CarrierPigeon

iOS Online Messenger With P2P Offline Relaying to Recipient

Chris D'Angelo	Ifeoma Okereke	Riley Spahn	Abdullah Al-Syed
Columbia University	Columbia University	Columbia University	Columbia University
cd2665@columbia.edu	iro2103@columbia.edu	rbs2152@columbia.edu	aza2105@columbia.edu

ABSTRACT

We have created a messaging application for iOS with the added capability of relaying messages peer to peer to a recipient while the sender is without a network connection. The use case for the app describes its purpose best. Suppose Elaine would like to send a time sensitive text message to Jerry while she is in the subway that she will be late to dinner. Elaine does not have an internet connection and will not have one until it is too late to reach Jerry. Elaine uses our application named CarrierPigeon to draft and send a message while offline. A fellow CarrierPigeon user, named Kramer, sitting next to Elaine, silently receives Elaine's message so that he may "carry" the message to Jerry should he exit the subway and gain access to a network connection first. In essence every user of CarrierPigeon becomes a kind stranger that can carry your message onto your recipient. This relaying of messages is done by default and it is not an option in the application to turn this feature off. The relaying of message is done seamlessly from a user's perspective. The "carriers" have no knowledge of the content of the message being forwarded.

1. INTRODUCTION

With the rapid adoption of smartphones and other communication devices has come the expectation of being seamlessly connected to everyone, anytime and anywhere. Unfortunately, for a number of both technical and policy reasons, this is not always possible. Oftentimes, it is not cost-effective for cell phone networks to provide coverage in certain locations such as remote areas or underground such as on the subway. This poses a challenge for users who have come to expect communications technology to be at their fingertips. A business-person traveling on a subway who is running late for an important meeting may not have the time to wait to get a cellphone signal to send a message. In cases like these, it would be useful to have someone else transmit the message on behalf of the original sender. This intermediate user's device, which serves as a relay in this case, may obtain a data connection earlier than the sender and, hence, could deliver the message on behalf of the latter. Against this backdrop, we propose an application that will run on iOS devices and which will be able to transmit text messages on behalf of users. The app will also be able to perform many other functions similar to a traditional messaging app as well as have certain limitations, which are described in later sections.

In many ways CarrierPigeon resembles the stock iOS Messages application or another messaging application such as WhatsApp. CarrierPigeon contains all the typical features one would find in an application for messaging. For clarity we enumerate the current feature set of CarrierPigeon:

> Sign In, Log-In, Sign-Up, Sign-Out, Add Friend, Accept Friend, Search Contacts, Search messages, View Messages, View Contacts, Create Message, Receive Message, Send Messages, View Meta Data on Message (See Date Sent, Date Received, To, From, etc), Messages Presented in iMessages Style Bubble UI, Change Password, Network Access Notification, In App Notifications of Message Arrival, Offline Access to Messages and Contacts, Messages

Outbox, Video Background Log-In, Git tag/commit Auto Increments Version Numbers, Offline Holding of Delivered Messages, Licenses and Credits Screen, Push Notifications, Retrieving Old Messages*, iPad Version*

*Partially implemented. These features may be viewed in the source code and/or are available but contain bugs.

In addition to the standard features of a messaging application we enumerate the following novel features implemented in CarrierPigeon. We will describe these features subsequently in greater detail.

Network Indicators of Peer to Peer Connection, Multipeer Connectivity Handshake Logging, Only Use Pigeons Option (turn off connection to XMPP server and only use fellow peers to relay messages), Counter for Messages sent Directly, Counter for Messages sent via Pigeons, Counter for Messages delivered by Pigeons*, View Messages in Outbox, View Messages in Outbox Carrying for Others, Seamless Peer to Peer Bluetooth/Wifi Handshaking via Multipeer Connectivity Library, Messages in Outbox are Sent to Peers (up to 5) on Pull-to-Refresh Contacts.

*Partially implemented. These features may be viewed in the source code and/or are available but contain bugs.

