Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Bachelor Thesis

A Bitmessage Client for Android™

Author Christian Basler
Adviser Dr Kai Brünnler
Expert Daniel Voisard
Date January 21, 2016

Trademarks

Abbreviations

POW	proof of work
CPU	central processing unit
GPU	graphics processing unit
VM	virtual machine
OS	operating system
PGP	Pretty Good Privacy
MIME	multipurpose internet mail extensions
S/MIME	Secure/MIME
JDBC	Java database connectivity
SQL	structured query language
NoSQL	Not only SQL
API	application programming interface
DOS	denial of service
nonce	number used once
IP	internet protocol
REST	representational state transfer
WIF	Wallet Import Format

Abstract

Even if you are encrypting your e-mails, you still can't hide who you're writing to. Your e-mail client might even reveal much more about you, your computer, and the software you use.

Bitmessage solves all this, but up until now there was no practical way to use it on mobile phones. That's where this thesis comes in.

Bitmessage is a peer to peer messaging protocol that builds a mesh network among the participating clients. Each client tries to maintain multiple connections to other network nodes and has an encrypted copy of every current message.

There are some unique challenges for mobile clients. For one, its users are very privacy conscious. Also, the protocol needs both huge amounts of traffic and a lot of CPU time.

It works by distributing every message to every client, so they can pick up the ones they can decrypt with the available private keys.

To protect the network from malicious flooding, clients must find a partial hash collision as proof of work in order to send a message. This is designed to be relatively slow even on desktop computers.

So typically you'd want as many CPU cores as possible, no shortage of power and a flat rate on internet access – a challenge for a mobile app.

Android™ has some challenges, too. To preserve resources, the operating system might kill any process at any time, especially those in the background that process incoming network objects. Then there are some major Java dependencies missing in the Android VM, most notably JDBC, used to access databases. And finally, the devices differ vastly in processing power. The implementation we propose solves these problems.

In addition there were two optional optimizations that require a server. First, Android provides a highly optimized method to synchronize data with a server, which we leveraged while still using the official Bitmessage protocol. Secondly, an option to let a server do the proof of work was added for weaker phones. For both options the user must give up some of his anonymity towards the server – the choice is theirs.

You can get more information and the latest news about the app at https://dissem.ch/abit

Contents

1 Introduction **6**
- 1.1 What is Metadata? . . . 6
- 1.2 How Can We Hide Metadata? . . . 6
- 1.3 What is Bitmessage . . . 6
  - 1.3.1 Advantages . . . 7
  - 1.3.2 Disadvantages . . . 7
  - 1.3.3 Risks . . . 7
- 1.4 Motivation behind this Thesis . . . 8
  - 1.4.1 Current state – what is missing? . . . 8
  - 1.4.2 How should it be? . . . 8
  - 1.4.3 Why is it hard to do? . . . 8
  - 1.4.4 Why me, and how do I intend to do it? . . . 8
- 1.5 The Jabit Bitmessage Library . . . 9
  - 1.5.1 Architecture . . . 9
  - 1.5.2 How to Use . . . 10

2 The Bitmessage Protocol **12**
- 2.1 Bitmessage Commands . . . 12
- 2.2 Bitmessage Objects . . . 12
- 2.3 Encryption . . . 13
- 2.4 Proof of Work . . . 14

3 Naive Implementation **16**
- 3.1 Unexpected Problems . . . 16
  - 3.1.1 Bouncy Castle vs. Spongy Castle . . . 16
  - 3.1.2 JDBC . . . 16
- 3.2 Expected Problems . . . 17
  - 3.2.1 Proof of Work . . . 17
  - 3.2.2 Data Traffic . . . 17
  - 3.2.3 Storage . . . 18

4 Android Specific Challenges **19**
- 4.1 Application Lifecycle . . . 19
- 4.2 Deprecated Java Versions . . . 19

5 Optimizations **21**
- 5.1 Sync Adapters . . . 21
- 5.2 Server Side Proof of Work . . . 21
  - 5.2.1 Initial Idea . . . 21
  - 5.2.2 Acknowledging Issues . . . 22
  - 5.2.3 proof of work (POW) Protocol . . . 22

6 Artefacts **24**
- 6.1 App . . . 24
  - 6.1.1 User Interface . . . 24

6.1.2 Settings . . . 25
6.2 Server . . . 25
6.3 Installation . . . 26
6.3.1 Configuration . . . 26

7 Related Works **28**
7.1 E-Mail Gateways . . . 28
7.2 Bitseal . . . 28
7.2.1 User Interface . . . 28
7.2.2 Settings . . . 29
7.2.3 Power Usage . . . 29
7.2.4 Server . . . 30

8 Future Work **31**
8.1 Known Bugs . . . 31
8.2 Missing Features . . . 31

9 Conclusion **32**

1 Introduction

1.1 What is Metadata?

Metadata is information about data we create or access. This could be the websites we visit, what television programmes we watch, with whom we communicate and much more. The volume and quality of metadata greatly depends on the kind of data and the software that created it.

While encryption technology like Pretty Good Privacy (PGP) or S/MIME provides a secure way to protect content from prying eyes, it can't hide the header information of an e-mail: sender, recipient, subject and possibly much more.

Ever since the revelations of whistleblower Edward Snowden we learned that metadata – most notably information about who communicates with whom – is equally interesting and much easier to analyse than the actual content.[1]

1.2 How Can We Hide Metadata?

With e-mail, all metadata is plain text, even for encrypted messages. We might be able to encrypt the connection to the e-mail provider, and they might or might not encrypt their connections to other providers. We can only hope that both our and the recipient's e-mail provider are both trustworthy and competent. Can we really expect that from something we get for free? And can we be sure they're not forced to release what they know to some agency?[2]

1.3 What is Bitmessage

With Bitmessage we send a message to a sufficiently large number of participants, among them the intended recipient. Content is encrypted such that only the person in possession of the private key can decrypt it. All participants try to do this in order to find their messages.

It is a peer to peer protocol building a mesh network among the participating clients. Every client tries to maintain multiple connections to other network nodes and has a full copy of every current object.

Objects are encrypted using a public key. Every client tries to decrypt each object using its private keys, processing the ones where it succeeds.

Bitmessage lends many ideas from Bitcoin, which leads to some misconceptions. For one, there is no block chain, just a shared inventory of currently active objects. Also, the address does not contain the public key, just a hash. This is because Bitmessage uses different keys for encryption and signatures.

1.3.1 Advantages

A big advantage of Bitmessage is its inherent key management. The address contains a hash of the public key, and retrieving said key is an integral part of the protocol.

And the selling point of course is that everything is encrypted and signed, and there is next to no metadata an attacker could use.

Finally, the setup is very easy. You just need to install a client software, there's no setup necessary except a click to create a new identity (i.e. Bitmessage address) if your client didn't do that when it started for the first time.

1.3.2 Disadvantages

Of course the protocol uses a lot of resources. The traffic problem could be somewhat managed by splitting it into streams[3], but proof of work (POW), an expensive operation, is required in order to protect the network.

