Delft University of Technology

Contextproject - Blockchain

Architecture Design of the Network Explorer

■-■-■- BlockchainBoys -■-■-■–

Yinghao Dai, Max van Deursen, Geert Habben Jansen
Bram van den Heuvel, Ruben Keulemans, Tim Speelman

supervised by
Martijn Gribnau, Stefan Hugtenburg, Johan Pouwelse and Alberto Bacchelli

June 29, 2017

Abstract

In this document the architectural choices in creating an enhancement for Tribler are presented. Design goals for this architecture are ease of maintainability through modularity, reuse of existing code and writing easy to maintain documentation. Currently, Tribler is split into a daemon and a GUI which communicate over HTTP. Both the daemon and the GUI are extended to support our enhancement of the Network Explorer.

Contents

Chapter 1

Introduction

This document reports on all major design decisions that were made in building an interactive Network Explorer into the Tribler application. Tribler is an open source BitTorrent client with a built-in video player which allows anonymous for peer-to-peer downloading and sharing of content using YouTube-like *channels*. [1]. It is a research project used to experiment with decentralized systems and human cooperation. It is developed by students from Delft University of Technology and volunteers from all over the world on GitHub [2].

Developing software which will be part of such a project requires some specific considerations. These will be laid out in chapter 2 and corresponding goals will be formulated. Next, we will describe how we tried to achieve these goals in several different areas of the software architecture of our software. First, we will describe the different subsystems both in the existing Tribler software and our added software. We will then consider the persistent data management of Tribler and how our additions make use of persistence. Lastly, we will consider how Tribler handles concurrency and how our software fits in this model.

Chapter 2

Design goals

2.1 Tribler development

When considering the different design goals, it is essential to consider the way in which Tribler is developed. As an open source project, many different people work on the software. As Tribler is worked on mostly by students, many developers contribute only for a short while to the Tribler project. As such, there is little continuity in the development and many developers have to understand the existing architecture. There is also little time to maintain software as most students prefer to work on a new feature rather than invest time in making structural changes to improve code quality. Therefore, the *ease of maintainability* of the code should be the primary goal in considering architectural decisions.

As many students work on Tribler code, the relative amount of time spent learning to work with this code is higher than in most projects. We attempt to minimize this by writing software that is easy to understand, especially by people who did not work on the code before. After all, we want others to participate in finding errors and extending our software.

2.2 Ease of maintainability

Achieving maintainable software is attempted in a number of ways, modularity, the usage of existing code and easy to maintain documentation. These three points are elaborated upon below.

2.2.1 Modularity

Our software is split up into different modules, all of which have other responsibilities. This is done for several reasons. Firstly, this makes code easy to maintain as changes to a portion of the code are less likely to propagate to other parts of the code. Secondly, it reduces the time students need to invest into finding portions of the code relevant to their work. This allows them to work only with a small, relevant part of the codebase rather than large components which also contain a lot of other functionality.

2.2.2 Use of existing code

An attempt is made to minimize the introduction of *new ways of doing things*. That is, making use of existing code, dependencies and workflows as much as possible. Examples are using the same

configuration system, using similar methods of persistence and using the same methods of communication between the Tribler Core and Tribler GUI.

2.2.3 Easy to maintain documentation

As most of the Tribler developers do not have time to maintain the software, they prefer spending the smallest amount of time possible documenting their code changes and extensions. Past experience shows that external documentation is often neglected. Thus, it is necessary to find a way to document our code such that future developers feel only a very small threshold in updating and extending it.

Chapter 3

Software Architecture views

3.1 Subsystem decomposition

3.1.1 Existing codebase

Currently, Tribler is separated in a daemon and a GUI.[1] The Tribler daemon offers a HTTP API which allows the Tribler GUI to retrieve all the information it needs to display.

The Tribler daemon

Short of the GUI, the Tribler daemon is responsible for all functionality of Tribler. This includes all communication with peers, building and gathering a trustchain history, downloading torrents and providing search functionality.

The Tribler GUI

The Tribler GUI displays data it requests via the HTTP API. It contains multiple sub-windows which display information such as the current downloading torrents, search results and channels. It can also show statistics about the history of the user, which it retrieves from the database by requesting the information from the HTTP API.

The current statistics display

In the current Tribler codebase, there is a page containing several statistics of the user. Although this is called the "trust page", it could actually better be called "statistics page", for reasons that will be clarified in section 3.1.2. It fetches the data by performing an HTTP-request to the so-called *request manager*, which is still part of the Tribler GUI. The request is then propagated to the *trustchain endpoint* (since the statistics are part of the trustchain in Tribler), which is part of the HTTP API. In order to be able to build a response, the trustchain endpoint gets the statistics from the so-called *community*, which is part of the Tribler daemon. The community gets the information by retrieving the latest block of the given user from the database. The latest block carries information such as total amount of

[1] GUI stands for *Graphical User Interface*, the part of the software with which the user interacts. In contrast, the daemon is the background processes with which the user does not directly interact.

