FBase:

The next evolution of modularised code execution

M.J.G. Olsthoorn

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.

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

TUDelft

Preface

I started this thesis work with the Distributed Systems department after having previously done my bachelor thesis there. My bachelor thesis was on the concept of decentralized general asset markets that allowed clearance in the order of seconds. I became interested in how decentralized systems can distribute the control and burden of a system to the users of that system.

In my master's, I have specialized in the security of networks and systems. I wanted to use this new knowledge together with my previous experience from my bachelor to do research into topics that solve practical problems.

The origin of this current master thesis work stems from a project within the distributed systems research group called Tribler. The Tribler team focuses on creating disruptive technology that takes the control from central authorities and brings it back to the users. They have created a privacy-focused torrent client using a Tor inspired onion protocol that operates completely without any central control. This system relies on many different components that became tightly integrated and hard to maintain. This led to a more general problem within software engineering which sparked the idea of this thesis.

I want to thank my supervisor, Johan Pouwelse, for his guidance during my thesis. He provided me with helpful feedback during this time to help me improve my work. I would also like to thank the members of the Tribler team for their help and support during the development of the work and understanding the tools created by the lab. In particular, I would like to thank Martijn de Vos, with whom I got the chance to work on a paper during my thesis to test the feasibility of a concept used in this work.

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 the idea of re-usable software has been seen as the holy grail of software development. Even in the eighties, papers were already written about this topic [37]. Throughout the years, more and more research has been done on the benefit of re-usable software [19]. Studies have also been performed on how to re-use software in practice [35]. 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 [17]. 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 [10].

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 [34]. This evolution started with specific applications written for each use case and each platform it had to run on. These so called monolithic applications took extensive amounts of time to develop and could not be re-used.

(a) Shared Library.

(b) Plugin Architecture.

(c) Cross-platform Application.
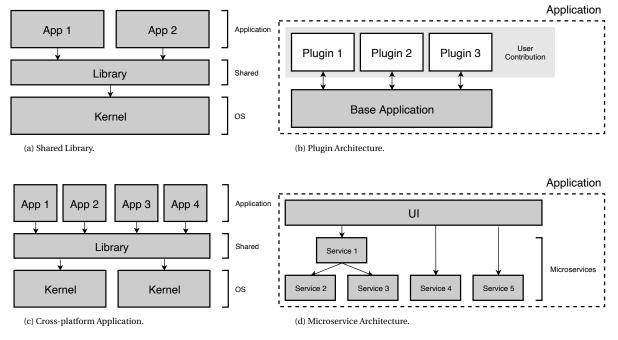
(d) Microservice Architecture.

Figure 1.1: Architectures in the evolution of software.

To reduce this time, system libraries were built to make it possible to run these applications on similar platforms. A visualization of this architecture can be seen in Figure 1.1a. 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.

A good example of this evolution is the Debian package system. 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 that application and the system would make sure it is available to the application. This possibility allowed these applications to be developed faster. [41]

These new shared code libraries provided numerous 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 to see. 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 visualization of this architecture can be seen in Figure 1.1b. 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 [28]. 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. A visualization of this architecture can be seen in Figure 1.1c. In the last decade, a major part of these cross-platform applications has 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 the 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 [15]. 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 [38]. The microservice architecture facilitates code re-use on a big scale with platforms like NPM (Node Package Manager) storing over 750.000 JavaScript packages [4]. A visualization of this architecture can be seen in Figure 1.1d.

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 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 their 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 its own. 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 consider 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 [31], 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 (Figure 1.2).
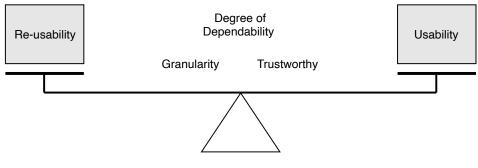

Figure 1.2: Balancing Re-usability and Usability

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 [20] [17]. 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 [16]. Morisio et al looked at success and failure factors in software re-use to identify key factors in its adoption [26]. 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. [11]

Lehman's laws of software evolution is a law describing the evolution of software. The law describes a balance between forces driving new developments on one hand, and forces that slow down progress on the other hand [21] [22] [18]. 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.

Studies have been done into the practical issues that the microservice architecture creates when used in a software application [14]. 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. Newman et al concludes in a different study that the microservice architecture was specifically designed for maximizing re-usability without taking into account usability [27].

Smart contracts, in particular Ethereum, is another software practice used today to solve the problem of code re-usability. However, since Ethereum 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 Ethereum 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

1.4.1. Research Aim

In the current research there exits a gap in balance between re-usability and usability. This works tries to fill that gap. 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?

1.4.2. Research Scope

In the field of software engineering the concept of reusable software is a big topic. Tackling and solving a substantial problem like the trade-off between re-usability and usability is not feasible within a Master thesis project. Therefore this work limits the scope of the project to a subset of this problem. This subset consists out of the re-usability of applications that can be broken down into generalized modules defined by this work (which can be found in Chapter 3). This thesis will not provide a solution for every possible application type or applications with extreme complexity.

Within this subset, this work will propose one concept in the form of a framework to tackle the remaining research aims of trustworthiness and dependability on external components. The theoretical and practical properties of the framework are examined on the effectiveness of the discovery protocol and the module communication. Furthermore, one non-trivial use-case is used to determine that the concept works and is viable.

1.4.3. Research Structure

Figure 1.3 shows how the research is structured. The background and related work (Chapter 1) form the requirements which are specified in Chapter 2. These requirements are used as the underlying principles of the proposed concept, called FBase, explained in Chapter 3. A proof-of-concept implementation will be created to support the evaluation of the concept. The details of the implementation will be discussed in Chapter 4. Chapter 5 will analyze the concept with a non-trivial use-case. Based on the evaluation a conclusion will be given in Chapter 6.
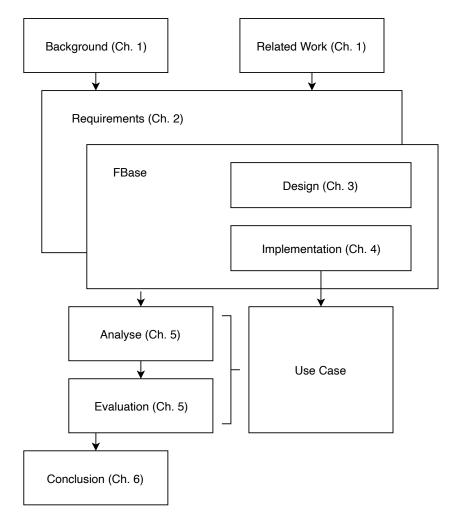
Background (Ch. 1)

Related Work (Ch. 1)

Requirements (Ch. 2)

FBase

Design (Ch. 3)

Implementation (Ch. 4)

Analyse (Ch. 5)

Use Case

Evaluation (Ch. 5)

Conclusion (Ch. 6)

Figure 1.3: Research structure.

