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/`.

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 [29]. Throughout the years, more and more research has been done on the benefit of re-usable software [16]. Studies have also been done on how to re-use software in practice [27]. But up until recently, there was more discussion about software re-use than actual software re-use. Even though most software uses the same blocks of code over and over again, almost all software is built from the ground up [14]. 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 [8].

1.1. Code Evolution

Over the years, the way we use code has evolved with the changing need of the users and society as a whole [26]. This evolution started with specific applications written for each use case and each platform it had to run on. These monolithic applications took a lot of time to develop and could not be re-used. To reduce this time, system libraries were built to make it possible to run these applications on similar platforms. This abstraction layer, however, was still limited to broader types of platforms e.g. Linux, Unix, Windows. These shared system 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 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 required for the application and the system would make sure it is available to the application. This possibility allowed these applications to be developed faster. [32]

These new shared code libraries provided a lot of benefits 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, the larger applications began to include plug-in systems. A plug-in system allows specific functionality to be added to an existing computer program. This enabled customization of applications, making it possible to reduce the size of the core application or separating source code because of incompatible licenses. This paradigm allowed the rapid development of extra features by both developers and the users of the application.

A well known example of a program with a plugin system is Winamp. The Winamp developers used a plug-in system to provide users with a customizable package that could serve each user's preference [22]. A large community formed around the application with different plug-ins for every imaginable feature [7]. This community building gave an incentive for other applications to implement similar plug-in systems.

Eventually programming languages were created that allowed the development of cross-platform applications that could be run on all common platforms. This eliminated the need to create separate binaries for each individual platform. These applications were either written in an interpreted language, e.g. Python, or in a pre-compiled portable byte-code format for which interpreters exist on all platforms, e.g. Java. In the last decade, a major part of these cross-platform applications have moved towards the web. These new web applications make use of existing cross-compatibility of web technologies, that were designed when the web became universal, that run on all platforms.

With the development of cross-platform applications, there was also a rise in the availability of code frameworks. A code framework provides particular functionality as part of a larger software platform to facilitate development of software applications. Software with common use-cases could make use of the abstraction provided by these code frameworks to create application-specific software with only limited additional user-written code. These code frameworks can be seen everywhere nowadays. Some examples of code frameworks are: Spring, WordPress, and the Android platform. Spring is a popular code framework for developing applications in Java. WordPress is a web framework that runs more than 25% of the websites on the internet [12]. The Android platform is the underlying framework that allows apps to be created for the Android mobile operating system.

A more recent concept in the development of software applications is the microservice architecture. This architecture breaks an application up into a collection of loosely coupled services. These services are normally small in size and have one use-case that they were specifically designed for [30]. This architecture facilitates code re-use on a big scale with platforms like NPM (Node Package Manager) storing over 750.000 JavaScript packages [4].

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.

Re-use is software development's unattainable goal. The ability to put together systems from re-usable 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 [6]. 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.

1.2.1. Re-usability vs Usability

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

To make a component more re-usable it needs to be broken down in smaller parts, that each handles only one task. Components with multiple tasks are harder to re-use 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 re-use, saving developers the need for maintaining their own version. However, to create complex application hundreds of small re-usable components would have to be used creating a problem of itself. How are all these components going to be managed? The largest part of this problem has to do with dependencies [6], an additional piece of code a programmer wants to call. 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 partly being addressed through Semantic versioning [23], but most of these are still unsolved today.

The context dependability of a component also greatly affects the balance between re-usability and usability. When a component depends on the context it is running in, it makes it impossible to move it to an different environment that does not have this context. For components to be more re-usable, this context has to be moved from code to configuration. However, if each small component has to be configured each time it is used, the application would become less usable.

Both the granularity and the degree of dependability on the context can improve the re-usability at the cost of usability. The key is to find a balance.

1.3. Component Terminology

Two different kinds of reusable components that are often integrated into applications are modules and plug-ins. Since these terms can have a different meaning depending on the application, the interpretation that this work will use is defined below:

- **Modules** are main functionality components that are used to break up the application into smaller subsystems that can more easily be worked on with different/larger teams. Modules can either by re-usable components or tied to the specific application, and should be able to operate independently
- **Plug-ins** are components used to extend the main functionality of the application without having to make changes to it. They are often created by the community of the application. The functionality in these plug-ins are often too small or too unique to integrate into the core application. Plug-ins depend on the services provided by the application, they do not operate independently. They are also tied to a specific application and can not be re-used for other applications.

