FBase:

The next evolution of modularised code execution

M.J.G. Olsthoorn

Technische Universiteit Delft - Master thesis

TU Delft

FBase:

The next evolution of modularised code execution

by

M.J.G. Olsthoorn

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday December 1, 2019 at 10:00 AM.

Student number:	4294882	
Project duration:	March 1, 2018 – December 1, 2019	
Thesis committee:	Dr. ir. J.A. Pouwelse,	TU Delft, supervisor
	Dr. J.S. Rellermeyer,	TU Delft
	Dr. ir. A. Aaronson,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

Preface…

M.J.G. Olsthoorn
Delft, November 2019

Abstract

The abstract should contain a brief overview of the research and the most important results

Contents

1 Introduction

For decades software re-use has been seen as the holy grail of software development. Even in the eighties, papers were already written about this topic [7]. Throughout the years, more and more research has been done on the benefit of re-usable software [4]. Studies have also been done on how to reuse software in practice [6]. But up until recently, there was more discussion about software reuse than actual software reuse. Despite the fact that most software uses the same blocks of code over and over again, almost all software is built from the ground up [3]. Today, this situation is completely different. Nowadays, almost every application re-uses software in the form of software dependencies. However, this re-use pattern is starting to become unchecked. The shift to re-usable software has happened so quickly, the risks associated with choosing the right dependencies are often overlooked [2].

1.1. Code Evolution

Over the years, the way we use code has evolved with the changing need of the users and the society as a whole [5]. This evolution started off with specific applications written for each use case and each platform it had to run on. These application took a lot of time to develop and could not be re-used. To reduce this time, abstraction libraries were built to make it possible to run these applications on similar platforms. These abstraction layers, however, were still limited to broader types of platforms e.g. Linux, Unix, Windows, Mac. These platform libraries could now be maintained and distributed separately. This led to easier development and applications that could be used on more systems.

The Debian package system is a good example of the beginning of this evolution. It made it possible for code that was meant to be used as a library to be packaged separately for both system and user code. This allowed applications to indicate which library would be requirement for the application and the system would make sure it is available to the application. This possibility allowed these applications to be developed faster [8].

These new code libraries provided a lot of benefit and speed to application developers, but to improve the ecosystem further a new step had to be made. At this point when applications were distributed they were static. There was no option to adapt the application to include features that the user would like. Also, users that wanted to add their own functionality had to go through the developers to accomplish this. To solve this, larger application began to include plugin systems. A plugin system allows different parts of the code to be changed or to add functionality to the application. This paradigm allowed rapid development of extra features by both developers and the users of the application.

A very early example of a program with a plugin system is Winamp. The Winamp developers used a plugin system to provide users with a customisable package that could serve each user's preference. A large community formed around the application with different plugins for every imaginable feature. This was the start of the plugin community.

When the whole application movement started to go to the web, this same plugin paradigm started to exist. These plugins allowed external parties to add functionalities to some of the biggest websites. A good example of this is Facebook plugins. Even now when Facebook is in a decline, people still actively use and rely on plugins hosted on Facebook.

This code re-use continued when web application started to use the micro-services architecture. This allowed web application to move towards modules that had very small tasks that they were specifically designed for. This facilitated code re-use on a big scale with platforms like NPM and reusable web components.

The decentralised application community eventually also started to work on modular applications in the form of smart contracts. Ethereum is a good example of this movement.

1.2. Code re-use

The constant factor during this code evolution is code re-use. The ability to make development easier and faster by making use of existing solutions already created by a different party.

Reuse is software development's unattainable goal. The ability to put together systems from reusable elements has long been the ultimate dream. Almost all major software design patterns resolve around extensibility and re-use. Even the majority of architectural trends aim for this concept. Despite many attempts in almost every community, projects using this approach often fail [1].

This is often attributed to one big problem: usability. The more reusable we try to make a software component, the more difficult it becomes to work with said component. This is a critical balance that needs to be worked on. The largest part of this problem has to do with dependencies.