As you might guess, the protocol doesn't scale well. As the user base grows and traffic increases, it might be too much for weaker clients (read mobile phones) to process. Again, streams are said to be the solution, but if there's a big inrush of network traffic the implementations might not be ready. This is a risk for this project, as a successful mobile client could cause such a surge.

If somehow your private key gets into wrong hands, they can read every message you ever received, even if you deleted them locally. Nobody can prevent an entity from collecting all encrypted objects just in case they might be of value someday. They can't read the messages you've *sent* though, unless they acquire your recipient's private key as well.

On the usability side, the addresses are barely human readable, and any method to make it look friendlier would most probably be either complicated to set up or make the client insecure, or both. Then most clients do not yet support attachments, and there's still some controversy on how to implement them into the protocol. Large attachments wouldn't be possible anyway, due to message size restrictions and POW.

1.3.3 Risks

As when you decide to use Tor, a software for enabling anonymous communication formerly known as 'The Onion Router', you could be flagged by some security agency as a potential threat.[4] By itself this isn't necessarily a problem, but if you're a journalist it might put your informants at risk, and combined with other red flags it's possible you are prevented from flying in or into the U.S.A.

1.4 Motivation behind this Thesis

1.4.1 Current state – what is missing?

Until recently there was no mobile client for Bitmessage, and the client that turned up since is stuck in its beta phase and doesn't work right now. The alternative is to use an e-mail relay server, but this means to give up the private key to this server and makes end-to-end encryption much more difficult to achieve for both parties. Therefore this might not be a viable option, especially if you can't run your own server.

1.4.2 How should it be?

We need mobile Bitmessage clients that allow the user to choose their levels of convenience, privacy and resource hunger. There will always be trade-offs between needed traffic, battery use and privacy, and for each user the answer might look slightly different.

1.4.3 Why is it hard to do?

Bitmessage is very wasteful with resources by design. All nodes receive and store all messages, and to protect the network POW is required for all objects that are distributed, meaning some very CPU-heavy calculations need to be done. The protocol wasn't developed with mobile users in mind, and while smartphones are getting increasingly powerful, there is at least the issue of battery use to watch out for, and many users have a limited data plan.

1.4.4 Why me, and how do I intend to do it?

I have seven years of experience developing Java applications, and was programming Android apps from the moment I had my "Android Dev Phone 1". As I developed Jabit, a Java implementation of the Bitmessage client, as my last project, I also have acquired extensive knowledge about the Bitmessage protocol.

There are a few optimizations that I intend to do:

- Connect to only one reliable node instead of eight random nodes. This should reduce battery usage, but yields some risk if the node is compromized. Also, the node must forward all messages to all connected mobile clients instead of the default eight random nodes.
- Don't save objects we can't decrypt. We can solely save their hashes, but this means we're using the network without supporting it. This also might be an attack vector.
- Only connect to the network if we're on Wi-Fi and charging. This means of course that we'll only receive messages when we're connected with a Wi-Fi and charging.

Of course every option has its own drawbacks, so they will be configurable. As for the POW: Jabit highly optimizes its calculation, which might be enough for modern smartphones.

Further optimizations might introduce a server component that *might* do

- Proof of work
- Request public keys, requiring us to give up some anonymity towards the server.
- Inform the client about new messages sent to its addresses. This would mean to give up our anonymity towards the server in the best case (which isn't supported by the protocol yet), towards the whole network (which is somewhat supported), or give up the private key to the server (which, for many users, is unacceptable).

To find out what is actually necessary, a naive implementation without optimizations will be done first.

1.5 The Jabit Bitmessage Library

The source code for Jabit is available at https://github.com/Dissem/Jabit

Jabit has been developed as a preparing project for this thesis. It implements (most of) the Bitmessage protocol in Java and can therefore easily be used both for server applications as well as Android apps.

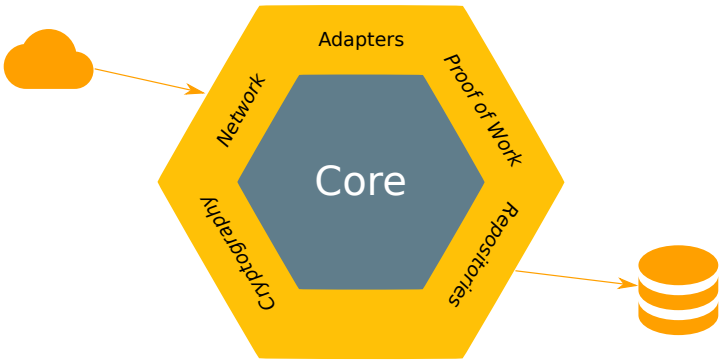
Almost all other clients are based on PyBitmessage, which is somewhat inconvenient as it always requires PyBitmessage to run along with the client. This isn't really a problem on desktop computers, but might render creating an easily deployable server application slightly more difficult, and is quite impractical on mobile devices.

1.5.1 Architecture

Jabit uses the Ports and Adapters pattern, also known as Hexagonal Architecture. This pattern isolates the core application from external dependencies. This vastly facilitates testing, as we can easily replace every dependency with mocks.

To achieve this isolation, we use adapters that encapsulate all necessary logic to communicate with an external dependency, e.g. a database. They communicate with the application via interfaces defined in the core module.

The core module contains all the data types and most parts of the protocol implementation. Ports and corresponding adapter implementations exist for data repositories, POW, network code and, more recently, cryptography (see 3.1.1 Bouncy Castle vs. Spongy Castle).



While there was hope to reuse the JDBC implementation of the *repositories*, it was clear that at some point someone might want to use a NoSQL database or some other means of storage. As explained in section 3.1.2 JDBC, an Android specific implementation unexpectedly came first.

The *proof of work* adapter exists in a single- and a multithreading version. The first mainly exists to show the concept, while the second one currently is always used to do the calculation, albeit remotely on a server or wrapped in some Android compatibility code. A future implementation that uses the GPU, which sometimes have thousands of cores, might be magnitudes faster and could easily replace the current worker.

The *network* code was put in an adapter as some Android specific tweaks were expected. This turned out to be wrong, but it didn't slow down development and might facilitate a future rewrite of the network code, so it still seems to be a good decision.

The *cryptography* adapter is a fine example why you should never rely on static helper classes. Rewriting the code to use an adapter was a pain and it wouldn't have been complicated to do it right the first time.

Adapter interfaces are all defined in package `ch.dissem.bitmessage.ports`.

1.5.2 How to Use

Add the Jabit dependencies to your Gradle build file:

```
1  apply plugin: 'java'
2
3  repositories {
4      mavenCentral()
5  }
6
7  dependencies {
8      compile 'ch.dissem.jabit:jabit-core:0.2.1-SNAPSHOT'
9      compile 'ch.dissem.jabit:jabit-networking:0.2.1-SNAPSHOT'
10     compile 'ch.dissem.jabit:jabit-repositories:0.2.1-SNAPSHOT'
11     compile 'ch.dissem.jabit:jabit-cryptography-bouncy:0.2.1-SNAPSHOT'
12     compile 'ch.dissem.jabit:jabit-extensions:0.2.1-SNAPSHOT'
13
14     compile 'ch.dissem.jabit:jabit-wif:0.2.1-SNAPSHOT'
```

```
15
16      compile 'com.h2database:h2:1.4.187'
17  }
```

Of course, Maven will work as well, or if you really want, you can add the dependencies by hand.

