Subscription Email

Final report

Student	David Lowe
Module	CM0343
Credits	40
Year	2012/2013
Project supervisor	Dr. Frank Langbein
Moderator	Prof. Ralph Martin

Abstract

Email is one of the most widely used applications of the Internet, but it suffers from two flaws in particular: insecurity and unsolicited bulk email. Various other online mail systems have been developed and adopted, but none of them provide authentication and confidentiality, while eliminating spam and keeping email's decentralised nature. PigeonMail is a protocol that offers these features, its design and its background were discussed in an interim report. In this final report, I give the specification of the protocol and I discuss a prototypical implementation of a PigeonMail server and client developed for this project. I then evaluate both the protocol and the prototype, based on the goals set out in the interim report, and I list any improvements that could be made in the future. I conclude with a reflection on learning and the benefits this project has personally brought me.

Table of Contents

Illustration Index

> *Note: Throughout this report, when referring to a generic human being, the third feminine singular person is used. It should be understood by the reader that the gender of the person referred to is intended to be neutral.*

Introduction

There are many online mail systems in use today. Some, such as email, are so popular as to be considered by some a cornerstone of the Internet. Others, such as Diaspora, are less well-known but offer advantages such as security and privacy. In my interim report, I explored four such online mail systems in detail: email, Facebook private messing, Diaspora private messaging and Stubmail. None of these online mail systems offer all of decentralisation, security and protection from spam at once. I saw the need for a new online mail system, naming it PigeonMail.

As discussed in the interim report, PigeonMail is comparable to email in many ways, with a few crucial differences. The first one is that the responsibility of storing messages relies on the sender's server, rather than the recipient's. A recipient must deliberately request messages from a sender's server to receive them. This means that sending messages to strangers is in effect impossible. One must first convince a user to add one to her white-list of contacts before being able to send messages to her. This completely eliminates almost the entirety of unsolicited bulk email, if not all.

The second crucial difference is that end-to-end encryption is expected and required. All messages are signed and encrypted using OpenPGP keys, meaning that the hosting servers cannot read the contents of those messages, only the intended recipients. In addition, all communication occurs over a TLS connection, further protecting the user from eavesdroppers.

With these two differences in place, the design of PigeonMail achieves what no other well-known online mail system can: decentralisation, security and protection from spam at once. This comes at a cost of being incompatible with existing online mail systems, such as email, and at the cost of disallowing communication between users who have not added each other to a white-list.

Seeing as the PigeonMail design did not need to maintain compatibility with existing online mail systems, I was at liberty to add some other differentiating features integral to its design. For instance, a PigeonMail user is identified by the fingerprint of her OpenPGP key, and not by a username and host name, as in email. The location of a user's server is not determined from her identifier, but by an entry in a public database of user to location map. Users are expected to assign names to their contacts, rather than rely on a published name, in order to avoid forgery attacks, following Stiegler's petnames system (2005). And finally, the unit of communication in PigeonMail is a conversation, not a message: every message is necessarily linked to a conversation, and all the messages in a conversation are viewable by the same people and hosted on the same server.

In this report, I present the specification of PigeonMail's protocol. This specification should be detailed enough to allow the reader to develop a compatible implementation. (Of course, it does not include the specifications of dependency technologies, such as TCP, TLS and OpenPGP, as they have been specified and standardised elsewhere.) I then follow this specification with a description of my prototype of a server and a client. I evaluate both the design of the protocol, and my implementation of it, leading to suggesting potential improvements to both. I end with a reflection on what I learnt during the accomplishment of this project during my third year of my Computer Science bachelors of science degree at Cardiff University.

Specification of the protocol

The protocol expects two agents with distinct roles: the server and the client. The role of each can be summarised as follows: the server stores conversations on behalf of one or more users, and the client simply fetches conversations from one or more servers to display them to the user.

The protocol maps onto the TCP/IP model in the following way:

1. It is completely oblivious to the link layer.
2. At the internet layer, it requires IP, either IPv4 or IPv6.
3. At the transport layer, it requires TCP.
4. At the application layer, it is built on TLS version 1.

TCP/IP is a very popular and obvious choice for most protocols designed to run over the Internet. In order to connect to the Internet, all devices must have a IPv4 or IPv6 address, so it would be foolish to attempt to use an addressing layer other than IP.

TCP provides a reliable stream connection: that is, it guarantees to deliver data in order and without duplication, or report an error. The alternative to TCP is UDP, and while it does not provide the same guarantees as TCP, it does offer performance benefits resulting from being connectionless. TCP was chosen over UDP for two reasons: first, for the guarantees that it provides that are suitable for an asynchronous online mail system, and second, for its compatibility with TLS[1].

TLS is the successor of SSL. It is a cryptographic protocol that provides end-to-end privacy for two peers. Each peer can provide a certificate proving the authenticity of its identity. TLS and SSL are very popular and trusted protocols, as they provide the basis of the security of HTTPS on the web. Because of its popularity, TLS has been examined by security experts and by malicious individuals many times. Also because of its popularity, multiple open source TLS libraries have been made available, simplifying the development of a TLS application. For these reasons, TLS was chosen as a dependency of this protocol.

PigeonMail's protocol is thus the layer on top of a stack of layers: TLS, TCP and IP.

One option would have been to depend on another layer in addition: HTTP. HTTP is a popular and well-understood protocol, used in particular by the web, but also for many other services and APIs, due to its extensibility and flexibility. (REST and SOAP are two architectures that use it, for example.) It is particularly suited for CRUD applications, a criteria PigeonMail matches. Its flaw is that its client authentication functionality (through either HTTP Auth or cookies) could not be extended simply enough to authentication through OpenPGP. For this reason, I decided that PigeonMail would not depend on HTTP. I discovered late in the project that TLS does in fact provide client authentication, even with OpenPGP, so this flaw of HTTP proved itself to be irrelevant.

Unlike UDP, TCP does not provide a message stream protocol. The TCP simply provides a protocol to send and receive bytes, it does not provide a mechanism to indicate the beginning or the end of logically grouped sequence of bytes. To overcome this, I wrote a very simple type-length-message protocol. Had HTTP been one of the dependencies of PigeonMail, this protocol would have been unnecessary, as the border between messages is defined in HTTP.

1 It is worth noting that Datagram Transport Layer Security (DTLS) provides communication privacy for UDP, as TLS does for TCP.

1 Type-length-message protocol

A PigeonMail frame[2] is the unit of communication between a server and a client in the PigeonMail protocol. Every frame is composed of a type header, at least one length header and at least one message.

- The **type** header is one byte in length. The protocol currently specifies these possible values for the type header:

Value (hex)	Meaning
01	Normal message, serialised in JSON, optionally signed with OpenPGP
02	Exception, serialised in JSON
03 to 0f	Reserved for future protocol improvements

- This header is then followed by one or more section, where each section is composed of:
  - a **length** header, indicating a non-negative integer in two bytes in big-endian order. If the value is 0, this indicates the end of the message, if the value is 256^2-1 , then this indicates that there will be at least one more section to be read.
  - a **message** field, containing the subsequent number of bytes of the message as indicated by the length field

For example, the string `"Hello world"` is encoded in ASCII by the following sequence of bytes:

```
22 48 65 6c 6c 6f 20 77 6f 72 6c 64 22
```

This sequence of bytes is 13 bytes long. We prefix the string with a type header, selecting `01` (meaning a normal JSON message), and with a length header `00 0d`. Because the string is only 13 bytes long, we do not need to follow this section with other sections. The resulting byte sequence is:

```
01 00 0d 22 48 65 6c 6c 6f 20 77 6f 72 6c 64 22
```

2 Request and reply format

Every request sent by a client to a server is always signed with OpenPGP. The signature is never detached, but included in one OpenPGP message. For example, this is a request to fetch all conversations owned by this particular fingerprint:

```
{
    "command": "get_all_conversations",
    "owners_fingerprints": ["6302 39CC 130E 1A7F D81A  27B1 4097 6EAF 437D 05B5"],
    "nonce": 294877488240439398709888049514396189207
}
```

The client then signs this request, and generates an OpenPGP formatted message with both the signature and the contents of the request. This message may be in binary OpenPGP format, or in ASCII armoured format, as in this example:

```
-----BEGIN PGP MESSAGE-----
Version: GnuPG v1.4.12 (GNU/Linux)
```

2 Please do not confuse a PigeonMail frame with a frame at the data link layer of OSI model. PigeonMail frames are always on the application layer.

```
owGbwMvMwMT4OO3GioenNP4xrtmTxJSRGdgYfaiaSwEIlJLzc3MT81KUrBSU0lNL
4hNzcuKT8/PKUouKE0sy8/OKlXQg6vLL84Bi8WmZeempRQVFmXklxUA90UpmxgZG
CsaWzs4KhsYGrgqGjuZuCi4Who4KCkbmToYKJgaW5gpmro5uCibG5i4KBqZOpkqx
UCPz8vOSU4GGGFmaWJibm1hYGJkYmBhbGltamBtYWlhYGJhYmhoCBcwMLSyNDMy5
ark65rAwMDIxsLEygXzAwMUpAPOYaAcLw/Z1a2Q3t8/QnM685nyb8DyRn+LF771P
l3B7z92nUGx4gK/mcAi7Tkqu4HQfzqnCD88dkfpxyCBBaeavpm+ujGwaR7ON1zrf
kFeT39bC+6T/vqphSspdwZXLf+3f8PWWbqBB+pqTYcfnbXg1WefGsyunHK4EaZ6p
O574eb7yqzviugvZfid2aQIA
=sNRd
-----END PGP MESSAGE-----
```

This OpenPGP message is then wrapped in a PigeonMail frame, by inserting the prefix of three bytes `01 02 40`.

