

# Implementing a Web Server

Daniel G. Taylor

## Table of Contents

<b>Introduction</b> .....	3
<b>Overview of HTTP 1.1</b> .....	3
<b>Existing Implementations</b> .....	4
<i>Apache</i> .....	4
<i>Lighttpd</i> .....	5
<i>Jigsaw</i> .....	6
<b>Overview of Oriven</b> .....	6
<i>History</i> .....	6
<i>Concept</i> .....	7
<i>Architecture</i> .....	7
<i>Data Flow</i> .....	8
<b>Implementation</b> .....	9
<i>Configuration System</i> .....	9
<i>Dynamic Module System</i> .....	10
<i>Sockets and Threading</i> .....	12
<i>Module Implementations</i> .....	14
<b>Integrating Python into an Oriven Module</b> .....	14
<i>Overview of Python</i> .....	14
<i>Basic Concepts</i> .....	15
<i>Results</i> .....	15
<b>Conclusion</b> .....	16
<b>Further Work</b> .....	16
<b>Bibliography</b> .....	17

## Introduction

In the last decade the World Wide Web has become one of the most prevalent things in our modern society. The amount of traffic traveling over the web every day is staggering. Each and every one of us probably use the web daily for communication, research, and fun, but the vast majority have no idea how it all works. This paper will describe one of the most important aspects of the web, the design and implementation of a web server, a piece of software that accepts connections from clients and sends them documents. That is, after all, the basic idea of the web.

I will begin with a quick refresher on the protocols used and existing implementations, and then delve into our design and implementation, finishing with some conclusions about the project as a whole and what we learned.

## Overview of HTTP 1.1

Before getting into web servers and their implementations I want to take a minute to review the Hypertext Transfer Protocol version 1.1. As defined by RFC 2616, “The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990.” [1]

A typical HTTP 1.1 connection will normally involve a client requesting a document from a server. To do this the client creates a TCP connection to the server and sends a command (usually GET), arguments (the document to retrieve), and the HTTP version that is being used along with several headers. The headers contain information about the client, such as its name and version, features, etc. The server completes the request and sends back a response containing headers and the document's data or an error code. This connection may stay live for several requests and is typically to port 80 on the server. [2]

The actual HTTP 1.1 protocol is rather complicated and covered thoroughly in many other places, specifically in the RFC itself, so for further information please refer to either the RFC or other documents dealing specifically with the protocol. [1] [2]

## Existing Implementations

There are several free software implementations of the HTTP 1.1 specification. Before attempting to write our own I wanted to get a good overview of some of the existing web servers, their strengths, and if possible any implementation details or documentation so that it would give us a general idea of how others have handled the problems we were going to face. The biggest is of course Apache, followed by Lighttpd. Jigsaw is thrown in as it is something of a reference implementation by the World Wide Web Consortium (W3C).

### *Apache*

“The Apache HTTP Server Project is an effort to develop and maintain an open-source HTTP server for modern operating systems including UNIX and Windows NT. The goal of this project is to provide a secure, efficient and extensible server that provides HTTP services in sync with the current HTTP standards.” [x]



“Apache has been the most popular web server on the Internet since April 1996. The November 2005 Netcraft Web Server Survey [x] found that more than 70% of the web sites on the Internet are using Apache, thus making it more widely used than all other web servers combined.” [0]

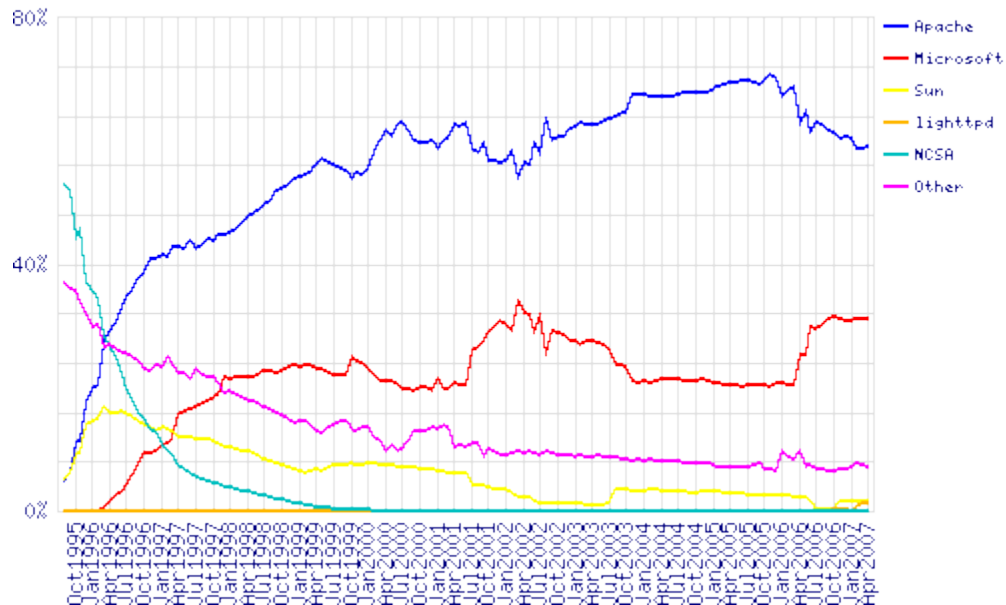


Figure 1. Web server usage statistics [6]

