National University of Ireland, Galway

CS & IT Final Year Project Thesis

PiTorrent

Author:
Sean Heffernan

Supervisor:
Dr. Des Chambers

A thesis submitted in fulfilment of the requirements
for the degree of Computer Science & Inforamtion Technology

April 2014

Declaration of Authorship

I, Sean HEFFERNAN, declare that this thesis titled, 'PiTorrent' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"Anyone who has lost track of time when using a computer knows the propensity to dream, the urge to make dreams come true and the tendency to miss lunch."

Tim Berners-Lee

Acknowledgements

Acknowledgements go here...

Contents

Abbreviations

IDE	**I**ntegrated **D**evelopment **E**nvironment
HTTP	**H**yper**T**ext **T**ransfer **P**rotocol
P2P	**P**eer-**T**o-**P**eer
GPU	**G**raphics **P**rocessing **U**nit
CLI	**C**ommand **L**ine **I**nterface
SSH	**S**ecure **S**hell
RPC	**R**emote **P**rocedure **C**all
SCGI	**S**imple **C**ommon **G**ateway **I**nterface
SPA	**S**ingle **P**age **A**pplication
MVC	**M**odel **V**iew **C**ontroller
AJAX	**A**synchronous **J**avaScript **A**nd **X**ML
DOM	**D**ocument **O**bject **M**odel
JSON	**J**ava**S**cript **O**bject **N**otation
API	**A**pplication **P**rogramming **I**nterface

Chapter 1

Introduction

1.1 Welcome

Welcome to my thesis for my project PiTorrent. PiTorrent is a web-based torrent management system that is aimed to be run on a Raspberry Pi. The system should essentially allow users to easily turn their Raspberry Pi into a file sharing system.

With this project I aim to fill a gap in the market for such a product. A small number of projects that claim to give a similar service already exist, but they are difficult to get working and involve installing and configuring a number of various programs. I will talk more about rival products later, I just want to say now that my project will solve many of the problems with the current paradigm.

So with that said, lets start at the beginning. What are torrents and why should we care about them?

1.2 Torrents and the BitTorrent Protocol

Traditionally, if we download a file from the internet, we open a connection with a web-server and download the file from this web-server. Just two parties are involved in the transaction, us (the client) and the web-server. This model is good and works well, but has some possible shortcomings. For example, say I wish to download the Eclipse Integrated Development Environment (IDE), which is located on a web-server in America. My friend, who I'll call Chris, also wishes

to download Eclipse. We are both located in Galway, Ireland. In order to download Eclipse, we both have to open connections to the web-server in America and transfer the same data across the Atlantic.

Now lets just think about this. When Chris and I are downloading the files from the remote web-server over Hypertext Transfer Protocol (HTTP), we are getting the web-server to send the same data across the Atlantic twice. Internet traffic sent across the Atlantic tends to be slower due to network congestion. We are also putting more strain that is necessary on the web-server, as both Chris and I are downloading the file in its entirety twice.

But what if Chris and I could somehow share our downloads, if we could each download a different half of the file from the web-server in America and share the data we have downloaded amongst ourselves. The advantages of this are clear to see:

- We only need to put half the "strain" on the web-server
- We can get increased download speeds as we are downloading off multiple sources
- We can download from a more "local" source - which can also help download speeds

Because of the advantages of a system like this, the BitTorrent protocol was created.

1.2.1 The BitTorrent Protocol

Bram Cohen, an American computer programmer designed the BitTorrent protocol in 2001. He released his first implementation of it in July 2001 [**?**]. Bram's vision was to enable simple computers, such as home PC's to share file amongst themselves. The protocol works as follows:

- A user who wishes to share data creates a torrent descriptor file
  - This is a .torrent file that defines segments in the data called "chunks"
  - A cryptographic hash is generated for each chunk, and the hashes are stored in this descriptor file
  - These descriptor files are quite small, typically a few kilobytes in size
- The user distributes the descriptor file by any means it wants to other users who wish to download the data

- The user then makes the data it wishes to share available through a BitTorrent node
- Other users can connect to this node, via links in the torrent descriptor, and download the data. They download data in chunks
- Once users start to download chunks, they can share the chunks they have downloaded with other users - reducing the load on the original uploader
- Any chunk downloaded can be verified by generating a hash for the chunk and checking this hash against the hash in the torrent descriptor file

The protocol is implemented by programs called BitTorrent clients. There are multiple clients available to download; uTorrent, rTorrent and Vuze are a few popular options that come to mind.

BitTorrent has become a well established protocol over the years, with many websites that serve large files adding BitTorrent download links.

1.3 The Problems with Running a BitTorrent Server

BitTorrent is a peer-to-peer (P2P) protocol where the overall "health" of a torrent relies on users keeping their torrent client open and uploading the torrent data back to other users. If all users stopped their torrent client after they downloaded the torrent, it would not take long until there were no "seeders" on the network, i.e. users who have the all the data and are uploading it back to the community. Some BitTorrent communities exist where users share files with each other exclusively, and each user is asked to maintain a 1:1 ratio - meaning they upload as much data to their peers as they download from their peers. Failing to keep this ratio will result in the user being removed from the community.

Because of this, serious BitTorrent users usually leave their BitTorrent clients running for extended periods of time - until they upload as much data as they have downloaded. This in itself may prove problematic, as it can be inconvenient to leave your laptop or PC running unattended for hours on end. Therefore enthusiasts usually resort to resurrecting old PC's or buying cheap PC's that they can use solely as BitTorrent servers that they run 24/7.

These servers are usually run headless (without a monitor, keyboard and mouse) and are administered via a network connection. As a result of this, the torrent clients that these servers

use need to be web-based, where users can log into their client over a network connection and monitor their torrents. There has been some advancements in this area in recent years with many good web-based torrent clients coming into existence.

1.4 The Raspberry Pi

The Raspberry Pi is a pocket sized computer that is developed by the non-profit group, the Raspberry Pi Foundation. The Pi was initially released in February 2012 as a cheap computer to promote the teaching of computer science to kids in schools. The Pi was priced at 35 USD and was capable of running a full Linux stack, as well as having a Graphics Processing Unit (GPU) of playing full HD 1080p video. The project generated huge hype and interest due to the "bang-for-buck" of the Pi's specifications. Any self respecting gadget enthusiast put themselves on a waiting list for a Pi.