1.3. Re-usability vs usability

The challenge we face when creating a highly reusable component is to find this balance between re-usability and usability.

To make a component more reusable it needs to be broken down in smaller parts, that each handle only one task. Components with multiple tasks are harder to reuse since each application has different use cases and therefore has to modify and maintain there own version of that component. Smaller components that handle only one task can be used as building blocks for bigger components making them easier to reuse, saving developers the need for maintaining their own version. However, to create a complex application hundreds of small reusable components would have to be used creating a problem of itself. How are all these components going to be managed. Some aspects to think about are:

- Is the API (Application Programming Interface) going to stay constant?
- How do we deal with breaking changes?
- How do we prevent dependency conflicts?

Some of these aspects are already being addressed e.g. Semantic versioning, but most of these are still unsolved today.

For something to be reusable it also needs to have a default un-configured state. If the configuration of the original author would be included in the component itself it would make the component less reusable. However, if each small component has to be configured each time it is used, application would become less usable for the developers making them.

1.4. Modules vs Plug-ins

There are two different kinds of reusable components that often can be integrated into applications: modules and plug-ins.

- **Modules** are main functionality components created by core members of the developing team that are used to break-up the application into smaller subsystems that can more easily be worked on with different/larger teams.
- **Plug-ins** are community created components used to extend the main functionality of the application by users of the program. These functionalities are often too small or too unique to integrate into the application by the core team. Plug-ins normally don't have full access to all functions within the main application and are therefore limited in their behaviour. They are also tied to a specific application and can not be reused for other applications.

The function of both kinds of components are, however, not different. They both provide a (small) piece of extra functionality to the application. It would therefore also make sense to both make them first-class citizens of the application instead of making plug-ins a secondary operator.

This distinction is often made to differentiate between the code of the original authors and code submitted by third-parties. These plug-ins are most of the time also not reviewed by the original authors of the project.

1.5. Dependency

A dependency is additional code a programmer wants to call. Adding a dependency avoids repeating work: designing, testing, debugging, and maintaining a specific unit of code. In this thesis, that unit of code is referred to as a module; some systems use the terms library and package instead.

1.6. Research Goal

This thesis focuses its work on developing a framework that continues the progression in the development of re-usable code. It tries to find a balance between the software practices of Today and the impractical concepts of the future.

There have already been many attempts to solve the goal of practical code re-usability. However, these attempts still left some problems open, that this thesis tries to solve. These problems include:

- How to find a trade-off between re-usability and usability?
- How to minimize the risk associated with the use of dependencies?
- How to ensure dependency availability in an efficient and secure manner?

The rest of this document is outlined as follows: in Chapter 2 will go further into the problems that this thesis tries to solve. In Chapter 3 will discuss the solution proposed to solve the problems mentioned in Chapter 2. In Chapter 4 will discuss the proof-of-concept implementation. In Chapter 5 will evaluate the proposed framework against existing solutions.

2

Problem description

Currently the Tribler application, consists out of a monolithic 120k lines of code that has been developed over the last 13 years by various researches, developers, and students. Through its many development phases and limited time projects, the application has become unmaintainable and unmanageable. The application incorporates the main components of torrent client and many different sub projects that are used for research. This creates a difficult environment to work in as the code base is very complex resulting in a learning curve of many months to years for the core components. This complexity also causes code to be duplicated and rewritten multiple times across the lifespan of the project.

This work sets out to create a unified framework for reusable developer modules and user plug-ins built on top of IPv8.

2.1. Scope

The work is focusing on the specific use-case and problems of the application Tribler. It will not provide a universal solution to the problem of re-usability. This work will also not be tackling the problem of managing external dependencies like language dependencies and system dependencies.

This work will limit itself to the underlying platform used by Tribler, IPv8 and its language (Python).

