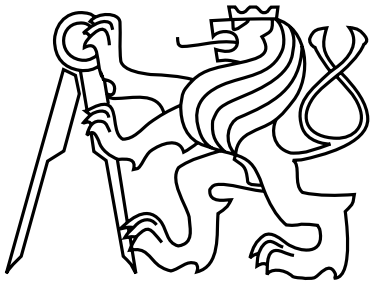
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Bachelor's Project

FatRat Download Manager Extensions

Luboš Doležel

Supervisor: Ing. Jan Žďárek, Ph.D.

Study Programme: Software Engineering and Management

Field of Study: Software Engineering

May 26, 2011

Aknowledgements

First of all, I would like to thank my supervisor for accepting my project and providing valuable advice throughout the writing of this work. Then I thank to thank David Watzke for user-testing of virtually every piece of code I wrote.

I would also like to thank my fiancée Pavlína.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000 Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 26, 2011 .....

Abstract

The goal of this work is to further develop FatRat – a download and upload manager for Linux – by adding segmented downloads support, enabling people to extend the software with easy-to-write extensions and by creating a modern web interface utilizing AJAX technologies. The primary benefit of this work is the enrichment of the range of currently available Linux applications of a similar focus, modernization of an ongoing project and the provision of an all-in-one solution.

Abstrakt

Cílem práce je rozvinout FatRat – manažer pro stahování a upload souborů na Linuxu – o podporu segmentového stahování, umožnit rozšiřování softwaru pomocí rozšíření, která by bylo snadné psát, a přidat moderní webové rozhraní používající AJAX technologie. Hlavním přínosem je obohacení nabídky obdobných programů na Linuxu, modernizace zaběhlého projektu a poskytnutí řešení, které nabízí vše v jednom.

Contents

List of Figures

Chapter 1

Introduction

FatRat is a universal download and upload manager for Linux. It has been under development since 2006. The primary programming language used in the project is C++ and the framework of choice is Nokia Qt 4.

The reason this project was conceived was the lack of a multifunctional download manager on Linux. At that time there were a couple of more or less promising applications such as Downloader for X[1] or KGet[2], but all of them failed to provide a solution that would enable the users to combine multiple download types along with advanced features such as speed limiting or segmented downloads. Also, there were many issues with stability, performance and usability as well.

During the early stages of development, several draft versions – including a GTK+[3] version – were created, but were scrapped in favor of a less generic design that on the contrary allowed for much faster development. For that reason, FatRat 1.0 was a monolithic download manager with no support for plugins and merely a single, graphical frontend. Notwithstanding, this version already boasted queue management, speed limiting, BitTorrent downloads and other features. Later on, support for plugins was integrated and other frontends were included to supplement the GUI[4].

The latest stable version offers, apart from the GUI, a web interface, a Jabber interface, torrent search and much more. The plugins provide subtitle search, easy extraction of archives and integration with certain file sharing services. This version is currently included in many high-profile Linux distributions including Debian, Ubuntu or Fedora Linux.

The work done within the semestral project and this Bachelor thesis is aimed at introducing a couple of modern trends to FatRat, some purely technological, others reflecting the current situation in the means of file distribution.

[1] An HTTP/FTP download manager using GTK+[3], this project has since been discontinued
[2] A simple download manager for the KDE desktop environment
[3] An alternative toolkit for building of graphical user interfaces on Linux
[4] Graphical User Interface

Chapter 2

Problem Description, Goal Specification

2.1 Goal Specification

The official specification is as follows.

- Add an XML-RPC[1] command interface.
- Based on the XML-RPC interface and the AJAX[2] technology, design and implement a web interface. The new interface shall be comparable in terms of functionality with the existing native graphical user interface and it shall provide a configuration means for (headless) server installations.
- Add a segmented download support, including speed limitations and priority downloading.
- Add a connection to Java Native Interface to provide a possibility of cooperation with external Java plugin modules.

Every of the following chapters of this thesis is split into three parts focusing on different categories: the new web interface and all the related work, segmented downloads support and Java extension support.

During the writing of this thesis, further progress has been made and I also started working on various additional ideas to these features. This work is also reflected in the text.

2.2 Existing Implementations

There are a dozen of projects providing similar functionality on Linux:

- Vuze – a Java BitTorrent client, recently reworked as a distribution platform client

[1] Remote Procedure Calls encoded in XML and transported via HTTP.
[2] Asynchronous JavaScript and XML

- KGet – a simple downloader integrated into KDE
- Deluge – a BitTorrent client with multiple user interfaces (including a web interface)
- KTorrent – a BitTorrent client for KDE
- FreeRapid Downloader – a downloader focused on web-based file sharing services
- JDownloader – another downloader focused on web-based file sharing services
- and many others

None of the above, however, provide a combined solution that would cover all of the user's needs at once.

Although one could argue that an all-in-one solution is against the so called "Unix philosophy", when it comes to graphical applications, this philosophy should no longer be applied so strictly.

The reason for this is simple: whereas traditional Unix console-oriented tools are, and are supposed to be easy to combine so as to reach the desired goal, it helps neither efficacy, nor usability to use several graphical applications at once to perform a single task – to download and upload files.

That being said, violating this philosophy does not inherently lead to code duplication. This is thanks to free software libraries that implement lower level tasks and therefore help eliminate the arduous work connected with Internet application development.

Chapter 3

Analysis and Solution Proposal

3.1 AJAX Web Interface

FatRat version 1.1, released in September 2008, introduced a web interface, which enabled its users to control FatRat remotely. This web interface did not use many of the modern JavaScript techniques and a lot of effort was put into ensuring that the interface works even with no JavaScript available on the client side.

The interesting part of this implementation was the server-side scripting that did not use PHP or native code to generate the HTML code sent to the client. The language of choice was JavaScript, or ECMAScript respectively, specifically the QtScript implementation[1]. FatRat had its own engine to process all HTTP requests, decode arguments, handle file uploads, extract the ECMAScript code in order to pass it to the QtScript interpreter and then push the result into the socket.

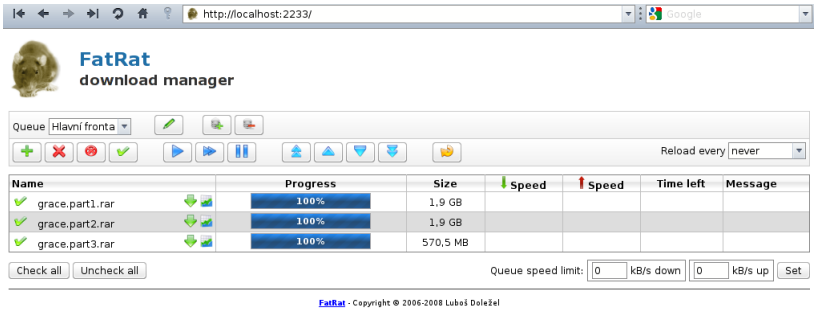


Figure 3.1: The old web interface

Even though the most of web browsers at that time did support AJAX reasonably well, the rationale behind this approach was not to discriminate against web browsers on smartphones, where the AJAX support was lagging behind. Since 2008 the mobile device market has been expanding rapidly and so have the mobile operating systems, which now include web browsers that are up to par with their desktop counterparts.

[1] QtScript is part of the Nokia Qt Framework

The plain old HTML way also involves a lot of page reloading and makes certain features hard, if not impossible, to implement. Owing to that, the question was not whether to write a new web interface or not, but which technologies to use.

3.1.1 Choosing the Framework

While it may seem obvious what technologies to use with AJAX – JavaScript and XML – I still should pick the right framework for the task and there are quite many to choose from. For client-side scripting I have considered jQuery and Ext. Ext would appear to be a clear choice for creating web applications that resemble desktop applications. On the other hand, jQuery and its sibling jQuery UI seemed to be fit for the purpose as well.

In the end I decided to go for jQuery for two reasons:

1. jQuery is very lightweight compared to Ext. Ext took very long to load when connected via EDGE[2] and the like and also put a lot of stress on the browser.
2. I had already used jQuery on multiple occasions and thus it would require only little time for me to become acquainted with jQuery UI.

The downside of jQuery UI I am aware of is the lack of certain user interface components.

3.1.2 HTTP Server Component

FatRat has its own HTTP server implementation based on polling – a single thread serves all the requests. It has worked well for the old web interface, although glitches did appear time from time. I believe it is for the best to remove this code and let a specialized library handle the task. Especially if I want to support HTTPS, as it would involve a lot of rewriting to add SSL support into the existing code.

I have chosen libpion-net, as it is continuously developed, has an elegant C++ interface and provides just the features FatRat needs. It is a Boost-based library that uses ASIO[3] for communication. And FatRat already uses Rasterbar libtorrent, which is also a Boost-based library and uses ASIO as well. That means no additional build or runtime dependencies. ASIO also serves as a "quality guarantee".

Starting up a web server with libpion-net is only a matter of several lines of code.

```
m_server = new pion::net::WebServer(m_port);
m_server->addService("/simple", new SimpleService);
m_server->start();
```

Figure 3.2: libpion-net example code

Implementing a new "service" is straightforward as well.

[2] Enhanced Data Rates for GSM Evolution — data connectivity via cellular networks
[3] Multiplatform asynchronous I/O library

```
class SimpleService : public pion::net::WebService
{
public:
  void operator()(pion::net::HTTPRequestPtr& request,
    pion::net::TCPConnectionPtr& tcp_conn)
  {
    HTTPResponseWriterPtr writer(
      HTTPResponseWriter::create(tcp_conn,
      *request, boost::bind(&TCPConnection::finish, tcp_conn))
    );

    writer->getResponse()
      .setContentType(HTTPTypes::CONTENT_TYPE_TEXT);
    writer->writeNoCopy("Hello world!");
    writer->send();
  }
};
```

Figure 3.3: libpion-net example service

3.1.3 Transporting the Information

If we take a look at modern web applications, we must admit that while the letter *X* in AJAX stands for XML, JSON seems to be used more frequently nonetheless. But neither JSON, nor XML defines a way of transporting commands to the server, and we need to bear in mind that a read-only application would be of little use. I could devise my own protocol, but that would hinder the development of any future applications serving as alternative clients (or frontends).

Two standardized XML-based protocols came to my mind: XML-RPC and its successor SOAP[4]. I decided to use the "lowest standard", i.e. XML-RPC, because it covers all the needs of the application and does not have the complexity of SOAP. Furthermore, I already had experience with writing and debugging XML-RPC-enabled applications.

There is a Qt library called QtSoap that adds SOAP capability to Qt applications, albeit only for the client use. Yet there is no official XML-RPC library for Qt, but that does not pose a problem, since I already have an XML-RPC client implementation ready from my previous work. The protocol itself differs only insignificantly between the client and server side, and therefore it required just a few more lines of code to add support for XML-RPC server use.

Finding an XML-RPC client implementation in JavaScript was a matter of seconds. After trying JS-XMLRPC, I decided to stay with jsxmlrpc as I am more comfortable with its interface. It is not as advanced, but it appears to be easier to extend if needed.

[4]Simple Object Access Protocol

3.1.3.1 Data Flow Compared

Using AJAX for data updating improves the user experience significantly. It removes any "flickering" during the page load (while the page is being rendered) and reduces the amount of transferred data.

This is what the data flow between the server and client looks like:

1. Request the HTML page (with all data such as the list of queues, the list of transfers and so on already in the HTML code).
2. Request all CSS files, images, scripts etc. (caching improves the performance here).
3. Repeat the above steps every n seconds to show new data.

Figure 3.4: Data flow in the old web interface.

1. Request the HTML page.
2. Request all CSS files, images, scripts etc.
3. Request queue and transfer information asynchronously via XML-RPC every n seconds.

Figure 3.5: Data flow in the new web interface.