2

Requirements

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. This chapter specifies the requirements that were identified from the background and related work in Chapter 1. The key property is permission-less code execution at near-zero cost.

The requirements are subdivided into three different categories namely Principle, Trustworthy Code, and Runtime Support.

2.1. Principles

These principles form the foundation of the framework and are necessary to achieve the research aim.

Decentralized

For the framework to be financially sustainable it needs to be decentralized. In centralized systems, the operating costs of a network are the responsibility of the entity in control of the system. This entity would need to have continued incentive to keep operating the network. In decentralized systems, the operating cost is split across all users of the network. The number of users will scale together with the cost of the network.

Decentralization also increases the reliability of the system as there can be no central infrastructural point of failure that could bring the network down. The system should have no central servers except for bootstrapping.

Self-governing

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

To be free of external influence, no central entity or central governance can be in control of the network. The network should be owned by everybody and nobody and therefore self-governing.

Since self-governing systems are run by the system itself and its users' input, there is no single owner of the network. If the original author disappears or is mandated to hand over control to another party, the overall system is not affected and can continue to function regardless.

An example of a platform that was taken down by external pressure from lawsuits is Napster, a peer-to-peer music sharing service[25]. Although Napster used decentralization for file sharing it was still controlled by a central company. This allowed lawsuits to be filed against this company to take down the service.

2.2. 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 aspect is very important for a code execution ecosystem. There are countless examples of applications being compromised by running untrusted code [5][2].

Open Ecosystem

For a user to trust the application they want to run, they also need to trust the framework running it. That is why every part of the code execution ecosystem must be 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 can be inspected to increase the trustworthiness of the code.

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 [12].

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 to reviewing it yourself. However, because the code review is crowdsourced to many different individuals, the eventual trustworthiness of the code will be higher.

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, applications 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]. Its solution to 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 [40]. They determined it poses a great risk to an 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 [32] [33]. Raemaekers et al found that backward-incompatible changes are widespread in software libraries.

Trust Function

Trustworthy code is a cornerstone of 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.

2.3. Runtime Support

A key bottleneck for re-usability and usability is the lack of runtime support within the execution environment.

Integrated Autonomous Dissemination

Centralized systems store all code libraries in one or multiple central locations. These locations require a lot of technical infrastructure which is not free. Besides this, they also have several downsides. One downside is that these locations are susceptible to the influence of governments. They can be blocked or shut down 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 it easier to use.

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.

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 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 [36]. 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.

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. The type of module and the connection between them determine the Maximum Complexity of Applications (MCA) which a single company, a global consortium, or open-source community can create. We devised the first architecture to take the MCA as the cardinal design optimisation.

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

- Strong encapsulation: hide implementation details inside components, leading to a 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 is 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 is 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.

Module Definition

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.

3

FBase Design

This chapter will discuss the design of the proposed framework, called FBase. FBase is a framework designed to create a usable ecosystem for the development, distribution, and execution of applications. This chapter will elaborate on the high-level structures within the framework. The implementation considerations and details will be discussed in Chapter 4. The evaluation of the framework is performed through an experiment (Chapter 5).

3.1. Overview

An overview of the architecture of FBase can be found in Figure 3.1. It shows the three different layers that make up the framework.
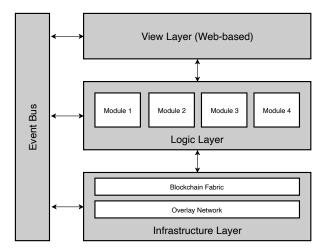

Figure 3.1: The architecture of the FBase framework

- **View Layer:** The view layer is responsible for the interaction between the user and the logic layer. This layer contains the view components for both the FBase framework and the user applications.

- **Logic Layer:** The logic layer is responsible for the execution of the user applications. It provides runtime support through the FBase runtime engine.

- **Infrastructure Layer:** The infrastructure layer is responsible for providing services to the FBase framework. These services include database storage, encryption primitives, and network capability. This layer contains the module distribution and overlay network.

These layers are connected by a system-wide **Event Bus** that is used for connecting different parts of the user application. Connecting these components to form the user application can quickly become unmanageable. To prevent this from happening, the FBase framework is built according to the event-driven

architecture style. In this style of framework, actions are taken according to events happening in the system. Every component in the framework can trigger events. Creating simple and maintainable logic. Input such as human interaction, network packets, creation of components, and system notifications, trigger these events and cause 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. The event bus is the main method for communications between different layers.

These layers together create the components needed to run and distribute modularized code in a decentralized fashion. The next sections will expand on the components that make up these layers.

3.2. Generic Modules

The FBase framework aims to support as many different use cases of modules as possible. However, to create a single module type that can support any type of interaction and behavior would be infeasible. That would require an interface between modules so generic and complex that it would not satisfy our requirement of a usable system. If such an interface would even be possible to create, it would have a substantial performance penalty because of the abstractions and complexities needed to support such as system.

The method that FBase uses to find a balance between this flexibility and feasibility is a system of four generic modules:

- **View Module:** The view module type contains the components that deal with human interaction. This module type is not required for a user application to be functional. The module represents a Graphical User Interface (GUI).

  To meet our goal of having a re-usable ecosystem, these GUIs have to be cross-platform compatible. This cross-platform compatibility enables the view module to be used on any major platform in use today e.g. Windows, macOS, Linux, Android, iOS. To accomplish this, view modules will be created using web technologies. Web technologies were chosen for the FBase user interfaces, as they are one of the only GUI technologies that allow for uniformly looking cross-platform user interfaces. They are becoming the current standard for these kinds of GUIs.

  Web technologies also allow for easy decoupling between the view layer and the logic behind it. A view module 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 an Application Programming Interface (API). This decouples the user interface part of the user application and allows it to be interchanged. Multiple different GUIs could be offered for the same user application. It also allows GUIs to be created with different purposes that could be run simultaneously. An example of this would be a music visualization plugin.

- **Application Module:** The application module type is the main module type in the FBase framework. This module type contains the main logic and it is the entry point into the user application. User applications can be either an isolated application on each user's system or a decentralized application that spans across the FBase network overlay.

  The logic in the application module type acts as the controller in the Model-View-Controller (MVC) architectural software pattern. Controllers act as an interface between the View module and the Service Module to process all the incoming requests, manipulate data, and interact with the GUI to render the final output.