The function of both kinds of components is, however, not different. They both provide 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 created by third-parties. Plug-ins are most of the time also not reviewed by the original authors of the project.

1.4. Research Goal

Over the last couple decades, extensive research has been performed into the field of software re-use. In this period, several survey studies have been done to see what different approaches were used in research literature for creating re-usable software [17] [14]. The surveys tried to make generalizations about the methods used to research if there is a common pattern among them. The approaches mostly centered around re-using code for common use-cases like code frameworks.

Other studies have worked on designing metrics and models for measuring progress in software re-use to identify the most effective strategies [13]. Morisio et al looked at success and failure factors in software re-use to identify key factors in its adoption [20]. The main cause of failures that they discovered was a lack of commitment by companies and projects.

A more recent study, attempted to build a framework for highly modular and extendable software systems, called Normalized System theory. This theory is based on a theoretical concept called system theory. This theory, however, takes the abstraction of modules to a level that makes it inefficient beyond usage. It takes this approach to make the system more agile. However, without simplicity all agility is lost. [9]

Lehman's laws of software evolution is a law describing the evolution of software. The law describe a balance between forces driving new developments on one hand, and forces that slow down progress on the other hand [18] [19] [15]. One of the forces that slows down the progress of new developments is the ability of developers using the development to understand and easily use the functionality of the development.

In the current research there exits a gap in balance between re-usability and usability. This works tries to fill that gap. Many people have tried solving the problem of software re-use, but it has proven to be a hard problem. There needs to be a trade-off between re-usability and usability.

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?
- Can we use social trust and crowdsourcing to improve security of libraries
- How to ensure dependency availability efficiently and securely?

The rest of this document is outlined as follows: Chapter 2 will go further into the problems that this thesis tries to solve. Chapter 3 will discuss the solution proposed to solve the problems mentioned in Chapter 2. Chapter 4 will discuss the proof-of-concept implementation. Chapter 5 will evaluate the proposed framework against existing solutions. We use a concrete use-case driven methodology to advance the cause of software re-usability.

2

Problem description

This work sets out to create a framework for the next evolution of modularized code execution to find a balance between software re-usability and usability. The key property is permission-less code execution at near-zero cost.

There should be no difference between modules and plug-ins. 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. The 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.

This framework attempts to solve the problems that exist in the software practices of today. One of these software practices is the microservice architecture. This architecture like explained in the introduction breaks an application up into a collection of loosely coupled services. However, this pattern was specifically designed for maximizing re-usability without taking into account usability [21] . Studies have been done into the practical issues that the microservice architecture creates when used in a software application [11]. Examples of these issues are the manageability of packages on NPM, no explicit dependencies, interfaces that are not well defined. The architecture makes use of completely decoupled services that communicate through REST APIs. This interface, however, limits the type of communication that can be send between the nodes.

Smart contracts, in particular, Euthereum is another software practice used today, to solve the problem of code re-usability. However, since Euthereum is based on a proof-of-work principle, it requires payment so execute actions on the system. This will make applications build on top of Euthereum subject to these charges. In many cases these charges will have the consequence that the application would be to expensive to use in practice. The Ethereum model is not long-term sustainable

2.1. Decentralized

No central entity or central governance should be in control of the network. The network should be owned by everybody and nobody. The system should have no central servers except for bootstrapping.

2.2. Self-governing

The system should be able to run on its own without supervision. As there is no parent governing entity. The system should be able to handle all tasks needed for operating the network by itself.

2.3. Trustworthy Code

Since this work is proposing a framework that is highly dependent on re-usable code, it is important that the user running the application can trust all its parts. This trust aspects is very important for a code execution ecosystem. There are many examples of application being compromised by running untrusted code [5][2].

2.3.1. Open Ecosystem

For a user to trust the application it wants to run, it also needs to trust the framework running it. That is why it is important that every part of the code execution ecosystem is open for inspection. Making the source code public allows users or external parties to verify the behavior of the system.

Next to opening up the framework for inspection, it is also of vital importance that each re-usable component used within the framework is able to be inspected to increase the trustworthiness of the code.

2.3.2. Crowdsourcing

DevID is a previous work of the authors of this thesis. It evaluates the possibility of using social trust as a way to increase the trustworthiness of code by using crowdsourced peer-review [10].