Due to the Pi's price tag, form factor and low power consumption they make an excellent torrent server. Raspbian, a port of the popular Linux distribution was created especially for the Raspberry Pi. Raspbian comes with over 35,000 packages especially compiled and optimised for the ARM architecture of the Raspberry Pi. As luck would have it, Raspbian comes with some torrent client packages that can easily be installed. However, many of these clients are quite CPU intensive and are reported to make the Pi sluggish. The torrent client that has the lightest footprint, and therefore the best suited for the Pi's limited resources is called rTorrent.

1.5 rTorrent

rTorrent is a text based BitTorrent client that runs over a Command Line Interface (CLI). It is written in C++ and was developed with a focus on high performance and low memory usage. It has become a very popular torrent client due to its stability and efficiency. But it is not without its issues. As rTorrent is a CLI based program, it has a learning curve as users need to learn keyboard macros such as pressing CTRL+s to start a torrent and CTRL+d to stop a torrent. rTorrent also is not web-based, although users can log into a server via SSH (Secure Shell) to administer their torrents over the command line.

In over to overcome these issues, numerous attempts were made by 3rd party individuals to code a web interface to rTorrent by calling functions on rTorrent's exposed Remote Procedure Call

(RPC) interface. Some of these attempts were quite good, with good products like ruTorrent and TorrentFlux being moderately successful.

The problem with these solutions is that they are by no means simple to get working. They rely on a number of different technologies that need to be installed and configured to work together. Technologies like PHP, Apache, and Simple Common Gateway Interface (SCGI) plugins for Apache to enable PHP and rTorrent to communicate with RPC messages. Remember that in many cases, the server on which you are wishing to install these products on are headless - so this installing and configuring needs to be done entirely over the command line, a daunting task to the uninitiated in Linux admin.

So this is where I come in. My idea is create a web interface for rTorrent that will do away with the hardship of setting up such a system. I propose to use UNIX sockets to communicate with rTorrent's RPC interface. With this approach I can remove the necessity to install a web server such as Apache, and configuring it with SCGI and PHP plugins. My system should be a "one click install" product that also incorporates modern development requirements like responsive design - something that rival products lack.

Chapter 2

Technologies Used

2.1 Foreword

One of the intentions from the outset of this project was to embrace new technologies. This does not mean that I jumped on the bandwagon of every up-and-coming hipster product coming straight out of Silicon Valley. I carefully and thoroughly investigated some of the leading "newer" technologies that I felt had genuine potential to become proven mainstream solutions. I discarded some of these technologies, others I chose to use at the core of my project.

2.2 Node.js

My project has two main sections, the front-end (which users will see on their web browser), and the back-end (a web-server on the Raspberry Pi). The back-end acts as the middleman between users and rTorrent. If a user clicks a button to stop a torrent in the front-end, the front-end sends a HTTP message across the network to the back-end, the back-end then builds a XML-RPC message to stop the torrent, and sends the XML-RPC message to rTorrent via a UNIX socket. A rTorrent "stop" function is called, and the response is sent back to the back-end, which in turn builds a HTTP response, and sends it to the front-end over the network. Once the front-end gets the response, it update the view to show the torrent has stopped.

The back-end is the core of my project, and choosing a technology to use as the back-end was the biggest decision I had to make while creating PiTorrent. The technology should be:

- ARM compatible - must run on an ARM CPU
- Lightweight - must not hog the Pi's limited resources
- Easy to install - preferably be in Raspbian's software repositories
- Easy to use - possible to get a prototype up and running quickly

After some consideration I decided to use Node.js as my back-end technology. Node.js is a platform that allows us to use JavaScript as a back-end programming language. It is geared at building fast and scalable net applications. It comes pre-compiled for ARM architecture, so it will run on the Raspberry Pi without any problems. Node's main selling point, and the reason I choose to use it in my project, is it's threading model. Node.js makes use of JavaScript's callback functionality to enable an event driven, non-blocking I/O model.

One of the most difficult issues to overcome in building server-side web application is threading. For example, with Java you would use an application server like WebSphere or GlassFish to handle thread generation and management for you. Even when using these middleware applications, dealing with threads still proves difficult. This is where Node.js I/O model comes into play. With Node, there is only one main execution thread. All I/O calls are non-blocking and when a result is returned, an event is generated and a callback is executed.

Here is an example of some JavaScript that runs under a Node.js environment. Node comes with a 'net' package that allows us to make use of sockets in our code. This example shows all the code required to connect to a UNIX socket (located at '/tmp/rpc.socket') and write and read data from the socket. Notice the layout of the code - when we try to connect to the socket (line 2), we do not wait for the connection to be established, the thread of execution moves onto the next instruction. In this case, that next instruction is line 11 - printing "Coffee time!" to the screen. Only when the connection has been established, a "connect" event is generated and the thread of execution runs the unnamed function which starts on line 3.

```
1  var socket = new net.Socket().connect("/tmp/rpc.socket");
2  socket.on("connect", function() {
3    socket.write("Hello!");
4    socket.on("data", function(data) {
5      // Do something with the incoming data
6    });
7    socket.on("error", function (err) {
```

```
8       console.log(err);
9     });
10  }
11  console.log("Coffee time!");
```

Listing 2.1: Node.js and event driven I/O

With this threading model, we don't have to worry about concurrency issues. This will enable us to build quality applications in a much quicker time-frame compared to other platforms.

Another selling point of Node.js is the fact that it is very lightweight. The current Node binary for ARM is 5MB in size. Compare this to Java 7 and GlassFish which is a whopping 140MB combined.

2.2.1 Express - a Web Application Framework for Node.js

Node.js by itself is quite low-level. It allows us to build web applications from the ground up, exactly the way we want them. However, the majority of Node users do not want to spend their time building authentication systems, or URL route handling systems, or other boilerplate code - they want to be writing business logic. This is where Express, a Node web application framework comes into play. Express takes care of all this boilerplate coding that every web project requires. By using the Express module in our Node applications we can use their URL route handling and other neat features like managing view engines, like Jade.

2.2.2 Jade - a Node.js Template Engine

HTML has been around a long time and can be quite tedious to work with.

2.3 AngularJS

AngularJS is a JavaScript framework which is a brainchild of Google. It aims to provide a mechanism for developing Single Page Applications (SPAs). What is does is essentially bring the Model-View-Controller (MVC) paradigm to front-end JavaScript coding, by separating logic from views.

Angular's killer feature, and what makes Angular's MVC paradigm possible is its two-way data binding. What this means is that we can define JavaScript variables in our front-end JavaScript code, and anytime these variables get updated, the HTML that the user sees gets updated also, with the new information. This concept is not the easiest to get across, so I'll explain with an example.