`ch.dissem.jabit:jabit-wif:0.2.0` is optional, you'll need it if you want to export to or import from the Wallet Import Format (WIF), which is used by PyBitmessage.

For Android clients you'll need to use `jabit-cryptography-sc` instead of `jabit-cryptography-bc`.

Next, create a `BitmessageContext`:

```
1  JdbcConfig jdbcConfig = new JdbcConfig();
2  BitmessageContext ctx = new BitmessageContext.Builder()
3      .addressRepo(new JdbcAddressRepository(jdbcConfig))
4      .inventory(new JdbcInventory(jdbcConfig))
5      .nodeRegistry(new MemoryNodeRegistry())
6      .messageRepo(new JdbcMessageRepository(jdbcConfig))
7      .powRepo(new JdbcProofOfWorkRepository(jdbcConfig))
8      .networkHandler(new DefaultNetworkHandler())
9      .cryptography(new BouncyCryptography())
10     .listener(new BitmessageContext.Listener() {
11         @Override
12         public void receive(Plaintext plaintext) {
13             // TODO: notify the user
14         }
15     })
16     .build();
17
18 ctx.startup();
```

Line 13 is where you do whatever you want to notify the user about new messages. They are already persisted, so there's no need to do that. Line 17 starts a Bitmessage node, i.e. connnecting your application to the network.

Then you might want to create an identity

```
1  BitmessageAddress identity = ctx.createIdentity(false);
```

and add a contact

```
1  BitmessageAddress contact = new BitmessageAddress(
2      "BM-2cXRfaXKbGyxuXfkkatU6vBhhz9vj7ReJ6");
3  address.setAlias("Chris");
4  ctx.addContact(contact);
```

to which you can send a message

```
1  ctx.send(identity, contact, "Test", "Hello␣Chris,␣this␣is␣a␣message.");
```

That's it, you've got a basic Bitmessage client.

2 The Bitmessage Protocol

2.1 Bitmessage Commands

Commands are messages between nodes, used to initialize a connection and exchange information about the network and available objects.

The **magic** is a number to mark the beginning of a command. It is always `0xE9BEB4D9`, but *could* be changed for test clients so they can't connect to the real network.

A **command** defines how the payload looks like and what the node is supposed to do with it. The commands used by the Bitmessage protocol are *version*, *verack*, *addr*, *inv*, *getdata* and *object*. Their uses are specified in the Bitmessage Protocol Specification.[5]

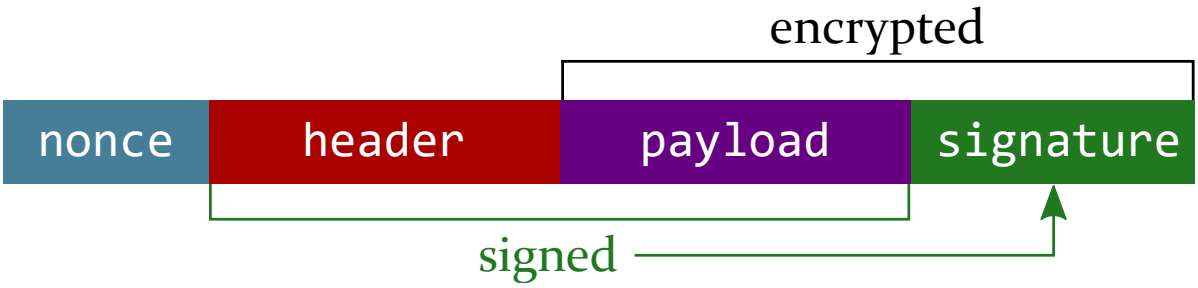
Size defines how many bytes of payload need to be read.

The **checksum** is used to discover transmission errors.

Payload is the data that belongs to a command. It could be anything from empty to a complex object containing a message.

2.2 Bitmessage Objects

Objects are distributed throughout the network using the "object" command. With exception of some legacy objects, they are signed and encrypted.



To prevent malicious flooding of the network and, to a lesser extent, spam, a **nonce** needs to be found such that a specific hash over the whole object represents a number lower than a calculated threshold. This is called proof of work (POW), and described in detail in section 2.4.

The object **header** consists of the expiration time, type and version of the object, and stream

number. (Streams are an optimization feature of the protocol, so it should stay usable if many people start using Bitmessage.)

The **payload** contains the actual data, depending on the message type.

The **signature** covers everything except the nonce and is encrypted along with the payload, thus covering the unencrypted payload.

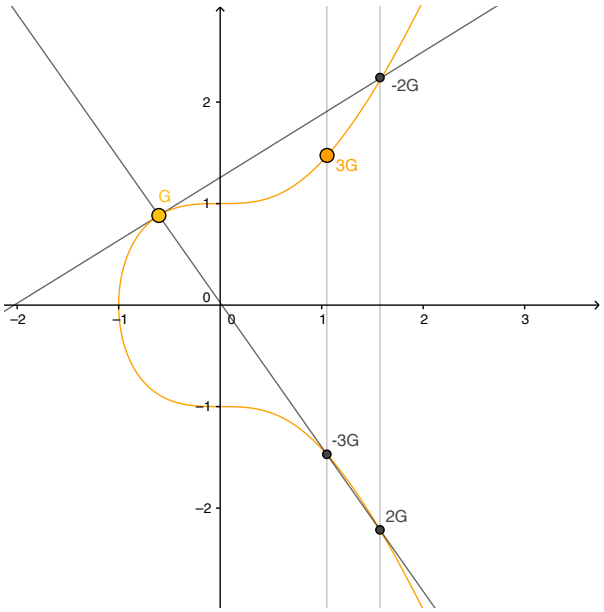
There are four different types of objects in the Bitmessage protocol: *getpubkey* is used to request a public key to some address, *pubkey* contains said public key, *msg* is a typical person-to-person message, and *broadcast* is a message that is broadcast to anyone who subscribed to the sending address – which needs to be known in order to subscribe.

2.3 Encryption

Bitmessage uses elliptic curve cryptography for both signing and encryption. The math behind it is rather complicated, yet bases on the established principle to use a mathematical operation that has an inverse that can't be calculated, except by searching all possibilities. Instead of the typically used huge prime numbers, a point on an elliptic curve is multiplied by a high number.

Advantages of elliptic curves are the fact that we don't need to search for big primes, but also that the keys can be smaller while having the same encryption strength.

Addition on the curve means finding the point where the line through those points intersects with the curve, and then find its reflection on the x-axis. If you add a point to itself, you use the tangent instead. In the example, G is multiplied by 3 by first adding $G + G$ and then $2G + G$, resulting in point $3G$.



Obviously you can't find a line from just one point, so there is no way to calculate $\frac{P}{3}$, and $\frac{P}{G}$ can only be found by searching all options, assuming $P = Gn$, which holds true in elliptic curve cryptography.