At the time being the above novel features are implemented with complete transparency for all users including the carriers. In an application built for release in the Apple App Store many of the features that provide information to the user regarding the status of "carried" messages would be hidden from view. In fact, messages that are being carried would be encrypted so the carrier has no possible knowledge of the message they are carrying. The current design is in place only for testing and transparency of the process.

This paper is organized as follows. We begin with a short overview of the use case and a preview of our implementation. We then enter a discussion of the implementation of the iOS Messaging app in the use case where the sender and recipient are both online and connected to a server which passes messages like any standard instant messaging service. We then discuss the steps taken to provide the novel functionality of peer to peer relay of messages to the end recipient. In addition we offer an overview of our development process and the tools we used to bring CarrierPigeon to reality. Finally, we provide an explanation of the outstanding issues in order to bring an application such as CarrierPigeon to market including but not limited to: battery life implications, security implications, and consumer features not yet implemented. We also offer a brief overview of related work in this area.

2. ONLINE MESSAGING

Before we delve into the details of how we implemented a multi-user online messaging application on iOS we would like to first provide an overview of the CarrierPigeon use case once more in context of the general architecture of CarrierPigeon. This architecture of course encompasses the capabilities for online and offline peer-to-peer message relaying.

In the standard messaging use case where both the sender and the recipient are online the message is sent via an XMPP stream. In the case where the sender is offline a message is kept in an outbox. If that device sees another device within range the message in the outbox will be relayed to that nearby stranger. When the stranger gains access to the internet (if they haven't already) the

message will be delivered to the XMPP server with the information from the original sender. The XMPP server mutates the message and delivers the message to the recipient as if it came from the original recipient. Figure 1 describes these two use cases.

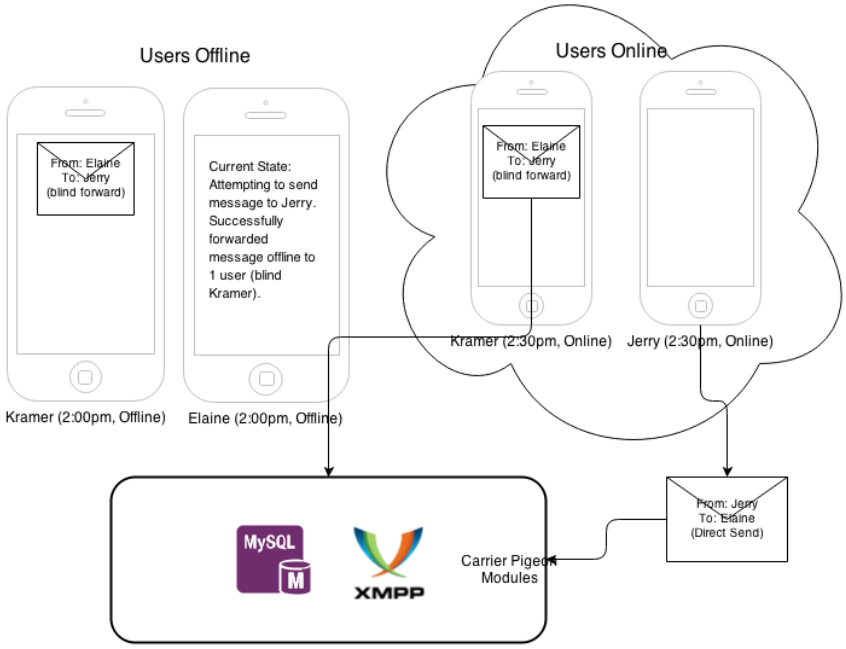

Figure 1: CarrierPigeon's two use cases and architecture overview.

2.1.1 XMPP

Introduced by Jeremie Miller in 1998 as Jabber and formalized by the IETF in 2004, the extensible messaging and presence protocol (XMPP) is a commonly used XML based communication protocol that was until recently in use by companies such as Google and Facebook to support large scale public facing chat applications. Though it is commonly used as a chat protocol, XMPP is a protocol for sending XML "streams" over a network. These XML streams are a set of XML "stanzas" exchanged by parties, client and server, over the network. Streams are delineated by the "stream" tag and define one connected session. XMPP streams are not bidirectional; in other words, for client-server communication, there will be one stream from client to server and and another stream from server to client [1].