3.1.4 Other Technologies

Being influenced by the presentations and speeches at the Google Developer Day[5], I reached the conclusion that HTML 5 is good enough for developers to gradually start using it in their applications. HTML 5 comprises a set of new tags and attributes, but the "buzzword" itself is often used in contexts, where it is meant to include the new JavaScript features that were introduced at the same time. These include APIs for local file access, canvas support, movie playback etc.

After checking the support for HTML 5 in various browsers, I found out that it is coming along, but it is not quite there yet. Hence all the code related to HTML 5 would be optional and would enhance the user experience, but the lack of support thereof should not impede usability.

Certain aspects cannot even be implemented without the new JavaScript APIs: e.g. there is no way of pushing local files (such as *.torrent* files[6]) through an XML-RPC call without them. The only conceivable workaround would be to add a special means of uploading files to the server outside the XML-RPC pipeline, but that would be rather dirty.

[5] An annual conference held by Google, where the company presents modern technologies and the newest technologies of their own.

[6] You need to supply a *.torrent* file or a link to one in order to actually *download* files via BitTorrent.

3.1.4.1 Interesting Features in HTML 5

HTML 5 has a lot to offer. It is the largest step forward in bringing web applications up to par with their desktop counterparts since AJAX. Where AJAX introduced the capability of loading data in web applications as needed and opened the window to the new world of web, HTML 5 finishes the job.

I have handpicked several features of HTML 5 from [8] that could be useful in the new web interface:

- Drag & drop – this can be used to drag files from the desktop into the web application. Could be used for file uploads (uploading .torrent files into FatRat), or for dragging transfers between transfer queues (although this functionality can be emulated using various user interface tricks).

- Canvas – the old web interface generates all images (especially transfer speed graphs) on the server side, pushes them to the web browser and there they get displayed. This is not the right way of doing things in modern web applications. One would expect a client application to request source data and render them itself and this is what the Canvas API enables web application developers to do.

  Canvas is often considered *slow*, however this is getting better as new acceleration techniques get introduced in web rendering cores. This approach also decreases the amount of data transferred between the client and the server.

- Media Playback – HTML 5 finally brings *audio* and *video* elements that offer media file playback. This could be used for file previewing, but comes with a catch: only several media formats are supported and this support cannot be extended in many web browsers.

  That is actually useful as is prevents proprietary and otherwise "problematic" media formats from spreading throughout the web, which in the end means higher web application compatibility. Needless to say this also prevents the user from playing all files his computer could play *outside* the web browser. This functionality needs to be reconsidered at a later time.

- File API – in order to push files through XML-RPC, the JavaScript code needs to have access to files on the local computer. That inherently poses a security risk, thus all web browsers keep stringent control over this feature – the user always has to select the file through the native file dialog. That should suffice for the use in FatRat's web interface.

- Webkit Notification Popups – this is a Webkit[7] extension, which enables web applications to show HTML popup windows that stay on top of all windows in the system. FatRat could use this to inform about transfers having been completed or if a captcha code needs to be typed in[8].

[7] Webkit is the rendering and layouting engine used in Google Chrome, Google Chromium, Safari and many other web browsers

[8] Java extensions will need this

This being an unofficial extension means there should always be an alternative codepath for non-Webkit browsers.

– WebSockets – as of now this is a draft, but still a very powerful feature. Up until now if the developer needed to push data to the web application in real time, he had to employ one of several known JavaScript polling[9] techniques, as AJAX does not provide any functionality for push[10]. These are often called Comet[11]. WebSockets bring a means of bi-directional communication to the web.

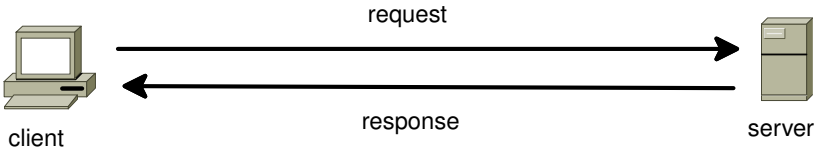


Figure 3.6: Before WebSockets – the client needs to ask for new data

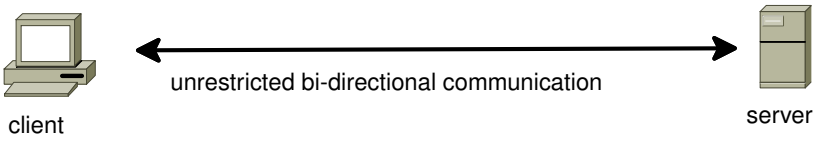


Figure 3.7: With WebSockets the data is pushed to the client as soon as new data is available

WebSockets could be used for notifying the client about new captcha images. WebSockets do not enable JavaScript to communicate with any TCP server. Instead, there is an HTTP overlay protocol being developed, where the client and server switch from HTTP to WebSocket mode after a handshake. As of April 2011, WebSockets are a moving target.

Despite implementations being already available in multiple web browsers, the protocol specification keeps changing in such a disruptive way, it is not possible to write a WebSockets server that would work in all browsers at once and would still work several months later. For instance, in March 2011 both the handshake and the data exchange protocols were reworked completely.

WebSockets may therefore be incorporated into FatRat at a later point in the future. Until then, server-sent events (the JavaScript `EventSource` interface) may be used instead, with data being pushed back to the server through other means.

3.1.5 Remote Configuration Consideration

FatRat uses text configuration files to save its configuration. This file is usually generated by the GUI, but FatRat's shift towards the server calls for alternative methods. The web interface should therefore allow for complete configuration, as the advised workaround –

[9] Polling: the client indefinitely keeps querying the server for new data

[10] Push: the clients register for events at the server, and the server then notifies them of any events being fired

[11] An allusion to AJAX – Ajax is a detergent sold in the United States, and Comet is a detergent as well

configure FatRat elsewhere in the GUI and then copy the configuration file – is anything but straightforward.

Remote configuration is not hard to achieve – several value/array retrieval and store methods in the XML-RPC API will do the trick. There also needs to be a method that would be called to apply all settings. The configuration dialogs themselves are trickier, though.

Currently, the configuration dialogs for the GUI are designed with Qt Designer and are stored in an XML structure (*.ui* files). Then there is the native code that fills them with data, checks the values entered and saves the changes. These XML files are not present in a compiled application – Qt has a generator that generates C++ code from the XML structure. I could write my own generator to generate XHTML dialogs from the XML (could be just a XSLT). Or I could go the other way around, create my own XML structure designed specifically for configuration dialogs, use that to generate XHTML code and maybe later on use that to generate GUI dialogs as well.

Both approaches have their cons and pros. Writing an application or a transformation file that would generate XHTML from *.ui* files would have the advantage of not having to write all configuration dialogs from scratch. The complexity of *.ui* files' structure is a problem, I would have to implement all the possible UI layouts, and many features do not have their direct XHTML counterpart (i.e. spacers). Some dialogs will probably need their specific version shown only in the web interface (e.g. warn the user that the connection will be interrupted if he toggles the HTTPS mode).

On the other hand, devising a specific XML structure for settings files would be quite simple. And more importantly, more extensible since adding something not supported by Qt Designer would not pose a problem. For instance, should I decide to use ECMA-Script/JavaScript for glue code (especially handling button clicks) in the settings dialog, it would require no dirty "hacks".

For the time being, I will write several XHTML settings dialogs for the web interface "by hand", so as to have something the users can use as soon as possible – this is a frequently requested feature. In the future, however, I could analyze existing settings dialogs and design an XML structure fit for this purpose.

3.1.6 Browser Integration

If we think about where users get the links they want to download, we inevitably reach the conclusion that it is the web browser. Therefore it makes sense to make the process of getting the URL from the web browser to FatRat as easy as possible.

FatRat already includes two options: a drop box and a clipboard monitoring tool. The drop box is a widget that hovers on top of all windows in the system. If there is a link dragged onto the widget, a "New transfer" dialog pops up. The clipboard monitor listens for clipboard contents changes, and if the new content is a text matching a regular expression, a dialog pops up enabling the user to add a transfer.

While both of these options are certainly helpful, they are still not so easy to use. For that reason FatRat should directly integrate with the web browser itself.

Various download managers, such as GetRight, integrate with web browsers by installing an NPAPI plugin[12] that "catches" clicks on links based on the file extension (suffix) or MIME type of the file and redirects the request to the download manager. That results in several problems:

1. It catches all specified link types with no exception. E.g. even if the user wanted to download a small file without getting the download manager involved, he would have no choice.
2. Catching only certain certain file types is at the same time very limiting *per se*.
3. After the user clicks on a link, a fullscreen information page is unavoidably displayed. It usually says "This download is now being handled by your Download Manager". The user has to navigate to the previous page in order to continue his work.

The only advantage of this approach is that you have a single plugin that works in all browsers that support NPAPI plugins, which is virtually all modern browsers. Yet I believe that this is not the right way to go because of the bad user experience.

3.1.6.1 Mozilla Firefox

Writing extensions for Mozilla Firefox is not one of the simplest tasks. Thankfully there is already an extension called FlashGot, which has already included support for FatRat. Apart from that it allows for integration with any download manager, provided that the manager accepts new URLs via the command line.

There is no need to write a specific extension for now.

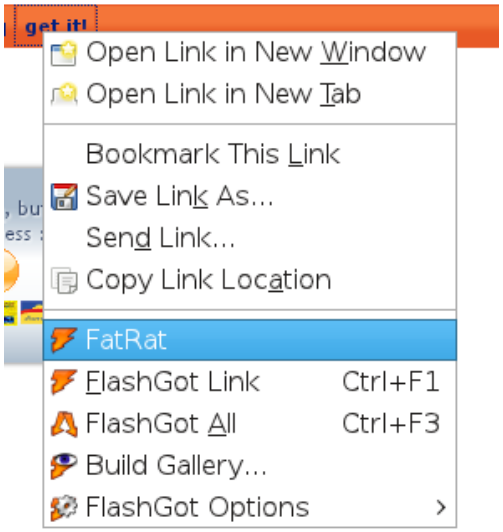


Figure 3.8: FatRat in FlashGot's context menu

[12] Adobe Flash, Java or a media player are good examples of NPAPI (or "Netscape") plugins

3.1.6.2 Google Chrome

Google Chrome is a relatively new browser that took my interest from the very beginning. It has no legacy from decades-old browsers and Google provides a good level of support for extension developers. Its extensions are written in JavaScript and HTML.

After examining the possibilities, the context menu appeared the best place where to add the integration itself, i.e. add a menu item like "Download this with FatRat". The only problem I see is getting the link across to FatRat. For security reasons, Chrome does not allow extensions to interact with the system through any of the official APIs.

The only possible way around this is writing an NPAPI plugin. Not an NPAPI plugin functioning as already described in 3.1.6. Instead, Google Chrome extensions may contain native NPAPI plugins that are loaded by the extension and expose its API to the extension. The NPAPI plugin is therefore never displayed to the user and does not catch any links itself. It merely serves as a gateway to the system.

NPAPI has a significant downside that I will have to entertain: native plugins in general need to be recompiled for every operating system and hardware platform.

For Google, this has simplified the model of trustworthy and possibly dangerous extensions. An extension containing an NPAPI plugin is considered possibly dangerous by default – during the installation, the user is warned that the extension will have full access to the system. Also the Google Chrome Extensions site serving as a central extension directory singles out all such extension for manual checking before they get published.

The NPAPI plugin could implement two ways of passing the link to FatRat:

1. Execute the "fatrat" command – if FatRat is already running, FatRat automatically passes the link to the running instance

2. Use the D-Bus interface – only works if FatRat is already running, but can be faster.

While I was experimenting with Chrome extensions, I got another idea how to make the extension even more appealing. Chrome extensions may examine all loaded pages and interact with them and although there are stringent rules for this, I could integrate not only with FatRat as a desktop application, but the web interface as well.