In the following example, key pairs are represented by pairs of the same letter, lower case for the private key, and uppercase for public keys. This corresponds to the convention of using lowercase letters for scalars and uppercase letters for points.

The user, let's call her Alice, needs a key pair, consisting of a private key

$$k$$

which represents a huge random number, and a public key

$$K = Gk$$

which represents a point on the agreed on curve – by default that's *secp256k1* for Bitmessage. Please note that this is not a simple multiplication, but the multiplication of a point along an elliptic curve. G is the starting point for all operations on a specific curve.

Another user, Bob, knows the public key. To encrypt a message, Bob creates a temporary key pair with private key

$$r$$

and public key

$$R = Gr$$

He then calculates

$$Kr$$

uses the resulting point to encrypt the message by calculating a double SHA-512 hash over the x-coordinate and sends K along with the message.

When Alice receives the message, she uses the fact that

$$Kr = Gkr = Grk = Rk$$

so she just uses Rk to decrypt the message.

The exact method used in Bitmessage is called Elliptic Curve Integrated Encryption Scheme or ECIES, which is described in detail on Wikipedia.[6][7][8]

2.4 Proof of Work

While invented as an anti-spam feature, Bitmessage uses proof of work (POW) mainly to protect the network. Users can demand higher POW for their identities as additional protection.

If not for POW, any malicious party could overload the network with an immense amount of messages, preventing other messages from being distributed and filling the hard disks of the weaker

nodes, practically disabling the whole network. Therefore an object is only distributed to other nodes if its proof of work is correct and its time to live isn't up.

The difficulty is calculated through a function of message size and time to live, i.e. a long lived message or a large one is more expensive to send, in terms of computation time.

$$d = \frac{2^{64}}{n(l + \frac{tl}{2^{16}})}$$

d target difficulty
n number of trials per byte
l payload length + extra byte
t time to live in seconds

The virtual *extra bytes*, currently at least 1000, prevent flooding the network with a great number of extremely small objects by adding some minimal difficulty that's necessary to send a message.

Time to live can be up to 28 days, which is the default for *pubkey* objects, but for normal messages two days are commonly used.

As proof of work, the client must find a nonce such that the first eight bytes of the object's hash (including nonce) represent a smaller number than the target difficulty d.

Note that the following pseudo code is optimized for readability. In a production implementation the loop should be as optimized (for speed) as possible, as it will be run so many times every nanosecond counts. Object creation and expensive conversions should carefully be avoided.

```
byte[] nonce = new byte[8]

do {
    nonce++
} while (
    lt(
      target,
      SHA512(SHA512(nonce + initialHash))
    )
);
```

initialHash	a single SHA-512 hash over the whole object except for the nonce
lt(a, b)	a helper method that takes the first eight bytes of both arrays and returns `true` if the ones from a represent a smaller number than the ones from b and `false` otherwise
$+$	in this context, a concatenation of both arrays `[1, 2] + [3, 4] = [1, 2, 3, 4]`
$++$	an increment by one, treating the array as a number `[1, 255]++ = [2, 0]`

3 Naive Implementation

The naive implementation attempts to use the Jabit Bitmessage library as it is, with as little mobile optimizations as possible. The plan was to either discard or improve it afterwards. Fortunately, improvement was possible.

3.1 Unexpected Problems

Most problems can be summarised as this: Android builds on the Java language, but not on the Java platform. While the programming language is the same and most libraries can be used without any restrictions, there are some subtle differences that make a programmer's life hard – some due to the limited resources of a mobile handset, others due to design decisions made by the Android development team.

3.1.1 Bouncy Castle vs. Spongy Castle

Jabit heavily relies on Bouncy Castle, a very popular open source encryption library.[9] Sadly, Android ships with a broken version of Bouncy Castle. Even worse, when building an Android app, the toolchain just discards any Bouncy Castle dependencies in favour of the built-in, broken version.

Roberto Tyley recognized this problem and built a fork of Bouncy Castle, called Spongy Castle.[10] It basically just replaces "Bouncy" with "Spongy" wherever necessary, so it doesn't get discarded during the build process. This works fine and is quite easily done. Unfortunately, it doesn't work on the desktop: the Oracle JVM requires Security Providers to be signed, which is done for Bouncy Castle builds, but not so for the Spongy derivation.

As forking Jabit wasn't an option, the whole cryptography part had to be refactored into an exchangeable module and implemented twice, in both a bouncy and a spongy manifestation.

3.1.2 JDBC

Android has its own application programming interface (API) to access the included SQLite database. While it's a nice, easy to use API, the Android team didn't deem it necessary to support JDBC, which is the Java standard API to connect to databases.

There is an open source project attempting to implement a Java database connectivity (JDBC) driver for Android's SQLite database called SQLDroid, which looked very promising. But as it turned out, it lacks essential features, such as returning the automatically generated key of an inserted row. It doesn't even have the courtesy of throwing a NotImplementedException for missing features, instead it just doesn't do anything and returns *null* where a result is required, making debugging unnecessarily tedious.[11]

Discovering SQLDroid was unfit for the job finally took more time than reimplementing all repositories using the Android database API. As the necessary ports already existed, the Jabit library could be left unchanged. Yet the futile attempt to use JDBC caused many adjustments, most of them only to be reverted later on.

3.2 Expected Problems

There were a few problems we actually expected. They need to be examined and possibly fixed as part of this thesis.

3.2.1 Proof of Work

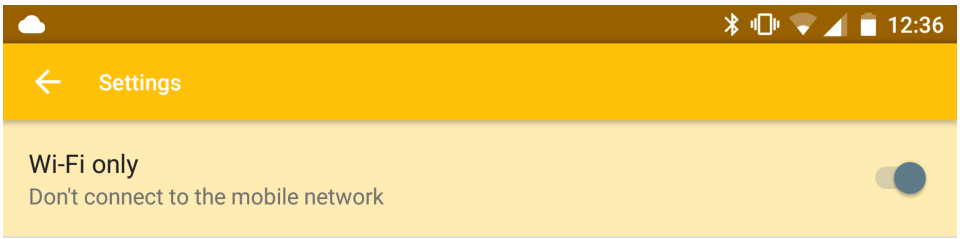
Although modern smartphones tend to have faster processors than cheap personal computers, proof of work (POW) for sending a public key takes more than 15 minutes on a device with four cores at 2.5 GHz. Even worse, during this time it uses so much power that the device discharges even when connected to a power supply.

But as it used the default time to live of 28 days, cutting it down to two days shortened that time to a much more acceptable 1-2 minutes. Weaker devices still want to rely on a server to do their POW, but for more privacy conscious people and those with stronger devices it is a viable option to do it on the mobile.

3.2.2 Data Traffic

Not creating a full node that also distributes the objects it receives certainly reduces the necessary internet traffic, but a Bitmessage client still needs to download around 1 GiB of incoming objects each month.

The relatively crude solution to this problem is an option to prevent connecting to the mobile network.