The server does not need to sign its replies using OpenPGP, since its authenticity is already assured by TLS and the server's certificate. Some sections of its reply (such as the contents of a PigeonMail human-readable message, or the title of a conversation) may be encrypted and signed with the corresponding author's OpenPGP key, guaranteeing that these sections are not readable even by the server.

3 Request-reply protocol

The protocol uses a simple request-reply pattern, with exceptional breaks in this pattern. Unlike traditional HTTP[3], many request and reply pairs can be included in a single TCP connection.

The client initiates the TLS connection with the server. For this, it will need the IP address (or the domain name) of the server, and it will need a port number. It will also need to to trust at least one certificate authority and have its public key available. The version of TLS used should be 1.0 or later.

As the TLS connection is being established, the client must validate the X.509 certificate of the server against its list of trusted certificate authorities. If the server's certificate is not validated by any of its certificate authorities, it must reject the connection to prevent man-in-the-middle attacks. This model is very identical to that of the web between servers and browsers, and was chosen for that reason.

The client does not need to provide a certificate during the TLS handshake (even though TLS does provide this functionality).

The client will then send to the server the following sequence of bytes (shown in hex), wrapped in a PigeonMail frame of type “normal” (see page 7):

```
50 69 67 65 6f 6e 4d 61 69 6c 30 2e 31
```

This represents “PigeonMail0.1” in the ASCII encoding. This is to allow future versions of the protocol to be developed. This technique is very similar to HTTP, which also sends a version header with every connection (but on every request, rather than just every connection).

When the server receives this version header, it will return a PigeonMail frame of type “normal”, containing a JSON structure representing a simple object with one key `nonce`. Its value is a random

3 Techniques to allow multiple HTTP request and reply pairs to be communicated over a single TCP connection have since been developed, known as “HTTP pipelining”.

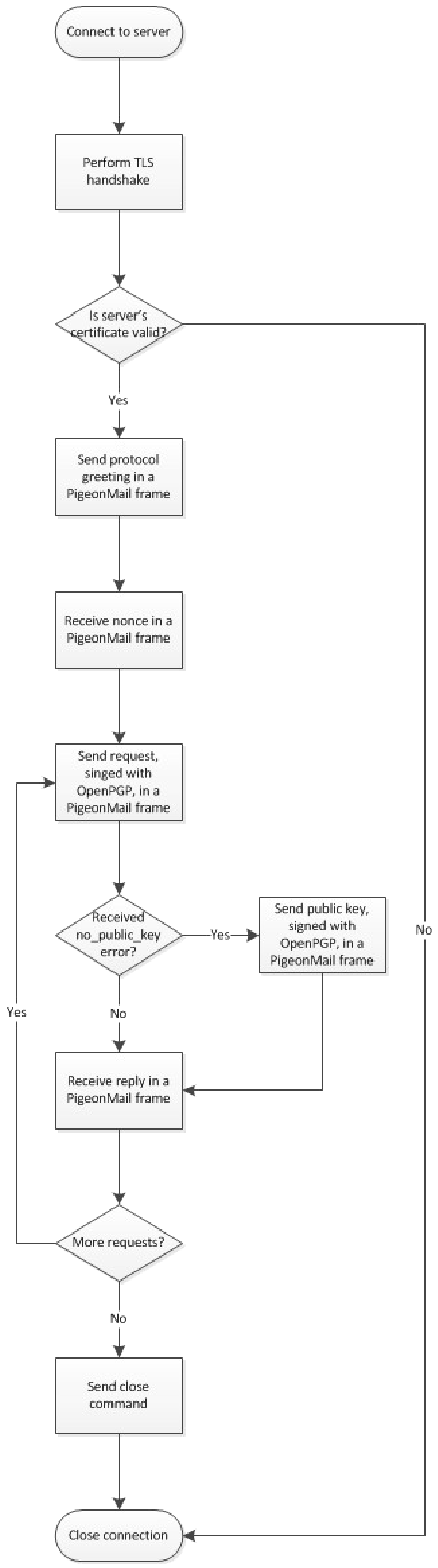

Connect to server
Perform TLS handshake
Is server's certificate valid?
Yes
No
Send protocol greeting in a PigeonMail frame
Receive nonce in a PigeonMail frame
Send request, singed with OpenPGP, in a PigeonMail frame
Received no_public_key error?
Yes
Send public key, signed with OpenPGP, in a PigeonMail frame
No
Receive reply in a PigeonMail frame
More requests?
Yes
No
Send close command
Close connection

Illustration 1: Flowchart for request-reply protocol non-negative 128-bit integer chosen by the server.

After receiving the nonce value, the client will send a PigeonMail frame containing an OpenPGP signed JSON structure. The JSON structure must be an object[4] with the keys `command`, `nonce`, `host` and `port` at minimum. The server will reply with a PigeonMail frame containing a JSON object, assuming it recognises the command sent, that the `nonce` value is indeed the one it set during the handshake and the `host` and `port` values match the address of the server.

The client may then proceed with any request, and wait for another reply. It may repeat this request and wait for reply process as many times as needed.

After the client has issued all of its requests and received all of its replies, the client will issue a special `close` command. Both the server and the client will then close the TLS and TCP connection.

All PigeonMail frames sent by the client to the server, with the exception of the initial version declaration, must contain JSON objects with the correct `nonce`, `host` and `port` values.

If at any point, the server is not able to verify the signature of the OpenPGP signed data because it lacks the singer's public key, it must return a PigeonMail frame of type “exception”, with the following JSON structure:

```
{
    "error": "no_public_key"
}
```

The client should then re-issue its request, including in its JSON object the key `public_key`, with its value being its exported public key. The server can then extract the JSON structure from the OpenPGP message (a public key is not needed for this), import the declared public key and verify the signature using the newly imported public key. If the verification fails once again for the same reason, the server should return a PigeonMail frame of type exception, with the following JSON structure:

```
{
    "error": "no_public_key",
    "closing": true
}
```

It should then close the underlying TLS and TCP connection.

The `host` and `port` values in each request protect the user's security. Without these values, the server could copy a signed request and send it to another PigeonMail server, masquerading as that user.

However, the `host` and `port` values on their own are not sufficient. Should a server ever change hands (such as during an acquisition), the previous administrators of the server should not be able to masquerade as users using stored copies of old signed requests. Because the server only changed hands and not address, the `host` and `port` values must remain the same. The `nonce` value ensures the request has been made for that specific connection.

The `nonce` value on its own would not be sufficient, as a malicious PigeonMail server could set the `nonce` value to be identical to that set by another server. In could forward any requests to the second server, gaining unauthorised access to the user's information.

JSON was chosen for a serialization format for several reasons:

4 What is known as an “object” using JSON's terminology corresponds to a map, a hash-map or a dictionary in other languages.

- It is quite expressive, not only can it represent key-value maps and ordered lists, but also a hierarchy of such values.
- It supports Unicode strings well, unlike Message Pack, which supports binary strings.
- It is schema-less: any JSON-encoded structure may be deserialised without requiring a schema defining the expected structure. This is important to ensure extensibility of the protocol: servers and clients should be able to set new key-value pairs, expecting incompatible peers to simply ignore those pairs. This is impossible to do without synchronising schemas in serialisation formats such as Protocol Buffer and Thrift.
- Because of its simplicity, it maps very well to native objects and structures in many programming languages, leading to good interoperability.
- It is quite popular, with JSON libraries existing for dozens of programming languages.
- It is a much simpler format than XML, whose advanced features are not needed for this application.

The disadvantages of JSON are primarily its space inefficiency and the performance penalty of serialisation and deserialisation. Because of this, PigeonMail has been designed to be extensible enough to use other serialisation formats instead, such as Protocol Buffer, Thrift, Message Pack, Blink or Cap'n Proto. The type field in the PigeonMail frame header could specify values for these formats. Message Pack in particular seems particular suitable, as it is schema-less like JSON.

This request-reply pattern is a very common pattern for applications on a network. One way of describing it would be to view the requests as procedure calls, made remotely from a client to a server. There are multiple Remote Procedure Call (RPC) protocols in use today, many of which have been generalised for multiple applications, for example Thrift, ZeroRPC and Java RMI. Instead of adopting one of these RPC protocols, this new PigeonMail specific protocol was designed instead, for two reasons:

- Many of these RPC protocols are programming language specific, whereas PigeonMail should be agnostic to the programming language used by the client or the server.
- None of the RPC protocols examined natively supported authenticating the client using OpenPGP.

The second reason manifested itself to be irrelevant once I discovered that TLS does support authentication using OpenPGP, but at the time of making this choice, I was unaware of this fact.

4 Conceptual units

To understand the PigeonMail protocol, a few conceptual units must be defined. These conceptual units were introduced in the interim report, but the upcoming definitions will be more detailed and will include implementation specifics.

User

A user represents a human being or a group of human beings who wish to communicate with other users through PigeonMail.

Every user must have exactly one OpenPGP secret key. In cryptographic terms, the OpenPGP secret key contains a private key used for creating digital signatures and certificates (known as a “primary key”), and at least one another private key for decrypting messages (known as a “subkey”).

From an OpenPGP secret key, an OpenPGP public key can be generated. It contains the corresponding public key used for verifying digital signatures and certificates, and all the public subkeys used for encrypting messages intended for that user. It also contains a digital signature proving the link between the primary public key and the public subkeys.

An OpenPGP public key can also contain user identifiers (UID). A UID can contain the user's real name, and optionally an email address and a comment. The OpenPGP public key will contain at least one certificate linking the primary key to the UID (known as a self-signature), and may contain other certificates generated by other users claiming that the UID is an accurate name and description of the secret key holder.

Similarly, an OpenPGP public key can also contain photo identifiers (PID). A PID is preferably a recognisable portrait photo of the user's face. As with UIDs, a PID should be self-signed, and the public key may contain certificates from other users.