The extension could detect any open FatRat web interfaces and add another context menu item for every open web interface. The menu item would pass the URL to the web interface, which would open up a dialog exactly like the desktop application. This will make using the web interface much more pleasant and elegant and will seemingly erase another difference between a web and a desktop application.

Looking back at the Mozilla Firefox integration, this idea could eventually justify a development of a FatRat-specific extension for that browser as well. But first I should get in touch with FlashGot developers and consult this idea with them.

Note: Starting with version 4, the Download Assistant extension for Chrome supports Linux and behaves very much like FlashGot. The above consideration was made at a time when there was no download manager integration extension available on Linux.

3.1.6.3 Opera

Starting with version 11, Opera supports extensions just like the other major browsers. I did a quick search through the documentation and failed to find a reasonable and unobtrusive way of integrating with this browser. At this time, context menus cannot be altered in Opera extensions.

There is, however, a way how to integrate manually. By creating a custom menu configuration file and adding

```
Item,"Download Link with FatRat" = Execute program,"fatrat","%l"
```

into the `[Link Popup Menu]` section, you get a certain degree of integration. The problem is that should Opera developers add more menu items into the menu, you will not see them, as by creating a custom menu configuration you fully override the default menu.

Writing an extension for Opera is therefore postponed at least until Opera developers allow for context menu modification.

3.2 Java Extension Support

3.2.1 Why Extensions

Why did I decide to implement extension support, when FatRat already could be extended with native plugins? The first reason is the plugins are hard to write – there is a lot of work involved. Plugins can modify the behavior of the application and add extra features, whereas extensions would only have a limited set of types. While you can add new transfer types with plugins (you can actually take any part of FatRat and break it out into a plugin), you need to handle all state changes, data serialization, add property dialogs etc. Extensions would have a single purpose and only implement a few specific methods.

Another reason is the long development cycle of FatRat. There may be a year-long gap between individual releases and it takes several weeks to get a new release into Linux distributions. This is very bad if you need to support file-sharing servers that do not provide a stable API. In most cases you actually need to parse HTML pages and pretend to be a regular visitor (and not a software robot). This way, many plugins would only work for a couple of weeks after the release.

Extensions can be released every day without much work involved and if available in platform-independent packages, there is no need for the packages to be packaged by distribution maintainers – they can be distributed separately. FatRat could retain its development cycle while keeping everything functional.

Even if I shortened the development cycle and did all the tedious work and released new plugins every week, many distribution maintainers would choose not to update their repositories that often or even bother with including the plugins. And if you take a look at the range of available applications supporting file-sharing servers, all of them have a means of keeping themselves up-to-date outside the distribution's package management system.

3.2.2 Why Java

Java is not the language of choice for application plugins or extensions. The only native application extended with Java extensions that I know of is LibreOffice. And it is because of Java why LibreOffice gets often called "sluggish". Why Java then?

First of all, Java is in no way responsible for the user experience you get with LibreOffice. Try running this office suite without Java support and it will not be any faster, you would only have to do without many features and extensions such as the PDF Import extension.

Java is often hated for its attitude toward memory management. The truth is the default JVM[13] settings are to blame for the better part of the problem; they are set to maximize performance at all costs: minimize garbage collection frequency and keep a lot of unused memory allocated so as to avoid later lengthy allocations. The JVM does not consume much memory if you tell it not to and there is even no reason for that happening in FatRat, considering the relatively little amount of Java classes the Java extensions would load.

[13] Java Virtual Machine — executes the Java code and provides memory management along with other necessities

State	Heap Size	Shared Resources
Before loading libjvm.so	164K	872K
After loading libjvm.so	860K	2968K
After creating a JVM instance	5140K	7420K
After running a simple Java class	5716K	7524K

Table 3.1: JVM memory requirements

I ran a simple test (see table 3.1) of the Java Native Interface[14], which clearly showed that the JVM does not consume more than a couple of megabytes *per se* and loading additional classes does not worsen the outcome significantly. Using Java would therefore not visibly affect FatRat's memory requirements.

Python and Ruby are probably popular choices for application extensions. I ruled out the latter immediately, because I would not like to force end-users to install *another* runtime environment along with *another* wrapper libraries. There is no Ruby installed on any of my computers. Moreover I have never learned Ruby and do not intend to do so.

Python would essentially be a good fit, however, just like in Ruby's case I did not want to limit the application to a single programming language. With Java *as a platform*, you can write code not only in Java, but in Python, Groovy, Scala, Ruby and other languages as well[15]. As a bonus, I get built-in support for application packages (.jar archives in this case), which beside the compiled code can contain other resources, too. That meant I do not have to devise a package scheme or write code for the unpacking or handling of packages.

The last advantage of using Java is the theoretical possibility of supporting plugins from competing applications[16].

3.2.3 Asynchronous Model

FatRat's Java interface is asynchronous, which follows the native API. That means no methods overridden (or implemented) in Java extensions are permitted block, as neither can any methods implementing transfers in native code.

This makes writing the code slightly more difficult but consumes less system resources, since all transfers can be handled in a single thread in lieu of n threads. But there is another significant advantage: asynchronous code is simple to terminate. There is no need for thread killing or any termination cooperation from the extension code.

There are several asynchronous APIs FatRat needs to implement for Java extensions:

- Waiting API – Starts a timer that calls back every second for n seconds (replaces `Thread.sleep`)
- URL fetching API – Replaces standard `java.net.URLConnection`, but automatically handles cookie exchange between the Java and native code and also applies current proxy settings. Additionally, replacing the standard API simplifies logging in this case.

[14] JNI – interface to interconnect Java with native code
[15] See Wikipedia for the complete list of JVM languages.
[16] FreeRapid Downloader and JDownloader in this case

– Captcha API – Accepts a captcha image URL and returns a solution, when and if available

3.2.4 Extension Types

I have specified three extension types, implemented as abstract Java classes. All extension classes would subclass one of these.

– Download extension – the most important type. Processes a URL given by the user, and when all the work[17] is done, passes a download URL back to the application.

– Extraction extension – extracts all URLs from so called "link folders". These are virtual folders containing multiple URLs to be processed by a download extension.

– Upload extension – allows users to upload files to file-sharing sites directly from the application.

3.2.4.1 Upload Extensions

Upload extensions will be the most difficult to do right. Unlike downloads, HTTP POST uploads have no universal way of handling interrupted transfers. Due to this fact, resuming of uploads is hard to implement.

File hosting servers have devised their own methods of upload resuming. The typical solution is to send the file in chunks[18], generate a unique file ID after the first chunk has been uploaded and let the client reuse the ID with every following chunk.

The code needs to reflect the fact that incomplete uploads are usually deleted after a period of inactivity.

The last thing is that FatRat must store the download link[19] (and possibly the kill link[20] as well) for every complete file upload.

3.2.5 Java Native Interface

The Java Native Interface is a simple interface for the C language. It provides low-level access to Java classes, methods, objects and primitives. While the interface itself supports C++ by simplifying a few details, it is still too clumsy and unnecessarily error prone – the programmer needs to remember to release all references, has to manually convert strings or paste in decorated Java method names.

A full-fledged C++ wrapper would therefore aid the development. It involves a lot of work, but it is an investment with a quick return.

JNI functions fall into the following categories:

1. Virtual machine management.

[17] Includes page parsing, captcha solving, countdowns etc.
[18] Not related to HTTP Chunked Encoding
[19] The URL the file owner can spread for others to download the file
[20] The URL the file owner can delete the file

2. Native method registration functions.
3. Functions replacing Java-specific operations (class loading, object instantiation, reference counting[21] etc.).
4. Method and property access functions.

The wrapper should focus on categories three and four. These are the most tedious functions to call manually and especially reference counting is highly error prone. Simple data types (such as integers or floating point numbers) map directly to native C/C++ data types, thus need no special care.

Wrappers around the most popular Java interfaces including `java.util.Map` or `java.util.Array` could be helpful as well.

3.2.6 Dealing with Captcha

Captcha[22] is a simple test to determine whether there is a real person visiting the web site or not. It usually consists of an image with a series of distored letters that are to be retyped and later verified by the server. Applications have no other choice than to either try to "crack" the captcha code or let the user pass the test. Virtually all file sharing sites use captcha to persuade the users to buy a premium service with no captchas.

Displaying a dialog to retype the captcha code is a simple task for desktop applications. FatRat is however not a traditional desktop application, and if the user wishes, it may not be a desktop application at all. It would serve no purpose if the user communicated with FatRat via Jabber[23], added a transfer where there is a captcha to be solved, yet had nowhere to type the captcha if the dialog only got showed on his PC while he is using a Jabber client at a bus stop.

There are therefore several issues that need to be resolved so as to provide a complete solution for captcha input:

- The "there is a captcha to be solved" event needs to be propagated through all user interfaces that are in use at the moment the event gets fired.
- The application needs to handle the situation when nobody inputs a captcha code. The transfer queue must not get permanently blocked because of such a captcha event.
- Consider various captcha cracking techniques to eliminate both of these issues.

I have experimented with captcha cracking. OCR[24] software cannot crack non-preprocessed[25] captcha images. Most of captchas in use on file sharing sites are very difficult to crack and/or

[21] This is not exactly used in Java, but is needed for the garbage collection to work in the native code
[22] Completely Automated Public Turing test to tell Computers and Humans Apart
[23] Jabber (or XMPP) is an instant messaging protocol. FatRat can be controlled via a Jabber chat session.
[24] Optical character recognition
[25] To recognize a word in a captcha, characters needs to be separated, background has to be blanked, character distortion needs to be fixed, and then there is a chance of guessing the letter

preprocess, hence I moved on to audio captcha cracking. I had a much larger success with that and found out that this is the weak spot of many of the targeted sites.

Nevertheless, as this remains a very complex problem, I decided to postpone my work on this and focus on making captcha typing as pleasant as possible.

This first issue is merely a programmatic one – there needs to be an API to register user interfaces and let the user interface code deal with the event. This is how different interfaces need to react:

- GUI: Display a dialog window
- Jabber: Send a link to the captcha image and define a command for the captcha solution input
- Web interface: Push the event to the web interface and display a dialog.

  There is a problem with getting user's attention in this case: a JavaScript overlay window may not be seen if the user is not watching the web interface at the moment. A popup window is likely to get blocked, the user would have to enable the application to use popups. A sound could attract user's attention to the captcha as well. The last solution are the Webkit notifications.

There needs to be a time limit for captcha input – a time limit would solve the second issue. The transfer queue would therefore not get blocked indefinitely, but only for a limited amount of time.

3.2.7 Extension Update Mechanism

Providing a simple way of updating and installing new extensions is crucial. FatRat should have an extension manager like many popular applications (such as Mozilla Firefox or LibreOffice). To improve user experience, the user should be notified of any available updates. This feature, however, should be optional and not unnecessarily annoying – the user must never be forced or feel forced to update.

The same extension manager should be able to offer new extensions for an easy installation.

The extensions need to have a versioning scheme. The update server must know what extension versions are currently available and the client must know what version is currently installed. I will implement this with a Java .MANIFEST file included in the .jar package. The manifest can contain metadata about the .jar file and this way the application would not get confused if the user adds or removes extension files manually.

3.2.7.1 Extension Installation

It is not a trivial task to update or remove extensions at runtime. The first problem is how to deal with extensions that are in use at the time of update or removal. Interrupting active transfers would be a wrong choice, because it may restart non-resumable transfers, i.e. scrap all the data that had been downloaded. On the other hand, it would be confusing to have some transfers using the old version of an extension and other transfers using the new one.

On the programming side of the problem, reloading .jar files in Java would probably require a classloader-per-extension approach. Should an extension be modified, the respective classloader would be destroyed and a new one would be instantiated. This would perplex the class loading code, because the native code would need to remember where from the requested Java code originates. Added to that, the package with core extension classes (defining basic extension interfaces and classes) is something the extensions will depend on. If that package gets updated, the application would also need to handle the situation correctly by reloading *all* extensions.