3.2.3 Storage

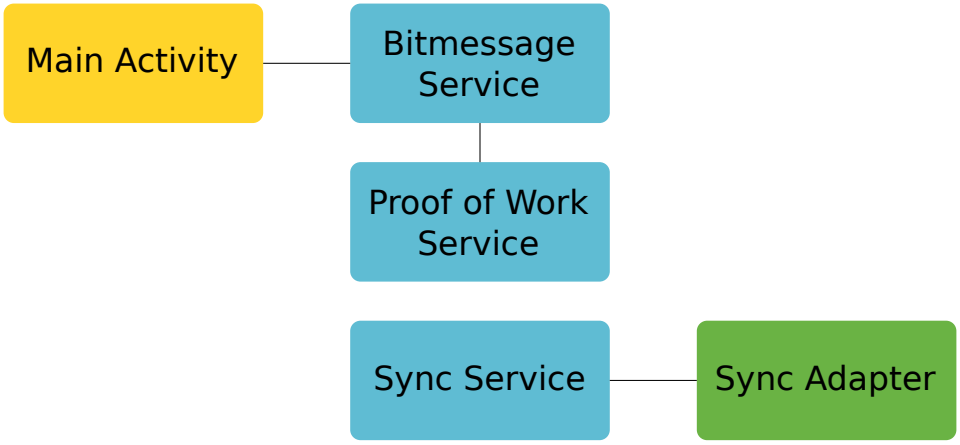
Limited storage isn't as big a problem as initially expected, experience showed that just barely over 100 MiB is used, which isn't a problem on current smartphones.

It shouldn't be too difficult to implement an option not to store any object data we don't need, but it certainly isn't a priority.

4 Android Specific Challenges

4.1 Application Lifecycle

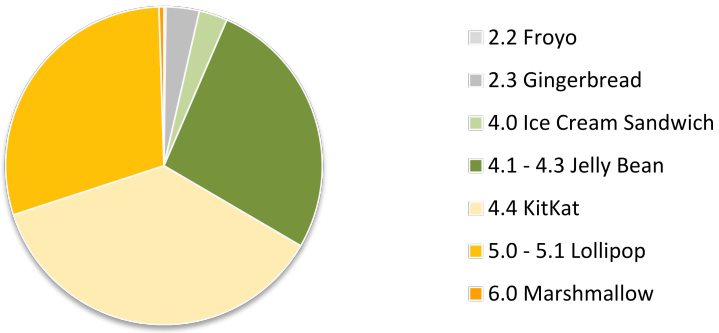
As with any good mobile system, Android doesn't hesitate to end any process of an application running in background, in order to save resources. Consequently anything that should be kept alive – network connections and POW come to mind – needs special treatment. This is done through foreground services. The networking part and POW hence needed to be moved into separate services. While not difficult, it needed some research to find out how it should best be done in our case.



It was decided to run all services as separate threads in the same process, which simplifies communication between those services, but has the risk of the whole app crashing if there is an error in a service.

4.2 Deprecated Java Versions

Jabit uses many features from Java 7, such as *try-with-resources* and `java.util.Objects`. Testing with older devices revealed that those features are only available on Android 4.4 'KitKat' and later.



As this would have meant a tedious search for the use of these features and their rewrite, it was decided to raise the system requirements from Android 4.0 'Ice Cream Sandwich' to 4.4 'KitKat' instead. This means that instead of 96.4% of the devices, the app now only runs on 66.6% of the devices that are actively using Google Play™.[12]

The Android development environment provides useful help to prevent such problems, but not if they originate from depending libraries such as Jabit.

5 Optimizations

5.1 Sync Adapters

Android provides an API that can vastly reduce battery usage for apps that regularly update data over the internet. The system triggers synchronization, so it can optimize its sleep modes. As it knows when the network access is done it can put the radio back to sleep immediately, unlike other kinds of network access where the system stays connected for some time in case a new connection needs to be established.

Imagine five apps synchronizing every five minutes, one after the other. In the worst case scenario the radio would need to wake up every minute, synchronize and could barely go back to sleep before the next app wants to synchronize. With the sync adapter the system triggers synchronization for all five apps and when all are done it can put the radio back to sleep.

Although it's not intended, synchronization can be done without any modifications to the Bitmessage protocol. What we need is a trusted node that is always available and therefore best run on a proper server. On successful connection the protocol exchanges all new messages, so we just need to connect, wait until new messages were exchanged, and then disconnect.

As Jabit is a Java library, it was trivial to create a server application using Spring Boot and Jabit. It turned out though that it's necessary to limit the number of connections in a Bitmessage server, therefore connections are now being severed after 12 hours or when a limit of 100 connections is reached.

5.2 Server Side Proof of Work

To reduce power consumption and possibly reduce the time to send a message, POW can be calculated by the synchronization server. This feature needs some changes on both server and client, and some custom extensions to the protocol.

5.2.1 Initial Idea

We first wanted the server to accept messages without POW, which it will compute and fill in, and then relay that message. Of course this would leave the server extremely vulnerable to denial of service (DOS) attacks – an attacker could just send a bunch of messages and the server would be busy the rest of its life.

Fortunately, Bitmessage has very secure authentication built in: every message is signed by a private key, and can be securely verified by anyone who knows the sender's address. We'll just send a message to the server that contains the actual message as content. All nonce fields are set to zero. The server checks if the sender is on a white list and then calculates the nonce and relays the complete message.

A broadcast message would be best qualified for this task, as the client wouldn't have to be configured with a recipient address. The server is configured with a list of addresses that are allowed to request POW and the client just needs to know the server's IP address or host name.

5.2.2 Acknowledging Issues

Unfortunately this approach has some issues. For one, sending a message would be very different using this method, requiring some major changes on the client side. Moreover, it would be impossible to generate acknowledgements.

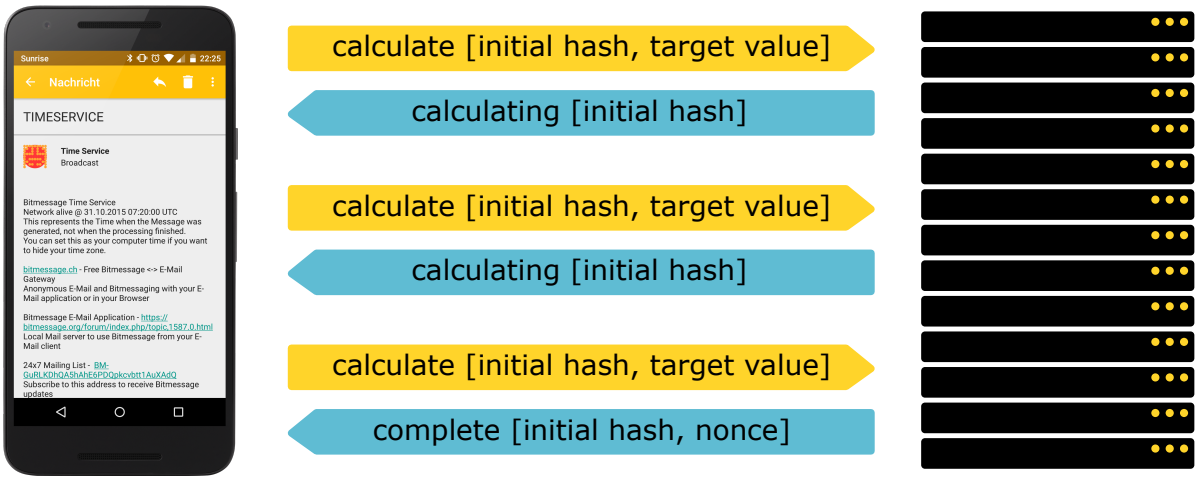
Acknowledgements are sent from the receiving client to notify the writer that the message was received. Think of it as a stamped addressed envelope delivered with a letter. The acknowledgement is part of the encrypted message the server can't read, so it couldn't calculate its POW.