There are two major versions of Apache, the 1.3.x series and the 2.x series. Many features were added for the 2.x version, however these features are not necessary and so many people choose to use the older series (which is still updated along with the new series).

Both versions have many of the features we were looking for and there is a lot of documentation on the internal API, however Apache itself is a rather large system, much more complex than what we had in mind for our implementation, which brings us to the next web server.

### *Lighttpd*

”Lighttpd (pronounced "lighty") is a web server which is designed to be secure, fast, standards-compliant, and flexible while being optimized for speed-critical environments. Its low memory footprint (compared to other web servers), light CPU load and its speed goals make lighttpd suitable for servers that are suffering load problems, or for serving static media separately from dynamic content.” [4]



Lighttpd originally started as a research project by a university graduate student to try and handle the C10K problem [9], that is 10,000 simultaneous connections without loss of service. Because

of the work that was put into Lighty for that problem it became very slim and fast, while retaining most of the popular features that larger servers such as Apache have. Lighty is now in use on many sites including YouTube.com, and is still actively developed and fixed when bugs are found. [4]

### *Jigsaw*

”Jigsaw is W3C's leading-edge Web server platform, providing a sample HTTP 1.1 implementation and a variety of other features on top of an advanced architecture implemented in Java.” Jigsaw is interesting because of the clean, well documented code and the plethora of documentation available about the design and its implementation. [3]



Jigsaw has objects that generate content for each URI that is processed. It has further logical separations of the data as it travels along its internal pipeline, depending on the resource and protocols in use and how the server is setup. The server works on the basis of frames, each of which get processed on the way through the server before a response is sent to the client. For more information and a thorough explanation of these concepts please refer to the Jigsaw project website. [3]

### **Overview of Oriven**

All of the server implementations just mentioned provided inspiration for the design of Oriven, however before we get to the design I will cover some of the history, basic concepts and goals we wanted to accomplish.

### *History*

The story of Oriven begins during the winter break of 2006. My brother Jens and I were both home in Germany, back from studying in the United States to visit family. We had several weeks with hardly a thing to do and began talking about taking up a project to keep us busy. Since we had both signed up for networking courses the following semester we decided to learn some network

programming to give us a heads up on what was heading our way in the classroom. I had done some extremely simple IRC chat bots before, but they were nothing compared to what Jens suggested, a full blown web server implementation. Neither one of us had the slightest idea how a web server works at that point, so we decided to go ahead and try.

### *Concept*

Our project was born, and we set out defining goals and requirements for ourselves. We spent several days drawing out diagrams, erasing them, redrawing them, screaming at each other, and finally settling on something sane for the general layout of the server and how we wanted everything to work. Several goals we immediately agreed upon included being:

- Simple – we wanted to keep things as simple as possible
- Light weight – keep the core server slim and add on to it as needed
- Modular – allow adding to or removing features through dynamic modules
- Multithreaded – let's not spawn tons of processes
- Secure – make sure we use safe functions to prevent possible security concerns

### *Architecture*

The architecture that we settled on includes a simple configuration system, module system that uses factory functions to create dynamically loadable objects, and a multi threaded approach to accepting and handling connections.

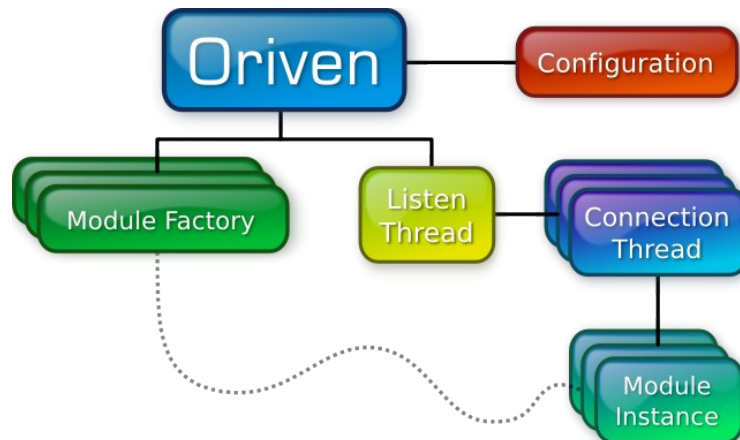


Figure 2. Oriven Server architecture overview

When the server is started the configuration system sets default settings and loads a configuration file. The configuration file includes which modules to load and passes this information on to the module system. The module system dynamically links with the module code and generates module factories. Once the modules are initialized the server opens a socket and begins listening for connections. The listen thread can spawn new handler threads as connections are made. Each handler thread has its own instance of the loaded modules created by the module factories and uses them to handle requests.