Say we are displaying a list of currently seeding torrents to the user on a HTML webpage. This information has to be updated from the back-end periodically. Without Angular, we would have to create an Asynchronous JavaScript and XML (AJAX) request to get the data from the server, and once we have the data, build the HTML the user sees with JavaScript, and append this HTML to the a Document Object Model (DOM) node.

This is not ideal, as our JavaScript code now contains view logic (HTML) - making the codebase less maintainable. AngularJS was built to get around this problem, by allowing us to define "models" that we can bind to, and display in the view. So, if were displaying the list of torrents using Angular, we could define the a JavaScript array of torrent objects as a model, and anytime a torrent is added or removed, our view would be updated with the changes automatically.

In the two code snippets below, we see this data binding in action. In our JavaScript code we perform a HTTP GET request on a URL every two seconds. When the result is returned, we bind the result (a JavaScript Object Notation (JSON) object) to "$scope.torrentResults". The "$scope" keyword is what allows us to bind data so that it is available in the view.

```
1  setInterval(function() {
2    $http.get(document.location.origin + '/torrents').success(function(torrents) {
3      $scope.torrentResults = torrents;
4    });
5  }, 2000);
```

Listing 2.2: AngularJS binding data with $scope keyword

Now that we have added the object to the scope, we can refer to the object in the view code. AngularJS achieves its MVC paradigm by allowing us to mix in Angular keywords with our view markup. Here is some Jade that creates an unordered list, and loops through our torrent object, displaying torrents as list items. In Angular, we use the "ng-repeat" directive to loop through data. Here we are also applying a filter that only displays seeding torrents (more on

filters later), and we are also ordering the torrents by their name. We use double brackets: "{{ }}" to display data to the user.

```
1 ul
2  li(ng-repeat="torrent in torrentResults.torrents | filter:seeding | orderBy:'name'")
3      strong {{torrent.name}}
4        p
5          span Downloaded : {{torrent.downloaded}}  Upload Speed: {{torrent.uploadSpeed}}
6        .progress
7          .progress-bar(style='width: {{torrent.percentDone}}%;') {{torrent.percentDone}}%
```

Listing 2.3: Jade with AngularJS

As you can see, there is now a nice separation between view logic and code. Also, as we are rendering our views on the client side, our back-end does not need to serve dynamic HTML pages, just static pages and JSON data. This is hugely beneficial as client side view rendering reduces the load on the Raspberry Pi. AngularJS has some other great features that I will mention later, but now I just wish to give you a taste for it and justify why I decided to incorporate it in my project.

2.4 Bootstrap

One of the biggest issues I have when designing web-applications is making them look good. I am a software developer, not a designer. This lack of flair for design is not the hindrance it once was to developers like me, thanks to Bootstrap. Bootstrap is a HTML framework that supplies developers with pre-built, styled components that they can use on their websites. These components range from buttons, to progress bars, to dropdown menus, etc.

Furthermore, Bootstrap is designed to be "mobile first", meaning that sites created by using Bootstrap will be responsive - working on all devices, regardless of their screen size. This feature is a big selling point for me, as the current rTorrent front-ends are not responsive and do not work well when being viewed on a tablet or mobile device.

Using Bootstrap is a no-brainer, all you have to do is download the Bootstrap CSS and JavaScript files and include them in you web-application. You can then make use of components by using specific Bootstrap HTML class names, e.g. to create a list of styled tabs you would use the following HTML:

```
<ul class="nav nav-tabs">
  <li class="active"><a href="#">Home</a></li>
  <li><a href="#">Profile</a></li>
  <li><a href="#">Messages</a></li>
</ul>
```

Listing 2.4: Bootstrap tabs HTML code

Producing the following:

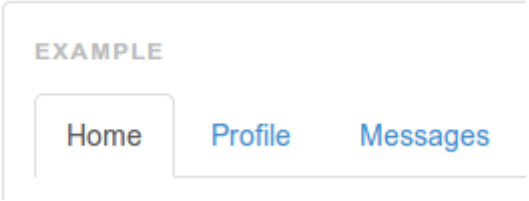

Figure 2.1: Bootstrap tabs

Chapter 3

Design and Implementation

3.1 Proof of concept

As I have outlined in chapter 1, my project should work by utilising UNIX sockets for communication between Node.js and rTorrent. Communication between the back-end and rTorrent traditionally used a web-server with SCGI plugins like so:

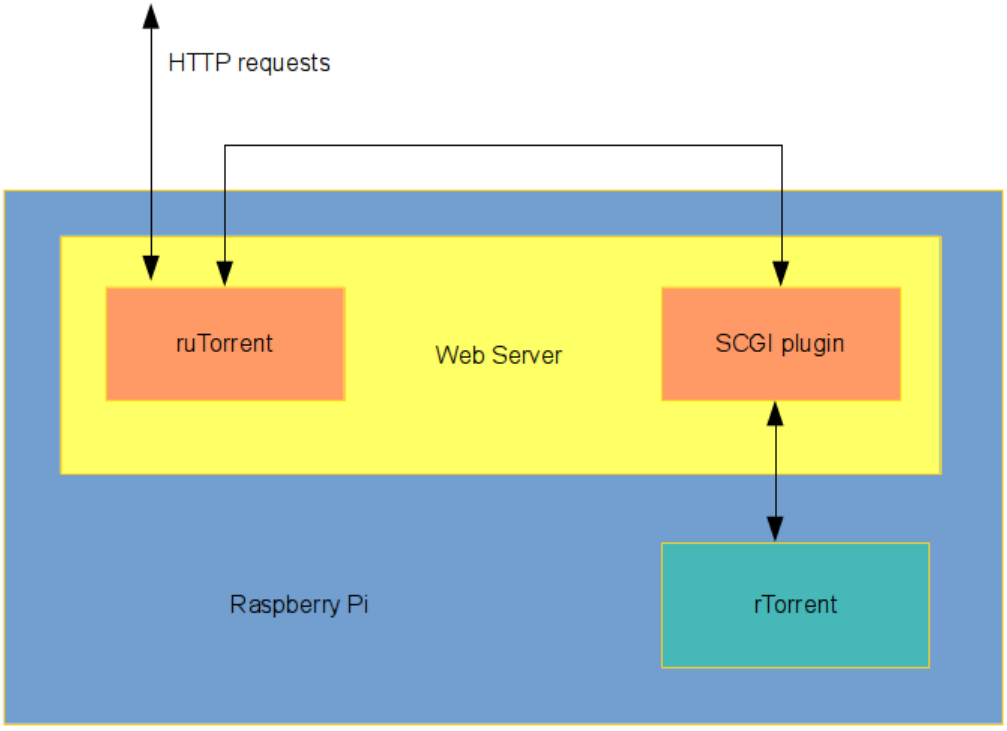

FIGURE 3.1: ruTorrent setup

Users would make HTTP requests to the ruTorrent back-end to get torrent information. The back-end then would then have to route requests to rTorrent via the SCGI web-server plugin. This is not ideal, as all XML-RPC requests between ruTorrent and rTorrent are routed through the SCGI plugin, and this method of communication is open to abuse. It may be possible for malicious individuals to send commands to rTorrent directly, by bypassing ruTorrent on an unsecured setup. An attack is possible as the SCGI interface runs off a port on the web-server, a port which may be accessible to unauthorized users on the network. My project aims to remove the SCGI dependency by spoofing SCGI requests in my Node.js back-end and sending these requests directly to rTorrent's XML-RPC socket like so:

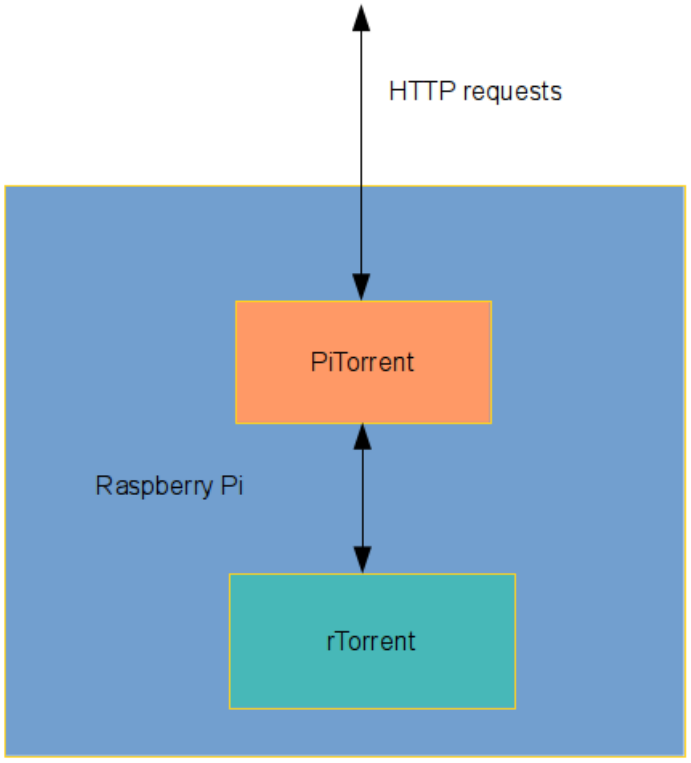

FIGURE 3.2: PiTorrent setup

This has the added benefit of being easier to set up - there is no need to install and configure surplus technologies.

3.1.1 XML-RPC Message Format

In order to talk about spoofing SCGI requests to do away with the SCGI plugin, I will need explain the format of a typical XML-RPC message. XML-RPC is a simple specification with the purpose of calling remote functions over a network. The format is built on top of HTTP.

A XML-RPC message is essentially just a HTTP POST request with a specific XML format as the request body. Here is an example XML-RPC request:

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: localhost
Content-Type: text/xml
Content-length: 181

<?xml version="1.0"?>
<methodCall>
  <methodName>getStateName</methodName>
  <params>
    <param>
      <value><i4>41</i4></value>
    </param>
  </params>
</methodCall>
```

Listing 3.1: A basic XML-RPC request

The header contains typical HTTP header information. The body contains an XML document with specific node names. The values contained in these nodes must specify the name of the remote method we wish to call, and any parameters we wish to pass to the method. Parameter values can be any of the following types:

- <i4> or <int>
- <boolean>
- <string>
- <double>
- <dateTime>
- <base64>
- <struct>

In the example above, we are passing an integer with the value 41 to the method "getStateName".

3.1.2 SCGI Message Format

SCGI is a wrapper around XML-RPC. As incoming remote XML-RPC messages are received, our SCGI plugin will wrap the entire XML-RPC message with a header and send the newly wrapped message to the RPC socket. The layout of a SCGI header can be described with the following thing:

```
1  headers ::= header*
2  header ::= name NUL value NUL
3  name ::= notnull+
4  value ::= notnull*
5  notnull ::= <01> | <02> | <03> | ... | <ff>
6  NUL = <00>
```

Listing 3.2: Layout of a SCGI header

The first header must have the name "CONTENT_LENGTH", with the corresponding value containing the length of the body. A "SCGI" header must also be present, with a value of "1". SCGI requests take the form of [length of SCGI header]":"[SCGI header]","[data]. Here is the SCGI request created by wrapping the XML-RPC request in Figure 3.1.

```
1  26:
2     CONTENT_LENGTH<00>181<00>
3     SCGI<00>1<00>
4  ,
5  <?xml version="1.0"?>
6  <methodCall>
7    <methodName>getStateName</methodName>
8    <params>
9      <param>
10       <value><i4>41</i4></value>
11     </param>
12   </params>
13 </methodCall>
```

Listing 3.3: Layout of a SCGI request

After researching XML-RPC and SCGI, I determined that I could indeed spoof SCGI requests within Node.js by replicating the SCGI protocol, and sending these requests to rTorrent's RPC

interface over a UNIX socket. Here is some Node.js code that I wrote as a proof of concept of creating spoofed SCGI requests:

```
1  function formatRequest(xml) {
2    /* Build the header of the request */
3    var header = "";
4    header += "CONTENT_LENGTH\000";
5    header += xml.length;
6    header += "\000";
7    header += "SCGI\0001\000";
8
9    /* Build the request */
10   var request = "";
11   request += header.length;
12   request += ":";
13   request += header;
14   request += ",";
15   request += xml;
16
17   return request;
18 }
```

Listing 3.4: Layout of a SCGI request

I tried sending some spoofed SCGI requests to rTorrent's RPC socket using Node.js and listening for responses (Node.js code to achieve this contained in Figure 2.1), and low and behold - it worked! I now knew that my project was viable and I could start building it.

3.2 Structuring the Project

Due to the Raspberry Pi's limited resources, I wanted the Node.js back-end to do as little computation as possible. One way to achieve this is by using a web development technique know as "client-side rendering". With client-side rendering, when you request a web-page from a server, the server sends you a static HTML page, along with any CSS or JavaScript code that the HTML page requires. The JavaScript code then makes another request to the server for any dynamic data needed to render the page (in our case - torrent information). Once the JavaScript gets the response, it modifies the original HTML page to include the data it received from the server's Application Programming Interface (API), and renders the page.

With this technique, the Pi does not need to generate dynamic HTML pages - instead it pushes this computation to the client. We also have the added benefit of solving another problem - how to continually send live torrent information to the client. In order to perform client-side rendering, we need to create server API's where live torrent information can be consumed by our JavaScript. These API's can then be polled at set intervals by our JavaScript to get up-to-date torrent information.

Here is the general structure that my project should take:

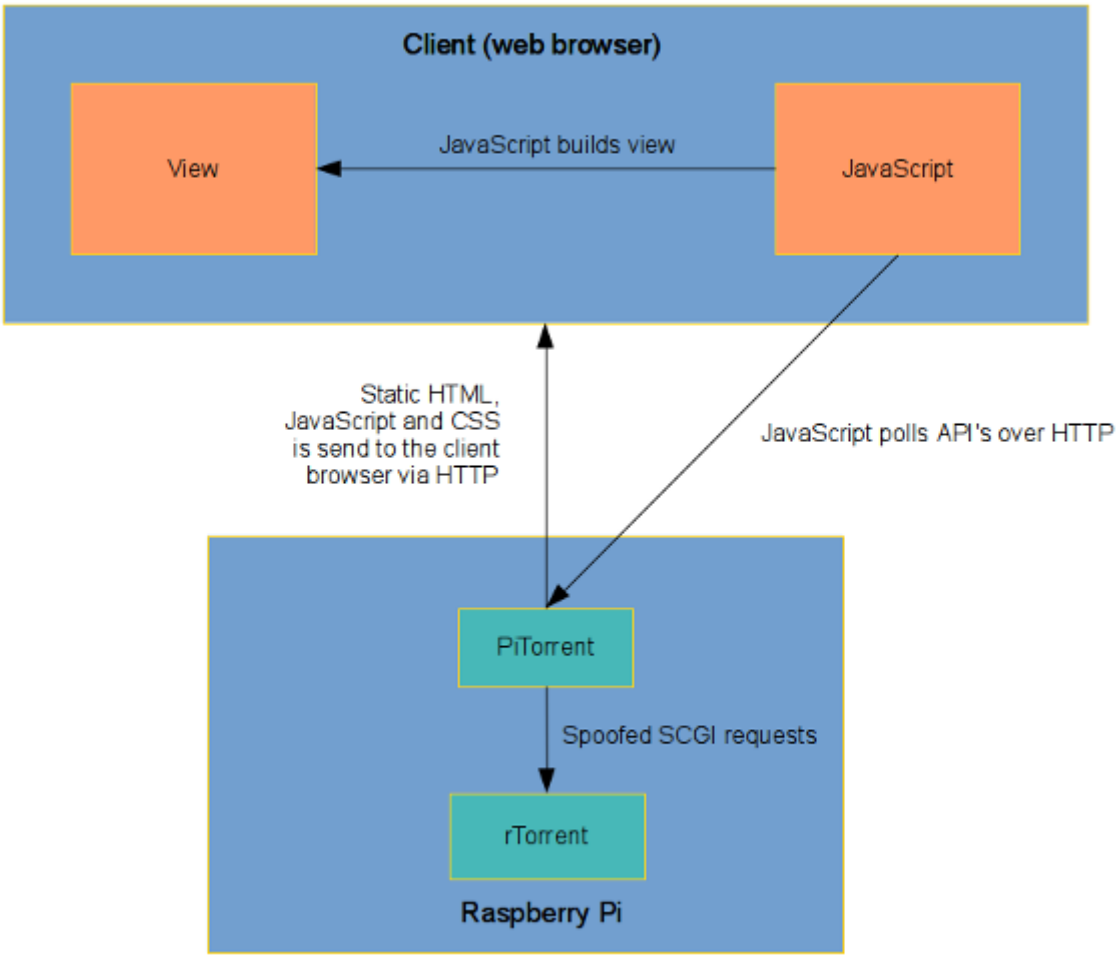

FIGURE 3.3: Project structure

3.3 Back-end development

Our Node.js back-end has two distinct jobs:

1. Serving static HTML pages, JavaScript and CSS files

2. Providing API's that provide up-to-date rTorrent information.

3.3.1 Serving static files with Express

Job number 1 is trivial with Node.js and the Express framework. When I talked about Express in Chapter 2, I mentioned that we could use it to accomplish a lot of commonly used tasks. Serving static files is one of these tasks. We can supply Express with a directory and Express will recursively serve all the files within this directory. Here is all the code required to serve content in the "public" folder on port 80:

```
1  var express = require('express')
2    , http = require('http')
3    , path = require('path')
4    , app = express()
5    , port = 80;
6
7  // Tell our Express app to use the 'static' middleware module and
8  // specify a directory where static content resides
9  app.use(express.static(path.join(__dirname, 'public')));
10
11 // Create a web server on port 80
12 http.createServer(app).listen(port, function() {
13   console.log('Express server listening on port ' + port);
14 });
```

Listing 3.5: Express serving static files

Without Express we would have to manually parse the URL of incoming requests to see what file they wish to download, check if the file exists, and if so, read the file contents into a buffer and pipe the buffer contents to a HTTP response. If the file does not exist, we would have to send a 404 message. Its clear to see why using a framework like Express is a no-brainer.

3.3.2 Creating rTorrent API's

Job number 2 on our list of server tasks is to provide a set of API's where rTorrent information can be accessed from. We should also be able to "control" rTorrent from these API's, e.g. we should be able to stop a torrent by sending a HTTP POST request to a certain API.

These API's will act as the connector between our front-end and rTorrent. If the front-end wants to see all the torrents that rTorrent is currently managing - the front-end JavaScript should send a HTTP request to a specific URL, say "http://localhost/torrents" (if the Node.js server was running locally). Once Node.js gets a request to this URL, Node should call some XML-RPC method on rTorrent's socket to get a list of all the torrents. rTorrent will respond with an XML document, which our Node server should parse, and send a human readable JSON object to the front-end.

I need to convert the rTorrent XML response to JSON as our front-end JavaScript library, AngularJS can only work with JSON objects. I had two options to convert this data - either convert it on the server and send JSON data to the front-end over HTTP, or send XML to the front-end, and let the front-end JavaScript convert it to JSON. I choose to convert it on the server as JSON is more compact that XML, and by having the server send JSON data over the network will reduce bandwidth usage.

Here I will go through all the steps in the front-end getting live torrent data from the server:

Firstly the following front-end code will call a server API ("/torrents"):

