Client for TRSST

CS3010 Individual Project Report

Ross Edwards

Project Definition

Student's name – Ross Edwards
Course – Computing Science
Project title – Client for TRSST
What is the project about? -
The creation of a desktop client for using the TRSST social network, having the "look and feel of Twitter" but with a few extra features – for example, the ability to compensate content providers directly via Bitcoin.
What is the project deliverable? -
A desktop client that will use TRSST to both send and receive messages from and to one or more users securely.
What is original about this project? -
This will be the first completely desktop-based GUI client for TRSST – the current desktop clients run the web client in its own browser engine or are command-line based.

List of Contents

1. Executive Summary

Current forms of social networking such as Facebook, Twitter, Google+, etc. do not inherently support features to authenticate one's authorship, encrypt entries so that only a certain audience may view them, protect an author's content from tampering by external parties, and to add 'paywalls' or 'tip jars' to individual entries.

The TRSST protocol started development as an open-source project by Michael Powers in 2013 after a successful crowdfunding campaign. Powers wrote a "white paper" (TRSST Project, 2013) describing a protocol for a blockchain-based social network that fulfills the above features, as well as giving some user stories regarding the protocol, from the perspective of both clients and servers, and developed a client as a demonstration of the protocol. At the time writing, the development can be considered to be 'abandonware', as the last commit to the GitHub repository was on the 22nd October 2014.

This report presents a continuation of the Powers' work on the TRSST protocol. It presents a C#/.NET-based client that implements the same features that the TRSST protocol defines as its objectives, namely supporting content authentication, encryption and compensation using a blockchain-based solution. Also proposed are some improvements/changes to the protocol, to conform to specifications for external technologies and data structures given by the TRSST client.

2. Introduction

2.1: Description of TRSST

TRSST is a social networking protocol designed to provide functionality that other social networks such as Twitter and Facebook do not have, specifically encryption, anonymisation and decentralisation.

TRSST's white paper (TRSST Project, 2013) states several main points to its development, including:

- Support of the open web
- Support of freedom of speech
- Support of privacy
- Support of compensation for the creation of content

TRSST achieves these aims in the following ways:

Support of the open web

TRSST can be accessed through the browser, making external applications (such as the one being developed) unnecessary for accessing the network. In addition, TRSST supports existing feed formats such as Rich Site Summary (RSS) natively, without the need for external applications, a feature not present in Twitter and Facebook. There are also means to ensure that content marked as private is kept hidden from all unauthorised parties, and content marked as public is search-indexable and available to everyone.

Support of freedom of speech

TRSST has functionality to afford authentication of authorship and anonymity at the same time, through the usage of asymmetric encryption, again a feature that the most common social networks do not have without the aid of external software.

Support of privacy

TRSST has functionality to support anonymous authorship and encryption in message to one or more people. All decryption keys are held by the users, and no server holds an unencrypted key, thus protecting encrypted content if a server hosting said content was compromised.

Support of compensation for the creation of content

All account IDs in TRSST are also valid payment addresses for cryptocurrencies (such as Bitcoins, Litecoins and Dogecoins), derived from the account's public key. Note however (at least in the case of Bitcoin), that this results in address reuse when recieving cryptocurrencies, which has the possibility of compromising one's anonymity as explained on the website of the Bitcoin Project (2015).

2.2: Project Objectives

The main four objectives of the project are the main points of TRSST described above. The criteria for judging whether each objective has been fulfilled is described below:

2.2.1: Support of the Open Web

This objective can be said to be completed when, if one is not logged into an account, they may access any content that any account has posted and has not encrypted. In addition, they may access said content without having to install any external software.

While two-way interaction with the original TRSST network through HTTP(S) was not achieved due to the network having both an expired HTTPS certificate and a lack of formal specifications regarding what the network will accept, a "mock" network was set up on a cloud storage service to emulate an external server that clients communicate with. Users that have not set up a TRSST account can view any unencrypted entry posted to the network, both through the developed software or through any other software that can parse Atom-published feeds. Failing that, content and the components required for manual authentication can also be viewed in a text editor.

While this objective was technically failed, effectively this objective could be achieved if the feeds were a) stored on a Web server as opposed to cloud storage, and b) if the client communicated through Internet protocols as opposed to a local/cloud file system.

2.2.2: Support of freedom of speech

This objective can be said to be achieved if both of the following statementes are true:
a) the only way for an account to be compromised is for the attacker to posess both the file containing the account's encrypted private key and the passphrase for decrypting said private key
b) gaining posession of these artefacts requires considerable effort relative to that of gaining possession of artefacts required to compromise an account on other social media.

In this context, "compromised" will be defined as the attacker being able to amned or append to to the account's feed without detection by readers.

With other forms of social media, all data regarding an account is stored on the servers, meaning that the owners of the servers have the ability to disclose and/or modify any and all information and posts at the request of external parties. TRSST differs from this by storing the account's private key, which is required to authenticate all actions by the account, on the account owner's local machine as an AES-encrypted file. Thus, content hosted on the public servers cannot be modifed by external parties without alerting all readers of the content, as described below.

In TRSST, every entry in a feed with the ECDSA private key of the posting account - and then appending the signature to the entry itself. To verify, the ECDSA public key is given with the feed and is used to verify the signature, using the entire entry sans signature as the 'message'. This ensures that every entry can be checked for being tampered with by external parties, as any change to the entry would cause the attached signature to no longer verify correctly.