Cargo Crev is a cryptographically verifiable code review system for the cargo (Rust) package manager [1]. It lets users cryptographically sign packages when they have deemed them to be safe.

Both of these systems make use of crowdsourcing to minimize the risk of users running undesired malicious code. This is a very important property, since manually inspecting all code running in an application can take a lot of time. By crowdsourcing this task to other individuals, the trustworthiness might be lower compared reviewing it yourself. However, because the code review is crowdsourced to many different individuals, the eventual trustworthiness of the code will be higher.

2.3.3. Dependencies

The current dependency trend is risky, developers trust more code with less justification for doing so. Since the recent explosion of code re-use systems, application have started shifting to using more and more existing libraries in the form of dependencies. This rapid shift, however, has caused developers to take along their perspective on code trustworthiness of classical dependencies like OS system libraries. The trust-ability of these new libraries is not as obvious as most developers believe.

H2020 FASTEN is a project that strives to minimize the risk associated with using dependencies [3]. It solution for this problem is performing static analysis of the code and creating a dependency graph. With this dependency graph, changes to the dependency can be detected. This allows inspection of the code that would be affected by this change.

When using dependencies without inspection, applications risk running code that contains bugs or has security exploits. Next to this, when the author of the dependency decides to change the purpose of their dependency or remove it entirely, the depending application becomes broken and useless. A study done by Xavier et al, has looked into the impact of breaking changes in dependencies [31]. They determined it poses a great risk to application, since the frequency of breaking changes and the impact are high in many cases.

Although Semantic Versioning is a system designed to indicate to developers when breaking changes have been made to the dependency, the system is not being used properly according to recent studies [24] [25]. Raemaekers et al, found that backward-incompatible changes are widespread in software libraries.

2.3.4. Trust Function

Trustworthy code is a cornerstone in software development. However, how is trust defined? Trust is a social notion. One person might need more or less information to trust a particular piece of code than another person. This is highly dependent on the social constructs of the user. Therefore, trust shouldn't be a fixed concept. Each user should be able to define a trust function that is used to determine if a dependency should be used or not.

2.4. Runtime Engine

To run an application, composed of modules, inside the code execution framework, a runtime engine is needed. This runtime engine is responsible for all tasks related to running of modules.

2.4.1. Integrated autonomous dissemination

Centralized systems store all code libraries in one or multiple central locations. These locations require a lot of infrastructure which is not free. Besides this, they also have a few downsides. One of these downsides is that these locations are susceptible to influence of governments. They can be blocked or shutdown when the government feels like the platform is not complying with its laws. Decentralized systems do not have this disadvantage. Another disadvantage of centralized library storage is that any library made by revoked at any time.

A system that has integrated autonomous dissemination of code libraries through decentralized methods, can not be controlled from outside the system. It also allows everything to be done from inside the framework making is easier to use.

2.4.2. Dynamic Loading

For the code execution framework to be easy to use, the user should not have to restart the application or load applications into the framework. This should be done automatically on-demand by the framework. Dynamic loading would also make sure that only the application that should be running are loaded into the system, so that unused applications will not waste computer resources.

2.4.3. Seamless Upgrading

Updating of dependencies is very important. Outdated dependencies that contain security exploits can seriously harm the system it is running on. So when a new version of a dependency is available on the network, it should automatically be distributed to all nodes that run applications depending on that dependency. There should be no user action required for the updating to happen. Once the dependency is available on the host computer, it should do an in-place replacement of the dependency in the framework.

A similar system has been proposed by Rellermeyer et al for Java OSGi modules [28]. In this paper, they devise a mechanism to extend the default functionality of OSGi modules to make it possible to upgrade them when used in a distributed method.

2.4.4. Module Interconnect

For applications to be built up out of modules, there has to be a way for modules to find each other and to communicate with each other. The way this connection is constructed is very important, since the code execution architecture defines the maximum complexity of the code that can be produced. JVM, Nodejs, CPAN perl modules and other real-world framework and the connection between modules determine the Maximum Complexity of Applications (MCA) which a single company, an global consortium, or open source community can create. We devised the first architecture to take the MCA as the cardinal design optimisation. Science what defines MCA? What constrains/boosts MCA? The interconnection fabric is the key determinant for the MCA. How data flows between modules, how the future-proofing is arranges, how any piece of code can interconnect with any other code, and how can we devise the universal module interconnector?