XML stanzas are the units of communication used by the client and the server. XMPP uses three standard stanzas: message, presence, and iq. The message stanza is the most straightforward of the stanzas and defines a message that is to be sent from one client to the server to be distributed to one or more other clients in a chat, group chat, or chat room. In addition to the message body, the message stanza also supports metadata such as message subjects and message threads. In Figure 2, we show an example of an XMPP message stanza below where Elaine is sending a direct chat message to Jerry. Presence stanzas are used by XML to notify the server of a client's state. The states may be that the client is available, unavailable, or currently composing a message. The IQ stanza is a general mechanism to support queries from the client to the server. Example uses of this stanza are capability or client discovery. XMPP was an ideal foundation for CarrierPigeon because there are many existing libraries and servers and the XML format make extending XMPP for CarrierPigeon trivial [2].

```
<message type="chat" from="Elaine" to="Jerry" id="17">
      <body>Hello</body>
      <active xmlns="http://jabber.org/protocol/chatstates">
      </active>
</message>
```

Figure 2: A standard XMPP Message Stanza with a CarrierPigeon special attribute *id*. Elaine sends a message to Jerry. Both Sender and Recipient are Online

2.1.2 Ejabberd Server

We built CarrierPigeon's server side portion using the the ejabberd XMPP server. ejabberd is a modular fully standards compliant XMPP distributed server written in Erlang. Erlang is a functional programming language with roots in the telecom industry. It was developed by Joe Armstrong at Ericsson in 1986 as a solution for long running, low latency distributed systems. Its functional nature discourages any shared state between the lightweight processes and code hot swapping makes it possible to introduce new modules or reconfigure the system without ever bringing the system down. According to a presentation by the creator, Joe Armstrong, Ericsson used Erlang to achieve nine nines of reliability or 99.9999999% up time.

CarrierPigeon uses ejabberd as the central management hub. All user management, messaging, and logging goes through ejabberd. Ejabberd considerably reduced the pain of deploying CarrierPigeon because we did not need to write from scratch any message routing or authentication. We augmented standard ejabberd with numerous existing modules. We moved from using the internal Mnesia database to the more standard MySQL. Mnesia, the default ejabberd database, is a distributed databases written in Erlang that is common in Erlang applications. We chose to use MySQL instead because we have experience managing RDBMSs and there are many commonly available management tools. We also augmented the standard ejabberd server using mod_archive to provide message archiving and mod_odbc.

2.2.1 Client Architecture

2.2.2 XMPPFramework

CarrierPigeon's iOS client was implemented using the XMPP Framework [21].The XMPPFramework provides an Objective-C implementation of the XMPP standard (RFC 3920) [1], popular XMPP Extensions [20] and the tools to needed to read and write XML. The XMPPFramework was structured using GCD, is parallel and thread-safe. The XMPP Core implements XMPP RFC 3920 specification (XMPP Stream, JID, Parser, Element, IQ, Message, Presence, etc) while the Extensions include roster support, automatic reconnect, Message Archiving, XMPP Ping and others.

The XMPPStream class of the XMPPFramework is the class that is primarily used to interact with and integrate XMPP's extensions. Before any communication can be done with the XMPPFramework, the XMPPStream must be authenticated using the SASL mechanism supported by the server.

3. PEER TO PEER TO SERVER MESSAGING

Before we dive into the details of how we implemented the novel features of CarrierPigeon, it is necessary to understand the details of the stages a message goes through in any case where the sender and recipient are not actively online at the time the message is sent and received.

Figures 3 and 4 illustrate how CarrierPigeon handles outgoing messages from the client and incoming messages to the server and client. When a user is about to send a message, the iOS client

first checks that the user is online to determine how the message should be sent. If the user is online, the message is sent through the server normally. When the server receives this message, it creates a message receipt and sends it to the sender to indicate that the sender's message has been received. The server then proceeds to deliver this message to the recipient. When the recipient receives this message, the iOS client creates a message receipt and sends it to the sender of the message.

