox, Google Drive and SkyDrive. Each of these services have API's that provide functionality to allow users to save information by logging into these individual accounts. The main API's which is being considered are Dropbox's Datastore API [6] and Google Drive SDK [7].

The Dropbox API has been investigated and tested. The DataStore API provided by Dropbox, allows the user to store records into multiple tables. The actual tables are schema less and records inserted are JSON style objects, meaning that the data stored is similar to the key value storage used by local storage. Dropbox currently has a limit of 10 MiB for its Datastore, which should be adequate since it is twice the size provided by local web storage.

Google Drive SDK was also investigated, as they have a rich SDK that allows per user data storage. Upon comparing Drive SDK to Dropbox API, Dropbox was chosen as the preferred cloud storage solution. Drive SDK is more focused on uploading, modifying and reading files as a way of storing user data. When storing application data, Drive has a special hidden Application Folder in which we can store and read

from files [7]. Our application is more suited to Dropbox's Datastore model which is more closely related to local storage making it simpler and quicker solution to implement. Google Drive SDK, currently has a limit of 10,000,000 queries/day which is a per app quota. Although Dropbox doesn't specify the exact number, the rate limit is done per user as opposed to per app, "*The API limits the amount of calls your app can make per user. The limits are high enough that any reasonable use of the APIs shouldn't come close to hitting them*" [6].This makes Dropbox more viable due to a per user quota rather than app quota.

Implementing this the Storage services using Datastore would require a default table, since Datastore uses tables by default. When Storage.save(key, value) is called, Datastore will save the key value pair in the 'default' table as shown below:

```
var defaultTable = datastore.getTable('default');

defaultTable.insert({

    key: value

});
```

When storing data for a specific table, the .getTable() function will just use the name of the table

Adding the cloud storage component introduces a form of centralization that may seem to counter the distributed nature of our application. Although these user profiles are stored and synched from centralized servers, the nature of communication is still decentralized. Essentially if the user chooses to use cloud storage, they will retrieve their contact list information from the centralized servers. The messages sent using the contact list information will remain as direct P2P communication. Encryption will also be performed, user profile information will be encrypted before saving onto the cloud, and will be decrypted after retrieving back from the cloud. This means even if someone had access to the users Dropbox account, the data for the application would be useless to them without the correct password.

3.2.4 Local Encryption Service Design

Security is a major component that is considered for this application. The storage service is used to store data into either a local storage on the browser or server side using a cloud based service. In using either of these services, data should not be stored in plain text. For local storage, it is less of a necessity,

however if someone has access to the device, they can easily copy the contents of the local storage, so the data in local storage should be encrypted. For cloud based storage service, the only way to guarantee that the cloud service provider does not view the data is to encrypt it before saving it onto their server.

In order to perform client side encryption, the Standford Javascript Crypto Library (SJCL) [8]will be used. The local encryption service will be built under this library. This library provides the necessary functions to encrypt and decrypt the data. The default encryption scheme provided in sjcl will be used, which encrypts using AES block cipher with a key size of 128 bits; this meets our minimum specification. SJCL also adds additional security parameters such as generating a random salt based on the password and hashing over multiple iterations (1000 iterations by default).

For the encryption system to work, the user is required to have a password. When a user first sets up his/her account, the user will be requested to set his or her password. Once this password is set up, the encryption service will be used to store it. The password itself will not be stored as a variable in the Encryption service module. Instead its hash value will be stored which can be done with SJCL since it supports SHA256 hashing. Once the hashed password is stored in the Encryption service, decryption and encryption of the data can be performed. As mentioned before, the Contact list is stored within the identity service as a JavaScript Object. The contact will be stored in the Storage service in a 'Contact' table. Each key value pair in the table will refer to an individual contact. The key for the contact will be the User ID and the value will be any other information related to that specific list. Currently this value is only the User Name, but other information may also be stored such as a display picture url and whether the individual belongs in a specific group. Before this data is saved into the storage, the Encryption Service will encrypt both the key and value pair using the hashed password.

This above mechanism makes it easy to create, update, add and delete Contact list information. For example if we want to delete a specific user, we will locate it using the encrypted User ID, and delete that key value pair from the LocalStorage / CloudStorage. A problem with this implementation is the fact that everything is based on a password which is not stored server side like traditional chat system. If the user forgets his/her password, there is no service to reset the password meaning the decryption of the contact list will not work which essentially makes the app unusable. This problem can't be resolved unless we provide our own service to store the user password which imposes cost and creates a centralized component that we would like to avoid. The best we can do is to place a warning to the user that the password can't be reset if forgotten. Another problem is if the user would change their

password. This means that all the encrypted data that is stored must be read, decrypted and then re-encrypted and saved back into Storage. This may be a time consuming process, but we assume users will not change their password frequently compared to other operations.

4. Discussion and Project Timeline

4.1 Discussion on Design

A table shown below indicating the number of hours invested for each student.

	Asif Arman	SangHoon Lee	Qi Liu	Danny Yan
Meetings and Brainstorming	5	5	5	5
Finding Consultant	1	1	1	1
Project Abstract	1	1	1	1
Project Specification & Risk Assessment Document	3	5	4	3
Research	29	27	30	18
Implementation	12	15	14	9
Total	**51**	**54**	**55**	**37**

Our project and design comprises multiple components which requires the knowledge of different upper-year engineering knowledge. Working with distributed systems, encrypting messages over the network, and implementing software architectural pattern are considered upper-year engineering knowledge which are used in our design.