After considering both sides of the issue, I decided not to implement a mechanism that would update extensions at runtime. Instead of that, the user will be offered an application restart. As FatRat does not take too long to start, this should not be bothersome and resolves the problem without excessive and error-prone code.

3.2.7.2 Extension Compatibility

Extensions may not be compatible between different versions of FatRat, because the API may evolve. API changes may result in extensions not showing properly in the application, failing to work or otherwise not behaving as expected. In this case I should follow the example of many other applications that store extensions in different directories for every version of the main application. That way old extensions will not get in the way after upgrade and should the user decide to downgrade, everything will keep working.

That being said, FatRat should offer the user to download all the extensions found in the previously installed version.

The extensions are not expected to be large in size (several kilobytes per extension). Files from previous versions therefore will not take too much disk space even if not pruned.

3.2.7.3 Extension Download Server

The extension download server will be available at `fatrat.dolezel.info`. The server hosting this domain has plenty of capacity to cover all possible future needs.

The directory hierarchy will be simple:

- `http://fatrat.dolezel.info/update/plugins/`
  - `1.2.0` – version per subdirectory
  - `1.2.1` → `1.2.0` – multiple versions using the same extension files may be implemented using symbolic links
  - `1.3.0`
    - `Index` – index file with version information and other metadata
    - `fatrat-something.jar` – extension files to be downloaded
    - `fatrat-something-else.jar`

All communication shall take place via ordinary HTTP GET requests. The URL for the Index file for version 1.2.3 will therefore be:

```
http://fatrat.dolezel.info/update/plugins/1.2.3/Index
```

The standard update procedure will be to first replace the .jar file and then update the Index file. This way even if a user happens to download a new extension between these two steps, the client will be able to handle the situation without any glitches. If the extension version indicated in the Index file is older than the version currently installed (which is exactly what may happen during this "race condition"), the client simply will not indicate that a new version is available.

The Index file's structure should be simple to edit by hand. Tab-separated values and extension-per-line strategy should be both simple and extensible. I have designed the following structure:

```
Extra-Field: all lines without tabs shall be ignored, unless known
Another-Field: 1.0.0

fatrat-something       20110403.1   Extension␣description
fatrat-something-else  20110408.3   Extension␣description␣2
```

The client will be able to construct a download URL by taking the Index file's URL and replacing `Index` with extension name and a `.jar` suffix like this:

```
http://fatrat.dolezel.info/update/plugins/1.2.3/fatrat-something.jar
```

New metadata fields may be added simply by adding new lines. Should the client not support these extra fields, it should simply ignore them.

Additionally, a change log may be made available in a file available at

```
http://fatrat.dolezel.info/update/plugins/1.2.3/fatrat-something.changelog
```

Unless a demand for alternative extension update servers arises, the update URL can probably be hard-coded in the application.

3.3 Segmented Downloads

Segmented downloads is a feature many download managers on Windows boast. It allows for parallel download of a single file from multiple servers. This functionality is quite controversial as it increases the load on both the Internet connection the download manager is used on and the servers the file is downloaded from.

Downloading from many sources at once also increases the overall TCP overhead. That means if used when not needed, it not only will not improve the download speed, it will slightly decrease it.

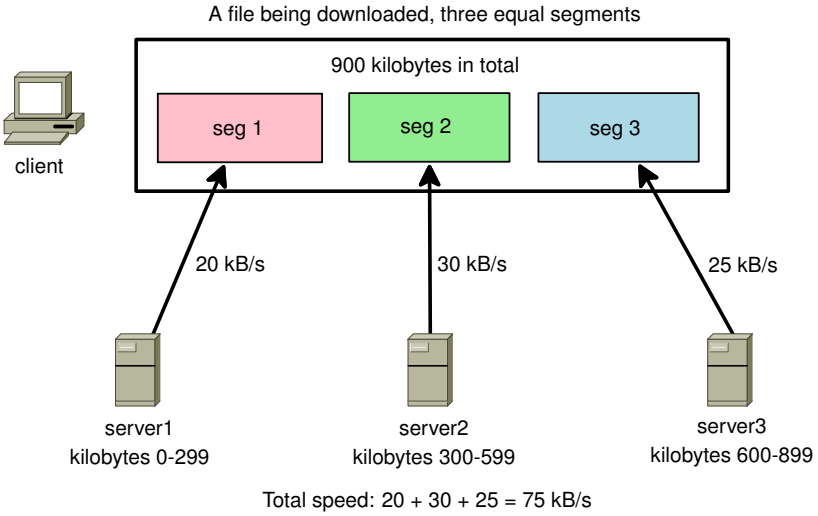


Figure 3.9: An example segmented download

3.3.1 Rationale

This feature was requested by many users from Russia, India and other eastern countries where fast connectivity is scarce. Many public servers in these countries do not possess enough connectivity to serve its users. It is not that useful in west-European countries since the servers the Europeans download from usually have plenty of connectivity to fully saturate the client's line.

An example use is a user with a 1 Gbit/s connectivity that wants to download a file, but all servers only have a 100 Mbit/s connectivity or simply do not have enough free bandwidth. The download manager with segmentation support could connect to 10 servers at once, request different parts of the same file and this way provide a total speed of up to 1 Gbit/s.

Notwithstanding, this feature is interesting from the technical point of view and provides a certain feeling of feature completeness, as I want FatRat to be up to par with proprietary download managers for Windows. FatRat is also exceptionally popular in exactly those countries this feature request originates.

3.3.2 Mirror Search

In most cases the user does not have multiple links for a single file. This is when mirror search – i.e. searching for other servers providing the same file – comes handy.

I have found several sites that index files available on the Internet and could serve for mirror search. The most significant are:

- FindFiles.com – used by GetRight
- FileWatcher.com
- The UK Mirror Service – focused on the United Kingdom

Basically all of them failed to supply a mirror for the most basic Linux-related queries, the only exception being FindFiles.com which was a bit more successful, but still not satisfactory. This means I need to find another solution or help building a working mirror search service.

I started off by capturing the data[26] GetRight sends to FindFiles.com. A few moments later I learned that the HTTP GET request (or actually a mirror submission) and server response look like this:

```
http://www.findfiles.com/find.src?file=$FILE_NAME&size=$FILE_SIZE
  &uri=$KNOWN_MIRROR_URL&getright=1
```

Figure 3.10: FindFiles.com query/submission

```
<a href="$MIRROR_URL">$FILE_NAME</a><br>
<a href="$MIRROR_URL">$FILE_NAME</a><br>
```

Figure 3.11: FindFiles.com response

This solution would require lots of users using FatRat in order to help make FindFiles.com a good service for searching the files Linux users want.

An alternative solution is to build and maintain a database of mirrors for popular download sites. These could include:

- SourceForge.net
- Debian.org
- Ubuntu.com
- FreeBSD.org
- Kernel.org
- GNU.org

FatRat could bundle a list of known mirrors for these sites.

[26] For network packet capture I used Wireshark.

3.3.2.1 Finding the Best Mirror

We cannot determine the speed of a mirror until we start downloading from it. However, we should prefer mirrors that are geographically closest to the client. We can do that by measuring the latency[27] and the number of hops[28].

The latency can be determined by sending an ICMP Echo Request and waiting for an ICMP Echo Reply to arrive. Sending ICMP messages on Linux systems requires root privileges and thus cannot be directly performed by the application. The natural solution is to use the `ping` utility, which has the setuid root bit set[29] and owing to that it can be used by non-privileged users.

The downside of this is we have to parse ping's output, the format of which is not standardized and may vary between implementations. The arguments needed may differ, too. For instance, `ping` on Solaris will require the `-s` parameter in order to output any usable data.

Measuring the number of hops is a bit trickier. We can use `traceroute`, but it may not be available on the user's system and it is likely to take a lot of time to complete. A smart workaround is to guess the distance by looking at the TTL[30] value of incoming ICMP Echo Replies. Also, luckily, it is a common practice to use the TTL value not as a "time" value, but as a "maximum number of hops" value[31]. The TTL value is decreased by one on every router the packet goes through and is part of `ping`'s output.

```
$ ping -nc1 fel.cvut.cz
PING fel.cvut.cz (147.32.192.13) 56(84) bytes of data.
64 bytes from 147.32.192.13: icmp_req=1 ttl=57 time=13.4 ms
```

If we traceroute the target, we can see the number of hops is eight:

```
$ traceroute -n fel.cvut.cz
traceroute to fel.cvut.cz (147.32.192.13), 30 hops max, 60 byte packets
 1  10.10.10.1  1.669 ms  2.672 ms  3.727 ms
 2  88.103.200.45  18.877 ms  20.972 ms  20.960 ms
 3  90.181.196.65  23.807 ms  31.806 ms  33.743 ms
 4  194.228.190.177  37.274 ms  39.320 ms  42.000 ms
 5  91.210.16.191  45.123 ms  47.314 ms  50.232 ms
 6  195.113.144.174  52.444 ms  53.093 ms  54.764 ms
 7  147.32.252.106  56.132 ms  43.122 ms  43.719 ms
 8  147.32.192.13  46.630 ms  45.493 ms  40.762 ms
```

It is clear that the number of hops is $n = X - ttl + 1$, where X is the initial TTL value. Now if we guess the X right, we can determine both latency and the number of hops quickly with a single `ping` command.

[27] The round-trip time of a packet between the client and the server.
[28] The number of routers packets cross between the server and the client.
[29] UID 0 and setuid bit set cause the process to be started with UID 0 (root) privileges.
[30] Time To Live – the remaining "time" until the packet is dropped. Used to avoid network loops.
[31] TTL was actually renamed to "hop limit" in IPv6 to reflect that practice.

Received TTL (ttl)	Initial TTL (X)
≤ 64	64
≤ 128	128
≤ 255	255

Table 3.2: Probable initial TTLs based on the received TTL

I have found a web page (see [7]) with a list of default TTL values on various operating systems. The default TTLs for the systems most widely deployed on web servers are 255, 128 and 64. That leads to the following conclusion:

Even if we guess the initial TTL wrong, it is not a disaster, especially if the distance value is accompanied with the latency information. Lastly, it needs to be noted that certain implementations (such as that of Solaris) do not output the TTL values and therefore cannot serve this purpose.

FatRat should probably contain a wrapper script handling differences between various ping implementations.

3.3.3 Dealing with File Corruption

File corruption is directly proportionate to the number of servers the file is downloaded from. Another aspect increasing the chance of a damaged file is the number of download resumes performed, although this assumption expects there is a bug in the transfer resuming implementation either in the client or the server.

The most intuitive way of mitigating this problem is checksumming. Most users probably will not be wary of this and will not provide a checksum themselves. This problem has already been solved – the Metalink standard defines the structure of XML files containing information about possible download servers and file checksums. Metalink files are already in use by many projects (especially Linux distributions). Metalink therefore solves both problems at once: aids in corruption checking and provides mirror URLs.

```
<?xml version="1.0" encoding="UTF-8"?>
 <metalink xmlns="urn:ietf:params:xml:ns:metalink">
   <published>2011-04-15T14:02:28Z</published>
   <file name="example.zip">
     <size>123456</size>
     <description>File description.</description>
     <hash type="md5">d8e8fca2dc0f896fd7cb4cb0031ba249</hash>
     <url location="de" priority="1">ftp://ftpserver.eu/example.zip</url>
     <url location="fr" priority="1">http://server2.com/example.zip</url>
   </file>
 </metalink>
```

Figure 3.12: Example Metalink4 file

Metalink allows even for combination of totally different download sources. A typical example is HTTP and BitTorrent.