Every OpenPGP key is uniquely identified by a fingerprint, which is cryptographic hash of the public primary key. It follows therefore that every user is uniquely identified by the fingerprint of her OpenPGP key. The user is free to add subkeys, revoke subkeys, add UIDs, revoke UIDs, add PIDs and revoke PIDs without changing her fingerprint. However, if she loses access to the primary key component of her OpenPGP secret key, she can no longer use PigeonMail with the same fingerprint. Because an OpenPGP fingerprint is resistant to pre-image and second pre-image attacks, it is highly unlikely that two users could ever share the same fingerprint without sharing the same secret key.

Every user must choose to associate herself with at least one publicly accessible server, identified by a fully qualified domain name (or an IP address), and a TCP port number. This identifier is called the location of the user. As discussed in the interim report, having a permanently online server is important for ensuring asynchronous communication, even when a user's client device is off-line.

From a usability point of view, the concept of a “user” should not be unfamiliar to most users. Users need only discover and learn the following pieces of information, and a good user interface would introduce these ideas gradually:

- A user is globally uniquely identified by a fingerprint.
- Every user is located at a domain name and port number, but these locations are not necessarily permanent.
- If a user loses access to her OpenPGP secret key, she loses access to her PigeonMail identity.
- Every user can publish names and photos of themselves, but other users are not obliged to use those names or photos themselves.

Message

A message is a piece of text, with meta-data attached. Its meta-data includes an ID, its author's fingerprint, its date of creation, the conversation ID for which it was intended, the fingerprint of that conversation's owner and the position of the message in the conversation.

In the prototypical implementation, the piece of text is always in plain text format encoded in UTF-8. However, there is in theory no barrier to implementing an extension allowing rich text in messages.

In order to interoperate, different PigeonMail server and client implementations do not need to agree on the format of storing messages, they simply need to agree on the format of exchange of messages, which is specified in the upcoming protocol specifications.

The textual piece of data will always be encrypted and signed when exchanged. This means that servers do not have access to the contents of the message, but they do have access to some of the meta-data such as the author's fingerprint and the conversation ID. While this reduces users' privacy, it does allow the server to deny access to unauthorized users, saving bandwidth and protecting it from denial-of-service attacks.

Every message is uniquely identified by the following four-tuple: *(conversation owner's fingerprint, conversation ID, message author's fingerprint, message ID)*.

From a usability point of view, a message is very similar to SMS text messages, email messages or Facebook private messages. It should not be a very difficult concept for most new users to grasp.

Conversation

A conversation is a group of zero or more messages ordered linearly. It also contains meta-data, including an ID, a title, a creation date, a shared secret, its owner's fingerprint, and a set of participants. Each participant is identified by a fingerprint, and is assigned read and/or write permissions to the conversation.

Similarly to messages, different PigeonMail server and client implementations do not need to agree on the format of storing conversations, but only on a format of exchanging information about conversations. The title and the shared secret of the conversation will always be encrypted, so that the server does have access to it. It does have access to the other meta-data, for similar reasons as discussed for messages.

When creating a new conversation, the client of its owner randomly generates a small amount of random data to use as a shared secret key. It then encrypts the secret key using the public keys of the recipients, so that only the recipients can decrypt it to read the shared secret key. The shared secret key is then used as a pass-phrase to symmetrically encrypt and decrypt the title of the conversation, and any messages added to the conversation. (The title of the conversation is also digitally signed by its owner, and each message by its author.)

A shared secret for the conversation may seem at first to be redundant, but it's necessary to allow adding new participants at a later stage. If the title of the conversation and the contents of each message were simply encrypted asymmetrically using the public keys of the recipients, any recipients at a later stage would be unable to decrypt old messages, nor verify their authenticity. Each message author in the conversation would have to re-upload a new encrypted and signed message, which may not be possible if those users' clients are not online. A shared secret for each conversation solves this problem, although it does increase the size of each conversation by a few hundred kilobytes, depending on the size of the shared secret.

A conversation and all of its messages are always stored on its owner's server. A conversation always has exactly one owner, and that owner has the privilege of setting the conversation's title, adding and removing participants, and granting and revoking read and write permissions to the conversation. A participant with the read permission can view the conversation and its message, while a participant with the write permission can also add new messages to the conversation.

It is important to understand that while a user must have added the conversation's owner as a contact to see that conversation, she does not have to have added all of its participants. One consequence of this choice is that it is easy to introduce one contact to another by simply creating a conversation in which both are participants. They will both be able to see each other listed as participants in that conversation, and may add each other to their respective contact lists if they so choose.

Every conversation is uniquely identified by the following two-tuple: *(owner's fingerprint, ID)*.

From a usability point of view, a user must come to understand these ideas:

- A conversation is a linear sequence of messages, all of which are viewed by the same set of users.

- Every conversation has one and only one owner.
- Every conversation has a title (unlike email, where every message has a subject).
- A conversation has a set of participants, some of which can only read messages, and some of which can add messages. This set of participants can be modified by the owner of the conversation.

5 Commands

Every PigeonMail server must implement the following commands. Each command takes a JSON object as input, and returns a JSON object. In the notations to follow, an item in italics indicates a place-holder.

It should be noted that in addition to the data provided in the JSON structure, the server also receives the fingerprint about the user performing the request, as well as the IP address from which the connection is made.

Every input JSON, in addition to the keys specified below, must also provide values for the following four keys:

- `command`: this will be the name of the command being requested (eg: `get_all_conversations`)
- `nonce`: this will be the nonce value required by the server
- `host`: the hostname or IP address of the server
- `port`: the port number of the server

In addition, the key named `public_key` is also reserved, its value is the OpenPGP public key of the client's user. It should be set after the server has returned a `no_public_key` error, and it should also be set for the `create_account` command.

A date or datetime is represented in this protocol by a JSON string consisting of the date serialized in the ISO 8601 format. If a timezone is not specified, the date is assumed to be in the UTC timezone. For example, the 30th of April 2013 at 17:55:54 UTC is represented as in this example:

```
{"date": "2013-04-30T17:55:54+00:00"}
```

A fingerprint must be represented in this protocol by a JSON string consisting of a sequence of 40 hexadecimal digits. The digits A to F may be represented in both upper-case or lower-case forms. The string may additionally contain zero or more space characters in any position in the string.

For legibility, it is recommended that a space be inserted between every grouping of four digits, as in this example:

```
{"fingerprint": "4592 D909 1CCC 12E9 0A0F 5627 0D13 6CB1 8CE9 798B"}
```

A conversation ID or a message ID must be represented in this protocol by a JSON string consisting of 32 hexadecimal digits, and it must match the published UUID 4 specification, that is, the 13th digit must be 4 and the 17th digit must be one of 8, 9, A or B. The digits A to F may be represented in both upper-case or lower-case forms. The string may additionally contain zero or more hyphens in any position in the string.

For legibility, it is recommended that a hyphen be inserted after the first eight hexadecimal digits, another one after the following four digits and one other hyphen after each following grouping of four hexadecimal digits, as in this example:

```
{"uuid": "2afba813-899e-4b02-87aa-8ad8af40b360"}
```

Even though conversation IDs and messages IDs match the specification of UUID 4, there are not meant to be universally unique. Rather, the tuple *(owner's fingerprint, conversation ID)* uniquely identifies a conversation, the ID on its own is not sufficient as a malicious user could deliberately choose an ID identical to that of the victim's. A conversation's ID should be randomly generated to avoid leaking any information. (For example, using an incremental counter for conversation IDs would leak a lower bound of the number of a user's conversations) Similarly, each message is identified by the tuple *(conversation's owner's fingerprint, conversation ID, message author's fingerprint, message ID)*.

Command `get_all_conversations`

This command fetches meta-data for all the conversations viewable by the client's user that are stored on that server and that are owned by the specified owners.

If no owners are specified, than only the conversations owned by the requester shall be returned.

If one or more of the owners specified do not have conversation storage privileges for that server, this command will simply ignore those owners. To determine which server to query for conversations from a particular owner, the client should use the location commands, not this command.

The meta-data returned for each conversation includes the conversation's ID, its encrypted title, the date of its most recent message, the fingerprint of its owner, and the encrypted secret of that conversation.

To decrypt the title, the client must decrypt the secret using its OpenPGP secret key, and then decrypt the title using a symmetric decryption algorithm, using the secret as a password. The specific asymmetric and symmetric encryption algorithms used for the secret and the title respectively are not specified, but they both must be in the OpenPGP format, which includes the algorithm name in its meta-data. Clients are advised to use the strongest algorithm available that is supported by GnuPG.

Input JSON:

```
{
    "owners_fingerprints": array_of_fingerprints
}
```

Example input:

```
{
    "owners_fingerprints": ["4592 D909 1CCC 12E9 0A0F  5627 0D13 6CB1 8CE9
798B", "5396 3A64 D28B 0800 05BF  3027 DC5E C5C9 0746 DFF2"]
}
```

Output JSON:

```
{
    "conversations": array_of_conversations
}
```

`array_of_conversations` is an array of zero or more objects in this format:

```
{
    "id": conversation_id,
    "owner_fingerprint": owner_fingerprint,
    "title_encrypted": title_encrypted,
    "date": date,
    "secret_encrypted": secret_encrypted
}
```

Example output:

```
{
    "conversations": [
        {
            "id": "2afba813-899e-4b02-87aa-8ad8af40b360",
            "owner_fingerprint": "4592 D909 1CCC 12E9 0A0F  5627 0D13 6CB1
8CE9 798B",
            "title_encrypted": "-----BEGIN PGP MESSAGE----- \nVersion: GnuPG
v1.4.11 (GNU/Linux) \n\nhIwD6kiFkFTr+i4BA/9s4yLwgFAoyvrM7...\n-----END PGP
MESSAGE-----\n",
            "secret_encrypted": "-----BEGIN PGP MESSAGE-----\nVersion: GnuPG
v1.4.11 (GNU/Linux) \n\nhIwD6kiFkFTr+i4BA/9I79z...\n-----END PGP
MESSAGE-----\n"
         }
     ]
}
```

Command `get_conversation`

This command fetches meta-data for the conversation, including a list of identifiers for the messages it contains, in order, and a list of participants, in no particular order.