Entries are also protected from being deleted outright by using "blogchains" - a variation of "blockchain" technology used in cryptocurrencies such as Bitcoin. A blogchain is formed by each entry containing within it the signature of the preceding entry before being signed. If an entry is removed, the mismatched signature of the (new) predecessor in the succeeding entry would ensure that the deletion is noticable to all users. If the succeeding entry is tampered with to update the predecessor signature, then the verification of the succeeding entry will fail, due to the predecessor's signature being a part of the entry content being signed. This allows for both authors of entries to prove themselves as the author whilst still staying anonymous, and for any tampering of the feed by an external party to be detectable by everybody, achieving the objective of supporting authentication.

2.2.3: Support of privacy

This objective will be achieved if content that the author has encrypted may not be read by any other party than who the content was intended for. TRSST achieves this by not storing any unencrypted key on servers (save for public asymmetric keys). This ensures that if a server hosting TRSST content is compromised, any encrypted content will be unable to be decrypted by any attacker, as keys for decryption are stored elsewhere.

2.2.4: Support of compensation for the creation of content

This objective will be achieved if the author of any account can be compensated for their work, without having to give up their anonymity.

While the developed software does not contain any functionality for sending or recieving currency to or from accounts, every account ID corresponds to a network-neutral Bitcoin address that can be used to compensate the author securely and robustly, as only the author holds the private key of the

Bitocin address (as opposed to services such as MtGox, that stored private keys on their own servers, that were susequently lost). Users can then use the generated BItcoin addresses and/or private keys to send and/or receive Bitcoins through other clients. As described above, this does carry with it the risk of losing anonymity, so in the strictist terms, neither the original TRSST client nor the developed software achives this objective.

2.3: The Atom Publishing Protocol ("AtomPub")

2.3.1:Overview of AtomPub

Content and resources published through TRSST must follow the Atom Publishing Protocol, as defined in RFC-5023 (Gregorio & de Hora, 2007) along with the Atom Syndication Format as definded in RFC-4287 (Nottingham & Sayre, 2005), in order to make the content machine readable through XML, and to allow newsreading software to read feeds and entries publishing through TRSST. In addition, the current TRSST client adds its own modification to AtomPub, in order to support the functionality to achieve the aims discussed in the previous section.

Content in AtomPub is represented as Resources – data objects identifed by and accessible through a Uniform Resource Identifer (URI). A Resource may be an Atom Entry (as described below in Section 2.3.2.2) or another piece of binary data. However, TRSST stores all binary data (or URL references to binary data) within entires, so therefore for the purpose of this report, "Resource" may be considered to be synonymous with "Entry".

A collection of resources in known as a "Feed". Resources belonging to

AtomPub transmits and manipulates resources through four HTTP methods:

- **GET –** used to retrieve feeds from a host
- **POST –** used to create new entries to be added to the feed given as the URI for the HTTP request
- **PUT –** used for editing existing member resources of a feed
- **DELETE –** used for removing one or more member resources of a feed

As TRSST is explicitly designed to protect the integrity of content and ensure that all changes to existing entries are detectable, PUT and DELETE are not used. Instead of a PUT request, a new entry is created through a POST request, which references the entry to be edited and contains the new elements (see section 2.3.2.2 below) for said entry. Instead of a DELETE request, a new entry is created through a POST request, which references the entry to be deleted and contains message digests of both the entry to delete and the entry preceding it, to maintain blogchain integrity. When a server hosting the entry to delete recieves this request, it replaces its copy of the entry to delete with a new entry that retains only the original's ID, time of publishing, time of last update, digest of previous entry and signature. All other elements are not retained.

2.3.2: Structure of an Atom syndication document

2.3.2.1: Feed

A feed is the root element of an Atom syndication document. A typical feed element consists of metadata pertaining to the feed itself, plus any number of entry elements.

Structure of a feed element

Child element	Description	Required?	Example value
id	Unique, permanent URI	Yes	`<id>http://example.org/myFeed</id>`
title	Human-readable title of feed	Yes	`<title>My Feed</title>`

updated	Last time feed was modified	Yes	`<updated>2015-10-23T08:21:37Z</updated>`
author	Author(s) of the feed (one per element)	Yes*	`<author>` `<name>John Doe</name>` `<email>JohnDoe@example.com</email>` `<uri>http://example.com/~johndoe</uri>` `</author>`
link	Identifies related Web page (usually itself)	No	`<link rel="self" href="/feed" />`
category	Specifies category feed belongs to	No	`<category term="technology"/>`
contributor	One contributor (possibly of many) to the feed	No	`<contributor><name>John Doe</name></contributor>`
generator	Software used to generate the feed	No	`<generator uri="/myblog.php" version="1.0">` Example TRSST Client `</generator>`
icon	Small image providing visual identification for the feed	No	`<icon>/myicon.jpg</icon>`
logo	Larger information providing visual identification for the feed	No	`<logo>/mylogo.jpg</logo>`
rights	Rights (e.g. copyright) information petaining to the feed	No	`<rights> © 2005 Aston University </rights>`
subtitle	Human-readable description or subtitle for the feed.	No	`<subtitle>Aston's TRSST Feed</subtitle>`

2.3.2.2: Entry

Entries typically make up the majority of a feed's data. They represent one 'post' or 'article' to be represented in the feed.

