Bern University
of Applied Sciences

Bachelor Thesis

An Android Client for Bitmessage

Author	Christian Basler
Tutor	Kai Brünnler
Date	September 20, 2015

Contents

1 Introduction

1.1 Current state – why is it bad?

Until recently there was not mobile client for the Bitmessage protocol, 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.

1.2 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.

1.3 Why is it hard to do?

Bitmessage is very wasteful with resources by design. All messages are being sent to and stored on all nodes, and to protect the network proof of work (POW) is required for all objects that are distributed. The protocol wasn't developed with mobile users in mind, and while smartphones are getting more and more powerful,[1] there is at least the issue of battery use to watch out for, and most users have limited traffic on their data plan.

1.4 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 great 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.

[1]Four cores is the state of the art with high end devices now, and there are even quite a few devices out there sporting eight cores.

- 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

- POW
- 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 is just a big NOPE).

1.5 Jabit 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. Adapters include the data repositories, network code and, more recently, cryptography (see BC vs. SC).

(TODO: image)

2 Part 1 – Naive Implementation

The naive implementation attempts to use the Jabit Bitmessage library as it is, with as little mobile optimizations as possible.

2.1 Unexpected Problems

Most problems can be summarized as this: Android builds on the Java language, but not on the Java platform.

2.1.1 Bouncy Castle vs. Spongy Castle

Jabit heavily relies on the Bouncy Castle library for all things encryption. Unfortunately, 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 favor of the built-in, broken version.

Some people recognized this problem, and built a fork of Bouncy Castle, called Spongy Castle. 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, this 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 derivate. As forking Jabit wasn’t an option, the whole Security part had to be refactored in an exchangeable module, and implemented twice, in both a bouncy and a spongy manifestation.

2.1.2 JDBC

Android has its own 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 JDBC driver for Android’s SQLite database called SQLDroid, which looked very promising. Unfortunately, it lacks essential features, such as returning the automatically generated key of an inserted row. Even worse, it doesn’t have the courtesy of throwing a NotImplementedException for missing features, instead it just does nothing and returns null where a result is required.

Unfortunately, discovering SQLDroid was unfit for the job took more time than reimplementing all repositories using the Android database API. As they were already implemented as adapters, no change was necessary on the Jabit library.

2.2 Expected Problems that Weren’t

2.2.1 Proof of Work

Due to the fact that modern smartphones tend to have faster processors than cheap personal computers, proof of work seems to be reasonably fast on those devices.