Distinctively, our project major component is to operate in a decentralized system. Therefore, our design will be bounded by the knowledge of distributed systems in order to achieve a decentralized system. Each peer or machine will operate as an independent entity or node in the system thus, dynamic hash tables will be required to allow communication between each node. TeleHash DHT, an open source project which we can include into our design allows us to achieve this communication.

The involvement of encryption as a upper-year engineering knowledge was because our project heavily emphasizes on the user privacy policy. Encryption is the major component of our system to allow the user privacy policy to be successful. For each message that will be sent over the network it will be

encrypted to prevent tampering and eavesdropping. TeleHash DHT enable us to achieve the encrpytion when the messages are sent over the network.

Besides features and functionalities, we have decided to organize and implement the design in a more efficient way. The software architecture of our design will include design patterns to enhance its performance and readability. To follow a good software coding standards, we have included AngularJS, a JavaScript framework in to our project to allow the usage of MVC (Model-View-Controller). MVC is a software architecture pattern for implementing the user interface by separating the internal levels of representing information.

From our current perspective, our design satisfies all essential specifications. After investigating which technologies that will be used in our project, we have carefully chosen the correct technologies to fulfill the essential specifications. Therefore, our design will satisfy all the essential specifications however, it cannot be say the same for the non-essential specifications. The non-essential specifications are mostly focused on performance for it is difficult to guarantee that our design will satisfy all of them. We believe that the design does not require any refinement before building the prototype. However, there can be changes to our design in the future if there are any unexpected obstacles that we did not foresee.

Due to our design which does not contain any electrical circuitry or medium, there are no potential hazards that we should be think about before starting to build our prototype.

4.2 Project Timeline

Start Date	End Date	Description
July 3, 2014	July 4, 2014	Compare, contrast and finalize type of cloud storage to use (Dropbox API, Google Drive SDK,SkyDrive API etc)
July 4, 2014	July 7, 2014	Implement cloud storage layer to work similar as local storage
July 4, 2014	July 7, 2014	Finish contact list implementation, including displaying availability and adding other users to contact list
July 7, 2014	July 11, 2014	Implement sign on screen to choose cloud/local storage
July 7, 2014	July 10, 2014	Implement working P2P communication between any 2 users
July 7, 2014	July 11, 2014	Use sjcl.js to encrypt user profile information (local and cloud)
July 10, 2014	July 14, 2014	Implement simultaneous conversations between users

July 12, 2014	July 16, 2014	Investigate and Implement real time component using Operational Transforms
July 19, 2014	July 22, 2014	Perform system testing and unit testing, fix and verify bugs
July 23, 2014	July 23, 2014	Demonstrate initial prototype and progress
Sept 01, 2014	Jan 30, 2015	Investigate and implement any remaining essential and non-essential features
Sept 01, 2014	Jan 30, 2015	Work on the aesthetic components and the GUI
Jan 01, 2014	Jan 12, 2015	Write and evaluate stress tests for the application
Feb 06, 2015	Feb 14, 2015	Prepare the Final Report
Feb 15, 2015	Mar 01, 2015	Finalize prototype, make sure a final stable build is live on an appropriate server
Mar 01, 2015	Mar 08, 2015	Prepare for the design symposium
Mar 19, 2015	Mar 19, 2015	Demonstrate final prototype

References

[1] D. Goodin, "Skype replaces P2P supernodes with Linux boxes hosted by Microsoft (updated)," 01 May 2012. [Online]. Available: http://arstechnica.com/business/2012/05/skype-replaces-p2p-supernodes-with-linux-boxes-hosted-by-microsoft/. [Accessed 01 June 2014].

[2] Microsoft, "Skype Privacy Policy," 2014. [Online]. Available: http://www.skype.com/en/legal/privacy/. [Accessed 01 June 2014].

[3] Tor Project, "Tor Project," Tor Project, 2014. [Online]. Available: https://www.torproject.org/. [Accessed 01 July 2014].

[4] J. Miller, "Telehash," 2014. [Online]. Available: http://telehash.org/. [Accessed 01 July 2014].

[5] Adobe Systems, "PhoneGap," Adobe Systems, 2014. [Online]. Available: http://phonegap.com/. [Accessed 01 July 2014].

[6] Dropbox Inc, "Dropbox," 2014. [Online]. Available: https://www.dropbox.com/developers/datastore. [Accessed 01 July 2014].

[7] Google Inc, "Google Drive SDK," 2014. [Online]. Available: https://developers.google.com/drive/web/about-sdk. [Accessed 01 July 2014].

[8] Stanford University, "SJCL," 2014. [Online]. Available: http://bitwiseshiftleft.github.io/sjcl/. [Accessed 01 July 2014].

[9] T. Simonite, "New Scientist Blogs," 07 December 2007. [Online]. Available: http://www.newscientist.com/blog/technology/2007/12/instant-message-irrelevance.html. [Accessed 01 June 2014].