Structure of an entry element

Child element	**Description**	**Required?**	**Example value**
id	Unique, permanent URI	Yes	`<id>http://example.org/myFeed</id>`
title	Human-readable title of feed	Yes	`<title>My Feed</title>`
updated	Last time feed was modified	Yes	`<updated>2015-10-23T08:21:37Z</updated>`
author	Author(s) of the feed	Yes*	`<author>`

	(one per element)		`<name>John Doe</name>`
			`<email>JohnDoe@example.com</email>`
			`<uri>http://example.com/~johndoe</uri>`
			`</author>`

content	Contains complete content of entry, or link to complete content	Yes	`<content>Lorem ipsum si dolor amet...</content>`
link	Identifies related Web page (usually itself)	No	`<link rel="self" href="/entry001" />`
summary	Short summary, abstract, etc. of the entry	No	`<summary>Lorem ipsum...</summary>`
category	Specifies category entry belongs to	No	`<category term="technology"/>`
contributor	One contributor (possibly of many) to the feed	No	`<contributor><name>John Doe</name></contributor>`
published	Time the entry was originally created	No	`<published>2003-12-13T09:17:51-08:00</published>`

source	Information pertaining to original feed (if entry has been copied from other feed)	No	`<source>`
			`  <id>http://example.org/</id>`
			`  <title>Fourty-Two</title>`
			`  <updated>2003-12-13T18:30:02Z</updated>`
			`  <rights>© 2005 Example, Inc.</rights>`
			`</source>`

rights	Rights (e.g. copyright) information petaining to the feed	No	`<rights> © 2005 Aston University </rights>`

*An author element may be omitted from the feed metadata if ALL entries in the feed contain an author element, and vice versa.

3. Background

3.1. Analysis of the current TRSST Client

The current TRSST client (hosted at https://github.com/TrsstProject/trsst) is a Java-based appliction that can be run from the command line or from a basic user interface using the web browser provided with the JavaFX architecture, designed to view the TRSST website.

3.1.1: Current Development

At the time of writing, the latest commit to the client's public GitHub repository was on the 22nd October 2014, which seems to give the impression that the project has been abandoned by its original creator, Michael Powers. In addition, the client itself appears to be incomplete, with several comment tags labelled as "TODO" and "FIXME".

However, as TRSST is technically not a social network but merely a protocol designed for them, and that said protocol has already been defined in TRSST's white paper and implemented, the lack of client support support does not seem to be a major hindrance in the development of the deliverable.

3.1.2: Extensions to AtomPub

As discussed in section 2.3.1, TRSST includes some extensions to AtomPub, in order to support several cryptographical functionality. However, in order to implement the functionality, the client does deviate from protocol in a few cases, for example, DELETE operations are not permitted, and PUT operations are treated as POST operations with the newly created entry containing the edited entry and a reference to the previous entry. Both of these deviations are to preserve the integrity of blogchains (and thus preserve the integrity of the communications sent by users).

3.2: TRSST protocol use cases and implementation

TRSST's white paper (TRSST Project, 2013) gives a total of 17 use cases for clients implementing the protocol. A short summary of how some of the cases are implemented by the client is given below:

3.2.1: User creates keystore and accounts

The original TRSST client creates a folder in the user's home directory on the user's machine, and stores both the account's signing and encryption keys (both generic ECDSA keys) encrypted with a passphrase in a single '.p12' file, and the account's feed in its own subfolder. The file and subfolder are both named with the ID of the account it relates to. Due to the key infomation being stored on the user's file system, it is the user's responsibility to preserve and secure the files and passphrase(s) themselves. The feed on the local system acts as a backup of the feed on the server, in case the TRSST network goes down or is compromised.

Critique

The downsides of this are that it is impossible to provide a "password reset" option in case the user forgets the passphrase, and that if the user wishes to use the account on a different device, they must copy the .p12 file containing the key and the feed information to the other device beforehand.

Users must also trust their hardware to be secure – if a device's operating system and/or drivers are compromised, then there is a good chance that private keys stored on that device are also unsafe.

3.2.2: User begins blogging

The orginal TRSST client posts entries by constructing an AtomPub <entry> element, in which the entry content, the signature of the predecessor (if applicable), an "activity verb" (using the namespace of, but not strictly conforming to the Activity Streams specification (Activity Streams Working Group, 2011), as verbs not in the given vocabulary (such as "encrypt") are used) and any "tags" or "mentions" are contained. For the entry itself, plus for each tag or mention, a Hashcash stamp is generated (see section), for the purposes of preventing "spamming". The client currently does not support syndication preferences or 'likes', and posting images and/or videos to the network seems to be broken.

Critique

Seeing as Activity Streams was not designed to handle encrypted social networks, I believe that a separate specification should have been made with the TRSST protocol in mind for easier access to and a concrete definition of which each actiivity verb is supposed to represent, as the white paper seemed to encourage development of independent clients using the protocol.

3.2.3: User creates a public entry

The original TRSST client signs the public entry after applying the Enveloped Signature (removes any previous XMLD-SIG elements within the entry) and Canonical XML (ensures that logically-equivalent XML elements are also byte-consistent) transforms to the entry, and creating a digest using the SHA1 hash of the resulting entry bytes. The digest is then signed using the ECDSA-SHA1 algorithm to produce a signature value that will then be used as the signature of the entry itself. Metadata about the signature is then stored along with the value in an XMLD-SIG `<Signature>` element and appended to the entry.

The inclusion of the predecessor in the signature is the core reason the blogchain is robust against editing without detection: at best, the attacker must choose between a valid blogchain but an invalid signature, or a valid signature but an invalid blogchain.

Critique

However, the current TRSST client does all its validation "behind the scenes" – no indication is given to the user interface or success or failure, which I believe to be an inconvenience to the authentication-conscious reader.

3.2.4: User creates a private entry

Entries are encrypted using a randomly-generated symmetric encryption key to encrypt an entry in the same format as a public entry. The symmetric encryption key is then encrypted using both the ECDSA public key of the intended recipient and the author, with each resulting cipher-key and the encrypted content itself placed in an XML-ENC (Imamura, Dillaway & Blair, 2002) element and appended to a 'wrapper' entry, which both has a predecessor element and is signed for authentication, and preservation of the blogchain. The wrapper entry is the appended to the account's feed in the same manner as a public entry.