2.2. Requirements

To realize this idea we have set out the following requirements with the client:

- **Crowdsourcing of code:** There should be no difference between modules and plug-ins. Everyone that wants to participate can create and add functionality to the application. Each user can also choose which functionality and therefore module they want to run on their instance of the application. This allows users to compose their own desired version of the application.
- **Source code inspection:** To make sure that users won't be running undesired malicious code. All modules will be inspected by making use of crowd-sourcing and trust-ability.
- **Trust function:** To determine how trust is created each user can download and select a trust function that corresponds with their view of what trust entails.
- **Live overlay:** All modules will be distributed across the network of users of the framework.
- **Dynamic loading:** When a modules is selected it should be downloaded and loaded into the application dynamically. Meaning the user should not have to reload the application for the new functionality to work.
- **Runtime-upgrades:** When new versions of modules will be published to fix bugs or add functionality, the module will automatically be distributed, downloaded, and loaded on the users system.
- **Developer communities around micro-services:** Each user can compose larger modules out of smaller ones or fork modules to represent their view on how it should be done. This should create a community around each module that could spark an ecosystem.

- **Self-governance:** The network should be owned by everybody and nobody. It should have no central servers (except for bootstrap) and be able to run on its own without supervision.

We will show the viability of the idea proposed in this work with a non-trivial use-case.

2.3. Related work

In the introduction, several related works were already mentioned. Ecosystems like Debian Package System, were one of the first big system that made use of reusable components on a large scale. It faced some of the same issues with dependencies but operates with central components and lacks source code inspection or user contribution. Another one of the mentioned systems was NPM. Node Package Manager is a highly reusable library manager for javascript modules. It, however, also makes use of central components and faces issues with dependency management.

Related work of plugins can be found in products like WinAmp. Which is a very famous old media player, which created the first community of contributing users around an application. These kinds of systems are also very popular in games, where they can be seen implemented all over. These systems, however, focus purely on the user contributing part and don't tackle the other requirements/issues mentioned.

3

Design

This chapter will expand on the design of the proposed framework. It will elaborate on the high-level structures within the application. The implementation considerations and details will be discussed in Chapter 4. The evaluation of the framework through an experiment will be done in Chapter 6.

3.1. Overview

An overview of the architecture of the framework can be found in Figure 3.1. It shows the three different layers that make up the framework. These layers are connected by a system-wide event bus that is used for connecting different parts of the application to each other.

These layers together create the components needed to run and distribute modularized code in a distributed fashion. The next few sections will expand on each of the mentioned layers and their components.

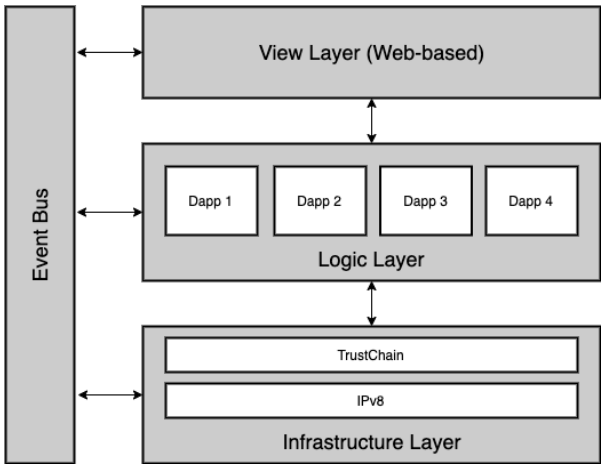

Figure 3.1

3.2. Event-driven Architecture

The framework is designed for running many different isolated applications. These applications consist out of separate modularized components that each run on different layers. Connecting these components together to form the application can quickly become a mess. To prevent this from happening, the framework makes use of an event-driven architecture. In such an architecture, actions are taken based on other actions happening in the system. Creating simple and maintainable logic. Input such as human interaction, network packets, creation of components, and/or system events, trigger corresponding actions in other parts of the system. An example of this would be the downloading of a module when a new one is discovered. This event system is located in the infrastructure layer and communicates with the other layers through the event bus. This event bus allows other parts of the system to hook on to specific events triggered by certain actions. To

allow components to hook onto such an event they have to register an event handler with the event bus for the types of events it wants to act on.