The question is how to handle this in FatRat. HTTP and BitTorrent are handled by a totally different codepath. Thankfully, the BitTorrent code supports HTTP seeds[32]. The user could therefore be offered a choice whether he wants BitTorrent to be the primary download mode with some chunks of data downloaded via HTTP or use merely HTTP with the possibility of manually creating segments and other specific features.

This will not make FatRat 100% compatible with Metalink, but the degree of support should be sufficient in most cases.

Final checksumming could be performed either automatically or manually. The first way would cause the transfer to fail with a descriptive message.

3.3.4 The On-Disk Format

Download managers such as GetRight use special on-disk file formats to store the downloaded data during the process. When the download finishes, the final file is reconstructed. That can be highly time consuming.

For FatRat, I could utilize the so-called sparse files. These are files that have "holes" inside of them, which exactly represents the real situation, where we, for instance, have some data downloaded at the beginning of a file and some in the middle. No reconstruction is needed at the end and non-segmented downloads can be switched to segmented downloads seamlessly. All common file systems on Linux support sparse files. The only exception to this is vfat[33], which is obsolete. Segmented downloads will therefore either not be supported on vfat or the file will have to be pre-allocated to the full size first.

The application needs to remember which parts of the file have been downloaded. While there is a Linux API[34] to detect holes in files, which could be used to retrieve this information, it cannot be considered reliable. Firstly, it is not guaranteed that the underlying file system will *precisely* replicate the data structure without any block aliasing[35]. And secondly, not all file managers replicate the sparse file's structure during file copying. That would result in a complete loss of information about downloaded parts.

This information therefore needs to be stored as part of the transfer's metadata. In FatRat's case, this data is stored in a file called `queues.xml`[36].

3.3.5 New Segment Creation

While the user will be able to create new segments and stop running segments, selecting the right place for a new segment is up to the application.

The first procedure is therefore the search for the best place for a new segment. Apart from what data has already been downloaded and where in the file it is, FatRat also needs to

[32] See [9].
[33] Also known as FAT or FAT32 with long file names.
[34] See FIEMAP ioctl in Linux kernel documentation.
[35] E.g. round the data block to a multiple of 512.
[36] See section B.1.2 for the file hierarchy.

remember where in the file the currently active download "threads"[37] are and what amount of data they are to retrieve. For example, in a 1000-kilobyte file being downloaded in a single thread we may have 0-500 kilobytes of already downloaded data and a thread downloading kilobytes 500-1000 ("allocated space").

1. Find the largest empty space between already downloaded data, consider the EOF[38] as a virtual beginning of another data.
2. If there is no data being currently downloaded before this empty space, allocate the whole empty space for a new download thread – operation is complete.
3. If there is an ongoing download (gradually diminishing this empty space), divide the size of that ongoing download by two. Because there is no way of changing the extent of requested data in HTTP after making the request, the existing download thread has to abort the transfer when the new end of allocated space is hit.
4. Allocate the second half of that empty space to a new download thread – operation is complete.

To put that into perspective, in the previous example this procedure would take the empty space between kilobytes 500 and 1000 and split it into halves. Therefore the existing thread would now only download kilobytes 500-750 and the new thread would download kilobytes 750-1000.

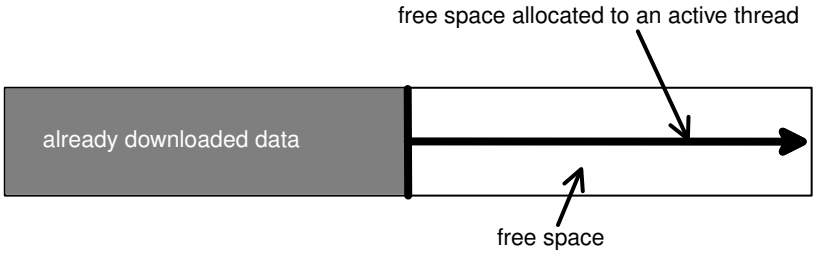


Figure 3.13: Before the example splitting

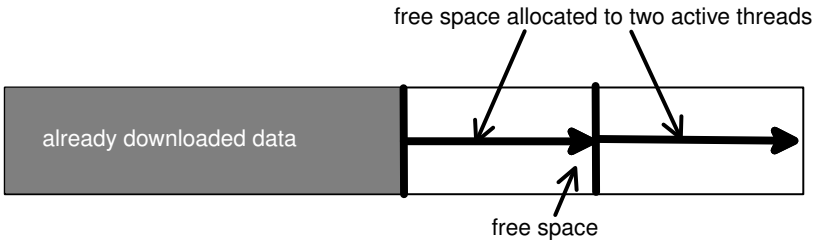


Figure 3.14: After the example splitting

If there were no active threads at that time, the new thread would have been granted the whole 500-1000 range.

[37] Only a expression commonly used to refer to an HTTP connection in segmented downloads. No actual thread is created in FatRat.

[38] End of File

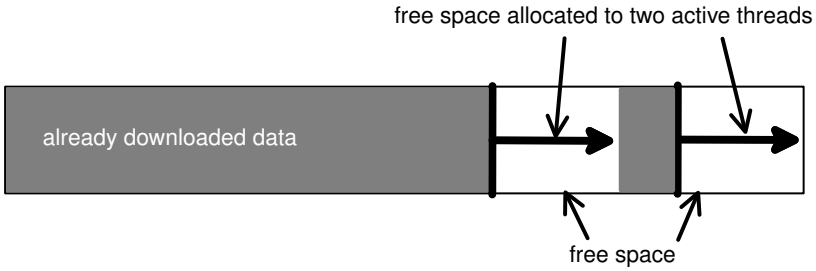


Figure 3.15: The same transfer after a while

When the thread completes its download, the application should start another thread for the same server the previous thread was using and thus maintain the number of active threads. This, however, comes with a caveat. As the whole file transfer nears completion, the free space depletes. This would result into a "fight" for a gradually smaller and smaller available space. That would cause extreme performance degradation. For that reason, we need to add an additional condition to step 1: do not do anything, if the largest empty space found is not at least X bytes long.

It is not easy to find the best value of X. The value should depend on how much data we can download in, say, five seconds. That can however change at any time and an inappropriate automatically detected value would be unpleasant to the user. I will set the value to one megabyte and let the user override it as needed.

Lastly, to avoid abuse of the segmented download functionality, only a single download segment should be used for new transfers. The user will have to add new segments manually.

3.3.5.1 Priority Mode

Priority mode would be a simple feature where the new segment creation algorithm would not spread new segments across the whole file, but instead would try to always create more segments in the first blank space in the file.

This would bring the behavior of segmented downloads closed to traditional "progressive" downloads. First parts of the file would be downloaded first and the last ones last. This may be useful for certain file types (e.g. movies).

3.3.6 Failure Handling

Without segments, failure handling is a trivial task. Should an HTTP or connection error occur, the transfer fails.

With segments, we need to do more thinking. I do not want to make decisions for the user, i.e. start downloading from another URL because the previous URL has failed. There may be a reason why the user picked that specific URL and for example kept the other one in the list of available URLs without the intention of using it.

For that reason, should a download thread fail, it will simple be removed. But if it was the last (or the only) download thread, the whole transfer will be marked as failed.

3.3.7 Speed Limiting

Speed limiting in FatRat is done on two levels. First there is queue speed limiting with a balancer that evenly splits the permitted maximum bandwidth between the transfers and also distributes the unused allocated bandwidth. Any possible balancing on the transfer-level is up to the transfer class[39]. In this case, it is the HTTP/FTP download code that needs to split the allocated bandwidth between the active download threads.

On Linux, we can have the balancing done automatically. The thing is with epoll[40] we can recurse epoll handles by including one epoll handle set in another handle set. Thus by performing speed limiting on the topmost epoll handle, we automatically limit all download threads.

This approach, no matter how elegant it is, will not work on FreeBSD, for instance. Kqueue[41] handle recursion functionality has been removed from FreeBSD in 2004[42], being deemed *utterly pointless* and because it was said to *severely complicate many things*. Note that this inherently breaks Linux emulation on FreeBSD, since handle recursion is a documented feature of epoll.

After analyzing the number of potential FreeBSD users (0.3% of all desktop Unix users[43]), I decided to ignore this platform for now. In order to support FreeBSD, the segmented downloads feature needs to be either removed or the polling code needs to be reimplemented.

[39] A transfer class is a class implementing a specific transfer type: HTTP/FTP, BitTorrent etc.
[40] Epoll is an effective method of event polling of many socket descriptors at once.
[41] Kqueue is a FreeBSD counterpart of epoll.
[42] See http://kerneltrap.org/node/2994.
[43] Based on Google Analytics data from FatRat's website. Confirmed by the data from AbcLinuxu.cz. Does not take into account users of Mac OS X.

Chapter 4

Implementation

4.1 AJAX Web Interface

4.1.1 Replacing the HTTP Server Code

I started off by replacing the initialization code of the previous web interface with code for libpion-net. libpion-net is designed as a service-based library and does not provide a typical disk file serving option. Thankfully, on libpion's web site there is an example service for this – `FileService`.

Being really just an example, it does not support advances features such as transfer resuming (HTTP Byte Ranges), but can still serve as a good basis. The web interface will, however, offer a file download capability – in the sense that from the web interface you can download the files that you have downloaded in FatRat. The aforementioned transfer resuming is therefore something I should later add to that code.

The example service is not part of the library as distributed in Linux distributions. Therefore it had to be added to FatRat's source code tree.

The next step was changing existing service-like functions into real services. These included especially PNG graph generating functions. This proved to be very simple and straightforward. Virtually the only work done was changing old "get a parameter" and "write to output" calls with new ones.

4.1.2 XML-RPC Service

I already had an XML-RPC client code written, which meant the code was able to create call requests and parse server responses. What had to be done was request parsing and response building. First I took a look at an example XML-RPC query [10] and a response (see figures 4.1 and 4.2).

They are not very different. It was apparent that most of the code can be reused, and by that I mean the parameter parsing code, which accomplishes the most complex task.

In order to enable transfer classes to have their own XML-RPC functions, I made the code easily (and dynamically) extensible with new XML-RPC functions. I also prepared a

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

Figure 4.1: An XML-RPC request

```
<?xml version="1.0"?>
<methodResponse>
 <params>
  <param>
   <value><string>South Dakota</string></value>
  </param>
 </params>
</methodResponse>
```

Figure 4.2: An XML-RPC response

generic routine that checks the arguments of an incoming request, since argument checking was a very repetitive task.

During the testing I hit minor glitches in the parameter parsing code. That happened because the client implementation had always been used only for communication with a single server[1], where these bugs have never exhibited.

4.1.3 XML-RPC JavaScript Client

The jsxmlrpc library I decided to use for XML-RPC communication from JavaScript did not require too many changes to work well. One notable exception was the support for transferring binary (Base64-encoded) data via XML-RPC. This I accomplished by adding a JavaScript Base64 library and a few additional lines of code in the library.

The other thing was error handling. XML-RPC-level exceptions should not happen (they would probably be a sign of a bug in the software), but TCP-level errors (server unreachable) should be reported to the user. Showing a traditional JavaScript alert dialog would not be very user friendly. Instead I added a red bar with an error description that shows up if the connection gets interrupted.

[1] OpenSubtitles.org – provides an XML-RPC interface to their services.

4.1.4 UI Layout

UI layouts are not one of jQuery's strong points. In HTML, it tends to be problematic to design layouts automatically expanding (or doing anything else) vertically. The reason for that is HTML (and CSS) was not designed that way. HTML was conceived as a tool for "web pages" and not "web applications" with the content spanning vertically as needed, and not as decided by the developer.

I have experimented with the jQuery UI.Layout plugin, which worked quite well up until the point I started to use other of jQuery's UI widgets – especially tabs. As I did not manage to make it adjust the layout correctly, I removed jQuery UI.Layout and tried to accomplish the task without it.

I ended up making the layout static (no pane resizing) with a bit of a help of JavaScript. The script in this case only resizes the application vertically based on the changed size of client area.