Jabit doesn't support acknowledgements yet, but it would be a pity to prevent it by design.

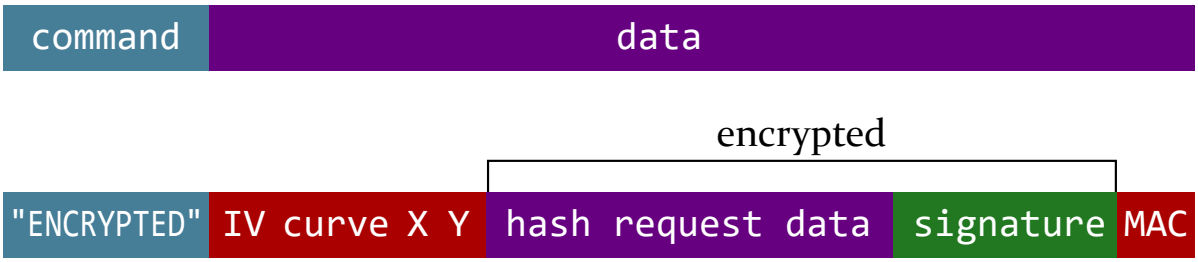
5.2.3 POW Protocol

Much of the planned design can be reused – the encryption used by broadcasts, signing the messages – but we need a wholly separate protocol, with some extra roundtrips to the server. Now the server just receives the parts needed for POW calculation: a hash and a target value.

When the nonce is found, it's saved on the database. The client will regularly poll for it (e.g. during synchronization cycles) until it retrieves the nonce. A great benefit of this method is the possibility to implement it as an adapter.



For this feature, a new command was introduced, which needs to be implemented on both server and client. It has the same structure as any Bitmessage command, and *custom* in the command field. As payload it contains a signed and encrypted block, containing the following structure:



The custom commands for the proof of work feature are encrypted and signed, including the kind of request or response. The *command* field therefore always says `ENCRYPTED`. Afterwards comes the *encryption header*, including the initialization vector (IV), the curve type (should normally be 0x2CA), and the X and Y coordinates of a random public key (see 2.3 Encryption, variable R).

The payload always consists of the initial hash (the base for POW calculation) and the kind of request or response. The data field contains the target difficulty for the `CALCULATE` request, and the calculated nonce for the `COMPLETE` response. For the `CALCULATING` response data is an empty array.

`CALCULATE` is also used to poll the server. This has the benefit that if there is a temporary problem on the server, it will automatically start POW calculation when the server is ready again.

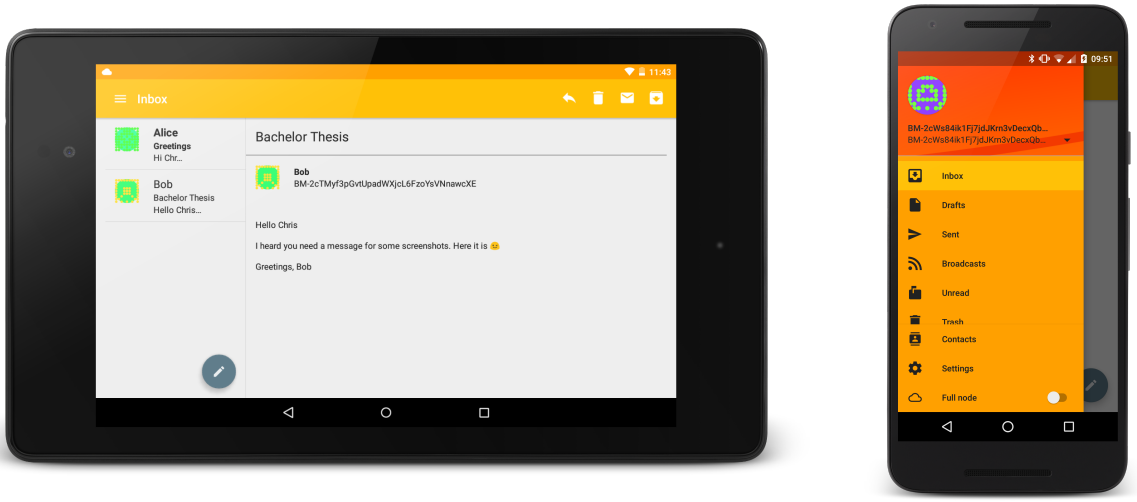
6 Artefacts

6.1 App

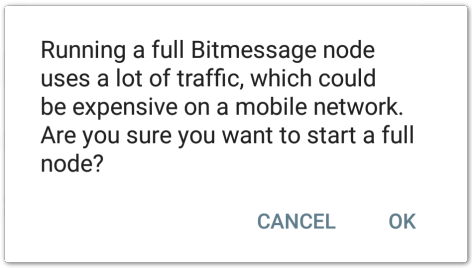
You can find the app's source code at https://github.com/Dissem/Abit

6.1.1 User Interface

Abit tries to imitate state of the art e-mail clients. Users of Android's stock mail client should feel right at home.

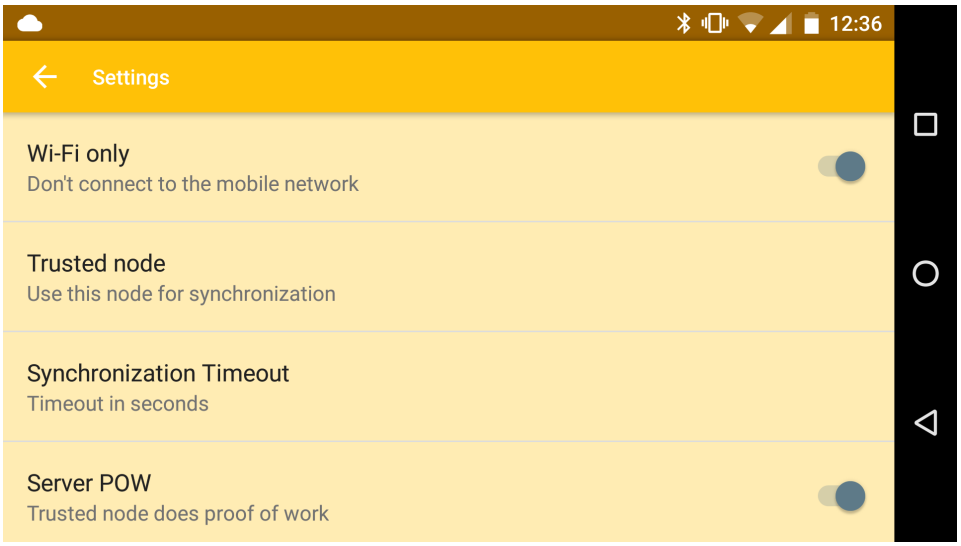


A curiosity might be the 'full node' switch at the bottom of the side drawer, which toggles a fully functioning Bitmessage node. As this uses a lot of traffic, the user is warned if they activate it on the mobile network. (Provided 'Wi-Fi only' is set in the settings, which is the default.)