Critique

There are flaws with this method: the encrypted symmetric key is placed in <EncryptedData> elements, whereas XML-ENC specifies a specific <EncryptedKey> element for encrypted symmetric keys, which led to confusion in the development process. In addition, encryption methods are not given, which can lead to confusion when decrypting entries outside of a TRSST client.

4. Preparation

4.1: Target platforms

As discussed in the previous section, the previous clients for TRSST have been either web- or command line-based. For the purpose of originality, the decision was made to make the software purely desktop-based with a graphical user interface, with all user interface elements having no direct interaction with the Internet, no usage of web browser technology within the software, and having the software work "out of the box", without any external software needed to be installed by the user.

As this project is relatively large in scale, the decision was made to target Windows 7 and above for the first iterations of the software, and then look at the feasibility of porting to other platforms, with Linux being the main focus for a port.

4.2: Choice of programming language

Due to the decision to make the client purely desktop-based, it would obviously require it to be programmed in a language that supports the development of native desktop applications. As use cases that may require or be better solved with a logical or functional programming language were not forseen, an object-oriented language was chosen to develop the software as a) I am most familiar with this paradigm, and b) there are several data structures present that lend themselves nicely to object-oriented languages. The two object-oriented languages I am most familiar with are C# and Java, and so I conducted an analysis of the two languages, to see which would be most suitable for developing the software.

As the two languages are syntactically similar, I decided to focus on two areas: portability, and external libraries relevant to the domain.

Programing Languages - Portability

Even though I believe porting the software to different devices and/or operating systems to be a functionailty that would be nice to have, I think it has a rather low importance compared to other functionalites. Due to the scope of the project, it seems very unlikely that the software will be ported to other operating systems, therefore this attribute has a very low weighting in my decision compared to domain libraries.

It is common knowledge amongst computer scientists that Java was designed to be a very portable language, however C# can also make use of Microsoft's .NET Framework for easier porting to Microsoft devices, and also the Mono framework (found at http://www.mono-project.com/) to achieve portability to multiple platforms, making C# not too far behind Java in terms of portability.

Programing Languages - Domain Libraries

TRSST makes use of the following cryptographical algorithms within its network:

- Elliptic Curve Digital Signature Algorithm (ECDSA) – used to generate a signature for authenticating entries, plus a variant (secp256k1, as defined in *SEC 2* (Certicom Research, 2010)) is used for account creation
- AES-256 – used to encrypt individual entries
- SHA-256 – used to generate an account ID/Bitcoin address from one's public key
- Elliptic Curve Diffie-Hellman (ECDH) – used to encrypt keys to share over TRSST

In addition, aside from AtomPub as discussed above, the following aspects of cryptography are also used:

- Base-58 encoding – used in the creation of account IDs and Bitcoin addresses
- RIPEMD-160 hashing algorithm – used to label binaries attached to an entry
- XML Signature standard (XML-SIG) – used with authenticating entries; one's ECDSA signature is embedded in an encrypted entry in this format.
- XML Encryption standard (XML-ENC) – followed when encrypting entries.
- Encryption and storage of keys on a local machine

Therefore, the programing language to be used must support the above encryption algorithms and aspects of cryptography. The below table shows how the two languages support these functions:

Algorithm / other aspect	**Support in C#**	**Support in Java**
ECDSA	Standard library	External library
AES-256	Standard library	Standard library
SHA-256	Standard library	Standard library
ECDH	Standard library	Standard library
Base-58 encode	External library	External library
RIPEMD-160	Standard library	Standard library
XML-SIG	Standard library	External library*
XML-ENC	Standard library	External library

*Library is provided by language developers (Microsoft for C#, Oracle for Java)

As you can see, C# has a greater degree of standard library support, relying only on CryptSharp for base-58 encoding with the standard library files being located in two namespaces (`System.Security.Cryptography` and `System.Security.Cryptography.Xml`), while Java relies on quite a few external libraries to achieve all of the functionality required. Should CryptSharp prove to have too much redundant and/or unnecessary functionality, there is always the option of implementing the algorithm manually.

In addition, it was easier finding documentation related to the above aspects in C# compared to Java, as guidance for implementing the aspect can easily be found on the Microsoft Developer Network website.

Programming Languages - Conclusion

Due to the greater standard library support and ease in locating documentation for advanced cryptography functionality, along with the various conveniences C# has over Java, the decision was made to write the software in C#, using Visual Studio 2012 as my main IDE, and Windows Presentation Foundation (WPF) to render the user interface. This decision also contributes to the originality of the project, as the current TRSST client was written in Java.

4.3: Software Architecture

The architecture of the software was designed using Simon Brown's "C4 model" (Brown, 2014), an iterative model designed to encourage designers to think about the most abstract aspects of the system and it's place in the environment (e.g. who it takes input from and sends output to) with each iteration of the design further increasing in concreteness in a series of four diagrams: a context diagram, a container diagram, one or more component diagrams, and zero or more class diagrams.

4.3.1: Context diagram

This diagram is designed to focus on what external systems the system as a whole interacts with, without delving into individual components. This diagram is supposed to help developers understand the place of the software in the surrounding environment it is located in.