4.1.5 Dynamic Script Loading

As it is entirely up to the transfer class to fill the area of "Transfer details" in the GUI, it is no different in the web interface. Transfer classes may override a method that returns the URL path where the script dealing with this functionality is available.

The web interface must load and start this script dynamically and remove it again after the user has switched to another transfer's detailed view, for instance. It took me by surprise that simply creating a HTML `<script>` tag and then removing it does exactly what I would expect it to. This excerpt loads a script:

```
var sc = document.createElement('script');
sc.type = 'text/javascript';
sc.src = t.detailsScript;
$('#details-subclass').append(sc);
```

And this one unloads it:

```
$('#details-subclass').html('');
```

As these scripts should only operate inside of the `details-subclass` element, this single line cleans everything up.

4.1.6 Image Reloading

Until the Canvas-based speed graph drawing is ready, I use the ordinary way of displaying such graphs – server-generated images. These images should be updated along with all other data in the interface. At first I tried to make the reloading work by setting the URL of the HTML `<img>` element to an empty string and then back to the original URL. Although the server sends HTTP headers indicating that the images should not be cached, it is partly cached by the browser anyway.

I had to force the browser to reload the image. Appending a random string to the URL (which is ignored by the server) proved working. In the end, instead of generating random numbers, I resorted to appending the current Unix time[2].

```
var d = new Date();
$("#tabs-tsg-img").attr('src', '/generate/graph.png?'
    + currentTransfers[0] + '&' + d.getTime());
```

4.1.7 Bandwidth Requirements

The bandwidth requirements of the web interface depend on the refresh interval set in the web interface. I was, however, interested in the worst case scenario, i.e. the bandwidth required for a data reload every two seconds (which could be more or less perceived as real-time updating).

Google Chrome provides a task manager that displays the amount of memory required for every opened web page and beside that also the amount of data transferred last second.

After testing all different views in the web interface, the highest amount of data transferred during a data load was approximately 14 kilobytes. This gives us about 7 kilobytes per second for the highest offered update rate. This sort of bandwidth is currently available even on slow EDGE mobile data networks.

Implementing gzip HTTP transfer encoding in libpion could push the requirements even lower.

4.1.8 Extension for Google Chrome

Writing the extension itself was easy – Chrome extensions are a mixture of HTML, JavaScript and JSON, which is something every web developer is familiar with. The only hurdle was writing an NPAPI plugin. The NPAPI does have documentation, but it is still very difficult to write a plugin from scratch.

I managed to find a few example plugins, but they all focus on developing a plugin with a user interface, whereas I needed to write a plugin with no user interface whatsoever, but with an API available. So first I had to remove all the code related to the UI, then get it to compile and start testing it.

The compilation was a bit tricky – I compile the code against the xulrunner headers and the author's of xulrunner seem to be renaming structure member names on a whim. For instance, I had to put this into the code to make it work on various versions of xulrunner:

```
#ifndef XUL_1_9_2
    link = args[0].value.stringValue.UTF8Characters;
#else
    link = args[0].value.stringValue.utf8characters;
#endif
```

[2] The number of seconds elapsed since Jan 1, 1970.

As you can see, it only checks for xulrunner version 1.9.2, as it remains to be seen what the variable gets renamed to in the future.

The most frustrating thing was that I managed to get the NPAPI plugin working in both Mozilla Firefox and Opera, but not in Google Chrome, which is the browser I was writing the plugin for. After spending several hours debugging the code, randomly swapping lines of code and changing return values, it started to work in Chrome as well. But I have never tracked down, what combination of changes actually fixed the code.

4.1.8.1 Integrating with the Web Interface

Enabling the extension to not only be able to pass links to the locally running FatRat, but to an open web interface as well is interesting. The first task was to find a way of reliably detecting that the page opened in a tab is a FatRat web interface. This actually needs to be done twice: first when the extension is loaded (we need to check all opened tabs) and then add a listener for all navigation events (for example when the user clicks a link or enters a URL).

Making a guess by checking the page title is wrong. Instead, I wrote a "content script" that is executed in every newly loaded web page and unlike the main extension's script has access to the DOM of the page. The content script is very simple – it checks for a `<div>` tag specific to the web interface:

```
elem = document.getElementById('fatrat-chrome-comm-div');
if (elem)
    chrome.extension.sendRequest({status: "fatrat"});
else
    chrome.extension.sendRequest({status: "other"});
```

At the first look, it may not be apparent, why to send any message back if the sought-after `<div>` is not found. If the user leaves the web interface by navigating to another URL, the extension reacts to this message by removing the tab ID from the list of active interfaces.

Then I had to construct a mechanism of passing the URLs to be downloaded to the web interface. I do that by injecting a script into that page. As Chrome blocks injected scripts from interacting with the scripts loaded in the page, I needed to find a way around this. That I achieved by firing a custom event, to which the in-page script listens. This is the injected code:

```
var customEvent = document.createEvent('Event');
customEvent.initEvent('startDownload', true, true);
var el = document.getElementById('fatrat-chrome-comm-div');
el.innerHTML = <the URL>;
el.dispatchEvent(customEvent);
```

And while the page is working on showing the "Add a new transfer" dialog, the extension switches to its tab.

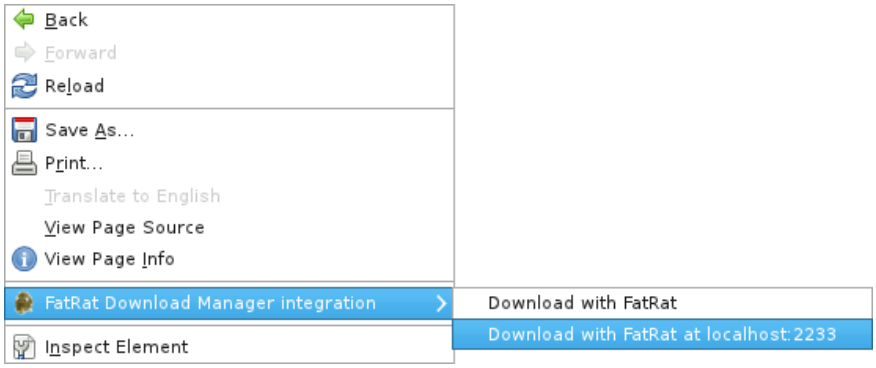


Figure 4.3: The Chrome Extension in Action

4.1.8.2 Extension Packing

As for the extension packing – creating a .crx file that can be distributed and installed – I never managed to automate this process in the Makefile. You need to use the Chrome browser itself to do the packing and the behavior of Chrome depends on whether you already have or do not have Chrome running[3], so the console interface to this functionality is rather useless. I could not find a workaround for this. That is why I need to do the final step of the build process manually.

Because the extension must be packed for every platform separately, there would have to be a manual intervention anyway (unless I had a compiler toolchain for every platform installed). At the moment I distribute the extension for Linux x86, Linux x86-64 and a universal extension, which only supports integration with the web interface, for other systems.

[3] This appears to be a known bug.

4.2 Java Extension Support

The first step was to write a C++ wrapper for the Java Native Interface. It took a lot of time to actually get the wrapper to a state where I could test it for the first time and see if it works at all.

4.2.1 C++ JNI Wrapper

The wrapper is supposed to wrap all `struct`s used in the JNI. During the process of writing these wrappers, several issues arose.

4.2.1.1 JObject vs. JClass

The first class I created was `JObject`, a counterpart of `java.lang.Object`. This wrapper class takes care of object construction, method calling and reference counting.

JNI has three types of object references: global, local and weak. As C++ does not have a way of distinguishing between local and global class instances from the class' point of view, I had to use solely global references throughout the code.

Then there was an important design choice to make. In Java, there is a `java.lang.Class` class used for class interfacing (reflection). This class, just like all other Java classes, is a subclass of `java.lang.Object`. Theoretically, I should follow the same scheme in my JNI wrapper as well. But after a closer look, Java has two ways of accessing classes. If `MyClass` were a class name, then `MyClass.someMethod()` would call a (static) method of that class, whereas `MyClass.class.someMethod()` would call a (non-static) method of a `Class` instance representing that class.

If `JClass` was a C++ subclass of `JObject` functioning as a counterpart of `java.lang.Class`, then one would differentiate between the aforementioned Java calls only by calling a different method – `jclass.call("someMethod")` (a method inherited from `JObject`) and `jclass.callStatic("someMethod")` (a method declared directly in `JClass`). That is too subtle a difference and from my perspective it would be very confusing. Moreover there would be no way to call static methods of `java.lang.Class` – not that there were any useful static methods, `Class.forName()` has a native JNI counterpart, so there is no need to have access to these static methods. It is only another reason why `JClass` should not subclass `JObject`.

What I did was to make `JClass` a standalone class with a member `JObject` instance. This way static class method can be accessed directly (just like in Java) and Java methods of `java.lang.Class` are one call farther, or more specifically, behind that `JObject` instance. This approach makes the `JClass` class more of a `class`-counterpart rather than a `java.lang.Class`-counterpart. But that did not seem to be quite the right way either, as `JClass` seems to be the right place for various class-related helper methods. The final definition therefore is that `JClass` is a generic class for everything Java-class related, but only if you need to access some specific methods of `java.lang.Class`, you call getClassObject() to get directly to that instance.

Java code	C++ code
MyCls.method()	JClass("MyCls").callStatic("method")
MyCls.class.getAnnotation(...)	JClass("MyCls").getAnnotation(...)
MyCls.class.spcMethod()	JClass("MyCls").getClassObject().call("spcMethod")

Table 4.1: Java and C++ wrapper calls compared

4.2.1.2 Strings

Strings are a special type in Java and it is the same in JNI. This is because it would be awkward to construct Java strings by calling Java methods. We would probably have to construct them from byte arrays, i.e. create a Java byte array first and then make the necessary calls. JNI eliminates this process by introducing functions for direct string creation and extraction.

`JString` was therefore another subclass of `JObject`. It was not necessary to wrap any methods of `java.lang.String` for easy access. Java strings are immutable, which means that for any change they need to be re-constructed. Making `JString` a very thin wrapper, the purpose of which would be only to convert a Java string to and from a `QString` (a Qt class), is therefore sufficient.

4.2.1.3 Wrapping jvalue

The JNI type `jvalue` is implemented using a C `union` [6]. It is used during Java method invocation, where it serves as a "transport type" of method arguments.

```
typedef union jvalue {
    jboolean z;
    jbyte    b;
    jchar    c;
    jshort   s;
    jint     i;
    jlong    j;
    jfloat   f;
    jdouble  d;
    jobject  l;
} jvalue;
```

There is no doubt that unions can be used in C++ too, but it is a bit clumsy. I had to a devise a way of automatically converting values and class instances into `jvalue`s. It had to be something that would work invisibly and hide the implementation detail – the `jvalue` type – altogether.

I chose `QVariant` (part of the Qt Framework). This class automatically wraps all simple data types and with a bit of help can wrap all possible class instances. After it wraps all variables from the caller, it is up to the wrapper to retrieve the value from `QVariant` and convert it into a `jvalue`.

So instead of using this:

```
jargs args[3];
args[0].s = 5;
args[1].f = 3.1f;
args[2].j = 7777;
obj.call("something", args);
```

we can write this:

```
obj.call("something", JArgs[4]() << 5 << 3.1f << 7777);
```

Using `JArgs` is still a bit ugly, though. This is where C++ 2011[5] could help, or so-called variadic templates, to be exact. In C++ 2003, if we wanted to replace the previous example with a simpler code such as this:

```
obj.call("something", 5, 3.1f, 7777);
```

we had to define a new template-based overloaded `call()` method for every possible argument count. In C++ 2011 however, we can make this possible with a clean piece of code like this:

```
template<typename T> static void addArg(JArgs& args, T arg)
{
    args << arg;
}

template<typename T, typename... Args> static void addArg(JArgs& args,
    T arg, Args... xargs)
{
    addArg(args, arg);
    addArg(args, xargs...);
}

template<typename... Args> QVariant call(const char* name,
    const JSignature& sig, Args... xargs)
{
    JArgs ja;
    addArg(ja, xargs...);
    return call(name, sig, ja);
}
```

[4] A typedef declared elsewhere: `typedef QList<QVariant> JArgs`

[5] A new standard of C++, previously known as the C++0x draft. It is gradually being implemented in compilers.