However, if the user is offline, the iOS client checks to see if peers are available to help the sender relay the message. If peers are available, the iOS client, using Apple's Multipeer Connectivity Framework [10], relays this message to a maximum of five peers who would deliver the message if and when they have internet connection. A server will recognized a relayed message because unlike a typical message it contains a "reallyFrom" and "reallyFromId" tags. The server alters the message's "from" and "id" attribute values to the "reallyFrom" and "reallyFromId" attribute values respectively and delivers the message to the recipient normally as if there was no intermediary. If a user receives a message with the "reallyFrom" and "reallyFromId" attributes, it forwards the message to the server. Message handling in the iOS client is primarily handled by the CPMessenger class [18].

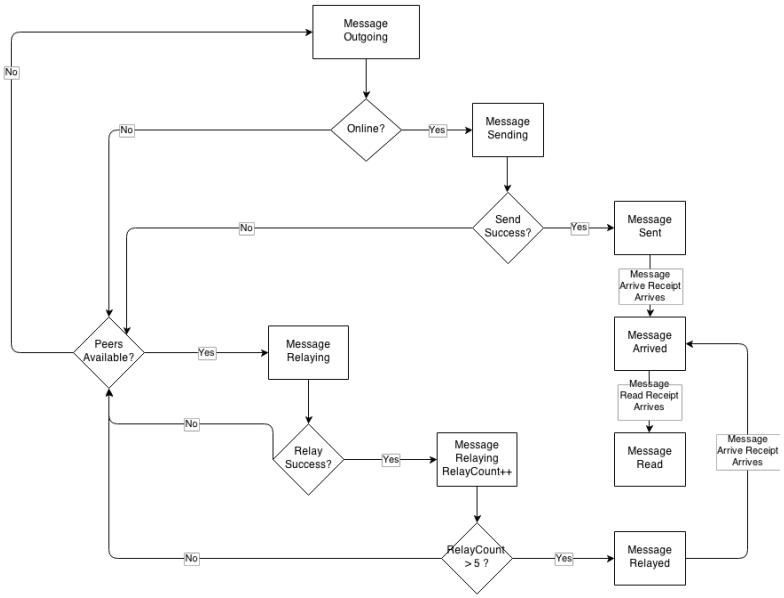

Figure 3: Message Lifetime for Outgoing Messages (Client)

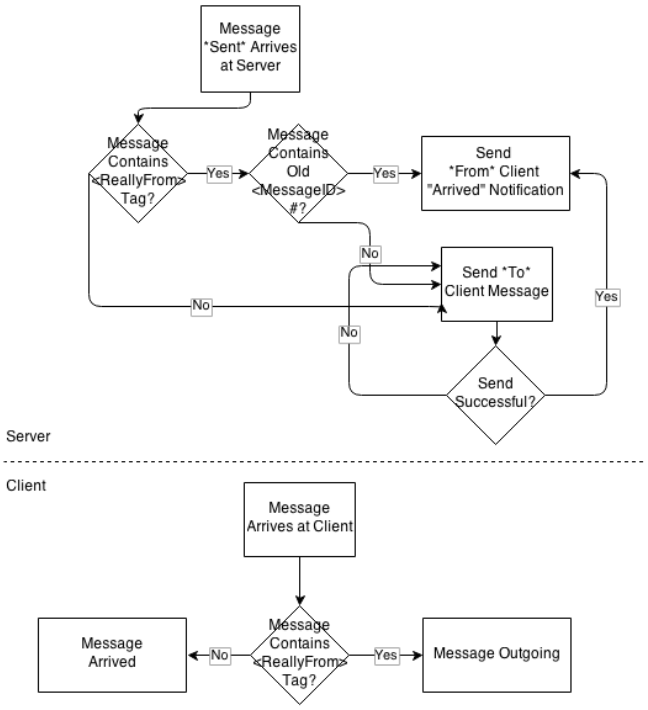

Figure 4: Message Lifetime for Incoming Messages (Server & Client)

3.1 Server Architecture

3.1.1 Server Side Forwarding

During CarrierPigeon message forwarding, it is unlikely that the carrier will be in close enough proximity to the destination client to deliver the message before the original sending client connects to a network and is able to send the message itself. To handle this most common case CarrierPigeon must facilitate a method for the carrier to deliver the message to the destination client without direct interaction and it must accomplish this with minimal overhead beyond the original service. To facilitate indirect delivery we introduce mod_bot_relay and the CarrierPigeon augmented message stanza show in Figure 5.