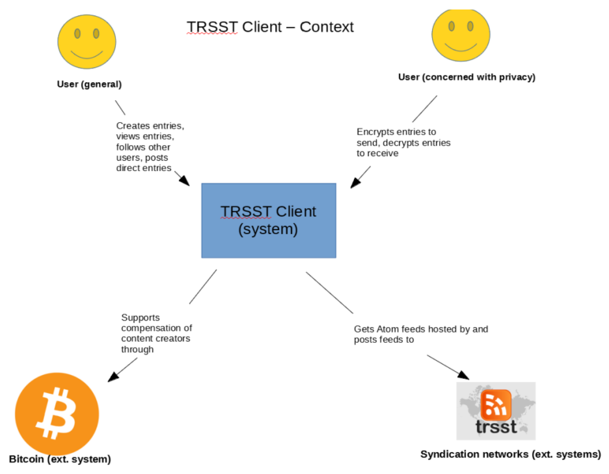
TRSST Client – Context
User (general)
User (concerned with privacy)
Creates entries, views entries, follows other users, posts direct entries
Encrypts entries to send, decrypts entries to receive
TRSST Client (system)
Supports compensation of content creators through
Gets Atom feeds hosted by and posts feeds to
Bitcoin (ext. system)
trsst
Syndication networks (ext. systems)

Fig. 4.3.1: Example of a C4 context diagram.

4.3.2: Container diagram

This diagram focuses on parts of the system, often divided by the technology they use, e.g. databases, file systems, user interfaces, etc. This diagram gives an overview of how the different technologies communicate with each other and the overall 'shape' of the software system.

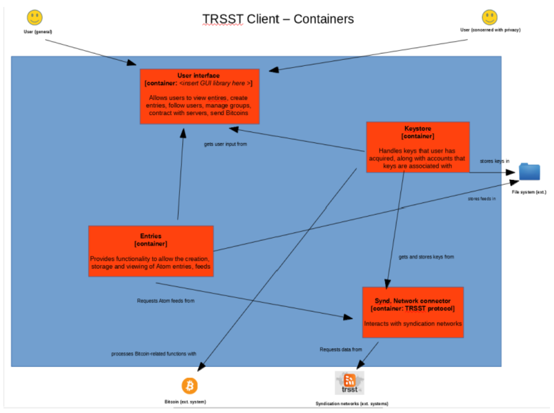
TRSST Client – Containers
User (general)
User (concerned with privacy)
User Interface
[container: <insert GUI library here >]
Allows users to view entires, create entries, follow users, manage groups, contract with servers, send Bitcoins
Keystore
[container]
Handles keys that user has acquired, along with accounts that keys are associated with
gets user input from
stores keys in
File system (ext.)
stores feeds in
Entries
[container]
Provides functionality to allow the creation, storage and viewing of Atom entries, feeds
gets and stores keys from
Requests Atom feeds from
Synd. Network connector
[container: TRSST protocol]
Interacts with syndication networks
processes Bitcoin-related functions with
Requests data from
Bitcoin (ext. system)
trsst
Syndication networks (ext. systems)

Fig. 4.3.2: Example of a C4 container diagram

4.3.3: Component diagram(s)

Each component diagram divides a single container into its major logical components and the interactions between them. Each component is labeled with what it is, its responsibilities and additional details about the technology being used and/or the implementation thereof.

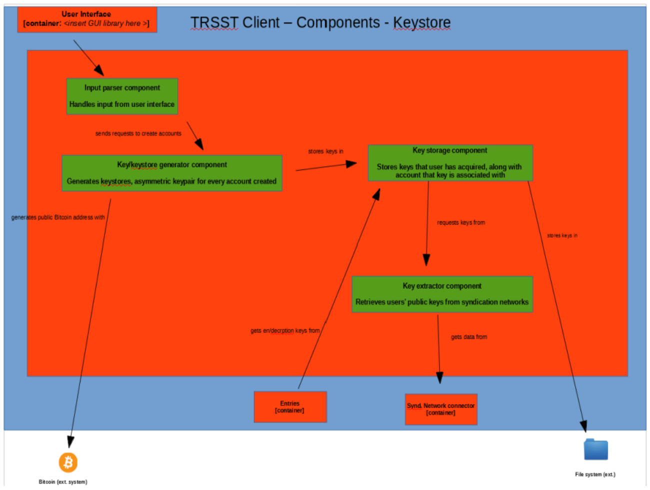
User Interface
[container: <insert GUI library here >]
TRSST Client – Components - Keystore
Input parser component
Handles input from user interface
sends requests to create accounts
Key/keystore generator component
Generates keystores, asymmetric keypair for every account created
stores keys in
Key storage component
Stores keys that user has acquired, along with account that key is associated with
generates public Bitcoin address with
requests keys from
stores keys in
Key extractor component
Retrieves users' public keys from syndication networks
gets en/decrption keys from
gets data from
Entries
[container]
Synd. Network connector
[container]
Bitcoin (ext. system)
File system (ext.)

Fig. 4.3.3: Example of a C4 compnent diagram.

4.3.4: (UML) Class diagram(s)

If more specific implmentation details are required for a single component, then one or more UML class diagrams are made to illustrate them.