#### *Data Flow*

When a connection is established from a client, the listening thread spawns a handler thread, which handles the connection by sending the request through one or more modules. The modules process the request and return a response that is sent back to the client.

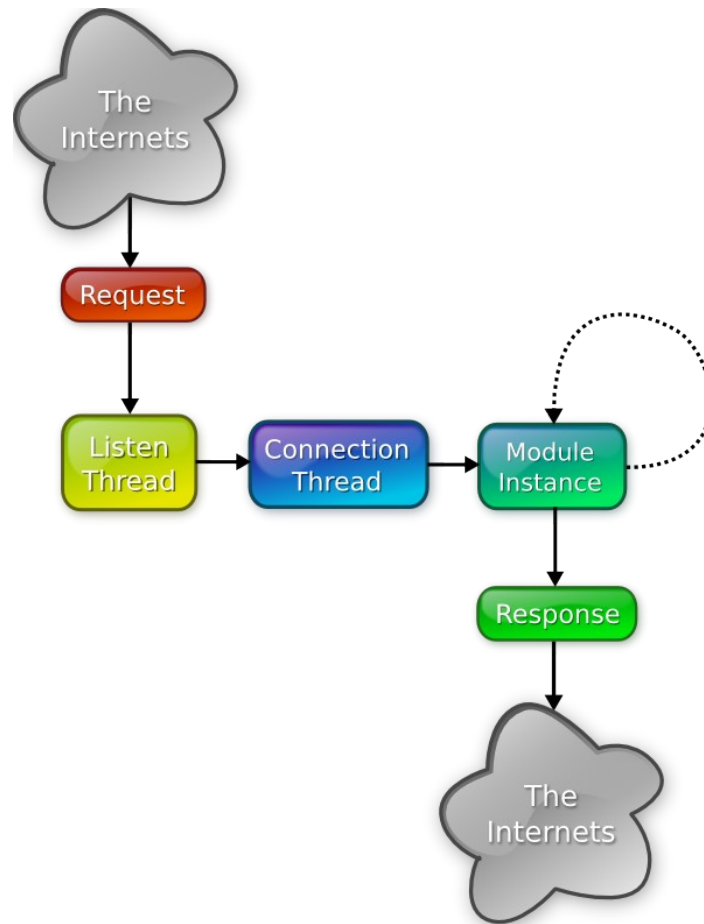


Figure 3. Packet flow through Oriven Server

## Implementation

For the actual implementation of Oriven we decided to use a lower level programming language to give us the most control over how features are implemented. We decided to use C++ so that we could take an object oriented approach to the server, and decided we would still like to be able to use higher level languages, such as Python, through the use of dynamically loadable modules.

### *Configuration System*

The configuration system is as simple as possible while remaining flexible. A configuration file consists of several namespaced directives, such as “server.ip = 127.0.0.1”, which would set the IP address of the server to the local host address. One other directive is supported. The keyword “include”

allows the configuration file to be split into multiple files. We chose to do this so that the mimetype database could be stored in its own file to keep the default configuration file clean.

The configuration file is read line by line. Each line is stripped of whitespace and processed. First, a check to see if the line is empty or a comment is done, and if so it is ignored. Next, we attempt to split the line on the '=' character, which outputs an error message on failure and moves to the next line. If the split is successful, we parse the key and value and set the proper value in the ServerInfo structure, shown below.

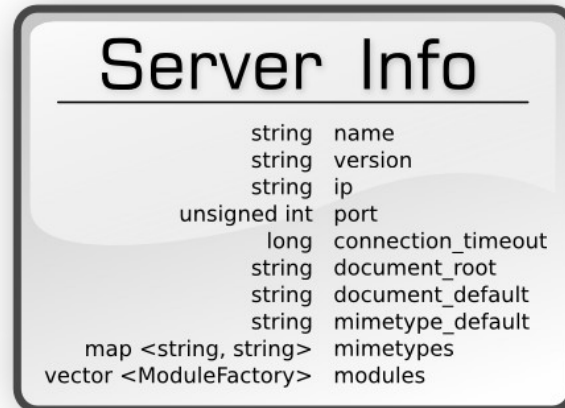


Figure 4. ServerInfo structure

### *Dynamic Module System*

The dynamic module system exists so that we can load features dynamically at runtime. This is accomplished through the use of the dlopen, dlsym, and dlclose functions in dlfcn.h. They allow you to open a shared library object and link to functions within it. The system is C-based, so passing objects back and forth is somewhat difficult. The solution we decided on is to subclass a Module object and provide a function to create new instances of that subclass within the module itself.