To optimize the MCA some properties of the interconnection fabric have to hold:

- Strong encapsulation: hide implementation details inside components, leading to low coupling between different parts. Teams can work in isolation on decoupled parts of the system.
- Well-defined interfaces: you can't hide everything (or else your system won't do anything meaningful), so well-defined and stable APIs between components are a must. A component can be replaced by any implementation that conforms to the interface specification. Rest is not ideal for this, native code is better Interfaces can't be fixed since application could be anything
- Explicit dependencies: having a modular system means distinct components must work together. You'd better have a good way of expressing (and verifying) their relationships.

Currently, there are some attempts at creating a universal module. One of these attempts is UMD. Universal Module Definition (UMD) strives to create a module that can run on all platforms, e.g. browser, nodeJS, that run JavaScript. UMD, however, doesn't have to deal with a lot of complexities since the interface on all platforms are almost identical and JavaScript has a common pattern for interconnection modules. Another platform that is trying to create a universal module is Java's OSGi. This platform allows modules conforming to the standard to be used interchangeably with other modules.

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 5.

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.

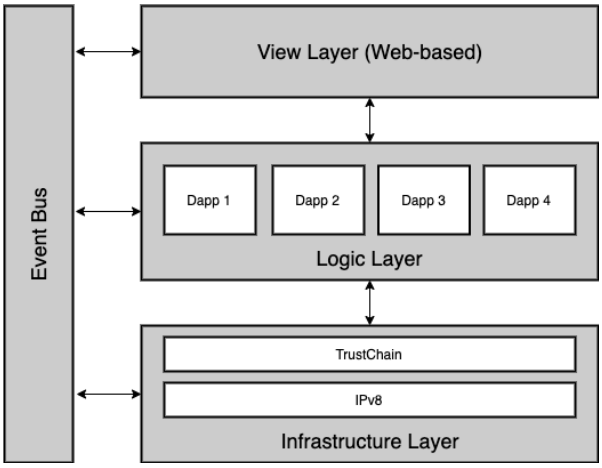

Figure 3.1

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
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.

5.1. 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 5.1 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.

5.2. 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

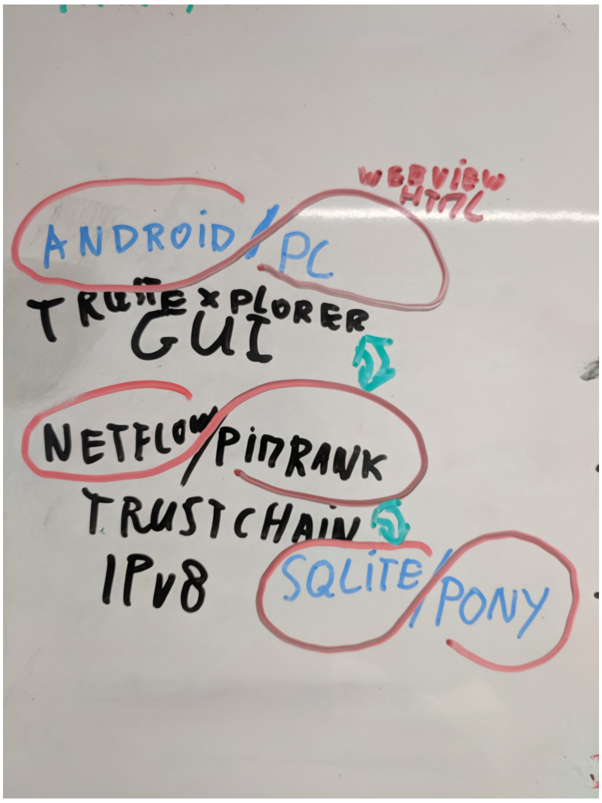

Figure 5.1

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 5.2 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 5.3 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.