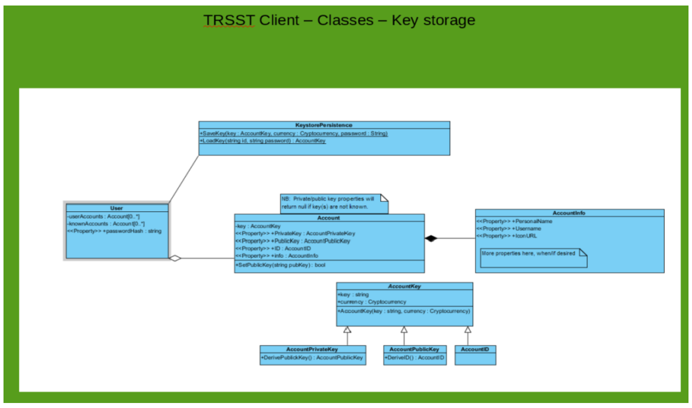
TRSST Client – Classes – Key storage
KeystorePersistence
+SaveKey(key : AccountKey, currency : Cryptocurrency, password : String)
+LoadKey(string id, string password) : AccountKey
User
-userAccounts : Account[0..*]
-knownAccounts : Account[0..*]
<<Property>> +passwordHash : string
NB: Private/public key properties will return null if key(s) are not known.
Account
-key : AccountKey
<<Property>> +PrivateKey : AccountPrivateKey
<<Property>> +PublicKey : AccountPublicKey
<<Property>> +ID : AccountID
<<Property>> +info : AccountInfo
+SetPublicKey(string pubKey) : bool
AccountInfo
<<Property>> +PersonalName
<<Property>> +Username
<<Property>> +IconURL
More properties here, when/if desired
AccountKey
+key : string
+currency : Cryptocurrency
+AccountKey(key : string, currency : Cryptocurrency)
AccountPrivateKey
+DerivePublickKey() : AccountPublicKey
AccountPublicKey
+DeriveID() : AccountID
AccountID

Fig. 4.3.4: Example of a C4 class diagram.

5. Deliverable

5.1: Keystore Generation and Storage

For the generation of cryptocurrency keypairs and addresses, the NBitcoin library was used. Even though the software in its present state only supports generating account keys and IDs through Bitcoin, the component for account generation was implemented to be able to handle any number of public-key cryptocurrencies through the Factory Method design pattern (Gamma et al, 1994), in which there is one class per cryptocurrency to handle keypair generation, and a "factory" class that uses these classes to generate and return keypairs as needed.

Storage of keys is done by first deriving the account's unique ID from the key, then encrypting the keys themselves using the AES algorithm using a given password as additional entropy, then storing the account ID, and the encrypted keys in a folder on the user's local machine. Keys are loaded into memory by specifying an account ID and the password used to save the key. The given ID is searched for in the specified keyfolder and the encrypted keys are retrieved. The given keys are then attempted to be decrypted and, if successful, said keys are returned and placed in an object design to represent an account's data.

5.2: Feed and entry generation

5.2.1: Feed generation

Feeds are generated from the results of a GUI dialog in which the user specifies the details of the account to be created. These details are then placed in an Atom `<feed>` element, along with the signing and encryption public keys generated along with the account.

The structure of the `<feed>` element's children and the information they represent is as follows (elements with user-defined content are denoted in *italics*):

Atom elements:

- 1 `<updated>` element: The year, month, day, hour, minute and second (in that order) the feed was last edited or appended to.
- 1 `<id>` element: The network-neutral Bitcoin address of the account, derived from the account's private signing key.
- *1* `<title>` *element*: A heading for the feed. In terms of social media, this would be analogous to a profile name. If no profile name has been given, the value of the `<id>` element doubles as the profile name instead.
- *1* `<subtitle>` *element (optional):* A summary of the feed. In terms of social media, this would be analogous to a profile description or 'bio'.
- *1* `<icon>` *element (optional):* A URI relative to the server's base storage folder, pointing to an image file. In terms of social media, this would be analogous to an avatar or profile picture.
- 1 `<author>` element: As a required element in an Atom feed, contains the following child elements:
    - *1* `<name>` *element*: Contains the same content as the `<title>` element.
    - *1* `<uri>` *element (optional)*: Supposed to contain a IRI reference associated with the user, however contains 'dummy' data, specifically what the value of the element would be if created by the original TRSST client.

- 1 `<link>` element: A URL pointing to where the feed may be found. Contains dummy data, specifically where the feed would be hosted, were it accepted by the TRSST servers.

Non-Atom elements:

- 1 `<sign>` element: The account's public Bitcoin key as a Base64 string, used for verification of entry signatures.
- 1 `<encrypt>` element: The account's 2048-bit RSA public key, used for sending encrypted messages to the account.
- 1 `<Signature>` element: The digest, value, and metadata of the feed's ECDSA signature. For further information, see Section 5.3.

5.2.2: Entry generation

Entries are generated from the results of a GUI dialog in which the user specifies the details of the entry to be created. These details are then placed in an Atom `<entry>` element, along with the entry's signature and that of its predecessor.

The structure of the `<entry>` element's children and the information they represent is as follows (elements with user-defined content are denoted in *italics*):

Atom elements:

- 1 `<updated>` element: The year, month, day, hour, minute and second (in that order) the entry was last edited.
- 1 `<published>` element: The year, month, day, hour, minute and second (in that order) the entry was first created.
- 1 `<id>` element: The time the entry was made, in UNIX time, as a string of hexadecimal characters.
- *1* `<title>` *element*: A heading for the entry. In case of small, 'tweet'-like entries, this could contain the entire semantic content of the entry.
- *1* `<summary>` *element (optional):* The main text content of the entry.
- 1 `<link>` element: A URL pointing to where the feed may be found. Contains dummy data, specifically where the entry would be hosted (relative to the base feed storage folder), were it accepted by the TRSST servers.

Non-Atom elements:

- 1 `<verb>` element: The type of 'action' the entry denotes (post, reply, delete, etc.)
- 1 `<stamp>` element: Hashcash proof-of-work stamp for the entry (see section 5.x).
- 1 `<predecessor>` element (if not the first entry posted): The signature value of the entry that was posted previously by the account.
- *0 or more* `<category>` *elements*: Accounts the author wishes to bring this entry to the attention of (mentions), or keywords relating to the entry (tags).
- 0 or more additional `<stamp>` elements: One per `<category>` element, for anti-spam purposes (see section 5.x)
- 1 `<Signature>` element: The digest, value, and metadata of the entry's ECDSA signature. For further information, see Section 5.3.