3.3. View Layer

The view layer contains the components that deal with human interaction. These components consist out of user interfaces created using web technologies. The decision for using web-based user interface was made because it is the current day standard for making cross-platform compatible GUIs and it allows for easy decoupling between itself and the logic behind it.

A view layer component consists out of a HTML, CSS, and javascript website. This website is run as a standalone component and connects to its logic counterpart through a REST API. This decouples the user interface part of the application and allows it to be interchanged. Multiple different GUIs could be offered for the same application.

When a new view component is added to the system, it needs to know how to connect to the logic component of the application. It does this by triggering an event on the event bus, specific for the type of application it belongs to, indicating it is requesting an endpoint address. The logic component is subscribed to this event. Its registered handler will return the REST API endpoint address back to the view component through the event bus.

To define a view component, a special file has to be created: view-component.json. This definition file stores the attributes and the settings of the view component. Attributes of the file include: name, version, app-tag (Application tag used for hooking on to the logic component). Each view component also needs to have a directory named public which contains the index.html file. An example structure can be found below.

- view-component.json
- public
  - index.html
  - Other HTML/CSS/javascript resources

3.4. Logic Layer

The logic layer contains the components that deal with the functionality of the application. Each component is defined by a file called: component.json located in the root of the component's directory. This file stores the properties and the settings of the logic component. The default properties that need to be defined are: name, version, app-tag, type. The component can be one of three types: Code, Overlay, or Service

3.4.1. Code Component

The code component consists out of a Python script without decentralized functionality that is executed on the host system. These types of components can be used as simple code scripts or as building blocks for bigger and more complex components. An example of this would be an updater script or an interface implementation.

In addition to the default properties a logic component defines, the code component also defines a function that needs to be executed when the module is run.

3.4.2. Overlay Component

The overlay component consists out of a decentralized overlay built on the IPv8 network. This component requires all files necessary to run an IPv8 overlay. This type of component runs in a shared environment and can have access to other overlay components in this same environment.

In addition to the default properties a logic component defines, the overlay component also defines the overlay class and overlay settings

3.4.3. Service Component

The service component consists out of a twisted service. This service type can be used to run processes in the background or isolate certain processes from other processes in the system.

In addition to the default properties a logic component defines, the service component also defines the service class.

3.4.4. Versioning
A system without versioning can quickly become infected and would make it difficult to track on which iteration the system is running. That is why each component has a component name and a version. The name property is a unique value for each component used to differentiate it from other components. The version property is value that is incremented every time a change has been made to the component. Together these properties form the identifier of the component.

3.5. Infrastructure Layer
infrastructure layer is responsible for providing network functionality and lower level functionality like storage. It accomplishes this through multiple different modules. Twisted is responsible for allowing pseudo multi threading through event-driven architecture. IPv8 is responsible for providing overlay functionality to run decentralized applications in and on the framework. TrustChain is responsible for providing a decentralized blockchain storage. LibTorrent is responsible for providing file transport services.

3.6. Identity Profiles
In peer-to-peer systems each peer in an overlay has to have an identity. This identity determines the trust and association within and across overlays. This identity can be shared between different overlays or each overlay can use its own identity. If two overlays use the same identity, one overlay can benefit from the built up trust and reputation of another overlay. However, actions performed by one overlay can also have a negative trust impact on the other overlay. To allow applications to choose between the having a shared identity, having its own identity, or having an pseudo-random identity, the framework provides a configuration option in the component.json to select what kind of identity profile is preferred..

3.7. System Strategies
Since the framework deals with untrusted executable user code, the framework provides several different strategies that the user can select from to protect their system against possible threats from running this code.