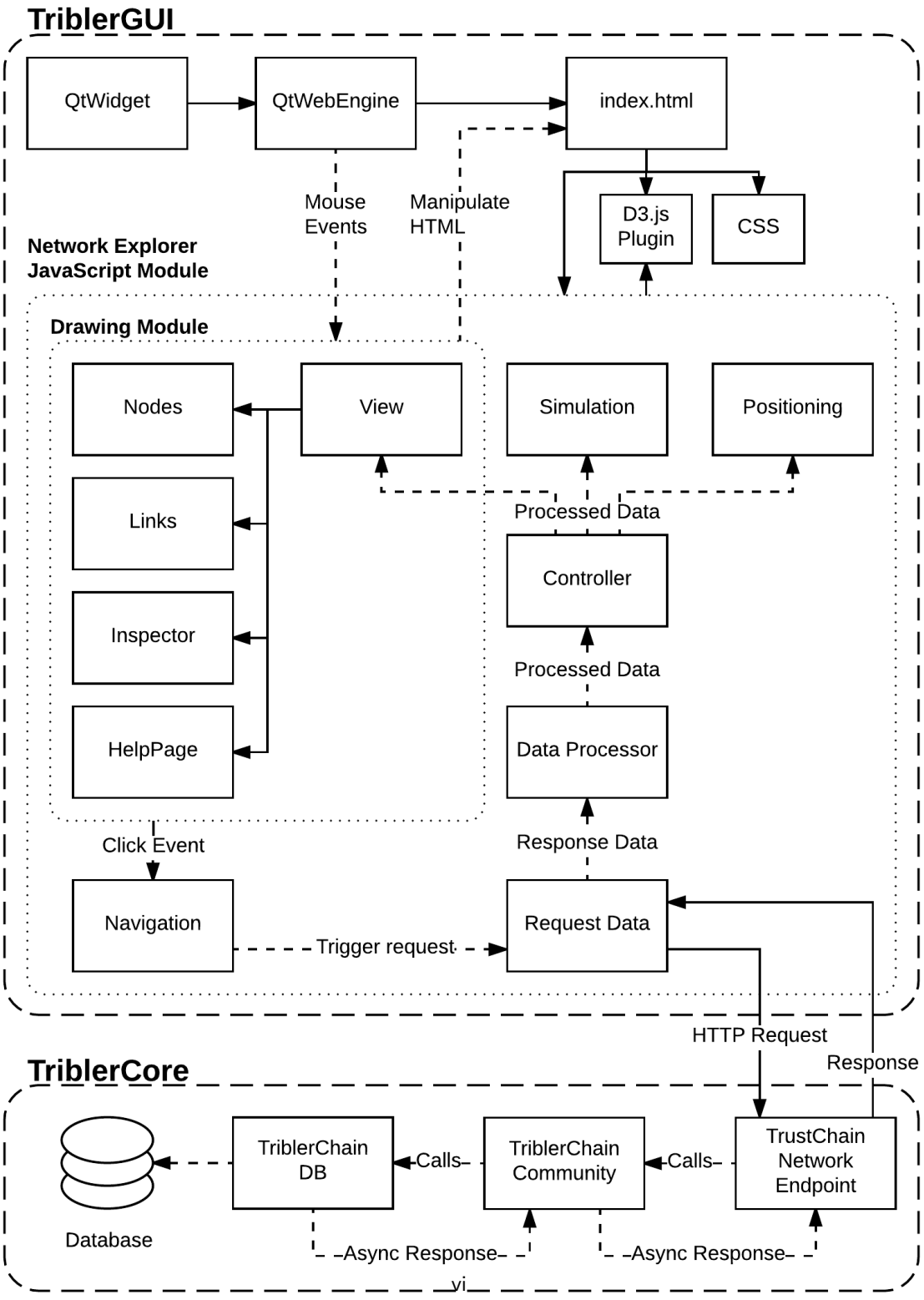

Figure 3.1: Network Explorer Subsystem Decomposition

uploaded/downloaded content. Other necessary information can be deduced: if the sequence number of the latest block is known, then the total number of blocks is known as well. Upon having received the information, the community puts the statistics in a Python dictionary, so that they can be looked up by the trustchain endpoint. The latter then builds a JSON-object to put the data in, as is usual for an HTTP-message. The Tribler request manager puts it into a Python dictionary again, in order to make the process of looking up information easier. It might seem superfluous to convert a Python dictionary into a JSON-object and then into a Python dictionary again, but this is exactly how sending an HTTP-message works. Similarly, a written letter is put into an envelope, whereas the envelope has to be opened at the other side again, as that is exactly how sending a letter works. Now the statistics page has received a Python dictionary containing all the information that it needs. It can then show these statistics on screen, possibly within a graph.