5.2.3: Encrypted entry generation

Encrypted entries contain the bytes of an unencrypted entry (see section 5.2.2) encrypted with a randomly generated AES-256 key, which itself is encrypted with the RSA public key of the recipient and stored in an XML-ENC `<EncryptedData>` element. This element is then placed in a 'wrapper' unecrypted entry and then posted to the account's feed. Unlike the orginal TRSST client, encrypted entries conform to the XML-ENC specification.

5.3: Signing feeds and entries

For the purposes of authentication, an XML Digital Signature (XMLDSIG) as defined by Bartel et al (2008) is created for each feed and entry within a feed. This signature is represented as a single `<Signature>` element within the feed/entry, which consists of a `<SignatureValue>` element which contains the value of the signature, and a `<SignedInfo>` element, which contains the information relation to the transforms, algorithms and digests used to determine the signature value. In TRSST, this includes:

- A `<CanonicalizationMethod>` element, which specifies the canonical form of the XML document, to ensure consistency between different instances of the same feed/entry when calculating the value of the signature. TRSST uses Canonical XML.
- A `<SignatureMethod>` element, which specifies the algorithm used to generate and validate the signature. TRSST uses the ECDSA-SHA1 algorithm for this purpose.
- A `<Reference>` element, which specifies the algorithm used for calculating a message digest, transforms to be applied before digesting, and the value of the digest itself. TRSST uses the SHA-1 algorithm to digest, and applies the transforms to specify the signature as an enveloped signature (i.e a signature contained within the document it signs), and then to convert the feed/entry into Exclusive Canonical XML, which specifies that the signature itself is to be excluded when calculating the digest and signature values.

The signature value is calculated by first converting the feed/entry into Canonical XML, then by converting the feed into a SHA-1 digest, and then signing the data using ECDSA with the private key of the authoring account. After the digital signature and digest have been calculated, the `<Signature>` element is created and populated with the appropriate values, and is appended as a child of the feed or entry.

5.4: Generation of Hashcash proof-of-work stamps

To prevent 'spamming' of messages (which could constitute a possible DoS attack against a TRSST network), mentions (which can flood a single user's notifications) or tags (which can cause a tag to 'trend' illegitimately), networks may use the Hashcash mechanism (Back, 2002) to force clients to compute a "proof-of-work" stamp to throttle the rate of posting entries. One "proof-of-work" stamp is required for each entry being posted, as well as a separate stamp for every mention or tag.

A stamp is generated by first generating a header consisting of various attributes delimited by the ':' character. In TRSST, this is, in order: the Hashcash version number (always "1"), the number of leading zeros required in the final output (from 1 to 160 zeroes, generally 20) ,the date the message was sent (in the format "YYMMDD"), the ID of the feed (if an entry is being made) or the text of the mention/tag (if a mention or tag is being made), and the unique ID of the entry (its timestamp as a hex string). If any of these fields contain one or more colon characters (':'), then they are converted to the full stop character ('.'), to prevent interference with the field separators of the stamp.

Then, hexadecimal numbers are generated, until the 160-bit SHA-1 hash of the concatenated header and number has the amount of leading zeroes as specified in the header. Given the number of leading zeroes as *z*, the client has a $1/2^z$ chance of generating a number that will give the SHA-1 hash required. Generally, clients start with 0 and increment by 1 for every unsuccessful attempt, so the proabability of finding a valid number is more accurately expressed as $1 / (2^z) - n$, where *z* is the number of leading zeroes required and *n* is the total number of unsuccessful attempts. An example proof-of-work stamp is given below:

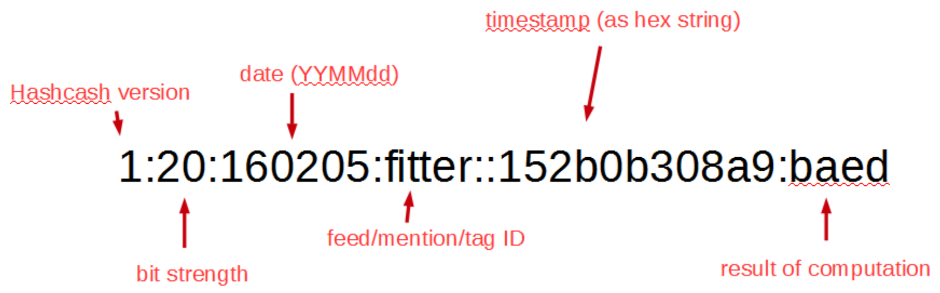

Fig 5.4: Structure of a Hashcash stamp.

To verify this stamp, a client will first check each field to see if it matches the corresponding data in the entry, then see if the 160-bit SHA-1 hash of the stamp has a number of leading zeroes greater than or equal to the number specified in the "bit strength" field of the stamp. Syndication servers may reject entries with proof-of-work stamps that are invalid or have too little bit strength. The developed client accepts stamps that are valid and have a bit strength of 20, and entries that have one of these stamps per entry, and additional stamps for each tag and mention.

Further avenues of development could lead to functionality in which the bit strength required for a stamp to be accepted starts low, and increases with each entry made in a given time period, allowing for genuine users to have a more responsive experience with better usage of their device's resources, while spammers will have their posts further throttled.

6. Evaluation

6.1: Secure storage of account keys

I wasted a good period of time trying unsuccessfully to implement a PKCS keystore in C# after misreading my notes – I read my notes as keys stored had to be in a PKCS keystore. However, on a second reading, it turned out that TRSST stated that the keystore need only be encrypted, the specific method is left to the developer's discretion. I think this was due to noting what the white paper said and what was done in the client in a way that the two seemed to be 'merged' – leading to confusion and wasted time on my part.