3.7.1. Download and Retention Strategy
The framework allows the user to configure and replace the download and retention strategy. This strategy is responsible for choosing which components get downloaded and how long they are kept on the system. For the distribution of components it is necessary to download packages that might not be used by the host system itself, but are solely for the intent of distributing. Some users might want to take a different approach to accomplish this. The framework addresses this by allowing parts of its code to be replaced by other components written by a third-part or by the user itself.

3.7.2. Isolated Execution
Since all distributed components have to be executed on the host system for them to function, it can pose a security risk by running untrusted user code. To minimize the risk that this poses, the framework allows components to be run inside of an isolated execution environment using Docker. When this method is used an execution environment is setup inside of the docker engine and the code will be mounted inside of this container. This container will then be able to run the code in isolation. This method, however, will prevent other applications running on the system from communication to it. It does allow the view layer to communicate with the isolated components since this makes use of network sockets.

4

Implementation

This chapter discusses the design principles and implementation details of the system described in the previous chapter. This work took a prototyping approach to get to a functioning prototype rapidly and improve from there. The sections below we explain the different functionalities that were tackled in chronological order.

4.1. Module Distribution

The first step that was taken to undertake this project was module distribution. Distribution was chosen as the idea hinges on the ability to setup an integrated content distribution network that would work efficiently and scale. Since this is not the first time this is done and there already exist excellent solutions out there that could accomplish this. Below I will list the different protocols considered.

4.1.1. Protocols

TFTP

Trivial File Transfer Protocol (TFTP) is a very simple and old file transfer protocol. It is mostly used in older enterprise equipment and is not really used anymore today. This has to due with the downsizes of the protocol in that it has no security built-in and has no verification that the content has arrived intact.

FTP(S)

File Transfer Protocol is a newer protocol than TFTP, but still older than the other alternatives. This protocol is mostly used for transferring content to web servers. For that purpose this protocol functions well because it is lightweight, provides content verification, and is simple. The downside for our use-case is that it isn't secure by default (gets routed through a HTTPS connection), doesn't support file transfer resumes, and doesn't scale well.

Web protocols

Web protocols like HyperText Transfer Protocol (HTTP) and its secure variant HTTPS are a very common transfer protocol in the current day internet. It is used by all major Linux distribution to distribute the system packages, by websites for downloading content and watching videos. This protocol supports file transfer resumes, encryption. It, However, doesn't scale well when the same content has to be uploaded to multiple users and doesn't natively provide content verification.

BitTorrent

BitTorrent is the protocol used by all bittorrent clients. It provides encryption, content verification, file transfer resumes, and scales very well when large amounts of the same contents has to be distributed thanks to its mesh architecture. That is why this protocol was selected as the basis of the module distribution of this work.

4.1.2. Module transfer protocol

Several small experiments were conducted to test the feasibility of the BitTorrent protocol with the regards to this work its use-case. These were related to choosing a suitable BitTorrent implementation, testing the

creation of a torrent and downloading this just created torrent on multiple other nodes. We made use of magnet links to transfer the information required to download the torrent. Ones these experiments were deemed successful, we had to find a way to distribute this magnet link through the network without using the traditional method of content indexing services. The method that we chose is described in the discovery section.

4.2. Discovery and Voting protocol

When a suitable transfer protocol is chosen, the next step was to make it possible for modules to be discoverable by all nodes in the system. Since we were already building our framework on top of the IPv8 peer-to-peer communication library. We decided it would be a good fit to use this to accomplish our goal, since it was very suited for bulk small size data gossiping. So this became out chosen method of module discovery.

Since IPv8 also provides a block-chain storage back-end it was an perfect opportunity to

4.3. Module Design

4.4. Event-Driven Architecture

4.5. GUI integration

4.6. Code review

5

Mobile App

To test the robustness and the flexibility of the framework, an experiment was performed to try to create a proof-of-concept prototype of an Android application that could run the same stack of code to extend the ecosystem to mobile platforms. Since the two major mobile platforms (Android, iOS) only run applications custom made for these platforms, different methods had to be explored. Because iOS has a very restricted development environment and strict security policies, this route was not further explored.