The contents of a conversation's messages are not returned, and must be fetched individually using the command `get_message`. This is to allow more aggressive caching: if a conversation has been updated with new messages, the client need only download the new messages, not the old ones.

Input JSON:

```
{
    "owner_fingerprint": fingerprint,
    "id": message_id
}
```

Output JSON:

```
{
     "success": true,
     "title_encrypted": title_encrypted,
     "owner_fingerprint": owner_fingerprint,
     "secret_encrypted": secret_encrypted,
     "participants": array_of_participants,
     "messages": array_of_messages,
     "date": date
}
```

If the command fails for whatever reason, the following structure is returned:

```
{
    "success": false
}
```

The `title_encrypted`, `secret_encrypted` and `date` values are identical in meaning as the ones explained in `get_all_conversations` (page 15).

The `array_of_participants` is an array of zero or more objects matching this schema:

```
{
    "fingerprint": fingerprint,
    "read": boolean_value,
    "write": boolean_value
}
```

The `array_of_messages` is an array of zero or more objects matching this schema:

```
{
    "author_fingerprint": fingerprint,
    "id": message_id,
}
```

Command `get_message`

This command fetches the contents of a particular message, as well as some meta-data: the owner of the message and the date it was originally posted.

The returned contents of the message are encrypted symmetrically using the secret included in the meta-data of the conversation. The client therefore needs to have the results of the `get_conversation` command stored or cached at some level.

Input JSON:

```
{
    "conversation_owner_fingerprint": conversation_owner_fingerprint,
    "conversation_id": conversation_id,
    "author_fingerprint": author_fingerprint,
    "id": message_id
}
```

Output JSON:

```
{
    "contents_encrypted": contents_encrypted,
    "author_fingerprint": author_fingerprint,
    "date_posted": date_posted
}
```

Command `add_conversation`

This command creates a new conversation and stores it on the server. It is expected that most servers would only allow a particular set of users to perform this operation, perhaps users that have paid for a subscription to a service, or who have previously agreed to the service's terms and conditions.

The client must specify the conversation's ID, it is not specified by the server. This is to allow a client to migrate conversations from one server to another without changing the conversation's ID, if the user so desires. If the conversation's ID clashes with an already existing conversation ID owned by the same user, the server must return an exception.

Input JSON:

```
{
    "id": conversation_id,
    "title_encrypted": title_encrypted,
    "secret_encrypted": secret_encrypted,
    "participants": array_of_participants
}
```

The array_of_participants is an array of zero or more structures of this format:

```
{
     "fingerprint": fingerprint,
     "read_permission": boolean_value,
     "write_permission": boolean_value
}
```

Output JSON:

```
{
     "success": true
}
```

Command `add_message`

This command creates a new message and adds it to the specified conversation. The server will store the message with the passed encrypted contents and the specified date, and it will append it to the existing list of messages. This means that messages in a conversation are sorted in the order in which they were received by the server, and not in the order of date piece of meta-data, as these may be maliciously incorrect.

Input JSON:

```
{
     "conversation_owner_fingerprint": conversation_owner_fingerprint,
     "conversation_id": conversation_id,
     "id": message_id,
     "contents_encrypted": contents_encrypted,
     "posted_date": posted_date
}
```

Output JSON:

```
{
    "success": true
}
```

Command `update_participants_for_conversation`

This command enables the user to add or remove participants from a conversation, or modify the permissions assigned to each participant.

It is important that if any participants are added that the same secret be uploaded, but encrypted for all the intended participants, including the new ones. The secret must not change, so that the existing participants are able to read new messages using their cached secret, and so that new participants can read existing messages. The old value of `secret_encrypted` would be useless to new participants, as they are unable to decrypt it.

If participants have been removed, the plain value of the secret should not change. This means that if the removed participants gain access to the new messages, they will be able to decrypt them. Therefore, it is critical that the server deny access to removed participants. While this is problematic, one cannot simply change the plain value of the secret, as the old messages would then become undecipherable by new participants. Instead, a history of encrypted secrets would have to be kept, a feature not currently designed in the protocol.

Input JSON:

```
{
    "owner_fingerprint": owner_fingerprint,
    "id": conversation_id,
    "secret_encrypted": secret_encrypted,
    "participants": array_of_participants
}
```

The `array_of_participants` should conform to the same schema as `array_of_participants` in `add_conversation`.

Output JSON:

```
{
   "success": true
}
```

Command `get_location`

Before a client can fetch conversations owned by a particular user, the client must first be able to determine the location of that user's server. To do so, it may query as many PigeonMail servers as it wishes using this command, `get_location`. The most recent and cryptographically verified location record should be taken as authoritative.

The Dove implementation of the client queries all known PigeonMail servers starting with the user's until it finds the location of a contact. Even after this exhaustive search, if it still does not discover the contact's location, it will prompt the user to enter her server's hostname and port number, as a last resort.

Input JSON:

```
{
    "fingerprint": fingerprint
}
```

Output JSON:

```
{
    "signed_data": signed_data,
    "public_key": public_key
}
```

The `public_key` value is the exported OpenPGP public key of the specified user, as stored on that particular server.

The `signed_data` value is the location record, together with its signature. Once the signature has been validated, the record can be extracted, and it should be a JSON structure matching this schema:

```
{
    "creation_date": creation_date,
    "expiration_date": expiration_date,
    "host": host,
    "port": port
}
```

The client must verify that the signature matches the record, and that the signature is from the fingerprint requested. It should also check that the `creation_date` is not in the future, and that the `expiration_date` is in the future or none. Once this is done, it can then adopt the host and port values as the authoritative values for the location of the requested user. If the client already has a previously fetched location record, it should use the one with the most recent creation date to establish the location of the user.

Command `update_location`

Using this command, users may publish the hostname and port number of the server they trust to store their conversations on as many PigeonMail servers as they wish. The more servers that hold an up-to-date location record, the more likely contacts will be able to find an up-to-date location record quickly.

Input JSON:

```
{
    "signed_data": signed_data,
    "user_public_key": public_key
}
```

Output JSON:

```
{
    "success": boolean_value
}
```

The `signed_data` structure should be the same as that for `get_location` (page 19). The fingerprint of the OpenPGP key signing the location record does not need to be the same as the fingerprint of the client's user. In other words, the client may submit location records on another user's behalf, if the client has previously obtained a signed location record from that user.

The server must verify that the signature is valid. It then extracts the location record, which is in the same format as specified in the specification for `get_location`. It verifies that the `creation_date` is in the past and that the expiration date is in the future or null, and it then stores the location record as it is with the signature in its storage, assuming it does not already have a more recent location record for that user.

Command `get_public_key`

This simple command simply fetches the OpenPGP public key of a specific user identified by her fingerprint, if the server has the public key available.

Input JSON:

```
{
    "fingerprint": fingerprint
}
```

Output JSON:

```
{
    "public_key": public_key
}
```

Command `create_account`

The role of a PigeonMail server is to store conversations and to serve them to the appropriate requesters. Because disk space and bandwidth are finite resources, each server only allows a limited number of users to store new conversations. To gain the permission to create new conversations on a particular server, a user must create an account using this command.

In other online mail systems, such as email, the functionality of creating an account is not part of the protocol. Users may simply receive an email address and a password from their ISP or from a web page, for example. In PigeonMail, this is not sufficient, as instead of passwords, OpenPGP key-pairs are used for authentication instead. For security, it is essential that the client generate the OpenPGP secret key, rather than the server. The process of creating a new identity and a new account, therefore, must begin with the client.

The client must generate an OpenPGP secret key, optionally asking the user for a passphrase to protect the secret key. The client then calls this command, uploading the corresponding OpenPGP public key. If the server returns the status `done`, then an account has been created, and the client may proceed to create new conversations. If the account has previously been created, the server should return the status `done`. If the server refuses to create the account for any reason, the status should be `denied`.

Accompanying the public key, a signed location record for that user must also be uploaded. The server may require that the location record be recent, for that server and signed by the user. If the server allows users to create an account without publishing a record, the server will simply store conversations, and it is unlikely that other users will request from it data. This use-case may be useful for back-ups, for example.

Input JSON:

```
{
    "public_key": public_key,
    "signed_location_record": signed_data
}
```

Output JSON:

```
{
    "status": status
}
```

Command `update_public_key`

This command allows a user to upload an OpenPGP public key, belonging to any user. OpenPGP keys contain multiple cryptographic public keys, user IDs, photo IDs and certificates. If the server already has an OpenPGP public key, it should merge the two OpenPGP public keys together and store the result.

Input JSON:

```
{
    "user_public_key": public_key
}
```

Output JSON:

```
{
    "status": "done"
}
```

Command `close`

This command allows the client to inform the server that no more requests are forthcoming, and that the underlying TLS and TCP connection may be closed. Once the client has sent this request, it close the connection and does not wait for a reply.

Input JSON:

```
{
    "command": "close"
}
```

Output JSON:

The server does reply to this request, it simply closes the connection.

Design and implementation

To demonstrate the viability of the project and to test and iteratively improve the design, a prototypical implementation of both the PigeonMail server and client was completed. The names "Loft" and "Dove" were chosen for the server and the client implementations respectively, as they are words that belong to the pigeon topical family of words.