6.2: Lack of formal specifications for the TRSST protocol

The lack of a formal specification for the TRSST protocol has also hampered the development. While it has granted some leeway, for example in allowing me to use an RSA key instead of an ECDSA key for encryption as there is both more documentation for RSA, and the ECDSA was not designed with encryption in mind. However, this lack of specification is what caused the main TRSST servers to not accept feeds created by the developed client, but accept feeds from the original client, among other difficulties that caused the lack of specification to be more ill than good.

However, the TRSST white paper denotes many optional functions and/or non-exhaustive lists of rules for both clients and servers using the protocol (e.g. "*Servers may also reject entries for other* *reasons...*"), so this vagueness may be by design, in which case it may be helpful if there was a specification for servers to have a publically-accessible document that specifies what entries it will and will not accept, allowing for clients to tailor the same content to different servers.

6.3: Usage of the C4 software model

The usage of the C4 process (see section 3.x) did, on the whole, allow for greater clarity in the design at the context, container and component level, allowing for easier implementation of code.

While the "shape" of the context and container diagrams remained similar with only minor tweaks in the implementation phase, and that the context diagram helped solve a problem in which communication with the TRSST network was not working, I believe that drawing class diagrams proved to be too much detail for an early stage in the project, as I often missed a minor detail, or assumed that a single class could hold a feature, where it fact it would have higher cohesion to split the feature further, et cetera. As a result, the architecture of the software at the class level turned out to be completely different to what was planned in the class diagrams, and time was wasted in a failed attempt to implement the code as the class diagrams dictated.

6.4: Generation of feeds and entries

Despite the Atom feeds and entries generated by the developed software not following the 'letter' of the TRSST protocol and subsequently being rejected by the main TRSST network, the feeds and entries did conform the the 'spirit' of the protocol, in that they did support (to an extent) the 4 main aims of the protocol, as described in Section 2.1. Given this and that the client works when communicating with a "mock" server in the local filesystem, I would call development of this feature an overall success.

Further development may be to write software for a server than can parse and accept feeds generated by the client, or accept any TRSST-compliant feed in general, as well as giving information as to what feeds it will accept (see section 6.2 above).

6.5: Encryption of entries

The changes I made from the original client in terms of encryption have improved how encrypted entries are handled, particularly by third-party applications, given that the `<KeyInfo>` element and related child elements allow a client to grab and use a specific key directly, as opposed to encrypted entries generated by the original client, which must attempt decryption with all available keys which is both less efficient (particularly with large sizes of encrypted content) and difficult to parse with other software design to read XML-ENC style content.

7. Conclusion

In this project, software demonstrating the TRSST protocol for social networking, its main objectives and their implemenmtation was introduced. The TRSST protocol is designed to guarantee availability of content, protect anonymity of content creators if they so choose, provide proof of authorship, and support monetary 'tipping' of content creators. Improvements to the protocol, inspired from both background research and the process of implementing the client, have also been suggested and/or implemented into the software. These improvements are mainly concerned with ease of usage by users and developers, and properly conforming to external specifications (e.g. encrypted XML).

While the development of the software was plagued by difficulties both technical and personal, the software does provide a demonstation of a social network using the TRSST protocol, that demonstates most of the aspects of the protocol.

Future directions for the TRSST protocol include:

- the production of a formal specification for feeds and entries adhering to the protocol, to aid communication between clients and servers
- the development of demonstration software for running a TRSST relay server, that can communicate with clients over the Internet, and can parse entries that have been formatted according to the TRSST specification, as discussed above.

8. References

BARTEL, M., BOYER, J., FOX, B., LAMACHIA, B., SIMON, E. and editors (2008) *XML Signature Syntax and Processing (Second Edition)* [online] World Wide Web Consortium (W3C). Available at <https://www.w3.org/TR/xmldsig-core/#sec-SignatureValue> [Accessed: 7th June 2016]

BITCOIN PROJECT (2015) *Protect your privacy* [online] bitcoin.org. Available at <https://bitcoin.org/en/protect-your-privacy> [Accessed: 12th November 2015]

BROWN, S. (2014) *Software architecture and the C4 model* [online] Coding the Architecure. Available at <http://static.codingthearchitecture.com/c4.pdf> [Accessed: 24th October 2015]

CERTICOM RESEARCH (2010) *Standards for Efficient Cryptography 2 (SEC 2)* [online] Certicom Research. Available at <http://www.secg.org/sec2-v2.pdf> [Accessed: 28th September 2015]

GAMMA et al (1994) *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston:Addison-Wesley

GREGORIO, J. and DE HORA, B. (2007) *The Atom Publishing Protocol* [online] Internet Engineering Task Force (IETF). Available at <http://www.ietf.org/rfc/rfc5023.txt> [Accessed: 19th October 2015]

IMAMURA, T., DILLAWAY, B., SIMON,E. and editors (2002) *XML Encryption Syntax and Processing* [online] World Wide Web Consortium (W3C). Available at <https://www.w3.org/TR/2002/REC-xmlenc-core-20021210/Overview.html> [Accessed: 7th August 2016]

NOTTINGHAM, M and SAYRE, R. (2005) *The Atom Syndication Format* [online] Internet Engineering Task Force (IETF). Available at <http://www.ietf.org/rfc/rfc4287.txt> [Accessed: 19th October 2015]

TRSST PROJECT (2013) *Trsst: a secure and distributed blog platform for the open web* [online] TRSST Project. Available at <http://www.trsst.com/paper/> [Accessed: 10th September 2015]