4.2.1.4 Java Name Mangling

Java mangles method names (adds a signature) so as to distinguish between methods of the same name, but different arguments (method overloading). As opposed to name mangling in C++, Java mangling is standardized, well-defined and not that arduous to do manually. For instance the following Java method

```
int myMethod(int a1, String s2, float[] f3);
```

would have a mangled name

```
myMethod(ILjava/lang/String;[F)I
```

Despite having mangled many names by hand like this, I still kept making typing errors or type-name errors[6]. Writing method names like this into the source code also makes the code quite unreadable.

That is why I decided that name mangling is an implementation detail that should be kept hidden. I designed the `JSignature` class to be a signature builder. Every method manipulating the signature would return a reference to itself and thus allow method chaining. The argument part of the signature can be now constructed with this piece of code:

```
JSignature().addInt().addString().addFloatA().retInt();
```

This is much more comprehensible, but comes at a price. A manually written mangled name would be stored in a constant string – this, however, will be "computed" at runtime and stored in a dynamically allocated buffer. That is not something that would significantly affect performance, but should be noted.

4.2.1.5 Passing Data Buffers

In order to avoid excessive copying of large memory blocks, Java has introduced the `java.nio.ByteBuffer` class. Byte buffers can wrap buffers allocated and managed in the native code. This is useful for the API for fetching URLs in Java extensions as described in section 3.2.3. I wrote the `JByteBuffer` class to access this functionality.

As the native code is fully responsible for freeing the allocated memory, I used a `std::tr1::shared_ptr<char>` buffer as a member variable to handle that for me. Copying a `JByteBuffer` then does exactly what is expected: it creates another reference both to the `java.nio.ByteBuffer` instance and thanks to `shared_ptr` another "reference" to the data backing the Java object, too.

[6] For instance, you need to remember that the letter `J` constitutes a `long`.

4.2.1.6 Handling Java Exceptions

The native code has to check for an exception after every JNI call it makes. The best way of disposing of such exceptions in the wrapper code is converting them to C++ exceptions and throwing them to the caller. For that I wrote a `JException` class. This class inherits `std::exception`, stores the Java exception object and extracts the message from that exception.

It prepends the message with the full name of the exception's class, since in many cases Java exceptions do not carry a useful message, instead the exception's class is the message *per se*. The complete message can then be retrieved by calling the standard `what()` method.

This wrapping approach makes the C++ and Java integration even more seamless.

4.2.1.7 Locating the Java Runtime Environment

There is absolutely no standard JRE location in the system one could rely on. On many Linux distributions, JREs and JDKs are installed in `/opt`. On Debian, this location is `/usr/lib/jvm`. First of all I analyzed how others have dealt with the problem of locating the JRE. Most actually make a wild guess – they do so by having a list of standard locations and then they try to determine, which version of Java is the best. That I find just plain wrong.

My first idea was to extract the path of `/usr/bin/java`. That appeared to work only until I discovered that this being a symbolic link on my distribution is merely an exception. On many other distributions, this is actually a script that executes the currently set JRE's Java.

Then I came up with something that could work across all distributions and even operating systems, for that matter. I wrote a simple "executable" Java class, the `Main` method of which has a single line:

```
System.out.println(System.getProperties().getProperty("java.home", null));
```

This code prints the location of the Java Runtime Environment used to run that program. In this case, it no longer matters, whether `/usr/bin/java` is a script, a symbolic link or anything else. It is entirely up to the distribution, what JRE of the possibly several is used.

This Java class is accompanied by a helper Bash script that checks the environment for the `JAVA_HOME` variable (which would take precedence over using the detection class) and if a JRE is successfully found, it looks for a file named `libjvm.so`[7] in that installation. The location of this shared library differs between platforms as well; because of that I simply use `find` to locate the file.

To summarize the advantages over the commonly found solutions:

1. It has no hard-coded paths.
2. It works across all distributions and platforms.

[7] When ported to other platforms, it should check for `jvm.dll` or `libjvm.dynlib`.

3. It uses the Java version the user has currently set as default.

`libjvm.so` is the main library for applications that intend creating a virtual machine. It exports the `JNI_CreateJavaVM` function, which really is the sole entry point needed, as all other JNI functions are accessed through function pointers.

4.2.2 Mapping C++ and Java Class Instances

If we have a Java class with native methods and the native methods have some instance data to use (which should be always the case, unless the native method is static), we need to deal with the mapping of a Java class instance to a C++ class instance. From a certain point of view, they are actually the *same* object, but they are allocated separately, as C++ class variables reside somewhere else than Java class variables.

The aim of this mapping is to make the two classes behave as one. To explain this better, all you get from the JNI when Java invokes your native method is a reference to the Java object (`jobject`) and it is up to you to find the right value of the C++ `this` pointer. I did some research on how others have dealt with the problem. The tesseract-android-tools project (developed by Google) adds an extra variable to the Java class which stores the address of `this`. In their case the code uses an `int` for that pointer, hindering compatibility with 64bit platforms.

That apparently was not the right thing to do. Even if I used a 64bit Java `long`, it would be inherently broken.

Instead, I resolved to perform the mapping in the C++ code. For that I wrote a template class called `JSingleCObject`. It maintains a list of all objects of every subclass of itself and works only for subclasses that also inherit `JObject`. Then, upon request, it searches the list for a given `jobject`, while the trick is to use the `isSameObject` function of the JNI to do the comparison with the `jobject` wrapped by `JObject`.

4.2.3 Java Class Hierarchy

Every of the extension class that is supposed to implement a new transfer type for FatRat must subclass one of `ExtractorPlugin`, `DownloadPlugin` or `UploadPlugin` (see figure 4.4).

Also, to provide meta-information about the type of supported URLs etc., the class must have a corresponding annotation, i.e. one of `@ExtractorPluginInfo`, `@DownloadPluginInfo`, `@UploadPluginInfo`. For instance, `@UploadPluginInfo` contains an integer with the maximum permitted length of an uploaded file. `@DownloadPluginInfo` in turn carries a flag, whether downloads can be resumed on this server. Another interesting flag for download classes is a flag telling whether there are multiple simultaneous downloads possible. The native code handles this limitation using a mutex for every such download class.

- `@interface ExtractorPluginInfo`
  - `name` – a descriptive name
  - `regexp` – a regular expression; what URLs this class can process

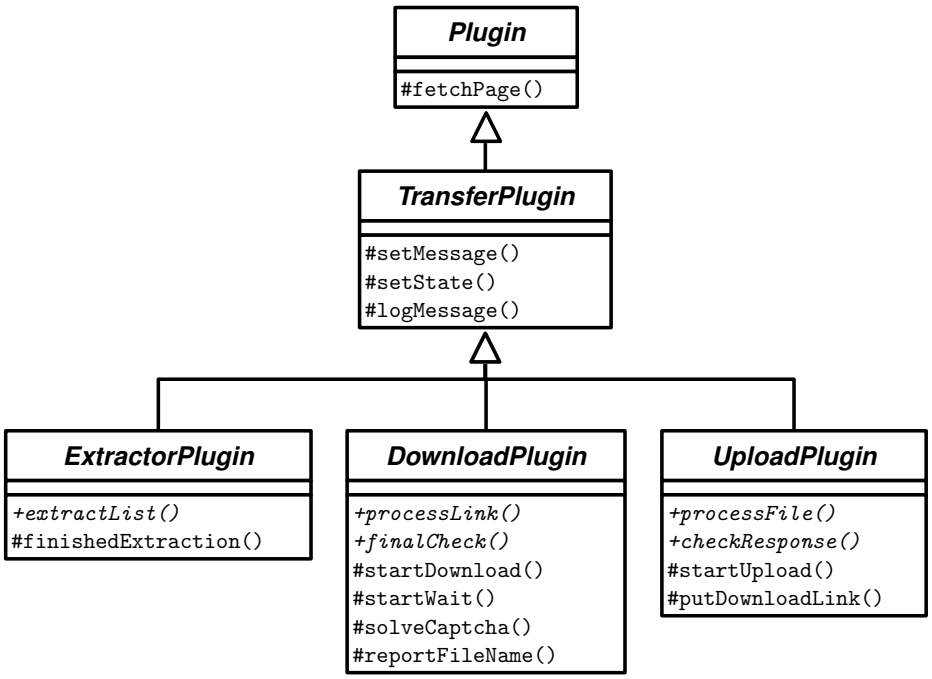


Figure 4.4: Java Class Hierarchy

  - `transferClassName` – hint, as to what transfer type is appropriate for handling the download of extracted URLs
  - `transferClass` – the same, but may refer directly to a Java class

- `@interface DownloadPluginInfo`
  - `name` – a descriptive name
  - `regexp` – a regular expression; what URLs this class can process
  - `truncIncomplete` – whether incomplete transfers should be truncated to zero length (i.e. whether resume is not supported)
  - `forceSingleTransfer` – whether there is only one transfer allowed from one IP address

- `@interface UploadPluginInfo`
  - `name` – a descriptive name
  - `sizeLimit` – maximum allowed length of files uploaded

4.2.4 Writing First Extensions

Because FatRat extensions will usually be very small in size, I decided they all should be part of a single Git repository, also along with the base Java classes seen in figure 4.4. I have written my own Ant build file that ensures that the Java source files are compiled and packed into appropriate files, so the single project has multiple targets (multiple .jar files).

No file sharing servers except for RapidShare.com provide a public API for accessing their services. A good way of emulating the behavior of a web browser and its user is to use Wireshark to capture all network packets, walk through a download session on that file sharing site and then reimplement that procedure in the code. Apart from that, one also needs to take a look at the source code of the pages the browser loads, of course.

Even though the code for different servers may differ a lot, the basic walkthrough for most servers is as follows:

1. Load the first page's URL (given by the user).
2. Virtually click the "free download" (or similar) button and load the next page.
3. Possibly solve the captcha (and load the next page).
4. Possibly wait X seconds.
5. Hand the real download URL back to FatRat.

While writing the first extensions, I have encountered a disadvantage of the fact that all extensions must work asynchronously. The asynchronous code based on anonymous classes[8] tends to get very messy.

Also, as a side note, one of the bugs I have hit (and fixed afterwards) is that the native code needs to take record of all asynchronous operations Java extensions request. Should the transfer be stopped or even removed, all these operations must be cancelled.

One thing I forgot to take into account and should add later is captcha reloading. Some captchas may be illegible and the user should be provided a button of some sort to load another one.

Another interesting remark I have is that certain captchas can be cached. If the application remembers the identification of the captcha that has been previously successfully solved, it may reuse it and avoid asking the user for a captcha solution during consecutive downloads. This perfectly works on Uloz.to.

4.2.5 Application Restart

As the application needs to be restarted for new extension versions to be used, I had to make the restart as simple and fast as possible. Thankfully this is very easy on POSIX systems. System calls in the `exec*` family replace the current process with a new one. Hence if we save the arguments used[9] for the current process and do an `exec` with them, our application restarts.

For this to work reliably, I had to ensure that no code in the application uses `chdir()`. Using this function would result in failure during `execvp()` if the process was started with a relative path to the executable.

[8] Anonymous classes are the simplest substitution for lambda functions (closures) in Java. Lambda functions are not available until Java 7.