To both remind the user something resource hungry is going on and keep the process alive, ongoing notifications are being shown for long running tasks.

6.1.2 Settings



If *Wi-Fi only* is selected, Abit will not synchronize unless it's connected to a Wi-Fi network. If you try to start a full node in this mode, you will be warned and asked if you want to continue. It is switch on by default in order to protect the user from using up his data plan.

The *trusted node* is needed for the synchronization feature to work (receiving messages while not being a 'full node').

Synchronization will stop trying to fetch object when the *synchronization timeout* is reached. This is a safety measure so it doesn't stay connected indefinitely if there's a problem, but might keep you from quickly getting all objects when the client runs for the first time.

If *server POW* is active, all POW will be done on the server. Please note that if there is a problem on the server, POW won't be done at all, and a server will only do POW for you if your identity is registered as an accepted client. In case of errors, POW will be done if the feature is turned off and the client is restarted, which you might have to do by force.

6.2 Server

The server's source code is available at https://github.com/Dissem/Jabit-Server

For the app to be really useful, it relies on a server component. If you don't require the POW feature it could be any Bitmessage client that can be accessed over a known IP address or host

name, but the provided server application is very easy to deploy and, as mentioned, can do POW for the mobile client.

The server also provides a REST API and a web interface, but as they didn't have any priority they don't work properly.

6.3 Installation

You can simply run `java -jar jabit-server.jar`. By default the server will provide a web interface on port 8080 and listen for Bitmessage connections on port 8444.

6.3.1 Configuration

Five configuration files will be created if they don't exist yet. If you modify them by hand, a restart will be required to apply the changes. The files may contain Bitmessage addresses with different meanings.

admins.config	Addresses in this list are allowed to administrate the server.
clients.config	Addresses in this list are allowed to request POW

The other three files concern the REST API:

whitelist.config	If there are any entries, only those addresses can be requested.
blacklist.config	Those addresses can't be requested.
shortlist.config	For those addresses, only the last five broadcasts will be shown

The configuration files can also be changed by a registered administrator, through sending a message with subject <*command*> <*list*> and a list of addresses in the body, each on a separate line. The commands are

set	replaces the list with the body content
add	adds the given addresses to the list
remove	removes the given addresses from the list

Example:

Subject	add client
Body	BM-2cUau5uxBYCK2Z2TVwUZnnNfYW5yyutekC BM-2cXDjKPTiWzeUzqNEsfTrMpjeGDyP99WTi

Adds those two addresses as clients to allow POW requests.

As it is a Spring Boot application, the file `application.properties` is used for all other configurations. The most important ones are:

```
1 server.port                      = 8080
2 server.address                   = 0.0.0.0
3
4 bitmessage.port                  = 8444
5 bitmessage.connection.ttl.hours = 12
6 bitmessage.connection.limit      = 100
```

1: *server.port*	On this port the server listens to HTTP requests
2: *server.address*	Is probably either 0.0.0.0, where the server listens on all interfaces, or 127.0.0.1 if you want to disable the web interface or route it through a web server.
4: *bitmessage.port*	On this port the server listens to incoming Bitmessage connections, e.g. from Abit users but also from random nodes.
5: *bitmessage.connection.ttl.hours*	The maximal time a client is allowed to stay connected to the server.
6: *bitmessage.connection.limit*	The maximum number of connections allowed for the server. If there are more connections, the oldest ones will be severed.

7 Related Works

7.1 E-Mail Gateways

A free e-mail gateway can be found at https://bitmessage.ch.

When using an e-mail gateway, you can use your preferred e-mail client, so we can assume that daily user experience is at least as good as with the native app. Setup isn't very complicated, but still not as easy as two taps for a new identity.

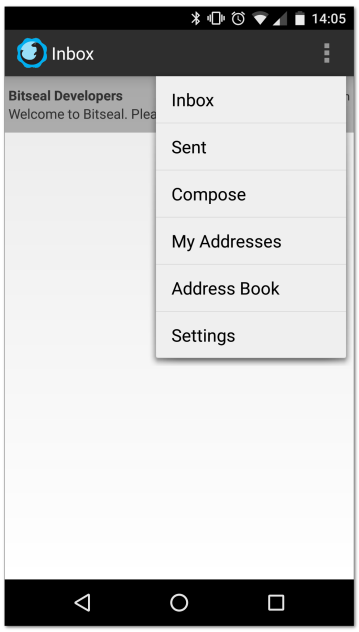
As for security, we must really trust the gateway provider, because they possess our private key. Due to the way encryption is done in Bitmessage, there is no other way to solve this problem except for using PGP or a similar product, but this is quite troublesome to set up and use for both parties.

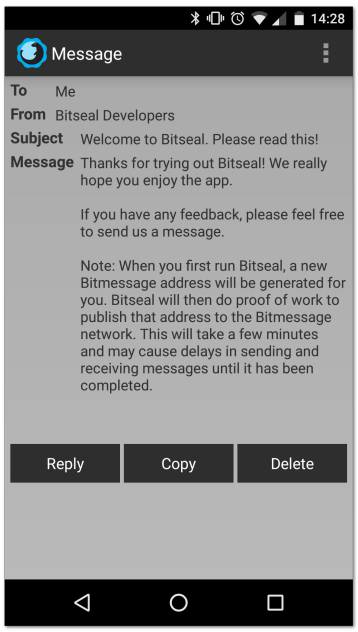
7.2 Bitseal

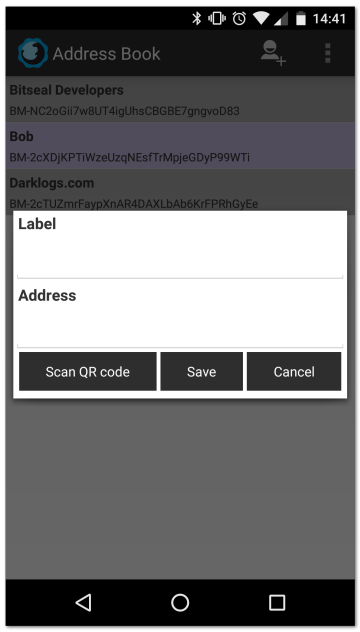
Bitseal isn't available on Google Play anymore, but its sources can be found on GitHub:
https://github.com/JonathanCoe/bitseal
It isn't actively being developed right now, and unfortunately the servers needed for it to work aren't currently running. It would be possible to set up a private server, but this is out of scope of this thesis.