The Android platforms allows app developers to run Java, Kotlin (Java based), and C. The desired framework language (Python) does not natively run on this platform. Converting the project code and dependencies is not a simple or maintainable method. This approach, however, also would not work. To improve security, the Android platform makes use of app scanning to verify that the executables haven't been tampered with. This security method severly hinders the working of the framework, since more functionality is added by distribution of application through its peer-to-peer network. These new code inclusions would trigger warnings in the Android security system and would block the app.

To circumvent this, a un-official method was used to package all the necessary code, dependencies, and executables as a single file and execute this as a C service on the Android platform. To accomplish this, a project called Python-for-Android was used. Python-for-Android is a build script that compiles the desired Python system version and Python dependencies for the ARM platform and creates a directory structure that can be used to run on Android. In Figure 5.1 and overview of the Android app structure can be seen.

Since the Android app is needed to interact with the C service in the background, a part of the app had to be written in either Java or Kotlin. To keep this amount of code to a minimum, a decision was made to create all GUIs in web technologies, so the view layer can be shared between mobile and desktop platforms. This decision made it possibly to include a web browser as the only component written for the mobile platform. This web browser can then interact with the web server and REST API running on the C service.

To package the executable code in a way that would not trigger the Android security system, the code had to be bundled in a single file, disguised as a MP3. This format does not get checked by the Android security system and therefore can be used for the purpose of this work. Underneath the extenstion, the code is packaged as a GZIP Tar-archive. Upon running the Android application, this MP3 file is unpacked in the application space of the app and the C service is started with the right configuration to run the code.

In Figure 5.2 a screenshot can be seen of the framework running with a test dApp on the Android platform. Development was stopped after reaching the proof-of-concept stage as it is not the main goal of this work and the development cycle is very tedious and slow. Each time a change or addition is made to the Framework the entire app structure has to be rebuild. This process can take up to 20 minutes.

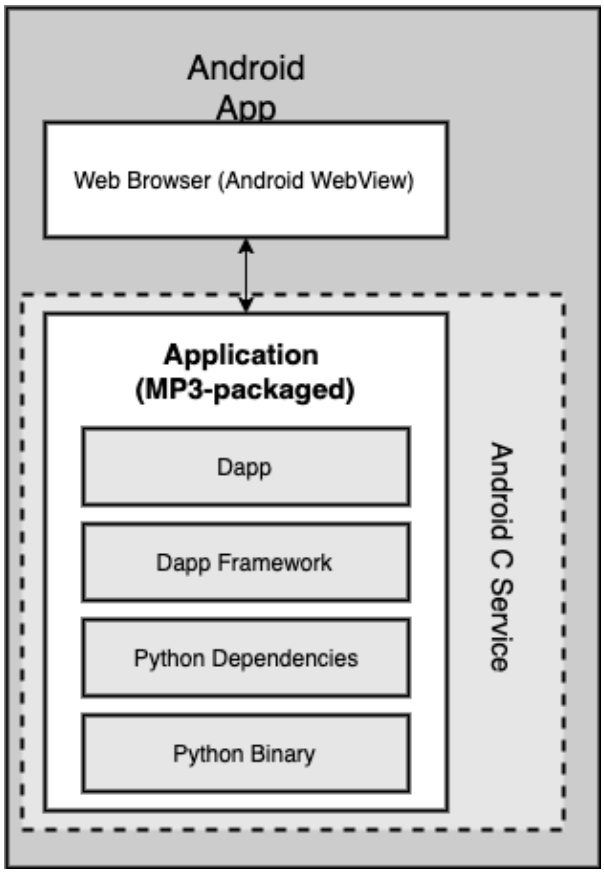
Android
App
Web Browser (Android WebView)
Application
(MP3-packaged)
Dapp
Dapp Framework
Python Dependencies
Python Binary
Android C Service

Figure 5.1


dApp
dApp
Done
test
4c69624e61434c504b3ae08e57f02b03e90726da888147a4359d7d
2
Tap here to fill entire screen

Figure 5.2

6

Experimentation and Evaluation

This chapter will propose an experiment and evaluate the framework described in Section 3. The evaluation will be performed based on the result gathered from the experiment.