- **Service Module:** The service module type provides services to the application module that have common use-cases. These re-usable services should contain the bulk of the code in a user application. Most applications share a significant portion of their functionality with other applications. The service module encapsulates those functionalities to make them more re-usable. A good example of such a service is authentication. Almost every user application needs a form of authentication. Writing a secure and well-structured authentication service is not trivial. When applications can re-use well build services it adds to the usability of the application and ecosystem.

  Service modules should be standalone services. They should provide functionality that is not dependent on other parts of the user application. Service modules also maintain their state. This makes sure the complexity of the functionality is encapsulated by the service.

Application modules can also be used as service modules. This can be achieved by marking the module with both types. An example of such a module is a zip code lookup system. Such a system can be used as either a standalone application with a GUI or as a service to another application.

Service modules can also be used to emulate plugin behavior where certain functionality is exchanged by a different one. This is done by changing the service module used in the user application. This ability would be similar to the Strategy software pattern. A use case of this would be changing the algorithm used to calculate trust in a network on runtime. It also allows service to be replaced when security vulnerabilities have been found that are not being fixed.

- **Package Module:** The package module type is added to make it possible to remove dependencies on existing code repository infrastructure. Many programming languages have a native package manager to house common use code libraries to make it possible to re-use code already used by others. However, these package managers almost all use centralized infrastructure and have many problems of their own e.g. revocability. FBase uses the package module to provide an alternative for these code repositories or act as a back-up.

  The Package module contains downloadable code libraries that are made available to the other modules in the system. Examples of use-cases for the package module are validation libraries and database abstraction libraries.

3.3. Module Discovery and Distribution

FBase makes use of a decentralized discovery and distribution network for its modules. Current networks for software distribution are often centralized and controlled by a company entity. By making use of a decentralized network, the possibility of the network being brought down by the company behind it, the government, or the law, can be eliminated. It also allows users to make the FBase network completely self-governing, so the system would be owned by everybody.

3.3.1. Identifier and Versioning

To distinguish different modules from each other, FBase makes use of an identifier. Each module has such an identifier, which is unique in the network. To guarantee this uniqueness a cryptographic asymmetrical key pair is used. An asymmetrical key pair consists out of two keys. One key is the public key to identify the object it represents, in this module. The other key is the private key which can be used to prove the ownership of the represented object. This private key acts as the credential for the author of the module to manage it.

The module identifier consists out of two parts. One part is the previously described public key. The other part is the versioning code. This versioning code is a hash of the entire module code. A hash is a short representation of a piece of data that is calculated in a one-way function. This method is chosen over more conventional versioning code like Semantic Versioning as it makes it significantly harder for malicious actors to spoof the module code corresponding to a version number. Since the hash represents the code.

The complete module identifier format is:

```
<public_key>.<module_hash>
```

New versions of the module have a different module hash and would be a separate entity in the network. Each module also has a timestamp of when the module was created. Together with the module identifier, this allows nodes to pick the most recent version of the module.

When a module wants to introduce breaking changes to the API, a different approach has to be used. Using the current approach for updates with breaking changes could cause dependent modules to break and stop functioning. In FBase when an update includes breaking changes a completely new module has to be created. This means it has a different public key than the previous versions. This approach was chosen because breaking changes are often accompanied by a change in the functionality of the module. FBase represents this change as a new module.

3.3.2. Discovery Protocol

The discovery protocol of FBase relies heavily on crowd-sourcing. It accomplishes this through voting. It is very similar to how code repositories on GitHub gain popularity. The way the discovery protocol works depends on the state of the network. Below different scenarios will be discussed depending on the state of the network. Figure 3.2 shows the message flow.
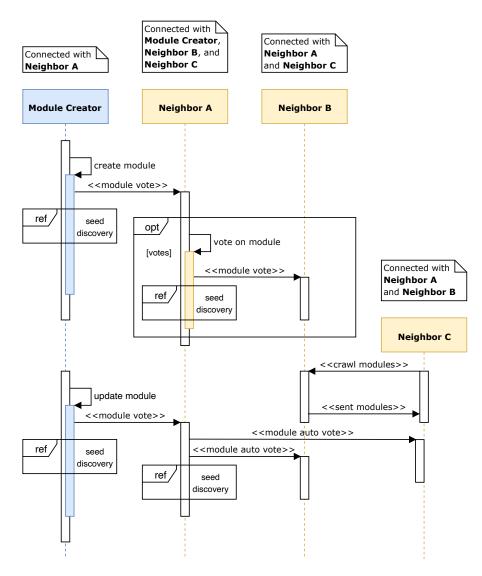
Figure 3.2: Sequence diagram of discovery protocol.

Scenario 1: New Module

When a new module is created the network is not aware of this. To make it possible for other nodes to discover this new module, the author will automatically cast its vote. This vote is stored on a blockchain and includes the public key of the voting node, a voting timestamp, and a module manifest. The module manifest contains the module identifier, module timestamp, and a description. The block representing this vote is sent to all connected neighbors in the FBase overlay network of the current node.

To give modules a better chance at discovery, FBase makes use of a mechanism called **Seed Discovery** (as shown in Figure 3.3). In this mechanism, 10 seed messages will each be sent through 10 intermediate nodes to reach their **Seed Point**. At each hop, these nodes select one of its connected neighbors at random, excluding the neighbor where the message originated from. The **Seed Point** will discover the module and make a simulated vote, called **Seed Vote**, that also informs its connected neighbors. This approach creates multiple different starting points from which modules can be discovered. This increases the possibility of a module to be discovered on a larger scale.

At the end of this scenario, the author and the 10 **Seed Points**, and their connected neighboring nodes know about the new module.
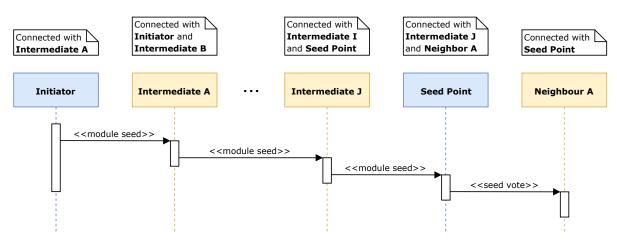
Figure 3.3: Seed discovery mechanism.

Scenario 2: Upvoting

When users discover new modules they can inspect the module and determine if they want to upvote it. When the user determines they do not want to vote on the module, the spreading is terminated at the current node. When they do want to vote on the module, the module information is distributed in the same way as in Scenario 1. This is done to promote the spreading of quality modules and hinder the spreading of bad quality ones.

Scenario 3: Updated Module

When a module is updated a new version has to be discovered by the network. To make sure critical updates are discovered as fast as possible, nodes automatically vote on updated nodes when they have voted on the previous versions. This fast updating is critical to prevent security vulnerabilities from persisting in code bases longer than necessary. In a framework designed around re-usable code, security vulnerabilities are a significant problem since many different applications rely on the same pieces of code. One vulnerability can affect many different applications.