1 Technologies used

Programming language: Python

Both the server and the client were implemented in Python. Python was chosen for the following reasons:

- My familiarity with Python meant that I was comfortable in it and could begin to produce working code reasonably quickly.
- Python has a wide variety of useful libraries, some of which are included in the standard library (the collection of libraries distributed with Python itself).
- Both the server and the client could be written in Python, allowing some code to be re-used between the two applications.
- Python is free and open source, an important consideration for a prototype that aims to be free and open source itself.

As you can see, the choice of Python was largely a personal one. PigeonMail was designed to be agnostic of the programming language used, and so the consequences of using Python to develop a proof-of-concept prototype are negligible.

In the author's opinion, Python's primary downside is its poor performance, especially when using multiple threads, as my prototype does. (The performance results are discussed in more detail on page 27.) Another language, such as C or Go, may have been a better choice for performance and reliability.

Database backend: MySQL

The server stores its data in a MySQL database, in configuration files, and in a directory reserved for GnuPG. The database stores details about location records, user accounts, conversations and messages (a specification of the tables used are included in the appendix, page 39).

I chose MySQL as a database implementation for the following reasons:

- My familiarity with MySQL meant that I was comfortable using it and deploying it, and I expected to begin using it effectively rapidly. This expectation proved to be less true than anticipated, as a significant number of the columns were in binary types, storing fingerprints or encrypted data, leading to query results that were not usable to a human reader. Also, it proved to be a hindrance in deploying the server to systems where MySQL was difficult to install for lack of permissions.
- The availability of a reliable Python library, MySQL-Python, binding the database to the application code.
- The good performance gained by using indexes, offered by any relational database.

- ACID compliance meant that much of the difficulty of designing a reliable and consistent server was taken care of.
- MySQL is free and open source, an important consideration for a project that aims to be free and open source itself.

In hindsight, I would have done better to simply serialise the server's memory in JSON format or in Pickle format to disk. I would have lost the performance and consistency advantages of MySQL, but I would have been able to deploy a working prototype quicker, which was priority in this case.

On the client side, only configuration and OpenPGP keys are stored in local storage. All conversations and messages are simply fetched as needed, a cache had yet to be developed.

OpenPGP implementation: GnuPG

PigeonMail's protocol requires functionality specified by the OpenPGP specifications. Instead of developing this functionality from scratch, GnuPG (GNU Privacy Guard) was chosen instead:

- GnuPG is the de-facto implementation of OpenPGP used on Linux distributions. Its popularity leads to a greater scrutiny of its security claims following Linus' law.
- A Python library for GnuPG named “python-gnupg” is available. Because GnuPG only offers a command-line interface intended for human operators, programmatically accessing GnuPG can prove to be complex, so depending on a library that implements this functionality is a must. This library proved to be non-comprehensive, and some functionality had to be written by hand, which the author intends to submit as patches to the python-gnupg project.
- GnuPG is free and open source, an important consideration for a project that aims to be free and open source itself.

While there are competing implementations of OpenPGP, none of them are both as widespread as GnuPG, and open source.

TLS library: OpenSSL

PigeonMail's protocol also requires that TCP connections be secured using TLS. OpenSSL was chosen for the following reasons:

- OpenSSL is the library used by Python's standard library, specifically, its `ssl` module, and it used by many other programming languages and applications. Because of this, Linus' law applies to ensure that many of its bugs and security holes are fixed.
- Because OpenSSL is included in the standard library, calling it from Python code should prove to be trivial. However, during the course of the project, I ran into multiple confusing errors about OpenSSL that might have been easier to debug if I had direct access to the library itself.
- OpenSSL is free and open source, an important consideration for a project that aims to be free and open source itself.

Because of its inclusion in the Python standard library, and its popularity, OpenSSL was chosen without much consideration of the alternatives. One alternative that is worthy of consideration is GnuTLS, as it offers some functionality that OpenSSL does not; in particular, it offers the ability to certify the authenticity of the connection end-points using OpenPGP rather than X.509 certificates. Taking advantage of this ability would drastically simplify the protocol, eliminating the need to sign and wrap the contents of PigeonMail frames with OpenPGP manually, keeping the protocol layers clearly distinct in purpose. This alternative is discussed in more detail in the section entitled “Future work“ on page 33.

Graphical user interface toolkit: GTK+

The client designed for the prototype offers a graphical user interface to the user. The author chose GTK+ for the following reasons:

- My familiarity with programming GTK+ meant that developing a prototype would be easier and quicker.
- There is a Python library that enables GObject introspection, allowing Python to access the GTK+ API.
- GTK+ is free and open source, an important consideration for a project that aims to be free and open source itself.

In addition, Webkit was used to render conversations in a GTK+ widget on the client, chosen for similar reasons as to GTK+. It was felt that the particular layout needed for conversations would be quicker to implement in HTML and CSS than using GTK+, and Webkit allows mixing these two technologies very well.

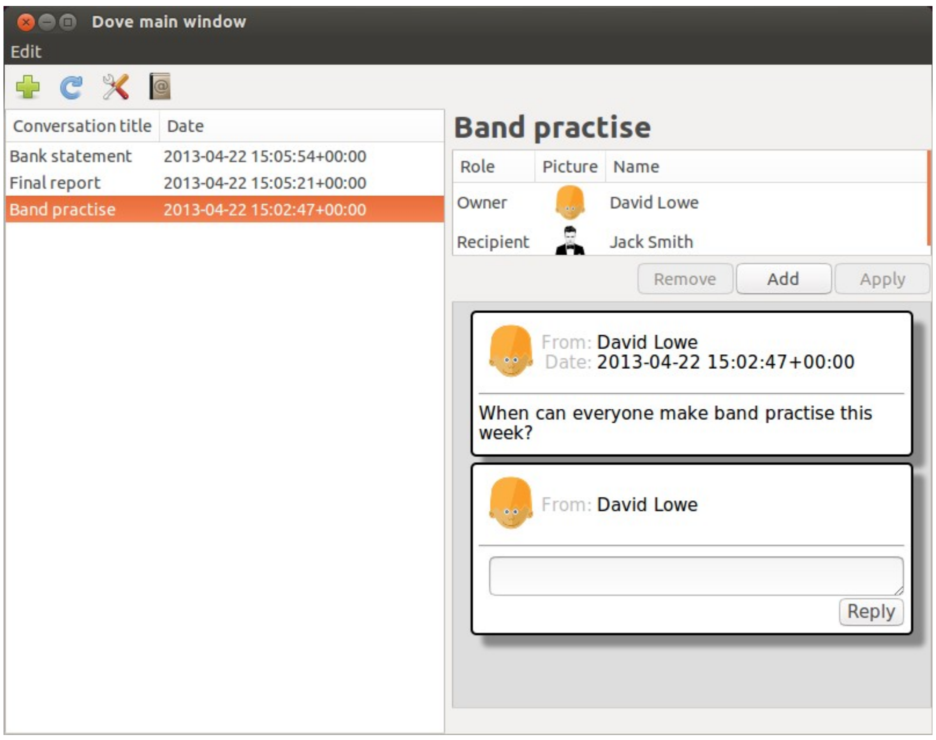
2 Server implementation: Loft

Loft was built on Python, MySQL, GnuPG and OpenSSL, technologies discussed in the previous section. Its model of execution, simply described, is a loop that looks for incoming connection requests. Once a connection request is received, a new thread is spawned, which continues to communicate with that particular requesting client, verifying the user's authenticity and fetching any data needed from the database. Once the connection with that client is closed, the thread is destroyed.

This model of concurrency is not very efficient in Python, because of the GIL (Global Interpreter Lock). When a thread attempts to execute a piece of Python code, it must acquire the GIL, which is shared amongst all threads in a Python process, only releasing once its done. (During I/O operations, however, it does not need to hold the GIL, allowing multiple threads to download and upload data concurrently.) Fortunately, a high percentage of time of each of Loft's threads is expected to be I/O bound, so the GIL does not eliminate parallelism completely. Even without the GIL, this model of concurrency is not very efficient because of the cost of context switches, the number of which grows with the number of concurrent connections. An evaluation of the performance of Loft is discussed on page 27.

An alternative model of concurrency is the asynchronous model. To summarise this model, only one thread is executed, tasks are assigned to a scheduler, and I/O operations are performed in a non-blocking manner. This model of concurrency is usually chosen for its performance advantages, but I chose not to use it, opting for the multi-threaded model instead, choosing ease and speed of development over performance of the prototype, once again.

3 Client implementation: Dove



Conversation title	Date
Bank statement	2013-04-22 15:05:54+00:00
Final report	2013-04-22 15:05:21+00:00
Band practise	2013-04-22 15:02:47+00:00

Band practise

Role	Picture	Name
Owner		David Lowe
Recipient		Jack Smith

From: David Lowe
Date: 2013-04-22 15:02:47+00:00

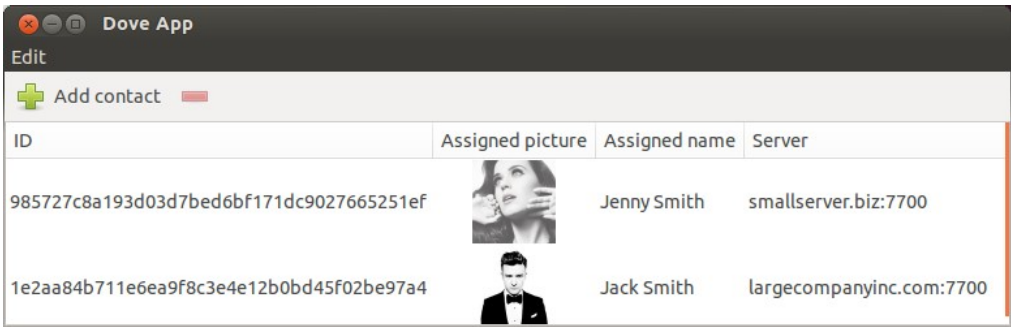
When can everyone make band practise this week?

