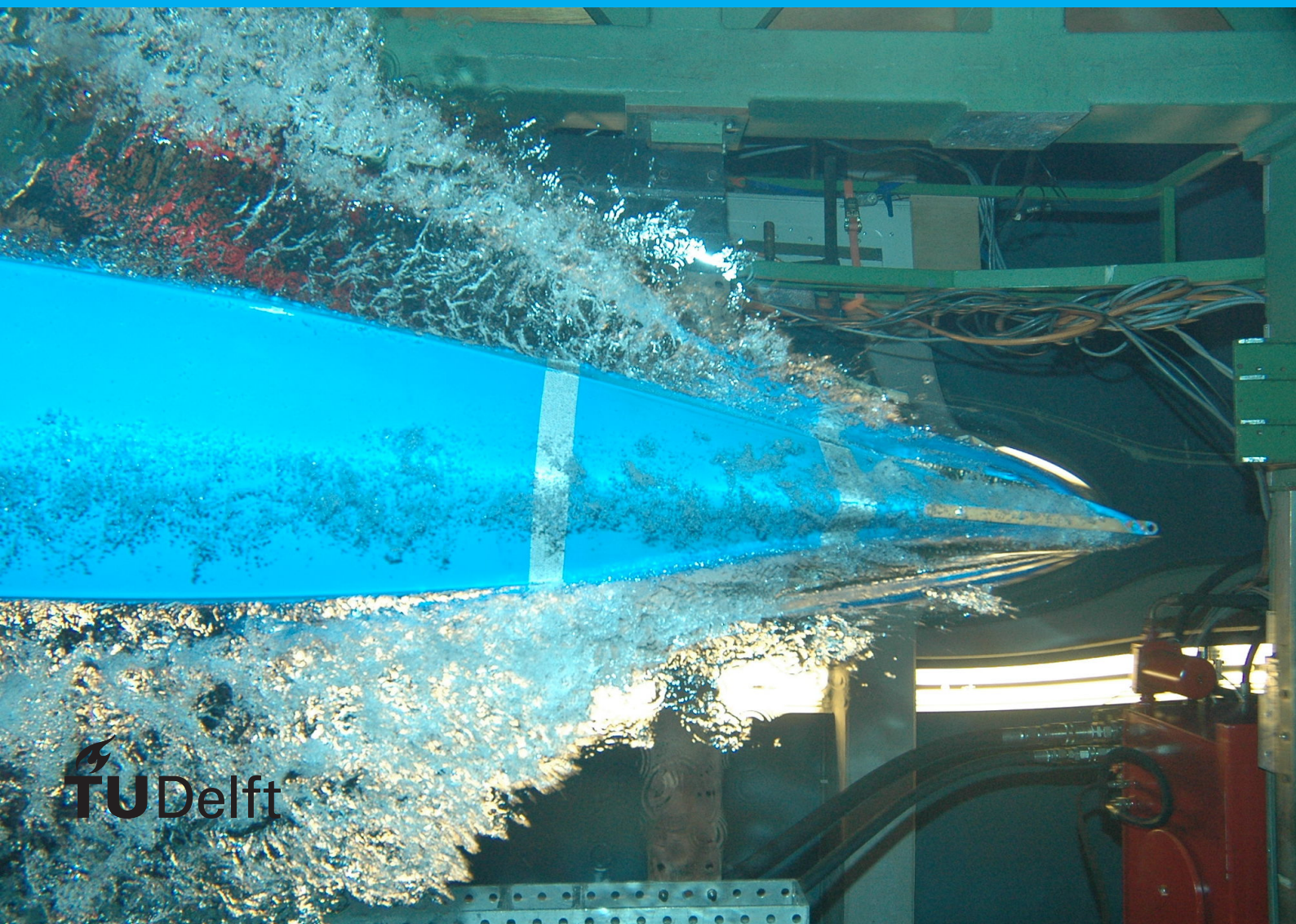
A unified framework for developer modules and user plug-ins

M. J. G. Olsthoorn



TU Delft

A unified framework for developer modules and user plug-ins

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 January 1, 2013 at 10:00 AM.

Student number:	1234567	
Project duration:	March 1, 2012 – January 1, 2013	
Thesis committee:	Prof. dr. ir. J. Doe,	TU Delft, supervisor
	Dr. E. L. Brown,	TU Delft
	Ir. A. Aaronson,	Acme Corporation

This thesis is confidential and cannot be made public until December 31, 2013.

TU Delft

Preface

Preface…

M. J. G. Olsthoorn
Delft, January 2013

Abstract

Contents

1 Introduction

This thesis focuses its work on a project called Tribler and its underlying network library IPv8.

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 is allows anonymous peer-to-peer communication by default. It is built on the underlaying 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 maked 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.

1.2. IPv8

IPv8 is the underlying network framework of the Tribler application. 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 favour of public keys. This removes the need for application that use this framework 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 patterns 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.

1.3. Code evolution

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

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

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

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

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

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

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

1.4. Code re-use

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

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

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

1.5. Re-usability vs usability

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

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

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

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

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

1.6. Modules vs Plug-ins

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

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

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

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

2

Problem description

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

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

2.1. Scope

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

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

2.2. Requirements

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

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

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

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

2.3. Related work

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

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

3

Design and implementation

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

3.1. Module Distribution

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

3.1.1. TFTP

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

3.1.2. FTP(S)

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

3.1.3. 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.

3.1.4. BitTorrent

3.2. Discovery and Voting protocol

3.3. Module Design

3.4. Event-Driven Architecture

3.5. GUI integration

3.6. Code review

4

Experimentation and evaluation

5

Conclusion

Bibliography