Bern University
of Applied Sciences

Bachelor Thesis

An Android Client for Bitmessage

Author Christian Basler
Advisor Kai Brünnler
Expert Daniel Voisard
Date January 6, 2016

Trademarks

Abbreviations

CPU	central processing unit
GPU	graphics processing unit
VM	virtual machine
POW	proof of work
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

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. Its users are very privacy conscious. 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 for accessing databases. And finally, the devices differ vastly processing power. The implementation we propose solves these problems.

In addition there were two optional optimisations 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 — it's their choice.

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

Contents

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, or with whom we communicate. 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.[7]

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?[3]

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.

Other that Bitcoin, Bitmessage does not have a block chain. This is a common misconception, as those protocols otherwise share many of their ideas.

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[9], 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 surge in network usage the implementations might not be ready. Unfortunately for this project, a successful mobile client could cause such a surge.

If somehow your private key gets into wrong hands, they might be able to 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 the might be of value some day. They can't read the messages you've *sent* though, unless they got your recipient's private key as well.

On the usability side, the addresses aren't really 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 supports 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 might be flagged by some security agency as a possible threat.[11] By itself this isn't necessarily a problem, but if you're a journalist it might be a risk for your informants, and combined with oder red flags you might be 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 very wasteful to the devices resources, draining the battery in little time. The alternative is to use an e-mail relay server, but this means to give up the private key to this server and end-to-end encryption is much more difficult to achieve. 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 allows 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 optimisations 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 compromised. 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 optimises its calculation, which might be enough for modern smartphones.

Further optimisations 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 impementation without optimisations will be done first.

1.5 The Jabit Bitmessage Library

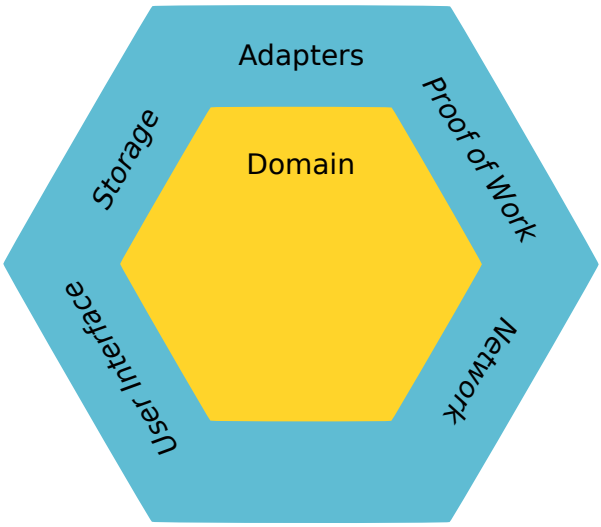
The source code for Jabit can be found at https://github.com/Dissem/Jabit

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 follows the Ports and Adapters architecture. There is a domain module which contains all the data types and most parts of the protocol implementation, and provides several ports, to which adapters can be attached. 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).



TODO: update image

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. Unfortunately, an Android specific implementation came first, as explained in section 3.1.2 JDBC.

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 immplementation 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.

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.[10]

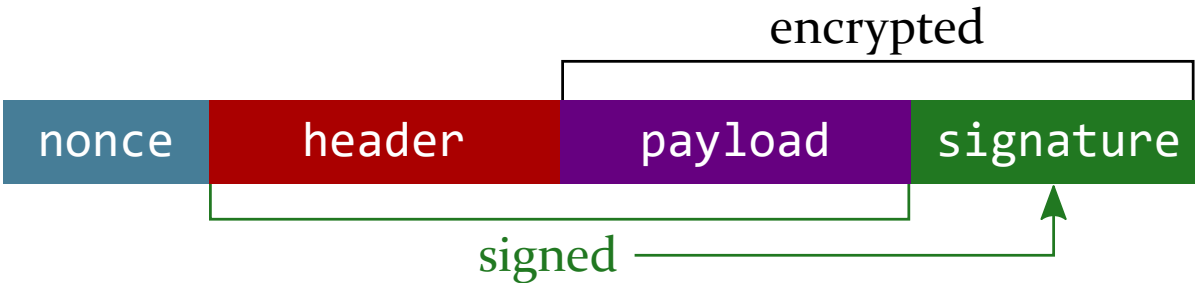
Size defines how many bytes of payload need to be read.