mod_bot_relay is an additional routing module written in Erlang that we integrated into ejabberd to handle server side message forwarding [16]. mod_bot_relay uses the new “reallyFrom” tag to rewrite the message stanzas for forwarding and will use the new “id” and “reallyFromID” tags to perform server side message deduplication in conjunction with the “messageID”. This purpose of message deduplication is to avoiding delivering the same message twice. This is relatively straightforward though online messaging where you have one source and one recipient but becomes more complex when introducing multiple carriers for the same message. mod_bot_relay deduplication is not yet implemented and will be completed in the near future. mod_bot_relay acts in addition to and not in lieu of the standard ejabberd router. mod_bot_relay sits behind ejabberd’s XML stream server and intercepts each stanza after it is parsed. If the stanza is an iq or presence stanza then mod_bot_relay simply passes it through without further examination. If the stanza is a message stanza then mod_bot_relay tests if the stanza has CarrierPigeon’s additional “reallyFrom” tag. If the tag exists then mod_bot_relay will rewrite the packet replacing the original from tag with the user and host parsed out of the “reallyFrom” tag. An example of a message arriving at the server from a carrier is show in Figure 5.

mod_bot_relay performs all of these actions with negligible overhead. We evaluated mod_bot_relay using two Python SleekXMPP workers. One worker sending messages and one worker receiving messages. To control for network variation we ran both workers on the same server on which the server ran. Normally mod_bot_relay introduces a few milliseconds of overhead. In any case the overhead from mod_bot_realy is dwarfed by the variations in latency that we see under normal circumstances and will go unnoticed by any user. Part of the reason mod_bot_relay performs well is that it is integrated with ejabberd's existing XML parser and manipulates Erlang data structures rather than reparsing XML. Figure 6 shows the results of our latency evaluation where both standard and relayed messages see a similar latency but with wide variations expected from a network service. Figure 6 shows a box plot latencies we observed when sending standard messages and relayed messages.

```
<message type="chat" from="Kramer" to="Jerry" id="17"
reallyFrom="Elaine" reallyFromId="3">
      <body>Hello</body>
      <active xmlns="http://jabber.org/protocol/chatstates">
      </active>
</message>
```

Figure 5: A standard XMPP Message Stanza with 3 CarrierPigeon additional attributes *id*, *reallyFrom*, and *reallyFromId*.

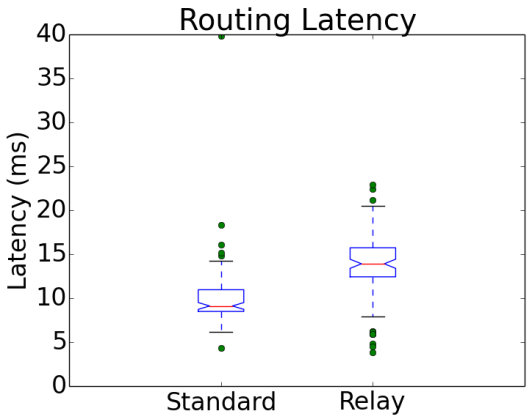

Figure 6: Latency Evaluation

3.1.2 Push Notifications

Push notifications are a service introduced by Apple in iOS 3.0 as a way for developers to send notifications from a remote service to the client device's notification center. Push notifications provide a standard abstraction layer that removes the need for significant portions of poorly implemented application background daemons that would otherwise be continuously polling remote servers and draining battery life. Push Notifications consolidate much of this functionality in an efficient central API to simplify application development with the added benefit of standard notifications across all applications and extended battery life. CarrierPigeon and other similar messaging applications use push notifications to inform the local application that unread messages are available to be fetched from a server [12].

CarrierPigeon's push notifications service was implemented using an ejabberd module and a PHP script [19]. The ejabberd module (mod_push_notifications) was programmed to send

notifications to a CarrierPigeon user when a new message for that user was received on the server (when the user was both online and offline). Because of CarrierPigeon's ability to allow multiple users on a single phone (via the sign in and sign out features), the push notifications PHP script was programmed to update the database with the user's username whenever a user signs in or signs out.