```
1  $http.get(document.location.origin + '/torrents').success(function(torrents) {
2    $scope.torrentResults = torrents;
3  });
```

Listing 3.6: AngularJS performing a HTTP GET

When Node.js gets a request for the "/torrents" URL it will create a specific XML-RPC request. This XML-RPC call attempts to get the hash, name and size of all the torrents in the rTorrent "main" view (rTorrent views include main, seeding, leeching, etc.):

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <methodCall>
3    <methodName>d.multicall</methodName>
4    <params>
5      <param>
6        <value>
7          <string>main</string>
8        </value>
9      </param>
10     <param>
```

```
      <value>
        <string>d.get_hash=</string>
      </value>
    </param>
    <param>
      <value>
        <string>d.get_name=</string>
      </value>
    </param>
    <param>
      <value>
        <string>d.get_size_bytes=</string>
      </value>
    </param>
  </params>
</methodCall>
```

Listing 3.7: Node.js generated XML-RPC request

A SCGI header is added to the request, and it is sent to rTorrent via a UNIX socket. A XML response is received. When the SCGI header is removed from the response, we get the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
  <params>
    <param>
      <value>
        <array>
          <data>
            <value>
              <array>
                <data>
                  <value>
                    <string>BB1D8144644F1F2CA9C2CDE60B3FB1B8A904A3C9</string>
                  </value>
                  <value>
                    <string>linuxmint-16-mate-dvd-32bit.iso</string>
                  </value>
                  <value>
                    <i8>1289748480</i8>
```

```
                    </value>
                  </data>
                </array>
              </value>
              <value>
                <array>
                  <data>
                    <value>
                      <string>5741F87F027608169473EE68D2EA7DDBBFA4EA4A</string>
                    </value>
                    <value>
                      <string>debian-7.4.0-i386-CD-1.iso</string>
                    </value>
                    <value>
                      <i8>679477248</i8>
                    </value>
                  </data>
                </array>
              </value>
            </data>
          </array>
        </value>
      </param>
    </params>
</methodResponse>
```

Listing 3.8: XML-RPC response

You can see why I chose to parse this data on the server and not send the raw XML to the client. The XML is very verbose and by converting it to JSON, I can make significant bandwidth savings. Once I have the XML response, I make the conversion to JSON and send the JSON object to the front-end over HTTP. Here is what that HTTP response would contain:

```
{
  "torrents": [
    {
      "hash": "BB1D8144644F1F2CA9C2CDE60B3FB1B8A904A3C9",
      "name": "linuxmint-16-mate-dvd-32bit.iso",
      "size": "1289748480",
    },
    {
```