6.1. Experiment

The experiment consists out of conducting a use-case study, by creating a fully functioning example that demonstrates the composition and construction of an application with interchangeable trust models. This application will consist out of 6 components:

- Test application GUI (view layer)
- Test application (logic layer)
- Trust algorithm 1 (logic layer)
- Trust algorithm 2 (logic layer)
- Execution engine (infrastructure layer)
- Transport engine (infrastructure layer)

Figure 6.1 shows an overview of the example application. The domain of trust was chosen since this is a very interesting use-case that has not been explored yet in other works. It allows users of a system to define their own notion of the concept of trust and apply this to their system without requiring extensive knowledge about each application they are using. For this experiment, this work makes use of two different trust algorithms: Netflow and PimRank. These two algorithms act as an exampl for this experiment.

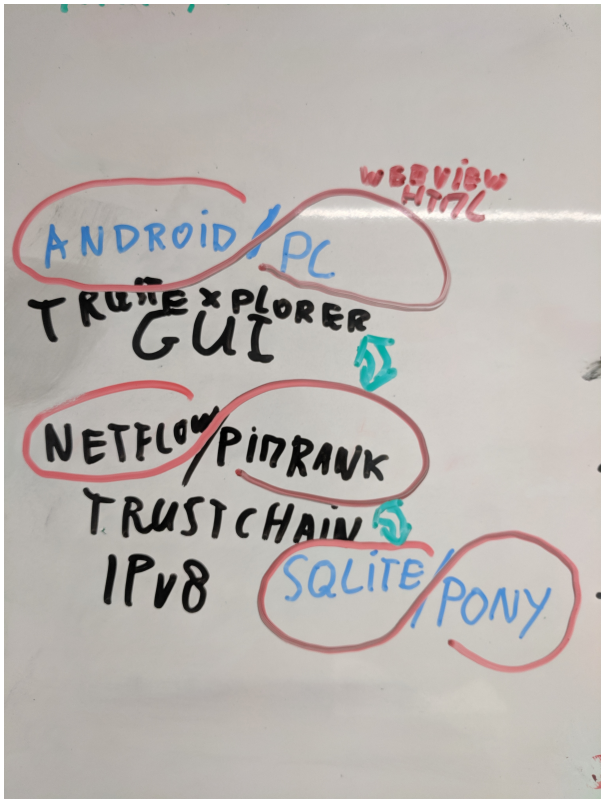
WEBVIEW HTML
ANDROID / PC
TRUE EXPLORER
GUI
NETFLOW / PAGERANK
TRUSTCHAIN
IPV8
SQLITE / PONY

Figure 6.1

7

Conclusion

References

[1] Reuse: Is the dream dead? URL `https://dzone.com/articles/reuse-dream-dead`.

[2] Russ Cox. Surviving software dependencies. *Communications of the ACM*, 62(9):36–43, 2019.

[3] William B Frakes and Kyo Kang. Software reuse research: Status and future. *IEEE transactions on Software Engineering*, 31(7):529–536, 2005.

[4] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software reuse: architecture process and organization for business success*, volume 285. acm Press New York, 1997.

[5] Václav Rajlich. Software evolution and maintenance. In *Proceedings of the on Future of Software Engineering*, pages 133–144. ACM, 2014.

[6] Donald J Reifer. *Practical software reuse*. John Wiley & Sons, Inc., 1997.

[7] Thomas A Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, (5):494–497, 1984.

[8] Stefano Zacchiroli. Debian: 18 years of free software, do-ocracy, and democracy. In *Proceedings of the 2011 Workshop on Open Source and Design of Communication; New York, NY, USA: ACM*, pages 87–87, 2011.

A

Module tutorial