3.2.1 Multipeer Connectivity

Multipeer Connectivity is a new framework introduced in iOS 7 to facilitate peer-to-peer discovery and communication between iOS devices over Bluetooth, a wireless network, or over peer-to-peer wifi. Multipeer communication supports message, data, and sharing of files and other resources. CarrierPigeon utilizes device discovery to find nearby carriers, establish a session and begin passing relevant XMPP stanzas for the “carrier" to forward. The relevant XMPP Stanza is stored in a NSManagedObject which is converted to an NSDictionary which is converted to an NSData object which is sent via Mulitpeer Connectivity. The carrier unpacks the object according to that protocol and stores the message until that carrier gains access to the network. CarrierPigeon utilizes multipeer connectivity in a fairly straightforward way but the service is flexible enough to build other novel applications. There are other less obvious applications such as disseminating application and operating system updates through peers or providing a method for communication over Bluetooth when cellular service is unavailable [11]. In the CarrierPigeon iOS project the class responsible for Multipeer Connectivity communication is handled in the CPSessionContainer class [17].

4. IMPLEMENTATION DETAILS

This kind of multi-user application requires actual hardware device testing for the peer to peer messaging. We built an iPad version of the app specifically so that we could use our existing devices in concert with other devices. It is also possible to use the Apple’s Multipeer Connectivity library with a device and the desktop simulator. The majority of testing was performed over an actual device and the simulator signing in and out users on the device. This was simultaneously beneficial because it was critical that we correctly built the capability for each device to completely sign in and out users on a single device. Of course no testing is as useful as actual users. We created a beta distribution website [3] to provide easy installation to beta testers. TestFlight, a popular beta testing distribution service, had been acquired by Apple [4] during our development period. As a consequence, it seems that TestFlight stopped allowing new accounts therefore we required to produce our own solution based on the terrific guide written by Jeffrey Sambels [5].

Chris D’Angelo led development on the iOS platform and created the presentation. Ifeoma Okereke was very helpful in her assistance in iOS development and push notification server functionality. Riley Spahn led server deployment and implementation of novel CarrierPigeon XMPP relay module. All members contributed to the paper. Full team meetings occurred weekly in person and over Google Hangout. iOS Application code and server code was shared over GitHub. Documents were written in Google Docs. The Asana web application was used as our task manager. CocoaPods was used to manage external libraries used in the iOS project. iMovie and Screenflow was used for demonstration videos [6]. Apple’s Keynote was used for the creation of the presentation [7]. Adobe’s Photoshop was used for all iconography creation. External libraries used within the application can be viewed within the application under the rightmost tab.

We provide the iOS application completely open source. The server implementation we will keep closed source as it contains sensitive key information. The iOS project [8] and the ejabberd modules written to handle forwarding [16], push notifications [19] and to add server timestamps to message stanzas [22] are available on GitHub .

5. FUTURE WORK

5.1 Security and Privacy Concerns

We have not addressed many of the privacy concerns that are inherent in passing personal information through an untrusted third party. An adversary can read messages they are carrying, drop messages, or at worst they can alter messages before sending them on. We have discussed many solutions for handling such problems and some of them may be impassable to achieve certain levels of service.

To implement peer delivery where a carrier delivers a message directly to the recipient the intermediary must be able to read at least some metadata from message. In the future we should be able to provide message privacy without any loss of service using asymmetric encryption. When signing up for the service Elaine can generate a public key that the CarrierPigeon service can maintain and distribute to Jerry when he would like to send Elaine a message. Encrypting the body of the message will stop Kramer from knowing its contents while she is carrying it out of the subway but it does not her from modifying the message before passing it on. It is likely impossible for us to stop Kramer from modifying the message but we can at least alert Elaine to potential modifications by including with the message either a hash of the unencrypted message or implementing a digital signature scheme.