[9] New URLs to be downloaded need to be removed from the argument array, as they were offered for download at the time the current process was started.

```
void restartApplication()
{
#ifdef WITH_WEBINTERFACE
    delete HttpService::instance();
#endif

    g_qmgr->exit();
    Queue::stopQueues();
    Queue::saveQueues();

    if (execvp(g_argv[0], g_argv) == -1)
    {
        qDebug() << "execvp() failed: " << strerror(errno);
        abort();
    }
}
```

4.3 Segmented Downloads

The development of this feature had no interesting implementation details. There are just a few areas that are worth noting.

4.3.1 Resume not Supported

To allow segmented downloads, the file size must be known. Hence all transfers without this piece information must have this feature disabled. A special case is when resuming is not supported and this is detected: if this is the only active segment, then the download needs to be restarted. If this is one from n download threads, then the transfer may continue and this thread fails.

4.3.2 Visualization

I use graphical visualization for the currently active threads and previously downloaded data. Previously downloaded data is black, active segments are colored and the blank space is white.

Segment colors are assigned from a set of pre-defined basic colors. If this set is depleted, a random color is assigned. On hover, the speed of the segment and other relevant information is displayed.

4.3.3 File Names

As/if the HTTP client code is being redirected between various URL, the display name (and the on-disk file name) is being changed. This had to be limited, because it is certainly not desirable to go through the name changing process every time a new segment is started.

Because of that, only the first started segment is allowed to change the name like this.

Chapter 5

Testing

FatRat has the advantage of having a fairly large user base with many people watching the Git repository for changes. This is the best and most thorough testing I could get, because other people have different usage patterns and may combine actions in a way the developer would not imagine.

5.1 AJAX Web Interface

Personally, I have found many usability problems when I started using the web interface. Web interface and the XML-RPC interface are hard to test with unit tests, since almost everything is performed asynchronously. For example, during the testing I encountered a bug where transfer types run from Java extensions failed to fetch web pages if the transfer was resumed from the web interface. This bug was easy to reproduce and easy to fix, but developing a test suite that would be able to discover such issues would be very demanding.

What I did test for automatically was race conditions in the web interface's code. One race condition and memory bug were discovered thanks to this, both were in the image generation code (special progress bars for BitTorrent transfers). The race condition was serious, as it resulted in data being randomly written to the wrong socket. The other problem was basically just a typing error – as data is written into the socket asynchronously, we need to consider whether the data buffer will still exist at the time of the writing, and that is what I probably did not pay attention to.

To evaluate the web interface, I think it is something most download managers do not offer. Compared to that of Deluge, it does not resemble a native application *that much*, but still is something to be proud of.

5.2 Java Extension Support

Java extensions need to be regularly checked to see if they are still working. This is because of the frequent changes in the structure of web pages on download severs. Unfortunately, this cannot be done automatically in most cases, unless I had a captcha cracking mechanism

done. From the half dozen of extensions I have written, only the YouTube downloader exception can be tested in this manner.

As for the testing of the extension support code, the most valuable testing I did was starting and stopping transfers after random periods of time. This checked whether the code can handle the stopping of a running extension at any point and can recover from that when the users restarts the transfer.

The writing of many extensions was the most important part of the testing procedure. Different download servers rely on different sorts of data being correctly processed by the client – URL decoding, cookie handling, correct timing etc.

FatRat does not currently support as many download servers as other applications (such as JDownloader). To catch up with them, I need to attract more people to developing extensions for FatRat. Then again, captcha cracking is something that would entice more users and hence more developers.

5.3 Segmented Downloads

The most crucial criterion of segmented downloads is that the downloaded file is not damaged in any way. This is what I was checking for the most part of the testing. I kept adding and removing new segments and I also killed and restarted the application to check if this situation does not result in unexpected bugs.

Certain problems exhibited in the speed limiting code. It worked, but the final speed was different from the speed limit. It is surprising that the code that worked just fine with non-segmented downloads failed to do its job with segmented ones.

I had to walk through the time calculation and see where the flaw is. The problem is you cannot use a debugger, as pausing the program makes it operate divergently. I had to produce a lot of debugging output with information about data received from sockets and the decisions taken by the program.

Compared to other applications with this functionality, FatRat is not that aggressive by default – it does not open multiple connections to a single server unless the user explicitly tells it to. The rest is very much the same.

Chapter 6

Summary

The project has been under development for five years. The summary is therefore split into two sections: the first one is a reflection on the whole history of the project, whereas the other one concentrates on this Bachelor's project.

6.1 The Bachelor's Project

I have completed all the work as described in the assignment of this Bachelor's project. FatRat now has a modern web interface, can be extended with Java extensions and supports segmented downloading. But not all of the *extra* functionality I have discussed in Analysis has been finished to the point when it can be used by ordinary users.

I have learned how to interconnect native and Java applications and familiarized myself with many new web technologies. I now use the new web interface on a daily basis and come up with new ideas for improvements.

The work done during this project will be released as version 1.2.

6.2 The Five-Year Long Project

FatRat is something I started developing because I needed it and at the same time it is something that has always been fun writing. I also must not overlook the fact that it is only thanks to this project that I learned many of the exciting technologies I use.

The process of preparing the application for inclusion in Debian was also very educative, as all applications to be uploaded to Debian's repositories must adhere to strict rules.

What always kept me happy is the fact that many users mail me just to tell me how satisfied with the application they are. And even by looking at Google Analytics data I can tell that this project is not *just some* Czech-made application known only by a couple of friends, but something that has transcended national borders and found its user base in distant countries such as Russia or the USA.

6.3 Future Work

First of all I would like to finish some of the features that were not a mandatory part of the Bachelor's project, but I started working on them nonetheless. These include:

- Upload extensions including upload resuming.
- More features in the web interface.
- Use more HTML 5 features.

Other than that, there are many other things on my TODO list:

- Android client for remote control.
- Graphical (S)FTP client (for walking the remote directory tree).
- Extended file attributes storing (metadata).
- Even more powerful Jabber remote control.
- Captcha cracking.
- Extend the D-Bus interface.

Bibliography

[1] BRYAN A., TSUJIKAWA T., MCNAB N. The Metalink Download Description Format. `http://tools.ietf.org/html/draft-bryan-metalink-28`, as of August 16, 2010.

[2] FELDMAN, B. F. kqueue giant-locking. `http://kerneltrap.org/node/2994`, as of April 17, 2004.

[3] Google. Google Chrome Extensions – Developer's Guide. `http://code.google.com/chrome/extensions/devguide.html`, as of April 18, 2011.

[4] Mozilla Corp. NPAPI Documentation. `https://developer.mozilla.org/en/Plugins`, as of April 16, 2011.

[5] Oracle. Java 2 Platform Standard Edition 5.0 – API Specification, . `http://download.oracle.com/javase/1.5.0/docs/api/`, as of March 19, 2011.

[6] Oracle. Java Native Interface Specification, . `http://download.oracle.com/javase/1.5.0/docs/guide/jni/`, as of April 30, 2011.

[7] SEBASTIAN, A. Default Time To Live (TTL) values. `http://www.binbert.com/blog/2009/12/default-time-to-live-ttl-values/`, as of December 8, 2009.

[8] Wikipedia contributors. HTML5, . `http://en.wikipedia.org/wiki/HTML5`, as of May 2, 2011.

[9] Wikipedia contributors. BitTorrent protocol, Web Seeding, . `http://en.wikipedia.org/wiki/BitTorrent_(protocol)#Web_seeding`, as of April 30, 2011.

[10] WINER, D. XML-RPC Specification. `http://www.xmlrpc.com/spec`, as of April 21, 2011.

Appendix A

Abberviations

AJAX Asynchronous JavaScript And XML

API Application Programming Interface

CSS Cascading Style Sheets

EOF End of File

FTP File Transfer Protocol

GTK+ The GIMP Toolkit

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

ICMP Internet Control Message Protocol

JNI Java Native Interface

JVM Java Virtual Machine

NPAPI Netscape Plugin Application Programming Interface

OCR Optical Character Recognition

PHP *originally* Personal Homepage

POSIX Portable Operating System Interface

SOAP Simple Object Access Protocol

SSL Secure Sockets Layer

TCP Transmission Control Protocol

TTL Time To Live

URL Uniform Resource Location

XHTML eXtensible HyperText Markup Language

XML Extensible Markup Language

XML-RPC Extensible Markup Language - Remote Procedure Call

XSLT eXtensible Stylesheet Language Transformations

Appendix B

Installation and User Manual

B.1 Installation Instructions

FatRat uses the CMake build system, just like KDE 4.

An example compilation procedure:

```
cmake . -DWITH_BITTORRENT=ON -DWITH_SFTP=ON -DCMAKE_INSTALL_PREFIX=/usr
make
make install
```

The prefix can be changed to `/usr` using `-DCMAKE_INSTALL_PREFIX=/usr`. The default prefix is `/usr/local`, but may be implementation dependent.

Should you need to `make install` to a different directory, the traditional `DESTDIR` variable is supported.

B.1.1 Features

CMake may have issues finding your JDK installation. Should that happen, try exporting variables such as `JAVA_INCLUDE_PATH2`, `JAVA_INCLUDE_PATH` and `JAVA_JVM_LIB_PATH`.

Feature name	**Switch**	**Dependencies**
Translations	`WITH_NLS`	—
HTTP(S)/FTP(S) support	`WITH_CURL`	libcurl 7.18.2, Linux 2.6.2 or newer
BitTorrent support	`WITH_BITTORRENT`	libtorrent 0.15.0, QtWebKit
Jabber remote control	`WITH_JABBER`	gloox 0.9/1.0
End-user manual	`WITH_DOCUMENTATION`	QtHelp
Web interface	`WITH_WEBINTERFACE`	libpion-net
Java extensions	`WITH_JPLUGINS`	JRE 1.6 (JDK for compilation)
All of above	`WITH_EVERYTHING`	

Table B.1: Feature switch table

B.1.2 File Hierarchy

All shared data is stored in *prefix*`/share/fatrat`:

- `lang/` – Localization files
- `doc/` – Optional end-user documentaton
- `data/` – Other data
  - `remote/` – Data for the web interface
  - `java/` – Data for Java extensions
  - `btsearch/` – Parse information for the Torrent Search
  - `btlinks.txt` – Regular expressions that help identify .torrent links
  - `mirrors.txt` – Mirror lists for segmented downloads mirror search
  - `defaults.conf` – Default confguration options
  - `genssl.cnf` – OpenSSL configuration file for one-click certificate generation

User data is stored in ~`/.local/share/fatrat`, except for the configuration file located in ~`/.config/Dolezel/fatrat.conf`.

- `data/` – All files stored in this directory have precedence over files stored in prefix/share/fatrat/data Notably all installed Java extensions are stored here.
- `torrents/` – .torrent file storage, needs to be purged time from time
- `queues.xml` – Information about all queues and transfers in FatRat

FatRat installs a single binary in *prefix*`/bin/fatrat` and creates a symlink named `fatrat-nogui`, which is really just an alias for `fatrat -nogui`.

B.2 User Manual

The user manual is optionally installed along with the application. It is also available online at http://fatrat.dolezel.info/documentation.

Appendix C

CD Contents

The disc contains the current source code of FatRat and of the Java extensions code, along with this document.

- `src/` – Source codes (current Git snapshots)
  - `fatrat.tar.bz2` – Source code of FatRat
  - `fatrat-jplugins.tar.bz2` – Source code of Java extensions for FatRat
  - `fatrat-chrome.tar.bz2` – FatRat extension for Google Chrome
- `thesis/` – The source files of this document
  - `dia/` – Diagrams (made with Dia)
  - `figures/` – Images (in PNG or EPS)
  - `dolezel-thesis-2011.pdf` – The print version of this document
  - `dolezel-thesis-2011.tex` – The LaTeX source file
  - `reference.bib` – References
  - Other auxiliary files