From: David Lowe

Illustration 2: A screenshot of the main window of Dove

The design of the the mail client is similar to that of many email clients, such as Gmail, Sparrow, Mail on OS X or Geary. On the left, it displays a list of conversations that the user can view and/or add messages to, and on the right, it displays the currently selected conversation.

Contacts can be viewed and managed by opening the contacts window. Upon adding a contact, the user is guided through the process using an “assistant” step-by-step window (also known as a “wizard”), prompting the user to enter the fingerprint, a name and a picture for that contact. The user may choose to accept the published name and picture of that contact, but only if she does not have another contact with the same assigned name or picture.



ID	Assigned picture	Assigned name	Server
985727c8a193d03d7bed6bf171dc9027665251ef		Jenny Smith	smallserver.biz:7700
1e2aa84b711e6ea9f8c3e4e12b0bd45f02be97a4		Jack Smith	largecompanyinc.com:7700

Illustration 3: A screenshot of the contacts window of Dove

The user can change her settings in a preferences window. On the first run of the application, the user is prompted to create an OpenPGP secret key and to choose a server, published name and published picture using another “assistant”.

Testing and evaluation

1 Stress testing

While the focus of this project was not on performance and resilience, the prototypical implementation was stress tested to a certain extent. A Loft server was installed on each one of the PCs available in the Linux laboratory at Cardiff University School of Computer Science and Informatics. The home directory of the user to which I have access is a mounted network share, which means that all the Linux machines have the same home directory, and the read and write performance is not comparable to that of a local disk. For this reason, I requested that a MySQL database be created on the hard disk of the lab machines.

Once the Loft servers were installed on the Linux machines, I ran several tests. The tests depended on a script named `hound.py`, which places several requests repeatedly to a target server summarised by this pseudo-code:

```
for iteration = 1 to 30
    connect to target server
    call get_all_conversations(owners_fingerprints=[random fingerprint])
    disconnect from server
    record time taken for this iteration
endfor
for iteration = 1 to 30
    connect to target server
    for inner_iteration = 1 to 30
        call get_all_conversations(owners_fingerprints=[random fingerprint])
    endfor
    disconnect from server
    record time taken for this iteration
endfor
```

The results were as follows:

1. When the test was run from one client to one target server:
    - When each connection performed one request, the mean average time was 0.106573 seconds, the median average time was 0.109037 seconds, the maximum time was 0.130134 seconds and the minimum time was 0.088446 seconds. The standard deviation was 0.010903. For a prototype, this is acceptable, as 100 milliseconds is considered instantaneous by at least one usability expert, Jakob Nielsen (1993).
    - Where each connection performed thirty requests, a list of thirty time durations was recorded, from which a list of thirty mean average time durations was calculated. The mean average time from this list was 0.075998 seconds, the median average time was 0.075991 seconds, the maximum time was 0.077696 seconds and the minimum time was 0.073656 seconds. The standard deviation was 0.000941. This is a noticeable improvement over the previous result, demonstrating the benefits of reusing the same connection when one has multiple requests.

2. When the test was run from 30 clients to a single target server concurrently:
    - When each connection performed one request, the mean average time was 0.924290 seconds, the median average time was 0.901914 seconds, the maximum time was a very high 4.034535 seconds, and the minimum time was 0.098413 seconds. The standard deviation was 0.319832. The average numbers are satisfactory, however, the maximum time taken clearly is not, nor is the standard deviation. The performance of the system noticeably slowed as more connections were being handled concurrently by the server. It seems that the multi-threaded model used to handle multiple requests cannot handle even thirty concurrent connections, or perhaps there is simply a bug in the implementation.
    - When each connection performed thirty requests, a list of thirty time durations was recorded, from which a list of thirty mean average time durations was calculated. The mean average time for each request was 0.495118 seconds, the median average time was 0.490271 seconds, the maximum average time was 0.648319 seconds and the minimum average time was 0.302863 seconds. The standard deviation was 0.038352. This is an improvement over the previous result, demonstrating that the bottleneck does not lie in concurrent data retrievals from the MySQL database, but rather in setting up and tearing down multiple TLS connections at once. 495 milliseconds is still too high a number for a simple operation that returns no data, and it does demonstrate that the prototypical implementation is not ready for production use.
3. When the test was run from one client to 30 servers one after each other:
    - When each connection performed one request, the mean average time was 0.112657 seconds, the median average time was 0.111608 seconds, the maximum time was 0.242552 seconds and the minimum time was 0.080907 seconds. The standard deviation was 0.019774. These results are similar to the first set of results, demonstrating that a client can switch between multiple servers and expect similar results. If the bandwidth available to the client is sufficient enough, it could theoretically concurrently perform these requests. In practise, the bandwidth is not infinite, so there would be a performance penalty to increasing the number of PigeonMail servers from which to fetch conversations.
    - When each connection performed thirty requests, a list of thirty time durations was recorded, from which a list of thirty mean average time durations was calculated. The mean average time for each request was 0.077072 seconds, the median average time was 0.076778 seconds, the maximum time was 0.086612 seconds and the minimum time was 0.072377 seconds. The standard deviation was 0.002010. These numbers are an improvement over the previous results as expected, proving the performance gained by reusing an existing connection.

The spreadsheet with the numbers gathered from the tests are included in the files submitted with the final report, as well as the source code for the tests.

2 Decentralisation

One of the goals of PigeonMail is to achieve decentralisation, that is, that the user should be able to choose a vendor and communicate with other users relying on other vendors.

On the whole, PigeonMail achieves this. It is entirely conceivable that every user could choose a different vendor, or that they could choose to host their own server, while communicating with other servers hosted by other vendors.

Because the published locations of users can be domain names, PigeonMail is necessarily dependent on a domain-name resolution system. For simplicity, PigeonMail relies on the same domain-name system as that of the Internet in general. While the technology of resolving domain names is distributed, the authority to assign domain names and to revoke them is hierarchical, with ICANN, the Internet Cooperation for Assigned Names and Numbers, standing at the top of the hierarchical pyramid. Some users may find this dependency problematic, as the ICANN may follow political or idealogical policies which they do not agree with, or it may abuse its power.

Even if the published location of the user indicated an IP address instead of a domain name, ICANN still has an amount of authority, as it assigns IP blocks to various organisations and regions.

In addition, to verify the authenticity of PigeonMail servers, clients must ensure that the certificate provided by the server is verified by a trusted certificate authority. Even when the list of trusted certificate authorities is large, this still places a large amount of trust in a centralised authority figure, and it reduces the decentralised nature of the PigeonMail protocol.

PigeonMail is as decentralised as email, but not more, as they both allow multiple online mail servers to communicate with each other, and they both rely on DNS, IP addresses and trusted certificate authorities. For the purposes of this project, I find this level of decentralisation satisfactory.

3 Privacy

PigeonMail does offer a certain amount of privacy to users that other online mail systems do not, most notably, because of the end-to-end encryption used by the protocol.

Assuming that the security of a PigeonMail client is not compromised, a user can expect the following guarantees:

- The contents of a message will only be viewed by the key holders of the participants of the conversation. This includes any participants that the owner of the conversation may add after the message has been created.
- The title of a conversation will only be viewed by the key holders of the participants of the conversation[5]. This includes any participants that the owner of the conversation may add after the message has been created.
- The existence of a conversation, its participants and the number of its messages, can only be ascertained by users who have permission to read that conversation, and by the server that hosts that conversation.
- Once a participant has been removed from a conversation, that participant can no longer read its meta-data or its messages (even in encrypted form), nor can it discover the new list of participants. (A malicious or faulty server could leak this information however, as it has access to it under the previous guarantee.)

Notice that if the user does not trust her PigeonMail server, then the two previous guarantees do not offer much reassurance.

It is important to note the potential threats to a user's privacy, that a privacy conscious user should be aware of:

- PigeonMail servers can track which users request which piece of data at what times and from which owners. Even if they cannot read the contents of a message or a title, they can

5 Interestingly, this guarantee is not provided by OpenPGP implementations used over emails. The subject of an email is not encrypted, only its contents.

still form a partial social graph of a user's communications.

- Because every PigeonMail request made from the client to a server is signed using OpenPGP, the server could exploit OpenPGP's non-repudiation property to prove to a third party that the holder of the user's secret key made that request. Once a server has collected enough such records, it would be able to form a partial social graph of a user's communications, that is cryptographically credible to a third party. (Note that the property of non-repudiation only guarantees that the holder of the secret key made that digital signature. A user could claim that her secret key was compromised to avoid being associated with a particular digital signature in a court of law.)
- Similarly, every message or title is signed by its author, and that signature is non-repudiable. The server hosting the conversation or any of its participants could exploit this property to prove to a third party that the holder of a user's secret key authored a message or title.
- Users of PigeonMail are vulnerable to traffic analysis. Even though third parties cannot read the TLS encrypted traffic between PigeonMail clients and servers, they may still be able to detect the existence of the encrypted traffic, possibly deducing from it the likelihood of communication between two particular users.

4 Security

Wherever possible, well-known established cryptographic technologies were used instead of implementing new ones. Not only did this save time, but it also was likely to increase the security of the system, as an established open source library should have many more security vulnerabilities discovered and fixed than a private implementation by one student.

Because all connections are made over TLS, a client can be certain that an attacker with access to the channel of communication cannot decrypt any communication between two peers. In addition, because the server is authenticated using a certificate singed by a trusted CA, the client can be sure that a MITM attack is not being performed, with the attacker masquerading as the server. TLS also provides protection against replay attacks and against certain forms of traffic analysis (for example, the size of the communicated data can only be determined approximately from the size of the encrypted traffic, not exactly). TLS is also extensible, allowing the two peers to negotiate into using newer and stronger encryption algorithms.