```
9         "hash": "5741F87F027608169473EE68D2EA7DDBBFA4EA4A",
10        "name": "debian-7.4.0-i386-CD-1.iso",
11        "size": "679477248",
12      }
13    ]
14  }
```

Listing 3.9: XML-RPC response converted to JSON

I created a number of API's that the front-end could call. Here are just some of these API's:

- "/torrents" - list the names and basic information of all torrents
- "/stats" - list some global stats: upload rate, download rate, etc.
- "/files/:hash" - list the files for a specific torrent: "hash" is the torrent identifier
- "/peers/:hash" - list the peers connected to a specific torrent
- "/stop/:hash" - stop a specific torrent
- "/start/:hash" - start a specific torrent

3.4 Front-end development

Once I have API's to retrieve torrent data from, I need to display this data in a meaningful way. The API data is potentially always changing, so these API's need to be polled at regular intervals so users get up-to-date information about their torrents. Since the main purpose of the front-end is to display torrent data to the user, I choose to use the AngularJS JavaScript framework to help me manage the task of keeping the view up-to-date.

AngularJS has some neat features. I have already mentioned it's two-way data binding - which I use extensively in this project. This data binding allows me to poll the server API's continually with JavaScript code, and any changes in the data are automatically shown in the view. By using AngularJS I reduced the time taken to code the front-end by a significant amount. Angular takes care of a lot of the boilerplate code that is needed in updating data.

The front-end contains a lot of dynamic data. Anywhere I want to display this data, I insert AngularJS "{{data bindings}}" in my view markup. I can then use the "$scope" object in my

JavaScript code to associate these bindings with data. Now that I have an idea of how to show the user updated torrent data, I need to decide on the layout of my front-end.

3.4.1 Layout of the web-app

I decided to give the front-end a standard web-app layout:

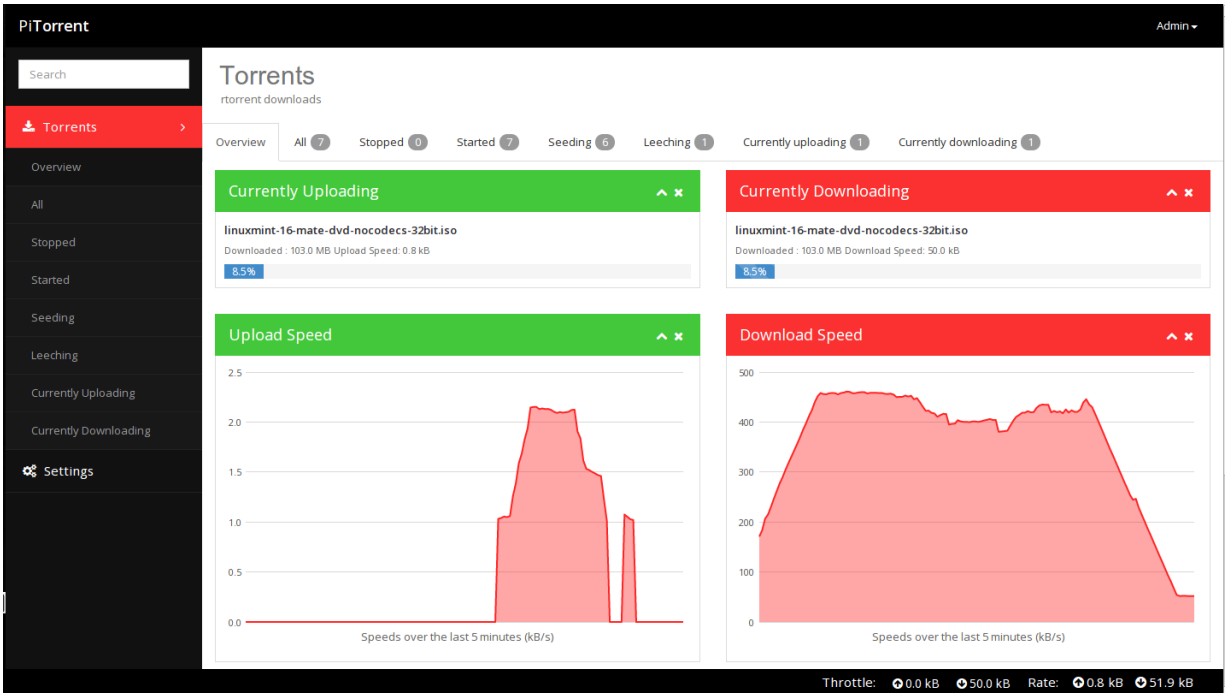

FIGURE 3.4: Front-end layout

There is a navigation panel on the left-hand side so that users can easily find there way around the app at all times. This panel is responsive, and will be hidden by default on tablets and other smaller screen devices.

There are horizontal bars running across both the top and bottom of the app. These bars are fixed to the top and bottom of the screen, so that if you scroll the page up or down, the bars will remain in place. The top bar has a PiTorrent logo on the left-hand side and a dropdown menu on the right-hand side. The dropdown is typical of a dropdown one would expect to see on the top right corner of we a web-app - it contains a link to the settings page, and a logout link.

The bottom bar has torrent speed and torrent throttle indicators, so that we can easily see what our upload and download speeds are at all times. A user can also click on the throttle indicators and be presented with a "popover" window where they can drag a slider to set upload or download throttle speeds. Here is what that popover looks like:

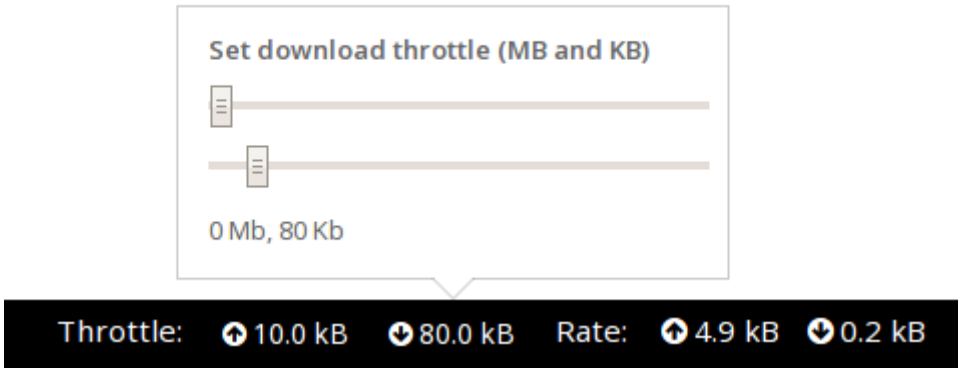