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 ar an optimisation 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 is magnitudes more complicated than the operation itself. Instead of the typically used huge prime numbers, a point on an elliptic curve is multiplied by a high number.

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

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.

TODO: try to find a graphic enlightening the key exchange

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.[5][6]

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 linear 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.

```
1 byte[] nonce = new byte[8]
2
3 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 is meant `[1, 2] + [3, 4] = [1, 2, 3, 4]`

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

3 Naive Implementation

The naive implementation attempts to use the Jabit Bitmessage library as it is, with as little mobile optimisations 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 decicions 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.[2] 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 recognised this problem and built a fork of Bouncy Castle, called Spongy Castle.[8] 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.[4]

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

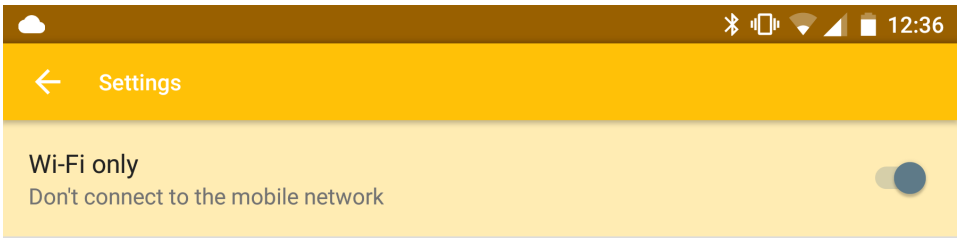
Although modern smartphones tend to have faster processors than cheap personal computers, proof of work (POW) for sending a public key takes more than 15 minutes on a device with four cores at 2.5 GHz. Even worse, during this time it uses so much power that the device discharges even when connected to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

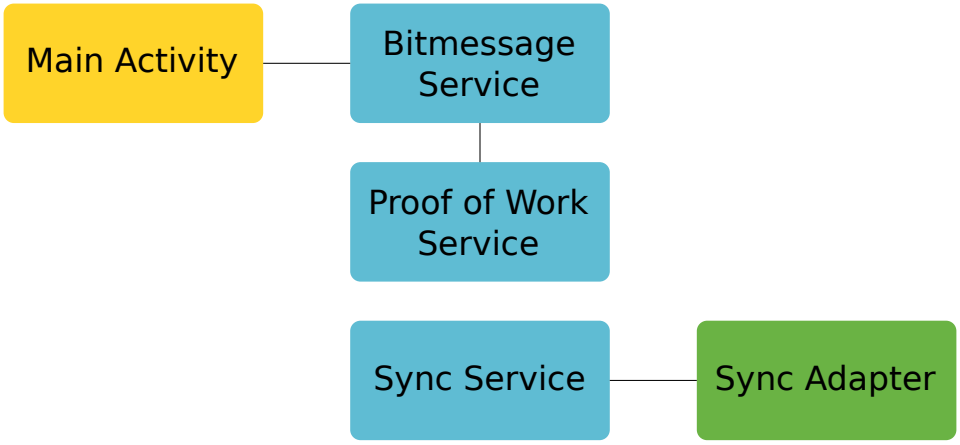
Limited storage isn't as big a problem as initially expected, experience showed that just barely over 100 MiB is used, wich isn't a problem on current smartphones.