The client is authenticated using an OpenPGP digital signature. A nonce value set by the server must be included in all requests, as well as the address of the intended server, to prevent replay attacks and MITM attacks, as discussed on page 13. While every step was taken to ensure the confidentiality and the authenticity of the data communicated by a client, this part of the system is probably the weakest from a security point of view, considering that it was designed and implement by a single developer with no external security audits.

The greatest danger with PigeonMail is not the loss of privacy, but the loss of identity. If a user loses her OpenPGP secret key (or its pass-phrase, if it has one), she can no longer identify using her old fingerprint. She must rebuild her network of contacts from scratch, convincing them again out-of-band to add her to their list of contacts. To prevent this, every user should make backup copies of their secret key, but only in secure locations, to avoid her identity being stolen. Key management is widely recognised to be the most difficult process to accomplish securely for any cryptographic system, and PigeonMail is no exception.

5 Spam free

As with security, the fight against unsolicited bulk messages is a continual one. It would be impossible to declare PigeonMail to be completely spam-free, as there may be new techniques to be discovered to deliver spam to users of online mail systems.

Because a user only begins receiving conversation updates from a hand-selected list of contacts, a spammer simply cannot send messages to as many users as possible. While a spammer could create a massive number of conversations, it is expected that almost none of these conversations would ever reach legitimate users. A spammer must first overcome the hurdle of being added to a user's contact list. To do so, the spammer must somehow persuade the user in a channel other than PigeonMail to add her as a contact. This task should be very difficult for a spammer with no interesting content to offer, and it should be very difficult to do in large numbers. Even if a spammer succeeds in convincing a user to subscribe to her conversations, she must avoid irritating or angering the user, as the user may choose to unsubscribe from her conversations. Thus, a bait-and-switch tactic is both costly in human resources, and not very effective for long periods of time.

The value of some of a contact's messages may be high enough to a user to deter her from unsubscribing, even when some of the messages are unwanted. But PigeonMail does not aim at eliminating all unwanted and uninteresting messages automatically, to do so would require a highly complex and artificially intelligent system. Instead, it merely aims at eliminating bulk unsolicited messages. Once the volume of messages becomes high enough to qualify for “bulk”, the user would simply unsubscribe, as it very difficult to provide numerous enough and valuable enough interesting to offset the cost of the unwanted messages in the eye of the user.

6 Usability

The prototype was made to be fairly similar to existing online mail user interfaces, without misleading users into assuming similarities when there are none.

For example, the list of conversations is displayed to the right, with the messages of the currently selected conversation displayed one after another in a pane to the left. The input box to add a message is displayed after the list of a conversation's messages, following the principal of affordance: by its very location the input box suggests to the user where the message will be added. One cannot create a message without first creating or selecting a conversation, this is intended to reinforce the concept of a conversation in the user's mind. The interface is deliberately similar to Facebook's private messaging user interface, as the concepts of a conversation and of a message are quite similar to Facebook's from a usability point of view.

Participants in a conversation are always displayed with the nickname and the picture that the user has assigned for them, in compliance with Stiegler's petnames system (2005). Fingerprints are only exposed to the user when she needs to add a contact, it is otherwise an implementation detail hidden from the user. The fact that nicknames are not allowed to be reused protects the user from forgery.

The most significant flaw with the protocol is that it does not allow the user to discover whether another user has added them, and whether they have received a conversation. As it stands now, users can create conversations for other users, never learning that those participants have not received the conversation because they have not added her to their list of contacts.

The prototype of the client suffers from two significant usability flaws, however, fixes to which are discussed in the future work section page 34. Firstly, Dove does not introduce a new user well to the concepts required to use PigeonMail successfully. On first run, the Dove client launches an assistant (wizard), guiding the user in creating an OpenPGP secret key, publish a name and a photo, and creating an account on a PigeonMail server. Once this process is finished, however, Dove's main window is simply displayed, with an empty list of conversations. No explanation is given as to the importance of adding contacts. The user must discover on her own how to add contacts and for what reason. Secondly, Dove does not make it easy to add a contact. To do so, the user must open the contacts window and then launch an assistant, which requires the fingerprint of the user to add. Instead, it would be much better if the user could simply click on the name or the picture of a participant in a conversation to add her as a contact, or even if participants were added automatically to the list of contacts.

7 Extensibility

The PigeonMail protocol is designed to be extensible. Since all requests and replies are simple JSON objects, new key-value pairs can be added with no trouble. Clients should simply ignore keys that they do not recognise. Most of the improvements suggested in the future work section (page 33) could be implemented in this way.

If something drastic should need to change with the PigeonMail protocol, the version number declared in the initial PigeonMail request could incremented. If the server supports that version, the request-reply sequence would continue as normal, if not, an exception would return, and the client could use an older version of the protocol.

8 Platform independence and portability

Because PigeonMail is built on common and standardised Internet technologies (TLS and TCP), it should in theory be platform independent, compatible with any Internet enabled device. In practise, however, some users may find problems deploying it, as some firewalls block unrecognised protocols. I came across this issue myself during development: I discovered that Eduroam, the network used by multiple European universities, contains a firewall that perform packet inspection. If it detects traffic that does not match a recognised pattern, even when encrypted with TLS, it closes the connection.

Future work

1 Future work on protocol design

- Currently, the protocol only allows conversations with a finite number of participants. However, there is feasibly no barrier to designing public conversations. Conversations could be publicly readable, or even publicly writeable. The natural act of adding and removing contacts corresponds very well to subscribing and unsubscribing to mailing lists.
- The protocol currently allows hosts to implement very limited caching: that of messages only. The list of conversation IDs to fetch and the list of message IDs of a particular conversation to fetch is never cached. There are several ways this could be designed. The server could calculate a hash value generated using a deterministic hash function from the set of conversations available to a particular user and the date of their latest modification. If the hash generated by the server matches the hash included in the request, the server need not reply with the whole list again, and the client can simply rely on its cache. If not, the server and the client could synchronise their list of conversations using an algorithm that minimises data transfer using delta transfer, similar in concept to the Rsync algorithm.
- To gather updates, a client must poll one or many PigeonMail servers. This process is more expensive and slow than it needs to be. It would be better if a client could subscribe to updates, and then PigeonMail servers could push out notifications of updates using UDP.
- The current design of the protocol using TLS to secure the connection from eavesdroppers, authenticating the server-side with a certificate. All requests from the client are authenticated using OpenPGP signatures on top of that. TLS actually allows for client-side authentication other than X.509. For example, the library GnuTLS allows client-side authentication using OpenPGP. This would reduce the complexity of the protocol.
- The structures exchanged by PigeonMail servers and clients are simple structures serialized in JSON. While this allows for easy implementation, it is very space-efficient. Binary protocols, such as Protocol Buffer, Thrift, MessagePack, Blink or Cap'n Proto allow for a more compact serialization of simple data structures, in addition to some other features.
- A user can publish a name and a photo by creating a UID or PID in her OpenPGP public key. It would be useful if the user could publish more information about herself, such as phone number, address, email address, website URL and so on. This would allow users to have a detailed address book with up-to-date information.
- Published names and photos are publicly available. It would be better if a user could make certain names and photos available only to a select list of contacts.
- A user should be able to attach one or more encrypted file attachments to a message.
- The protocol could allow for extremely short replies such as “yes”, “no” and “acknowledged”. This would work especially well for conversations with a large number of participants, as a client's user interface could save space by simply displaying a tally of the number of “yes”, “no” or “acknowledged” replies. In many ways, this is similar to the “like” system on Facebook and the upvote/downvote system in other online mail systems.
- The conversation owner should be able to set a maximum number of characters allowed for each message. In this way, a conversation owner could encourage conciseness for particular conversations.

- Messages should be able to contain rich text, not just plain text.
- It would be ideal if the information available to the servers was minimised, even meta-data about conversations and its participants. It may be possible to implement this without subject servers to DOS attacks by using fully homomorphic encryption. (Homomorphic encryption allows an agent to perform operations on encrypted data such as addition and multiplication, generating the correct encrypted result, without decrypting the data at any point.) This form of encryption is very new among cryptographic technologies, and it would take much more work to ensure its security and performance than simply relying on existing well-understood technologies. However, the benefits it provides to the user's privacy should not be neglected.
- A user is expected to upload her OpenPGP public key and her location record to as many PigeonMail servers as possible to ease her discoverability. This means that as the number of commonly used PigeonMail servers increase, the speed and reliability at which this is done decreases. It would be better if an algorithm used to only upload the public key and the record to a fraction of the servers were used, such as a distributed hash table.

2 Future work on implementation

The implementations of the server and the client were only meant to be prototypes, so it is to be expected that they are not feature-complete.

The client:

- The client should be able to connect to servers over IPv6 as well as over IPv4. As the Internet begins to run out of IPv4 addresses, there is a growing movement of migration to IPv6 addresses, and the client should be future-proof.
- The user should be able export her OpenPGP secret key and import it in another Dove client in a user-friendly way.
- In order to encourage users to back-up their OpenPGP secret key, the Dove client could periodically refuse to use the locally stored OpenPGP secret key, asking to import one from another location instead. This is meant to encourage good back-up practise, in the same way that fire drills are meant to encourage good procedure upon the discovery of a fire.
- While the protocol does support signing another user's UID and PID, the Dove client provides no functionality to do so. The contacts window of the Dove client should allow the user to sign a contact's UID and PID, and to publish that certificate. It should also allow publishing revocations.
- Switch to GnuTLS to allow for client authentication using OpenPGP in a standardised manner.

The client should especially offer OpenPGP's web of trust functionality to the user in a well-integrated and intuitive manner:

- Alongside a contact's name, the user interface should display one of the following pieces of text: “(unverified)”, “(verified by contacts)” and “(verified)”. The first piece of text indicates that the client has no assurance that the person with the given name is indeed the holder of the secret key matching that fingerprint. The second piece of text indicates that at least one trusted contact, or at least three marginally trusted contacts, or the path of signed keys from the contact to the user is five steps or shorter. The third piece of text indicates that the user has herself verified the authenticity of the name.