FIGURE 3.5: Download throttle configuration

The main section of the app will display data to the user. I decided to use a "widget" theme with my data, where different data sections are partitioned across widgets. These widgets are then arranged in a grid format. The grid is collapsible, so if you are viewing the site on a smaller screened device the grid columns merge into one.

The default page when you log in is the "torrents overview" page. This page shows lists of currently uploading and download torrents (if any), along with graphs of upload and download speeds over the last five minutes. The overview page also has a "System Stats" and a "Add Torrents" widget.

The System Stats widget gives users information about the Raspberry Pi. Users can see uptime, load averages and memory usage, amongst other things. The Add Torrents widget allows users to add torrents to the system. Users can either upload a .torrent file from their computer, or they can remotely fetch a .torrent file by providing a URL.

Along the top of the main section, there is a set of tabs where users can view torrents in different states. States include: All, Stopped, Started, Seeding, Leeching, Currently Uploading and Currently Downloading. By clicking a tab, users can quickly see all the torrents in that state.

Users can click on a torrent and see detailed information about that torrent. When a user clicks on a torrent AngularJS will intercept the event and change the main section of the view to another view where we can see detailed torrent information. When users are in the "detailed information" view, they have three sections that they can look at: Overview, Peer and Files.

The overview section displays general information about the torrent (upload/download speeds, amount of data uploaded/downloaded, etc.). The Peers section shows a list of all the peers we

are connected to. The list contains information about each of the peers (IP address, the speeds we are uploading/downloading to/from the peer, etc.). Lastly, the Files section shows us a list of all the files in the torrent, and what percentage downloaded each file is. This can be handy for large downloads, where we wish to see if a specific part of the torrent has downloaded. Here is what the overview section looks like, notice the tabs to enter the other sections:

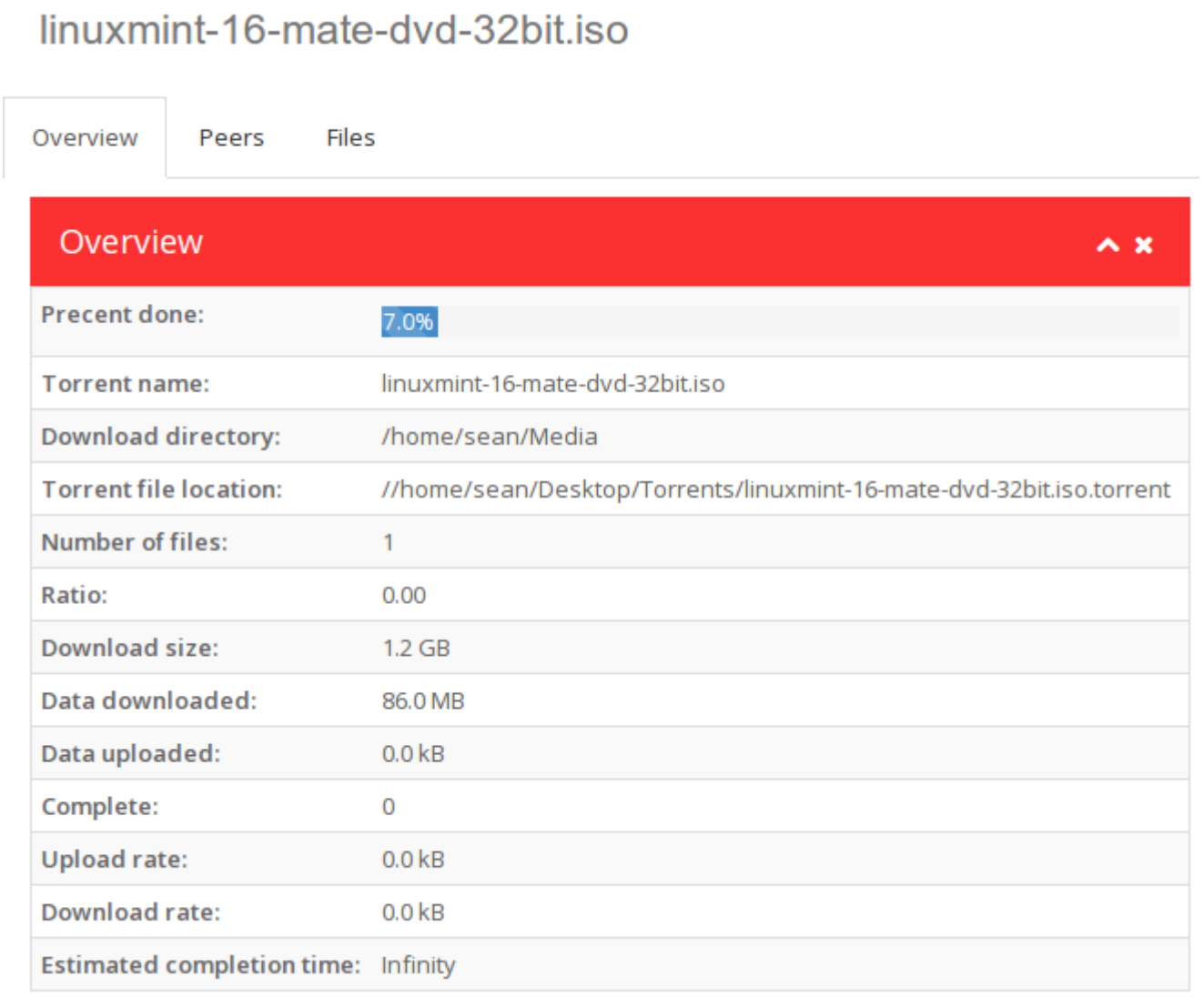

FIGURE 3.6: Overview section of detailed information view

Finally, there is a Settings section in the app. This section allows the user to change the system's settings. When I was coding the back-end, I made sure to include any environment specific parameters in a config file. This config file contains parameters like the location of the rTorrent RPC socket and the directory the back-end should monitor for .torrent files. The

settings section of the app enables the user to change these back-end parameters without editing the config file on the server via SSH or FTP.

Here is a screenshot of the settings section:

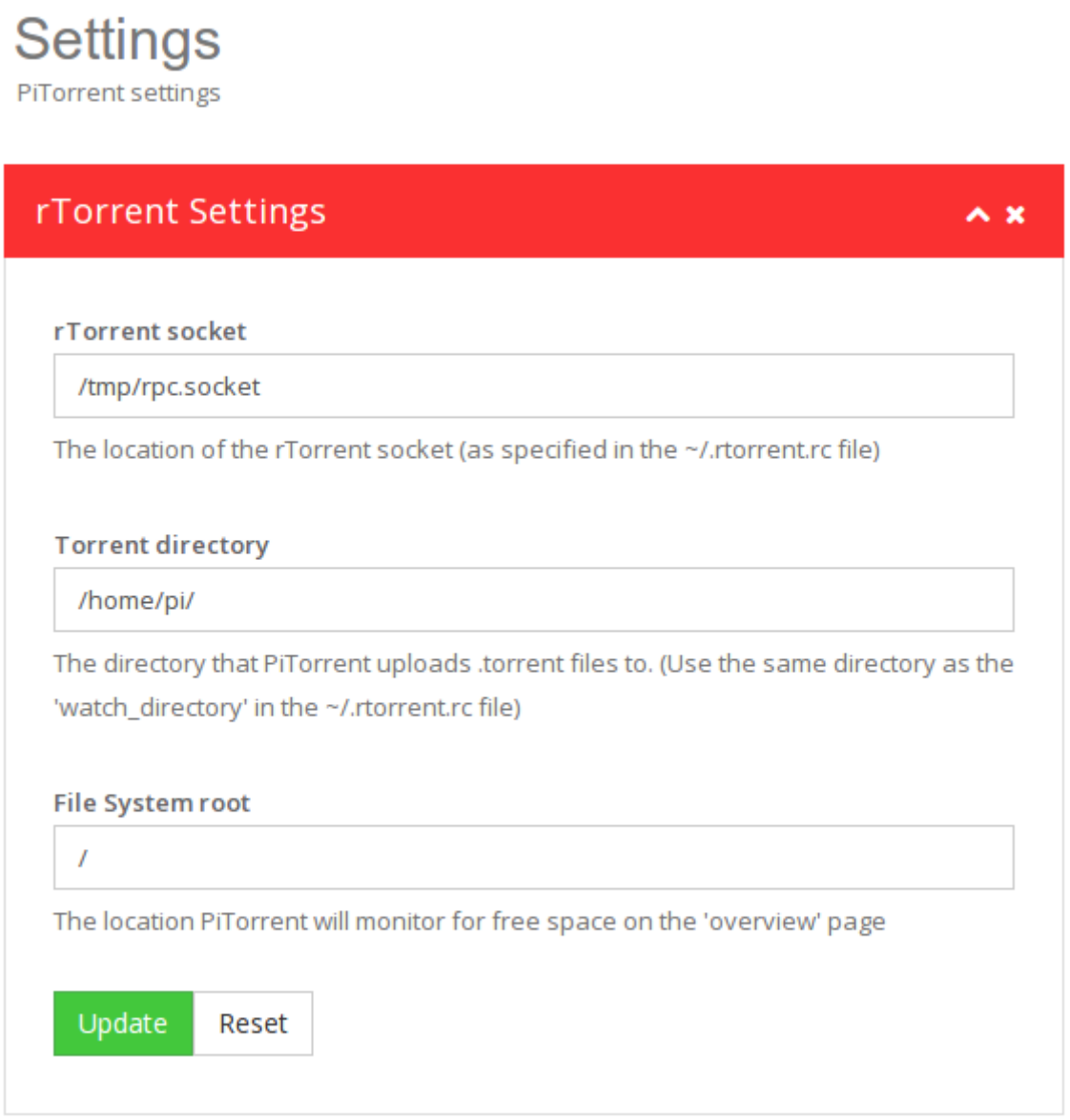

FIGURE 3.7: The settings section

Chapter 4

Results

4.1 How the project went

Appendix A

Appendix Title Here

Write your Appendix content here.