Scenario 4: New Node

The previous scenarios do not solve the bootstrapping problem, where nodes that are new in the network do not have any information about modules before the node joined the network. The node will only discover modules voted on by its connected neighbors after it joined the network.

To bootstrap new users, the node crawls the blockchain of its connected neighbors. During this crawling, it fetches the modules that have been discovered by these neighbors and stores them in its own blockchain.

Performance

The performance of the discovery protocol is determined by multiple factors.

One of these factors is the number of connected nodes each node has. This controls the speed at which modules can spread through the network. Making this value too low creates a scenario where modules have a difficult time being discovered when just created. Making this value too big could flood the network with (malicious) modules and create a Distributed Denial Of Service (DDOS) attack.

Another of these factors is the active involvement of users of the system. When users don't actively participate in the voting process, it makes it considerably harder for modules to be discovered by the entire network.

A third factor is how capable users are in determining the quality of modules. When users vote on almost every item, it would flood the network. Since each user would send a message to all its connected neighbors for every module it knows about.

By making use of an event-driven discovery protocol, FBase tries to find a balance between the speed at which modules can spread through the network and preventing a flood.

3.3.3. Distribution

When nodes have discovered modules, the module code has to be distributed to them. To meet the requirements that where determined in Chapter 2, the distribution mechanism has to be decentralized.

By making the distribution mechanism decentralized it allows the network to operate without a central point of failure. Another benefit of this approach is that the load of storing and distributing the module code is spread across the users of the network. When nodes download a module it increases the availability of that module within the network. This means that modules that are more popular and will be downloaded more often, have a higher availability to scale with this. Modules that are less popular will have a lower availability to correspond with the demand.

Each module must have a minimum availability to make sure that every module that is discovered is also available to download. To make sure this happens the author of the module always keeps a local copy of the module. Besides this copy, the Random Discovery mechanism will make sure that a minimum number of other nodes also have a copy to prevent a problem when the original author is offline. When a Random Discovery message is received that node automatically downloads the module to increase the availability of it temporarily.

If a node has a copy of a module that it has not used for a while this module is deleted from that node to keep the network lean and fit.

3.3.4. System Strategies

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

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-party or by the user itself.

3.4. Blockchain Organizational Principles

FBase requires a blockchain that can store tamper-proof and accurate data records. We now explore three common blockchain structures, displayed in Figure 3.4.
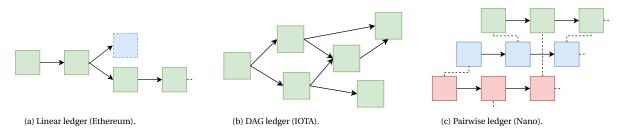

(a) Linear ledger (Ethereum).   (b) DAG ledger (IOTA).   (c) Pairwise ledger (Nano).

Figure 3.4: Three different structures of distributed blockchain ledgers. Each arrow points to the subsequent block in the chain.

Linear ledger:
Figure 3.4a shows the linear blockchain ledger used by Ethereum. The fundamental property of this ledger is that at least a majority of users agree on the exact sequence of transactions. A global consensus mechanism like Proof-of-Work or Proof-of-Stake prevents the double-spend attack where a malicious user intentionally creates a fork of their chain [39]. While providing a high level of consistency, the transaction throughput of these ledgers is often not high enough to facilitate record creation and modification by millions of users. This motivates us to consider different blockchain structures for FBase.

DAG ledger:
Another blockchain structure is the Directed Acyclic Graph (DAG) ledger, where each block can be referenced by multiple other blocks. This ledger structure, shown in Figure 3.4b, is adopted by blockchain platforms like IOTA and Dagcoin [30] [24]. IOTA is optimized for micro-payments within Internet-of-Things, and Dagcoin advertises itself as data storage for arbitrary data (e.g., documents or ownership records). Since these ledgers allow for different consensus mechanisms, transaction throughput is often superior compared to that of linear ledgers. However, they usually do not have the same consistency guarantees. While these ledger structures are more suitable for data storage, we consider current implementations unfit for the storage of

votes. The reason is that they either rely on a centralized coordinator (IOTA) or a fixed group of witness nodes (Dagcoin). Instead, our goal is to devise an infrastructure without any authority with leveraged permission.

Pairwise Ledger:

A third blockchain structure we consider is the pairwise distributed ledger. The key property of this ledger, given in Figure 3.4c, is that each user maintains and grows their individual chain with transactions. Each block holds exactly one transaction and optionally contains a (hash) pointer to a transaction in the individual chain of another user. Blockchain fabrics like R3 Corda, Nano, and TrustChain, use pairwise ledgers as their underlying data structure [29] [23] [9]. These platforms address the double-spending attack either by a trusted notary (Corda), a weighted voting system (Nano) or by guaranteed eventual consistency (TrustChain). In general, they provide superior scalability compared to linear ledgers as used by Bitcoin and Ethereum but lack global consensus.

We strongly believe that the pairwise distributed ledger is a suitable data structure to store voting records as transactions. Compared to linear and DAG ledgers, only data that the user is interested in is stored on that user's host. Pairwise distributed ledgers enable selective queries of data stored on the chains of other members, without the need for full data replication across the network. In FBase, each individual ledger stores all data associated with the actions taken by the user of that node, in a tamper-proof manner, and without global agreement.

3.5. Trust

Since this framework that is highly dependent on re-usable code, it is important that the user running the application can trust all its parts.

3.5.1. Verifiable Modules

To determine if a module can be trusted, the user should know what the module is supposed to do and what the module is actually doing. This is an important part of the voting protocol. Therefore, all modules should be verifiable. This means that the code in the module should be identical to the one the author controls and be open for inspection.

To accomplish this, every module should be cryptographically signed by the author. This digital signature can be verified by using the public key in the module identifier. This makes it impossible for malicious users to distribute a different version of the module. Only users who have access to the cryptographic key pair of the module can distribute new versions of the module.

The code base of a module must also be publicly available for all users of the network to inspect. This is achieved by distributing the code base openly through the network.

3.5.2. 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 having a shared identity, having its own identity, or having a pseudo-random identity, the framework provides a configuration option in the module configuration file to select what kind of identity profile is preferred.

3.5.3. Verified Identities

To further improve the trustworthiness of modules, users can optionally verify their digital identity. A verified identity is uniquely linked to a real-world entity. Software built on FBase can give preferential treatment to module authors that have verified their identity.