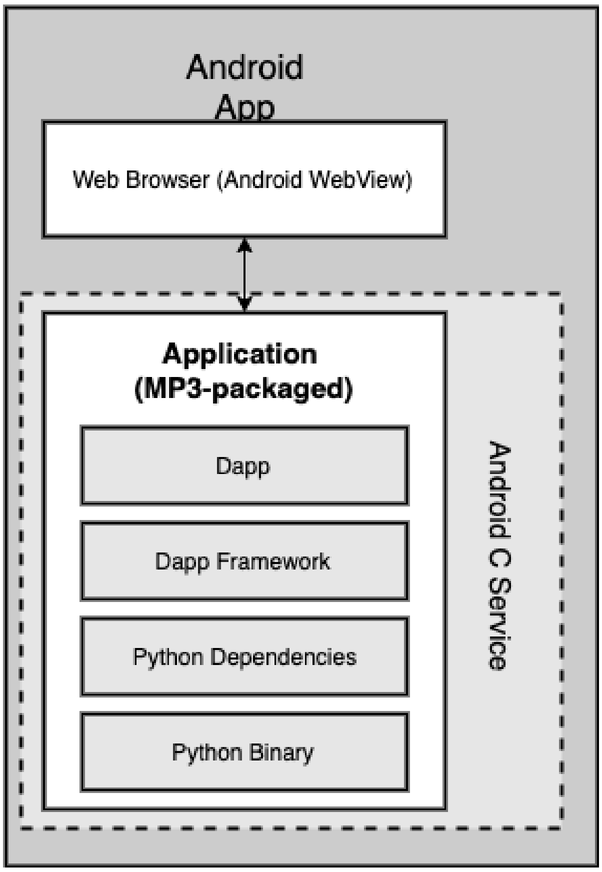
Android
App
Web Browser (Android WebView)
Application
(MP3-packaged)
Dapp
Dapp Framework
Python Dependencies
Python Binary
Android C Service

Figure 5.2


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

Figure 5.3

6
Conclusion

References

[1] Cargo crev. URL https://github.com/crev-dev/cargo-crev.

[2] Backdoor in docker image. URL https://arstechnica.com/information-technology/2018/06/backdoored-images-downloaded-5-million-times-finally-removed-from-docker-hub/.

[3] H2020 fasten project. URL https://www.fasten-project.eu/.

[4] This year in javascript: 2018 in review and npm's predictions for 2019. URL https://medium.com/npm-inc/this-year-in-javascript-2018-in-review-and-npms-predictions-for-2019-3a3d7e5298ef.

[5] Malicious code in purescript. URL https://harry.garrood.me/blog/malicious-code-in-purescript-npm-installer/.

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

[7] Winamp plugin community. URL https://web.archive.org/web/19981205123334/http://winamp.com/plugins/index.html.

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

[9] Peter De Bruyn, Herwig Mannaert, Jan Verelst, and Philip Huysmans. Enabling normalized systems in practice–exploring a modeling approach. *Business & Information Systems Engineering*, 60(1):55–67, 2018.

[10] Martijn de Vos, Mitchell Olsthoorn, and Johan Pouwelse. Devid: Blockchain-based portfolios for software developers. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 158–163. IEEE, 2019.

[11] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. In *Present and ulterior software engineering*, pages 195–216. Springer, 2017.

[12] Tom Ewer. 14 surprising statistics about wordpress usage. *ManageWP. Np*, 7, 2014.

[13] William Frakes and Carol Terry. Software reuse: metrics and models. *ACM Computing Surveys (CSUR)*, 28(2):415–435, 1996.

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

[15] Israel Herraiz, Daniel Rodriguez, Gregorio Robles, and Jesus M Gonzalez-Barahona. The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Computing Surveys (CSUR)*, 46(2):28, 2013.

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

[17] Charles W Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[18] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[19] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. Metrics and laws of software evolution-the nineties view. In *Proceedings Fourth International Software Metrics Symposium*, pages 20–32. IEEE, 1997.

[20] Maurizio Morisio, Michel Ezran, and Colin Tully. Success and failure factors in software reuse. *IEEE Transactions on software engineering*, 28(4):340–357, 2002.

[21] Sam Newman. *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015.

[22] INC NULLSOFT. The winamp mp3 player, 1999.

[23] Tom Preston-Werner. Semantic versioning 2.0. 0. *URL: https://semver. org*, 2013.

[24] Steven Raemaekers, Arie Van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 215–224. IEEE, 2014.

[25] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning and impact of breaking changes in the maven repository. *Journal of Systems and Software*, 129:140–158, 2017.

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

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

[28] Jan S Rellermeyer, Michael Duller, and Gustavo Alonso. Consistently applying updates to compositions of distributed osgi modules. In *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, page 9. ACM, 2008.

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

[30] Johannes Thönes. Microservices. *IEEE software*, 32(1):116–116, 2015.

[31] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. Historical and impact analysis of api breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147. IEEE, 2017.

[32] 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