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

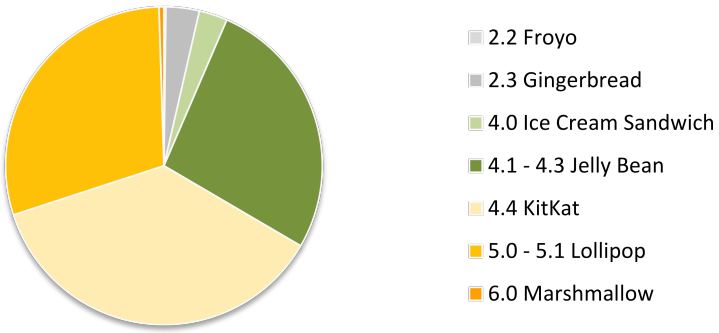
As with any good mobile system, Android doesn't hesitate to end any process of an application running in background, in order to save resources. Consequently anything that should be kept alive – network connections and POW come to mind – needs special treatment. This is done through foreground services. 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™.[1]

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

5 Optimisations

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 synchronisation, so it can optimise 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 synchronising 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 synchronisation for all five apps and when all are done it can put the radio back to sleep.

Although it's not intended, synchronisation 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 synchronisation 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. Ther 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.

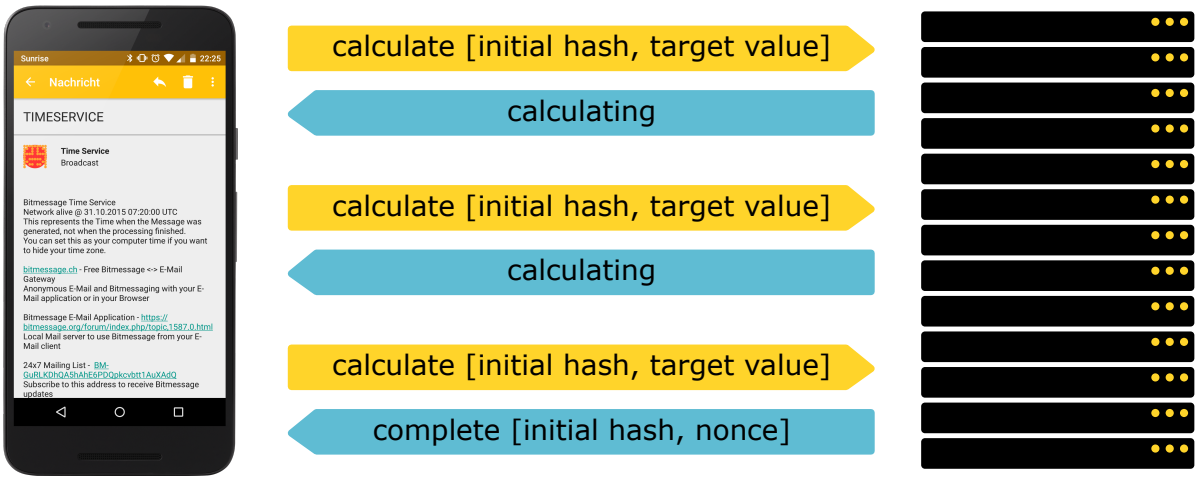
Acknowledgements are sent from the receiving client to notify the writer that the message was received. Think of it as a stamped addressed envelope delivered with a letter. The acknowledgement is part of the encrypted message the server can't read, so it couldn't calculate its POW.

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

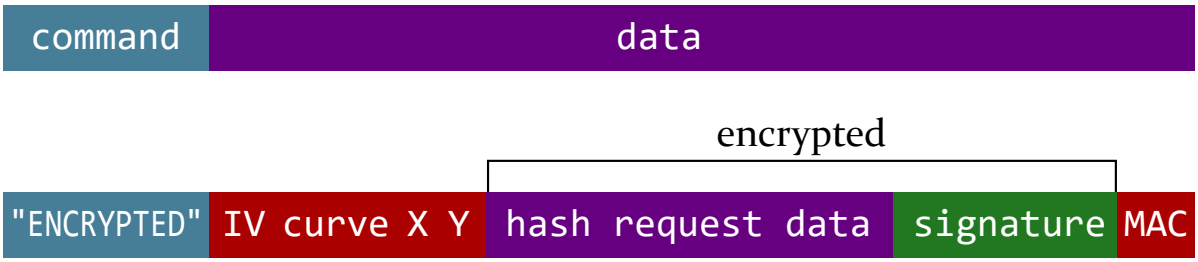
5.2.3 POW Protocol

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 initialisation 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 emtpy 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.

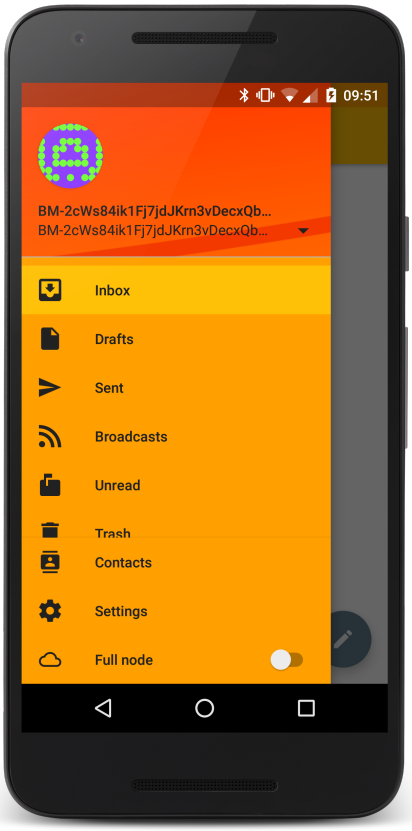
6 Artefacts

6.1 The App

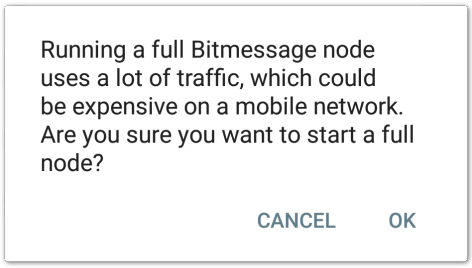
You can find the app's source code at https://github.com/Dissem/Abit

6.1.1 User Interface

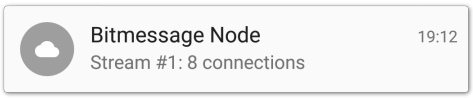