Identity verification can be done with an attestation given by a trusted third party like the government or a notary. Enforcing strong, long-lived identities in FBase is comparable with account validation that many centralized platforms use (e.g., the verification of a phone number). The requirement for verified identities addresses the Sybil Attack, where an adversary assumes multiple fake identities to influence or subvert the network [13].

Two solutions are proposed to achieve trustworthy importation of data: *challenges* and *TLS auditing*.

The first solution is to pose a challenge where the user importing the data, proves that they have control over this data. For example, when importing data from GitHub, we can require a public identifier (e.g., a public key) of the developer to be part of the "bio" profile field. This information can then be verified for correctness by other users who query the public GitHub API. Users who verify data are called *witnesses*. While this is a basic mechanism to ensure the accuracy of imported data, it heavily depends on the availability of a public API.

The second solution is TLS auditing [8]. The key idea is to proxy a TLS connection through a random witness, which then verifies and signs the data after the TLS connection terminates. When the TLS session finishes, the client gives the witness the private key used to decrypt HTTPS responses from the web service. Note that this way the witness is not able to decrypt the request made to the web service, which likely includes credentials or access tokens. The role of a witness can either be fulfilled by other entities in the network, or by a trusted notary service. Depending on the significance of data being imported, multiple witnesses can be used for this. Compared to challenges, TLS auditing works when access to a public API is absent but is more advanced.

3.6. Runtime Support

Once modules are discovered and distributed to the nodes in the network, they will be used to run in applications on the system. To accommodate this runtime support is necessary.

3.6.1. Module Interconnect Mechanisms

To combine the different modules into the user application the module interconnect mechanism is used. This mechanism uses a top-down search approach. When a module searches for a dependency, it sends a message on the event bus specifying the module type and name.

View modules can load Application modules. Application modules can load Service Modules and Package modules. Service modules can load Package modules. Package modules can load other package modules.

When the lower module receives the event for itself, it registers itself with the higher module.

View Module

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 to the view component through the event bus.

To define a view component, a special file has to be created: manifest.json. This definition file stores the attributes and the settings of the view component. Attributes of the file include name, version, and 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.

3.6.2. 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 set up 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 implementation details of the system described in the previous chapter. The thesis work aimed to create a proof-of-concept implementation to demonstrate the proposed concepts.

4.1. Overview

This work took a prototyping approach to get to a functioning prototype rapidly and improve from there. All major concepts that were discussed in Chapter 3 were implemented in the proof-of-concept code base. The code was developed alongside the research and served as an evaluation of the principles being worked on.

As the main purpose of the implementation is a proof-of-concept, this codebase is not optimized for a production environment. Various different performance and quality optimizations can be done to improve the quality of the implemented product. Examples of these optimizations include:

- Implement caching for module database

- Remove verbose validations found all over the code

- Making the implementation more crash-resistant

- Better integrate the torrent library by hooking into its event system for notifications

- Improve the user-friendliness

The final codebase can be found publicly on GitHub [1]. The readme for the project can be found in Figure 4.1. The programming language that was used for the prototype is Python. Python was chosen because it allows for rapid development and has access to a large collection of existing libraries. Since Python is dynamically typed and allows a lot of flexibility when it comes to security, it also made it easier to implement the proposed concepts in this programming language. The codebase consists out of 5000+ lines of code.

During the development of the implementation, several improvements were made and contributed to the overlay library used. This overlay library was only available on GitHub and needed to be used through Git Submodules. By working with the developers of the library we made changes to the library that allowed it to be distributed through the Python Pip package manager. This allows the library to be more easily integrated and used by other developers. Another improvement that was made to the library was a problem discovered in the REpresentational State Transfer (REST) protocol used for communicating with the library backend. This bug made it impossible for HTTP clients that implemented CQRS security to communicate with the library. We proposed a new base endpoint for the REST service that implemented this security protocol that all other endpoints extend from. This rewritten REST service was eventually merged into the main code base of the library. The pull requests for these changes can be found on the library's GitHub repository [2].

---

[1]https://github.com/mitchellolsthoorn/ipv8-module-loader
[2]https://github.com/Tribler/py-ipv8/pulls?q=is%3Apr+is%3Aclosed+author%3Amitchellolsthoorn

ipv8-module-loader

Setup

```
pip install -r requirements.txt
sudo apt install python-libtorrent
export PYTHONPATH="${PYTHONPATH}:."
```

Run single instance

```
twistd -n module-loader -s <state directory location>
```

Example:

```
twistd -n module-loader -s data/one
```

Run multiple instances on a single computer

```
twistd --pidfile twistd1.pid -n module-loader -s <state directory location 1>
twistd --pidfile twistd2.pid -n module-loader -s <state directory location 2>
...
twistd --pidfile twistdX.pid -n module-loader -s <state directory location X>
```

Example:

```
twistd --pidfile twistd1.pid -n module-loader -s data/one
twistd --pidfile twistd2.pid -n module-loader -s data/two
```

Figure 4.1: Readme of the IPv8 module loader project.

4.2. IPv8 - Overlay Library

IPv8 is the underlying network overlay library used in the implementation. It is responsible for providing **authenticated** and **encrypted** communication between different peers (computer nodes) in the system. The framework abstracts the notion of physical addresses (IP addresses) in favor of public keys. This removes the need for applications that use this library to keep track of where different peers in the system are and how to move data between them. IPv8 simplifies the design of distributed overlay systems.

Some other important aspects of the framework are its focus on:

- **Privacy:** where it is possible to choose if messages should be identifiable to all peers in the network or only to the peers absolutely needed for the network connection (doesn't include the receiver).

- **No infrastructure dependency:** allowing the network to function on its own run by the peers using the system. This is a very important aspect of the framework as it allows the framework to support itself, without needing external financing for server capacity.

- **NAT traversal:** making it possible to operate the network without static servers needed for overcoming the NAT issues that most peer-to-peer networks face.

- **Trust:** one of the most important aspects of peer-to-peer systems, as it is needed to mitigate free-riding issues in the network. In IPv8 trust is gained by recording a pattern of previous actions and storing

these on a blockchain structure called TrustChain.

The last main aspect of the framework is **extensibility**. IPv8 makes use of a concept called overlays. Where a virtual network is created in the system related to one specific application domain or topic where different peers can subscribe to. This is a very powerful mechanism to allow extendability and modularization of an specific application. This makes it possible for user applications to be run alongside the FBase framework.

4.3. Framework Structure

This section will go over the major components of the code and explain the folder structure used in the implementation. The root directory of the project consists out of two folders:

- **module_loader:** The module loader directory serves as a python package for the entire code base of the prototype. All the major components of the framework are located in this directory.

- **twisted/plugins:** The twisted plugins directory is a mandatory location for the storage of the framework's main application files. This requirement comes from the library Twisted which IPv8 is built on. This library provides pseudo-multi-threading by implementing an event-based scheduler.