This would require us to implement more key management on the CarrierPigeon service. Implementing these features would require a significant and complex key management implementation that was out of the scope of a semester project. None of these solve the problem of Kramer dropping messages that she is carrying from Elaine to Jerry and we do not believe there is a reasonable technical solution to this problem. We have explored but not implemented reward system where users are punished for dropping messages by limiting the number of messages that they can send over carrier.

5.2 Moonshot Features

Some advanced features that we may incorporate in the future include the ability to send photos as well as text messages, allowing app users to view how much data they have sent to their own recipients as well as data they have sent on behalf of others, allowing users to view their data sharing activity in more of a gaming aspect, allowing users to opt out of being “carriers” if they have done their fair share of relaying messages in a given amount of time.

Another possible feature that we could include is an artificial intelligence-based system for frequent carriers of messages. The idea behind this would be to investigate the possibility of carriers *delivering* messages to users offline. This would involve predicting where a user will be so that we can send a recipient's messages to peers that we predict will be in the recipient's location when the latter goes offline.

5.3 Battery Life, Online/Offline Listeners, xmppStreamDidDisconnect:withError:

Thus far in CarrierPigeon development we have assumed that the user has the app open at all times. In practice, this is not true. We made this affordance because Multipeer connectivity will only operate when an application is running. In the future, to make CarrierPigeon available when it is not open we intend to use iBeacons. iBeacons and the location manager framework provide callbacks that will be triggered when moving between an iBeacon region. iOS will launch the application and allow the application to execute to appropriate delegate functions. There are negative battery implications to require the application to constantly wake up and send broadcasts as a potential carrier. We could use various heuristics to limit the number of broadcasts such as limiting the number of broadcasts while network connectivity is available because, presumably, there will be few or no other users requesting a carrier.

Another limitation that requires user interaction is that we do not have an automated way for the application to be notified when the device regains network connectivity. Ideally we would like iOS to provide a callback for the operating system to wake up an application when the network is reachable but it currently does not. Additionally iOS does not allow network polling as one of it's supported background tasks. Instead of polling for network connectivity we could also have a background task that simply sends the messages over the network when available. Unfortunately, Apple allows network fetch but not network push operations.

5.4 Features in Development

We have other features either planned or in development. First among them is implementing the features to protect against the security and privacy concerns mentioned in section 5.1. We would also like to implement picture messaging in CarrierPigeon. This will require us to significantly augment our current infrastructure because XMPP is not suitable for large media transfer. We would need to implement an an out of band service to actually transmit photos. The current CarrierPigeon implementation is also missing a form of archive retrieval so right now there is no way for the client to populate itself with chat history. This is possible because we currently store all sent messages on the server and is in development on the "ImplementingArchiving" branch in our git repository.

Another feature nonessential the core functionality but is technically interesting is peer to peer direct messaging where a carrier can deliver a message directly to the recipient. Even though this feature is of technical interest it's unlikely to be a useful because the chance of a Carrier being within close proximity to a user before getting network access is low. However, peer delivery would be useful during periods when networks are unreliable or inaccessible during an event such as a conference. During those periods of network outage multi hop peer delivery where one carrier can pass a message onto a second carrier will also be useful.

We also have two user facing features that will improve the usability of the system. First is message read receipts where CarrierPigeon will notify the message sender when the recipient actually opens and reads the message. Second is allowing the user to upload an image to use as a profile picture. This may be problematic because it will require us to police images that users upload which is difficult to do in an automated manner. We will also finish features that we have partially implemented such as removing the user facing Multipeer Connectivity diagnostics, and an iPad version.

6. RELATED WORK

6.1 Delay-Tolerant Networking Research Group

The Delay-Tolerant Networking Research Group ("DTNRG") is a research subgroup of the Internet Research Task Force that focuses on the architecture and protocol design necessary to support communications in environments where end-to-end connectivity cannot be assumed such as disaster response, military and underwater sites as well as environments where Internet performance is degraded such as many parts of the developing world [13]. DTNRG's research evolved from related work that was carried out by other groups in the field of interplanetary space communications and is considered to be a generalization of that earlier work.