Abit tries to imitate state of the art e-mail clients. Users of Android's stock mail client should feel right at home.



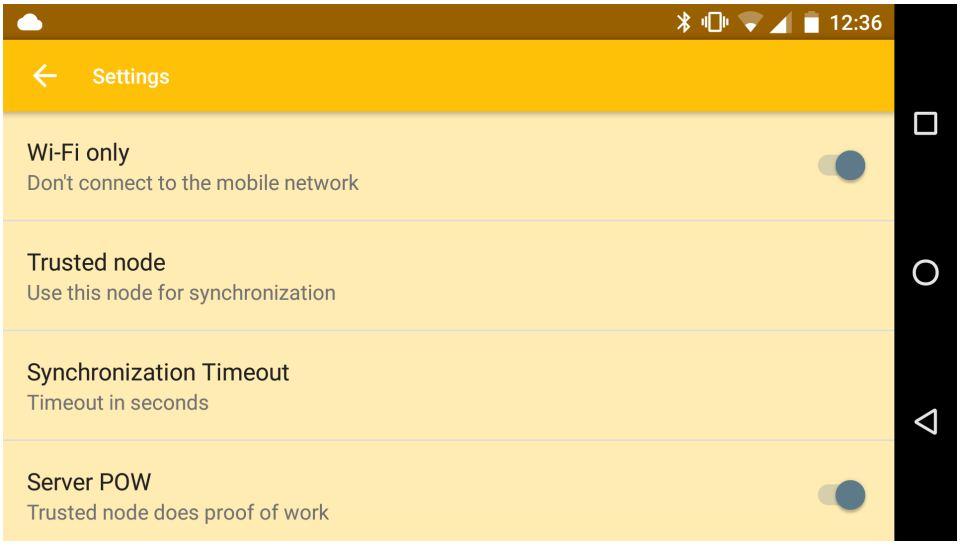
A curiosity might be the 'full node' switch at the bottom of the side drawer, which toggles a fully functioning Bitmessage node. As this uses a lot of traffic, the user is warned by default if they activate it on the mobile network. (Toggled via 'Wi-Fi only' option in the settings.)



To both remind the user something resource hungry is going on and keep the process alive, an ongoing notification is being shown.



6.1.2 Settings



If *Wi-Fi only* is selected, Abit will not synchronize unless its 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 synchronisation feature to work (receiving messages while not being a 'full node').

Synchronisation 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 registred 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 The Server Component

The server's source code can be found 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.

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.

TODO lists etc.

6.3.1 Configuration

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 diractly comparable to Abit.

7.2.1 User Interface

Bitseal

7.2.2 Settings

7.2.3 Power Usage

8 Future Work

8.1 Open Bugs

.

.

.

.

.

.

8.2 Missing Features

9 Conclusion

References

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

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

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

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

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

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

[7] 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.

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

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

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

[11] 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/.