Inside the module loader python package, the following structure is used:

- **CLI:** The CLI package contains all classes and logic related to the command-line user interface of the framework. This is the primary way to manage the network.

- **community:** The community package contains the overlay code. This includes the majority of all logic related to the FBase functionality.

- **event:** The event package contains all classes and functionality related to the main event bus of the FBase framework.

- **REST:** The REST package contains all REST endpoints needed to communicate with the FBase backend from other applications and services. This interface also allows user applications to communicate with their frontend

- **web:** The web directory contains the web frontend for the FBase framework. This is an alternative way of managing the framework next to the CLI.

4.4. Community

The community class is the main overlay class that manages all parts of the FBase module network. This class acts as the controller for all inputs and outputs of the system.

Figure 4.2 shows snippets of the community code base as the central part of the application logic.

4.5. Event Bus

The event bus that is implemented in the FBase prototype is built according to a PubSub design pattern. In this design pattern, modules can register themselves to receive messages when an event has been fired with a certain event type. These registered modules, also called subscribers, are stored in an array in memory. They only receive a call when that event is fired. Modules that publish events, called publishers, call the event bus with the message and the type of the message.

In the prototype, the event bus is implemented in code. For production use, however, it might be a good improvement to run the event bus as a separate process to speed up the performance when many events are being fired at a time. It would also allow multiple processes to be used to handle the actions performed by the subscribers.

4.6. Interface

There are two interfaces through which the FBase framework can be managed. Originally only the CLI interface was used. Later in development, the web interface was added to allow the framework to be controlled from any platform the framework runs. This decision was made since the framework itself could be run cross-platform, but it could only be controlled from platforms that support a CLI.

```
39   class ModuleCommunity(Community, BlockListener):
40       """
41       Overlay for the module network. Handles all communication and actions related to managing and maintaining the module
42       components.
43       """
44
45       # Register this community with a master peer.
46       # This peer defines the service identifier of this community.
47       # Other peers will connect to this community based on the sha-1
48       # hash of this peer's public key.
49       master_peer = Peer(unhexlify("3081a7301006072a8648ce3d020106052b81040027038192000403d4d2d1fc98e2e3a7b0127acffbbc1a"
50                                    "ade720955b6a3c8b4de1a25686f20b6e150591f1251252c4cd30bfaa8ca5f1d2c68327cbef939958c94d1"
51                                    "a441c20b10acd9c53e5c9023a6011d626e69290a4ef98c2588ab1eb2ca9a3fb6f08042e1c93c9bad60bbb"
52                                    "0f33fc156924e3914be9bd11d702fe1ab307c40634e97d476462d669ebc4a39c5dd10eb58ab2d8a86c690"
53                                    ))
54
55       # Override block class with custom module block
56       BLOCK_CLASS = ModuleBlock
57
58       def __init__(self, my_peer, endpoint, network, trustchain, bus, **kwargs):
59           """
60           Initialize module overlay
61
62           :param my_peer: Node identity peer instance
63           :type my_peer: Peer
64           :param endpoint: IPv8 endpoint instance
65           :type endpoint: Endpoint
66           :param network: IPv8 network instance
67           :type network: Network
68           :param trustchain: TrustChain overlay
69           :type trustchain: TrustChainCommunity
70           :param kwargs:
71           """
72           super(ModuleCommunity, self).__init__(my_peer, endpoint, network)
73           super(BlockListener, self).__init__()
74
75           self.trustchain = trustchain  # type: TrustChainCommunity
76           self.bus = bus  # type: EventBus
77           self.working_directory = kwargs.pop('working_directory', "./")  # type: str
78           self.ipv8 = kwargs.pop('ipv8')  # type: IPv8
```

Figure 4.2

4.6.1. CLI
The CLI interface operates through a command-line menu interface. This menu allows the user of the framework to run sets of actions. The general actions are

- **List all modules:** This action provides a list with context actions. Each of these context actions is performed on the module that is selected by the user.

- **Create test module:** Create an empty test module that can be used for testing the distribution of modules through the network.

- **Create module:** Create a module from a module definition that exists on the file system. The module should already be created. This action only processes the module and prepares the metadata for publishing it on the network.

The context menu for each module shows information about the module. It includes the name of the module, the description, the identifier and the number of votes in the network for the module. Next to this information, there are also context actions that are displayed and can be called. The context actions include:

- **Vote:** This context actions instructs the backend of the framework to sign the corresponding vote block that will be created. This vote block will then be distributed through the network.

- **Download:** The download context action retrieves the identifier of the module and starts the download of the module through the transport engine.

- **Run:** The run context action loads the module package namespace into the Python path of the framework. Afterward, it starts the application based on the instructions provided in the module.

Figure 4.3 shows a screenshot of the CLI interface showing the list of discovered modules.

Figure 4.3: CLI interface of the FBase framework proof-of-concept.

4.6.2. Web Interface

The web interface is a website built with HTML, CSS, and JavaScript that runs as a standalone service that communicates with the FBase backend through the REST API. This interface allows to view all discovered modules, download them and run them. This interface does not allow a module to be created.

4.7. REST Endpoints

The FBase framework provides several REST endpoints on its API for the web interface to operate. These API endpoints can also be used by other applications to perform actions on the overlay network.

- **Catalog:** The catalog endpoint is responsible for all actions related to the discovered modules on the network. This endpoint is mostly used for retrieving information about the modules available.

- **Downloads:** The downloads endpoint is responsible for managing modules that need to be downloaded or are already downloaded. Its actions include the downloading of modules, the deletion of modules from the host, and to retrieve the status of the downloaded modules.

- **Library:** The library endpoint is used to manage all modules that are currently being used by user applications on the framework.

- **Run:** The run endpoint provides actions to load and run the module that is specified.

- **Votes:** The votes endpoint is responsible for all actions related to voting and managing votes that have been performed.

4.8. Module Structure

This section will expand on the structure used for modules in the framework. Each module in the framework has to be a valid Python package. This is done to make it easier to import the module on runtime. This requires the module to have an __init__.py file in its root directory. Besides this, each module must also have a manifest.json file. This manifest file specifies the type of module and its information. Without this file, the module will not be detected and can not be run.

This manifest file contains a JSON dictionary. This dictionary contains key-value pairs for all the information required.

- **Name:** Specifies the name of the module used for displaying in the user interface

- **Description:** Specifies the description of the module used for determining if a user wants to download and use the module

- **Version:** Specifies the version of the module to determine if this is the most current version of the module

- **Type:** Specifies the type of the module, determining how it needs to be run and what kind of modules can require it.

- **Dependencies:** Specifies the other modules this module relies on.