As part of their research efforts, the DTNRG have commenced work on DTN-bone, which is a project to set up a global set of nodes running DTN bundle agents and applications. These nodes assist with remote management and control, interoperability, application deployment and testing, as well as operations across administrative boundaries; they are not meant to be a substitute for internal research test beds. In addition to DTN-bone, this research group has published a number of RFCs and Internet drafts that build on their work.

6.2 Firechat

Firechat is a recently released app from Open Garden that allows smartphones to connect directly without requiring an Internet connection or even cellular network coverage. Users, who only have to select a username, can communicate with other nearby users through a local chatroom. Designed for iOS 7, Firechat makes use of Apple's Multipeer Connectivity Framework, and boasts a host of features such as support for instant messaging, photo sharing, both local conversations as well as conversations that are not restricted by distance limitations. According to the developers, the app's battery consumption is low. Firechat runs on the iPhone, iPad as well as iPod touch devices, and it has added support for the Android platform recently as well.

Currently, devices using Firechat use their Bluetooth or Wifi radios to communicate with other devices nearby. While it is currently unsupported, future use of mesh networking technology, in which messages can piggyback on other devices, could enable Firechat users to communicate with other users who are not in their immediate vicinity [14]. Open Garden has said that it plans to release an SDK in the future that will support such multiplayer peer-to-peer connectivity [15].

Firechat's main limitation at the moment seems to be its lack of support for user identification and authentication. Users can communicate anonymously or masquerade as someone else without there being any sure-fire way for them to know who they are actually communicating with. While this may be advantageous in a "chat room"-like scenario, it poses challenges when a particular user wants to communicate with another user and does not wish for others in the chat room to see that message.

7. CONCLUSION

CarrierPigeon serves as a demonstration of an applications that provides the user connectivity where none exists. Creating a consumer ready application that is bug free is no small feat. There is much work to be done to make this concept work perfectly in the Apple App Store but this outline is an excellent beginning. The application uses an extremely reliable XMPP server ejabberd to serve users both completely online and uses XMPP server in combination with Apple's Multipeer Connectivity framework to usher messages from clients peer to peer. The application has been real world tested and serves well as a typical messaging client with added offline capabilities. It currently is a prototype but with more development time it is certainly something that could be sent to the Apple App Store. We believe that Multipeer Connectivity Framework is still an untapped resource. We believe that CarrierPigeon presents a novel application of the framework and one that is necessary in certain marketplaces. Ultimately, this same idea could easily be extended to platforms outside of iOS.

REFERENCES

[1] RFC 3920- http://tools.ietf.org/html/rfc3920
[2] RFC 3921 - http://tools.ietf.org/html/rfc3921
[3] http://www.christopherdangelo.com/carrierpigeon
[4]
http://9to5mac.com/2014/02/21/apple-acquires-beta-testing-platform-testflight-through-burstly-purchase/
[5] http://jeffreysambells.com/2010/06/22/ios-wireless-app-distribution
[6] bit.ly/carrierpigeondemo
[7] bit.ly/carrierpigeonpresentation
[8] http://github.com/mychrisdangelo/CarrierPigeon
[9] http://www.erlang.org/

[10]
https://developer.apple.com/library/ios/documentation/MultipeerConnectivity/Reference/MultipeerConnectivityFramework/Introduction/Introduction.html
[11] http://bgr.com/2014/03/24/ios-7-mesh-network-feature/
[12]
https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG
[13] https://sites.google.com/site/dtnresgroup/home
[14]
https://www.yahoo.com/tech/firechat-network-free-chat-could-be-big-and-now-its-81597433839.html
[15] http://opengarden.launchrock.com/
[16] https://github.com/rlyspn/mod_bot_relay
[17]
https://github.com/mychrisdangelo/CarrierPigeon/blob/master/CarrierPigeon/CPSessionContainer.m
[18]
https://github.com/mychrisdangelo/CarrierPigeon/blob/master/CarrierPigeon/CPMessenger.m
[19] https://github.com/IRuth/CarrierPigeonPushNotifiCations
[20] http://xmpp.org/xmpp-protocols/xmpp-extensions/
[21] https://github.com/robbiehanson/XMPPFramework
[22] https://github.com/IRuth/mod_server_timestamp