As it aims to be a full Bitmessage client as well, Bitseal is more directly comparable to Abit.

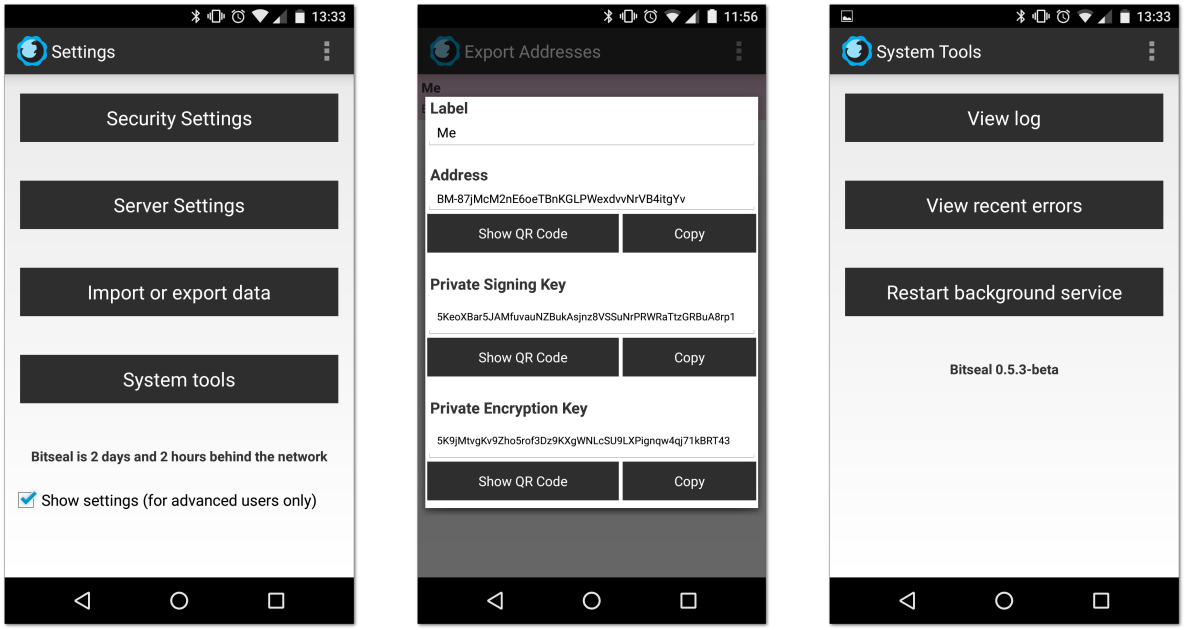
7.2.1 User Interface







7.2.2 Settings



The security settings let the user enable database encryption. This adds some security if the device is compromised.

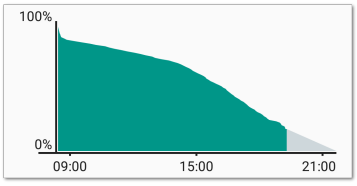
The server settings let the user add and remove servers. Other than Abit, Bitseal supports multiple servers to be set.

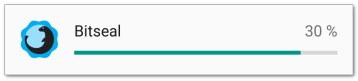
Import or export data lets the user export and import their private keys. While this is a planned feature for Abit, we prefer the wallet import format used by PyBitmessage rather than exporting the encryption and signing keys separately.

The system tools are very useful for debugging, sporting an option to view the log and recent errors, as well as restarting the background service.

7.2.3 Power Usage

A first test showed that Bitseal used about 30% of the device's power throughout the day. While POW for the public key is clearly responsible for the first power drop, the second decline isn't that easily explained.





Abit hardly ever turns up when in synchronization mode, and keeps quite low even in 'full node' mode. As long as it's not doing too much POW it won't be a nuisance.

7.2.4 Server

Bitseal relies on a custom server component for some of its crucial functions. However it still requires to download all network data and does POW on the handset.

Abit on the other hand can be fully functional without a server, and theoretically can use any node for its synchronization feature. Only if you want to do POW on the server you'll need the one described in 6.2 Server.

8 Future Work

8.1 Known Bugs

- If a user tries to import their identity as a contact, the private key is deleted.
- A rare rendering error exists where different lists of messages are being drawn on top of each other.
- The server's user interface doesn't work.
- There is still no icon!

8.2 Missing Features

- Drafts aren't stored.
- The user interface for sending broadcasts is still missing.
- Identities:
  - rename
  - show as QR code
  - share or copy to clipboard
  - import and export
- Jabit:
  - It doesn't support acknowledgements yet.
  - Stream support must be tested and probably fixed.

9 Conclusion

I'm quite proud of this app. As long as you don't start a full node it hardly ever turns up in the battery usage statistics, its data usage is under control and I think it looks quite nice, too. There is always room for improvements, but it's time to let a broader audience test it.

As often it wasn't the big questions that took the longest, nor the ones I expected. I've spent a lot of time exploring how to do things in a way that is both consistent with the Android ecosystem and keeps the code maintainable.

One lesson I learned the hard way: Never take shortcuts. The first mistake was using static methods for the cryptography. Then there are some changes to the Jabit library that I didn't create proper tests for, and I fear they are just about to come back and haunt me: during the last week I noticed some irregularities during key exchange *and* synchronisation. I think I fixed the key exchange, but I'm not 100% sure synchronization is working reliably.

A project like this is never quite done, it needs recurring attention. It was fun and fascinating work, but the real challenge will be finding people willing to contribute. So if you feel intrigued by this app, please contact me. Together we can make sure this project will survive.

References

[1] James Ball. Nsa stores metadata of millions of web users for up to a year, secret files show, 2013.
http://www.theguardian.com/world/2013/sep/30/nsa-americans-metadata-year-documents.

[2] National security letter.
https://yale.app.box.com/NSL-Attachment-Unredacted.

[3] Jonathan 'Atheros' Warren and 'AyrA'. Bitmessage wiki: Stream, 2015.
https://bitmessage.org/wiki/Stream.

[4] Kim Zetter. The nsa is targeting users of privacy services, leaked code shows, 2014.
http://www.wired.com/2014/07/nsa-targets-users-of-privacy-services/.

[5] Jonathan 'Atheros' Warren and Jonathan Coe. Bitmessage wiki: Protocol specification, 2015.
https://bitmessage.org/wiki/Protocol_specification.

[6] Wikipedia: Elliptic curve, 2016.
https://en.wikipedia.org/wiki/Elliptic_curve.

[7] Wikipedia: Elliptic curve cryptography, 2015.
https://en.wikipedia.org/wiki/Elliptic_curve_cryptography.

[8] Wikipedia: Integrated encryption scheme, 2015.
https://en.wikipedia.org/wiki/Integrated_Encryption_Scheme.

[9] The legion of the bouncy castle.
https://www.bouncycastle.org.

[10] Roberto Tyley. Spongy castle.
https://rtyley.github.io/spongycastle/.

[11] Sqldroid.
https://github.com/SQLDroid/SQLDroid.

[12] Dashboards.
http://developer.android.com/about/dashboards/index.html.