4.9. Module Distribution

The module distribution method was chosen based on the ability to set up 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.

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 efficiently when large amounts of the same content have 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.

BitTorrent also has the advantage of having a Distributed Hash Table (DHT). This DHT stores all information about who has a copy of the content and the metadata needed to download it in a distributed fashion. This removes the need for FBase to implement its own mechanism for this.

BitTorrent is the protocol used in the proof-of-concept.

4.10. Discovery and Voting

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 our chosen method of module discovery.

Since IPv8 also provides a block-chain storage back-end it was an perfect opportunity to also implement voting on the IPv8 library.

When a vote is done a block is created to represent this vote. This block is then stored on a blockchain. The reason we store the votes on the blockchain is to make votes irrefutable. Voting is unidirectional. Permission should not be needed from the other side. This is represented in the blockchain by only one party signing the vote. Storing votes on the blockchain also prevents malicious use of votes to promote one's module.

4.11. Blockchain

The FBase prototype is built on the TrustChain ledger introduced by Otte et al. We identified two advantages of TrustChain over other pairwise distributed ledgers like R3 Corda and Nano.

First, TrustChain focuses on fraud detection instead of prevention and as a result, does not require network-wide consensus. This makes TrustChain a lightweight and simple data structure.

Second, TrustChain is already used as transaction fabric within a self-sovereign, decentralized identity system, described in the work of Stokkink et al. Availability of a self-sovereign identity system aligns with our requirement for strong, long-lived identities.

5

Evaluation

This chapter will evaluate the proposed framework, FBase, described in Section 3. The evaluation consists of two parts, which will show if FBase, satisfies the requirements determined in Chapter 2:

- Testing if the concept works and is viable.
- Examining the effectiveness of the discovery protocol

5.1. Testing the Viability of the Concept

To determine if the concept works and is viable, this thesis uses one non-trivial use-case. This use-case comes from a project called Tribler.

5.1.1. Tribler

Tribler is an open-source community-driven decentralized BitTorrent client being developed and researched at the Delft University of Technology. Its main feature is that it allows anonymous peer-to-peer communication by default. It is built on the underlying network library IPv8, also being worked on by the same group.

Besides handling the tasks of a standard BitTorrent client, Tribler also makes it possible to:

- **Search for content:** allowing the program to operate independently of external content search providers that could be blocked and made it immune to limiting external actions such as legal constraints. Which is happening more frequently nowadays.

- **Torrent anonymously:** routing torrent traffic through anonymized tunnels that operate using the same principle as the TOR stack. Providing pseudo-anonymity for the two end and other observing parties.

- **Accumulate trust:** all torrenting metadata is stored in a way that is not linked to an physical identity or an IP address. This data is then translated into a trust score by calculating the ratio between the amount of traffic communicated across the network. A positive seed ratio (the ratio between uploading content and downloading content) indicated a positive trust value.

- **Trade trust:** With this trust system it is possible to prioritize or refuse services for particular users. To increase the incentive for having a large seeder network and therefore a high trust value, Tribler allows users with a large amount of uploaded content to exchange this gathered trust for currency on the built-in marketplace inside the Tribler application.

This trust value, expressed as reputation inside the Tribler application, can be described as an up- and download currency in a reputation-based peer-to-peer network. When a peer uploads more than it downloads, the reputation of that peer increases, and the peer can download more effectively.

5.1.2. Trust Experiment

The experiment consists 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 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)

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 example for this experiment.

5.1.3. Mobile App Experiment

To test the robustness and 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 platform 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 severely hinders the working of the framework, since more functionality is added by the distribution of applications 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 possible to include a web browser as the only component written for the mobile platform. This web browser can then interact with the webserver 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 an MP3. This format does not get checked by the Android security system and therefore can be used for this work. Underneath the extension, 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.

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.

5.1.4. Result Interpretation

One part of evaluating FBase is testing if the concept works and is viable. Through the use of a non-trivial use-case, it was demonstrated that real-world practical problems can be solved using this framework.
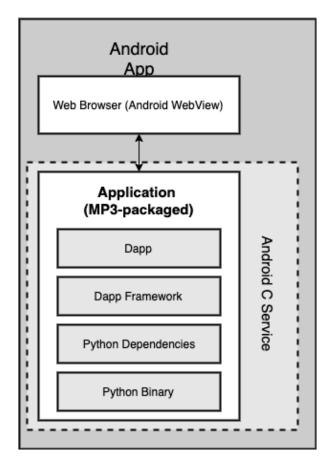
Figure 5.1: Android app architecture.

The main advantage that the framework provided in the use-case was the introduction of modularity. On one hand, it has the benefit of flexibility and variety in use. On the other hand, modularization improves the manageability of maintenance for complex software like Tribler.

One of the disadvantages that followed from the use-case was the increased time that modules on FBase took to develop the application compared to implementing it in a monolithic architecture. A second more general disadvantage is that the module interconnect limits the complexity of the interaction between modules. This disadvantage did not limit the development of this use-case but changes the way applications need to be developed.

5.2. Effectiveness of the Discovery Protocol

One of the most important components of FBase is the module discovery protocol. To make this platform usable, this mechanism has to perform sufficiently effective without crippling the network with its overhead. The discovery protocol is compared to existing methods and the effectiveness is evaluated based on a theoretical sensitivity analysis.

5.2.1. Existing Methods

One of the simplest and easiest approaches that can be taken to discover new modules is flooding information packets throughout the network. This is the fastest approach for discovering modules, however, it will create a big strain on the network when then network becomes larger. It also allows malicious nodes to DDOS the network with relative ease.

Another approach is crawling. When crawling, nodes ask their neighboring nodes for the modules they have discovered. This approach is much less intensive on the global network as it works on a localized view. This method will eventually create a global coverage for module discovery, however, this method is very slow. Each module has to be propagated through each local view. This method is not suitable for large scale module discovery.

5.2.2. Sensitivity Analysis

FBase makes use of a discovery protocol based on a voting mechanism to prevent DDOSing while still being able to effectively discover new modules. It uses selective flooding on a local scale. Flooding is a effective way to reach a large number of nodes connected in a graph as can been seen in Figure 5.2. Even when there is a significant overlap in the neighbors of the nodes, thousands of nodes can be reached in very few steps.
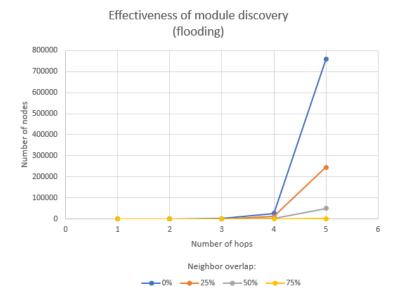

Figure 5.2: Effectiveness of module discovery (flooding).