3.1.2 Our extension

The main thing that can be deduced from these statistics, is the reputation of a certain user. For example, if a user downloads much more than he or she uploads, this user is considered a *free-rider* and will have a bad reputation. However, it is not directly clear what is meant by "much" or "much more". To decide whether a certain amount of downloaded content can be considered "much", this number needs to be compared to other numbers. As a user cannot be expected to indulge in the statistics of millions of other users in order to make a comparison, they should be offered a little bit of help. Therefore, Tribler is extended with a clear and interactive visualization of the reputation of different users in the network. This is where "reputation" is distinguished from "statistics": a user can see their own statistics, but only knows something about their "reputation" after having compared the statistics to that of their neighbors and other users.

Extension to the Tribler daemon

In order to accomplish this, the current Tribler daemon needs to be extended. For instance, it is necessary to keep track of more information, such as the amount of data that flows from user A to user B, rather than only the total amount of outgoing data at user A. In order to pass this new information to the Tribler GUI, a HTTP API is needed as well. As described in the previous section, that means an endpoint are also necessary.

Extension to the Tribler GUI

Of course, the Tribler GUI needs to be extended as well. In the first place, a new page on which the reputations for all users will be visualized, is needed. On this page, a graphical representation of (part of) the network is given, and the users' reputation are clearly indicated (for example by using colors). Furthermore, the user should be able to interact with this display, in order to retrieve the same information about other users that are not yet shown. The Tribler GUI should react to this, request additional information using the HTTP API and show this on the user's screen in a clear way. Whenever a free-rider or cluster has been detected, it should also be shown clearly on the new Network Explorer page. That means there should be much interaction, not only between the user and the GUI, but also between the GUI and the HTTP-API (or request manager). Since Python does not have any modules that give us the freedom to create an interactive visualization of the network, we use a web view inside Python. The web view renders a HTML page inside the GUI window in which we can use the D3.js

JavaScript library. The D3.js library enables us to freely create an interactive visualization of the network and we can link this visualization to the HTML page using SVG.

3.2 Persistent data management

3.2.1 Existing codebase

Tribler currently has several ways of achieving persistence. It creates a state directory on the user's hard drive in which it stores several files.

Configuration options

All options with which Tribler can be configured are stored in plain text files. These can be read using the configuration library *ConfigObj*, which is shorthand for *Configuration Object*. These objects provide a simple *key, value* store accompanied by several helper methods to make usage convenient.

Downloads

Partial downloads are stored as files on disk accompanied by metadata, which are saved in '.state' files.

Database

Tribler also makes use of an SQLite database. It stores information about collected torrents, previously met peers and discovered trustchain keys. The database is also used to store the trustchain history.

3.2.2 Necessary storage

It is most likely that the extension will have some configurable options. These can be stored in the existing configuration files and objects.

The data which the extension will display is contained within a table in the existing SQLite database of Tribler, but not in a way that we can use to query it efficiently. Our application uses a separate table inside the same database. We use this table to store the necessary data, transformed into a format we can use, for each transaction in the main table. This way, we can optimize the trade off between memory usage of storing extra data and the speed provided by optimizing SQL queries.

3.3 Concurrency

Tribler makes use of the concurrency framework Twisted. Twisted is an event-driven programming framework with a focus on computer networks. As Python is single-threaded, true paralellism is not possible. Asynchronous code, however, is possible and using Twisted allows for convenient coding with callbacks. The extension will be run on this same Twisted-thread, but at this time asynchronous code doesn't seem to be necessary.
To future proof our code, we do use the twisted framework in the back-end of our application. Receiving a request sets up a chain of callbacks that get the data from the database, format the data to a graph readable by the GUI and send a response containing the graph data. This way, the process of building a graph can be interleaved to prevent blocking of the entire back-end.

Bibliography

[1] Tribler. (n.d.). In Wikipedia. Retrieved May 1, 2017, from https://en.wikipedia.org/w/index.php?title=Tribler&oldid=777445841

[2] Tribler. (n.d.). On GitHub. Retrieved May 1, 2017, from https://github.com/Tribler/tribler/graphs/contributors