- In a similar way, a photo associated with a user should some visual representation to indicate whether the photo has been verified or not.
- The user should be able to indicate that she has verified the authenticity of a name or a picture. For instance, clicking on “(unverified)” or “(verified by contacts)” could open an assistant (a wizard) guiding the user through the process of verifying the user's identity. The process should be easy to follow and as free of jargon as possible. For example, the assistant could encourage the user to phone the contact and ask her to read out her fingerprint over the phone. The assistant could then ask for the user's permission to publish the certificate. Once this process has been completed, the piece of text displayed next to a server
- The user should also be able to indicate how much she trusts a particular user to verify the identity of her contacts, from one of the following levels: “I don't know or won't say”, “I do NOT trust”, “I trust marginally”, “I trust fully” and “I trust ultimately”. If one fully trusted contact or enough marginally trusted contacts have published a certificate for the identity of a particular user, the piece of text displayed next to the latter's name will be “(verified by contacts)”.
- The user should be able to change her level of trust in a particular contact to verify other contact's identity. She should also be able to revoke any certifications of identity she has issued herself, and publish revocations.

Server:

- Switch to GnuTLS to allow for client authentication using OpenPGP in a standardised manner.
- The exceptions provided to the client are very vague and generic, they could be made more informative and standardise in the protocol.
- The prototype's performance in handling concurrent requests is dismal, it could vastly be improved.

Reflection on learning

This project was completed as a required part of the third year of the Computer Science BSc degree at Cardiff University. Its goals are therefore primarily educational in nature, and it must also be evaluated in that light.

I can say without a doubt that I have learnt a lot while designing and implementing this project. For instance, I studied or used many technologies that were new to me. Before this project, I was not even aware of the existence of binary protocols such as Protocol Buffer, MessagePack and Thrift, nor socket libraries such as ØMQ (pronounced “zero-em-queue”). Studying them proved to be a delight and influenced my design of the protocol, even if I did not use them.

During the second term of the year, I took the module “Distributed Systems Technologies”. This module, together with the recommended reading, (especially Distribute Systems Concepts and Designs by George F Coulouris et al) proved to be very instructive, and I regret not coming across this material earlier in the life of the project. I discovered that the request-reply pattern used by PigeonMail is very common and well studied pattern, and I learnt about alternative patterns that could be useful for a project like this (subscribe-unsubscribe, etc). I also learnt about message queues and RPC, which I ruled out of this project.

In order to understand the guarantees provided by technologies such as TLS and OpenPGP, I studied cryptographic primitives. The book Everyday Cryptography (Martin 2012) proved to be particularly valuable in this regard, giving me a much deeper understanding of the concepts such as confidentiality, authenticity, non-repudiation, pre-image resistance and second pre-image resistance., the latter two of which were fundamental in choosing an OpenPGP fingerprint as the user's identifier. I also studied OpenPGP and GnuPG, one of its implementations, discovering for example that asymmetrically encrypted messages only encrypt the contents of the message, but not the identity of the intended recipients, and that digitally signed messages always provide non-repudiation. Designing a secure communication system allowed me to understand some of the choices made for OpenPGP much better, such as the importance of the verifying the validity of a person's OpenPGP key, and the methods of building a web of trust.

I also learnt more about what keeps me productive when designing and implementing a project like this. The iterative style of development suits me well: it allows me to gradually introduce features and functionality, keeping the effort-result cycle short and frequent. It also allows me to change the design of the project at will, an advantage made highly feasible by the fact that I am the main stakeholder and the setter of the requirements. Inspired by my experience during my year in industry, I was sure to develop unit and integration tests for the server implementation. This allowed me to catch bugs at a much faster rate than I would have otherwise. It also gave me confidence in the reliability of the software, and a confidence to refactor the code when I needed to. Although I had experience with Subversion and Bazaar in the past, I decided to teach myself Git and use it for this project. I found Git to be less intuitive than Subversion and Bazaar, but I am glad I used it, as it is a widely used tool in the programming community and it will be useful in future. Although Git's benefits are primarily for collaborative development, I did find it very useful to be able to divide my work on the code into chunks, regularly comparing any new code I had implemented with previous versions. At first, I did not develop tests for Dove, and by the later stages of life of the project, my development speed slowed, both because of the increased complexity of Dove, and because of the increased cost of manually testing it by running the GUI client by hand. I developed some unit tests for sections of the code that did not depend on the GUI, but given more time, I would have learnt and used a testing toolkit suitable for a GUI.

While this project is in one sense never-ending, as it could always be improved, I did not manage to complete all the tasks I had set out to do before the deadline. Most notably, I did not work on the first-contact problem at all, which was something I set out to do in the plan I had originally drawn up. In hindsight, I think I could have achieved this goal had I been more intentional in where I spent my time. For example, I spent a lot of time researching technologies speculatively, (most notably ØMQ), increasing my knowledge at the expense of allowing some work to be dropped before the deadline.

Another primary goal of this project is to achieve a high grade to contribute towards my degree. The allocation of my time did not always reflect this priority. Since my grade is pramarily based on this report, I should have spent a large chunk of my time writing and re-reading it. In reality, I only spent the last few weeks writing it, failing to submit a complete draft to my supervisor at the end of the spring holidays as we had originally planned. Part of this was due to a case of writer's block: getting started on writing even a paragraph could take me hours. I discovered some techniques to help me overcome this problem. For example, I tried to takes breaks not at the end of task, but in the middle of it, while I was still highly concentrated and in the flow. Although counter-intuitive, this meant that as soon as I returned to the page after my break, I could continue where I left off, avoiding writer's block. I tried a lightweight version of the pomedoro technique, which is summarised simply by working in 25 minute chunks, with five minute breaks. This technique was helpful, as it allowed me to relax and recuperate without feelings of guilt, and to keep my levels of concentration high for 25 minutes at a time. In contrast, I found it quite easy to spend time programming the project. This project was a good learning experience, as it allowed me to discover habits that keep me productive, and others that do not. I still do not feel I have broken the negative combination of perfectionism and writer's block that kept me from finishing the report in advance. Deadlines are crucial as long as they are externally set. I have yet to be able to keep myself to self-set deadlines.

Conclusions

PigeonMail has the potential to be a viable alternative to online mail systems such as email. It would be unrealistic to say that the protocol is ready as it is, as I previously discussed, there is still many improvements that could be added, a few of which, such as the caching, are essential.

However, even with its problems, PigeonMail at its current iteration does succeed in its primary goal: to be a online mail system that is decentralised, secure and free of spam.

References

Martin, K. M. 2012. *Everyday Cryptography*. Oxford University Press, New Delhi.

Nielsen, J. 1993 *Response Times: The 3 Important Limits* [Online]. Available at http://www.nngroup.com/articles/response-times-3-important-limits/ [Accessed 3 May 2012]

Stiegler M. 2005. *An Introduction to Petname Systems* [Online]. Available at http://www.skyhunter.com/marcs/petnames/IntroPetNames.html [Accessed 14 December 2012]

Wolf, Misha, and Charles Wicksteed. 1998 *Date and time formats. W3C NOTE NOTE-datetime-19980827.*

Appendix

1 SQL tables for Loft

```
DROP TABLE IF EXISTS `users`;
CREATE TABLE `users` (
  `fingerprint` BINARY(20) NOT NULL,
  `add_conversation_permission` tinyint(1) NOT NULL DEFAULT 0,
  PRIMARY KEY (`fingerprint`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

DROP TABLE IF EXISTS `conversations`;
CREATE TABLE `conversations` (
  `owner_fingerprint` BINARY(20) NOT NULL,
  `id` decimal(39,0) NOT NULL,
  `title_encrypted` VARBINARY(10240) NOT NULL,
  `secret_encrypted` BLOB,
  PRIMARY KEY (`owner_fingerprint`, `id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

DROP TABLE IF EXISTS `messages`;
CREATE TABLE `messages` (
  `owner_fingerprint` BINARY(20) NOT NULL,
  `id` decimal(39,0) NOT NULL,
  `conversation_owner_fingerprint` BINARY(20) NOT NULL,
  `conversation_id` decimal(39,0) NOT NULL,
  `contents_encrypted` BLOB NOT NULL,
  `position` int(11) NOT NULL, -- not necessarily in date order
  `date_posted` DATETIME NOT NULL, -- in UTC
  PRIMARY KEY (`conversation_owner_fingerprint`, `conversation_id`, `owner_fingerprint`, `id`),
  KEY `conversation_key` (`conversation_owner_fingerprint`, `conversation_id`, `position`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

DROP TABLE IF EXISTS `conversations_participants`;
CREATE TABLE `conversations_participants` (
  `conversation_owner_fingerprint` BINARY(20) NOT NULL,
  `conversation_id` decimal(39,0) NOT NULL,
  `participant_fingerprint` BINARY(20) NOT NULL,
  `read_permission` tinyint(1) NOT NULL,
  `write_permission` tinyint(1) NOT NULL,
  PRIMARY KEY (`conversation_owner_fingerprint`, `conversation_id`,`participant_fingerprint`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

DROP TABLE IF EXISTS `public_location_directory`;
CREATE TABLE `public_location_directory` (
    `fingerprint` BINARY(20) NOT NULL,
    `host` varchar(1024) NOT NULL,
    `port` INTEGER NOT NULL,
    `creation_date` DATETIME NOT NULL, -- in UTC
    `expiration_date` DATETIME, -- in UTC
    `signed_data` VARBINARY(10240) NOT NULL,
    PRIMARY KEY (`fingerprint`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```