However, If the same analysis is run with the voting mechanism the results are not as effective. The reason behind this behavior is that only a small fraction of all users will vote on a specific module. For this analysis, this percentage is set at 1% of all nodes. Even in the most optimistic scenario where there is no neighbor overlap, the discovery stops after two steps with 24 nodes being aware of the module.

To circumvent this problem FBase uses the concept of Seed Discovery where multiple seed points are used for discovery when the author creates a module. This increases the chance for one of these starting points to overcome this barrier. This effect can be seen in Figure 5.3. However, when applying Seed Discovery only when a module is created the discovery process still halts after 6 steps with 101 starting points.
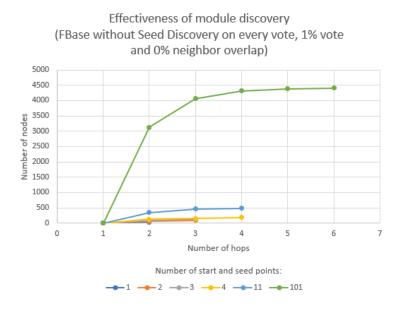

Figure 5.3: Effectiveness of module discovery (FBase without Seed Discovery on every vote, 1% vote and 0% neighbor overlap)

In a more realistic scenario where there would be a 25% overlap between the neighbors of nodes, the result would be even less effective as can be seen in Figure 5.4.
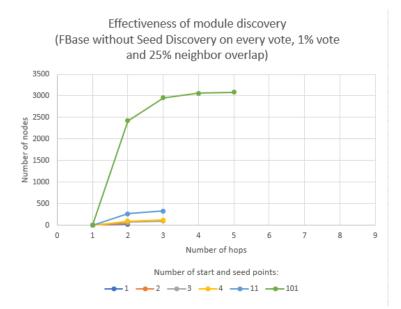

Figure 5.4: Effectiveness of module discovery (FBase without Seed Discovery on every vote, 1% vote and 25% neighbor overlap)

When the Seed Discovery is applied on every vote, a continuous discovery can be found with 11 starting points as can be seen in Figure 5.5.
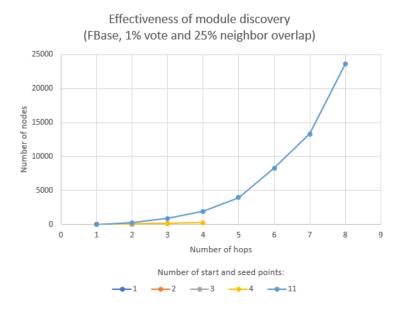

Figure 5.5: Effectiveness of module discovery (FBase without Seed Discovery on every vote, 1% vote and 25% neighbor overlap)

This approach results in an effective discovery without flooding the network and preventing malicious attempts.

5.2.3. Result Interpretation

The effectiveness of the discovery protocol is the second part that was analyzed in this evaluation. The sensitivity analysis indicated that a degree of flooding is necessary to accomplish effective network coverage without putting excessive strain on the network. When comparing different situations of selective flooding by introducing voting, it was shown that seed discovery is necessary on every vote for the discovery to reach global coverage.

6

Conclusion

6.1. Conclusion

Re-usability has always been a property software developers have tried to achieve. However, the complexity of applications increases when software is more re-usable. Since this complexity decreases the usability of the software, multiple concepts have failed to successfully implement the re-usable aspect in their software designs.

As software re-use has the potential of high development efficiency, it is still worth pursuing this goal. The proposed concept strives to find a balance between these two contradicting principles. The problem analysis led to the following research aims in developing the conceptual framework FBase:

- 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 main aim of this research is to find a trade-off between re-usability and usability. This work sets out to achieve this balance by limiting the granularity of re-usable modules to a distinct set of four component types. To further enhance usability, an ecosystem was proposed to mask the negative effects that are associated with re-usability. Previous attempts at solving the re-usability problem have mostly focused on an architectural level in contrast to FBase. By integrating sub-systems into one ecosystem it improves the usability of the framework.

The secondary aim of the research is to address the corresponding security problems that re-usability creates. The approach, to use social trust and crowdsourcing to improve the security of libraries, was selected as a platform with external dependencies that requires the notion of trusting people. Since trust is a social construct, which can not fully be solved by automated systems, it needs to be gathered from other users. The way to achieve this is by letting the combined expertise of all users in the system determine if a module can be trusted.

The third and last aim of this research is to ensure dependency availability efficiently and securely. An important aspect of dependency management is preventing code alteration in re-usable code during distribution. Another aspect is ensuring that new versions do not introduce security vulnerability to applications using them. Finally, dependencies always need to be available despite changing possible author intentions. Current popular approaches lack the previously mentioned aspects. FBase addresses these aspects by implementing an integrated autonomous discovery and distribution mechanism.

FBase uses the aforementioned approaches to satisfy the research aim but needs to work and be viable to be considered successful. An evaluation showed that FBase can be used in a non-trivial use-case to increase the flexibility and variety of its use, and improve the manageability of maintenance. The disadvantage, the limitation on the complexity of the interaction between modules, did not limit the development of this use-case but changes the way applications need to be developed.

6.2. Discussion

The research aims for this project have no singular answer. They form a guideline to approach a problem that has not been solved, despite numerous attempts. This work is making yet another attempt to solve that problem. It has taken into account the successes and failures of previous attempts to improve the current state of the field.

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] Tlsnotary - a mechanism for independently audited https sessions. *URL https://tlsnotary.org/TLSNotary.pdf*, 2014.

[9] Richard Gendal Brown. Introducing r3 corda: A distributed ledger designed for finanial services, 2016, 2017.

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

[11] 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.

[12] 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.

[13] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.

[14] 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.

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

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

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

[18] 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.

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

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

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

[22] 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.

[23] Colin LeMahieu. Nano: A feeless distributed cryptocurrency network. *URL https://nano.org/en/whitepaper*, 2017.

[24] Sergio Demian Lerner. Dagcoin: a cryptocurrency without blocks, 2015.

[25] Tom McCourt and Patrick Burkart. When creators, corporations and consumers collide: Napster and the development of on-line music distribution. *Media, Culture & Society*, 25(3):333–350, 2003.

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

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

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

[29] Pim Otte, Martijn de Vos, and Johan Pouwelse. Trustchain: A sybil-resistant scalable blockchain. *Future Generation Computer Systems*, 2017.

[30] S Popov. The tangle, iota whitepaper, 2018.

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

[32] 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.

[33] 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.

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

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

[36] 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.

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

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

[39] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *iNetSec*, pages 112–125. Springer, 2015.

[40] 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.

[41] 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.