A ModuleFactory contains information about a module, as well as pointers to dynamically loaded functions to initialize and cleanup the module and create new instances of the module for handler threads. ModuleFactory objects also contain the handle to the dynamically loaded shared object

code which is needed to later unload the module.

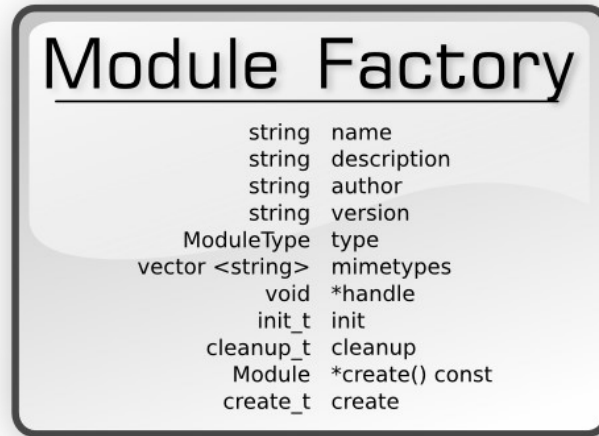


Figure 5. ModuleFactory structure

There are three types of modules: request filters, handlers, and response filters. A request filter is a module that can modify information about a request before it is sent to a handler and is optional. At least one handler must be present to handle a request; this is where the actual work is done, e.g. a document is read from disk and sent to the client. A response filter is a module that can modify the response after it has been generated by the handler and before it is sent to the client. Response filters are also optional. An example of a request filter would be something like Apache's `mod_rewrite`, which allows requests to be modified based on regular expressions. An example of a response filter would be something like `mod_gzip`, which compresses a response before it is sent back to the client.

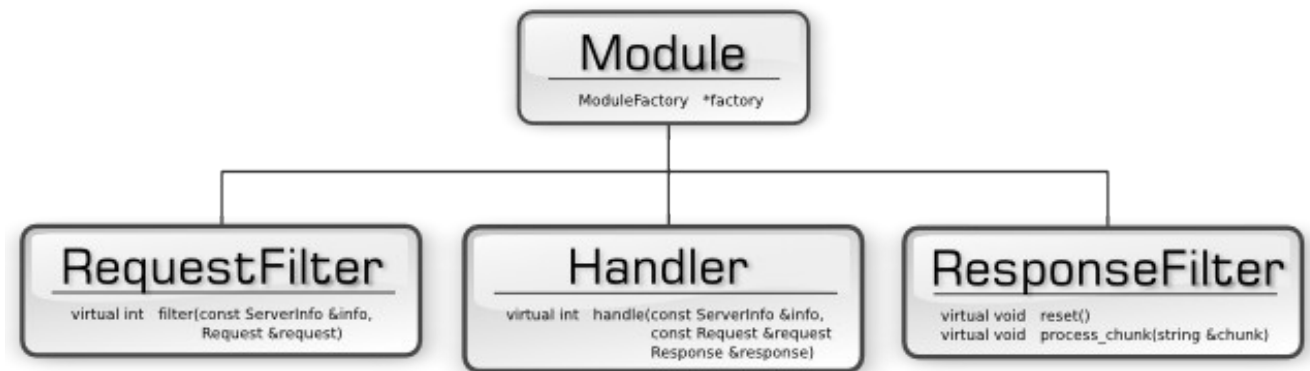


Figure 6. Module inheritance hierarchy

### *Sockets and Threading*

To handle socket communication in Oriven we created a stream-like object that handles much of the low-level functionality. The idea was to create an object that would act similar to `cin` and `cout` from the C++ `iostream` library and support the same operators. This provides an abstraction layer to the rest of the server, and specifically makes writing modules easier. The stream-like class is called `SocketStream`, and is coupled with `DummySocketStream`, which provides the operators to allow writing to the stream using e.g. “<<”.

The `SocketStream` is intelligent about the HTTP protocol. It contains a flag about whether or not headers have been sent yet and has a pointer to the associated `Response` object within it. When data is sent to the stream it will send the headers if needed and then proceed to send the data in chunks after having passed through the response filters (though this isn't entirely implemented yet).

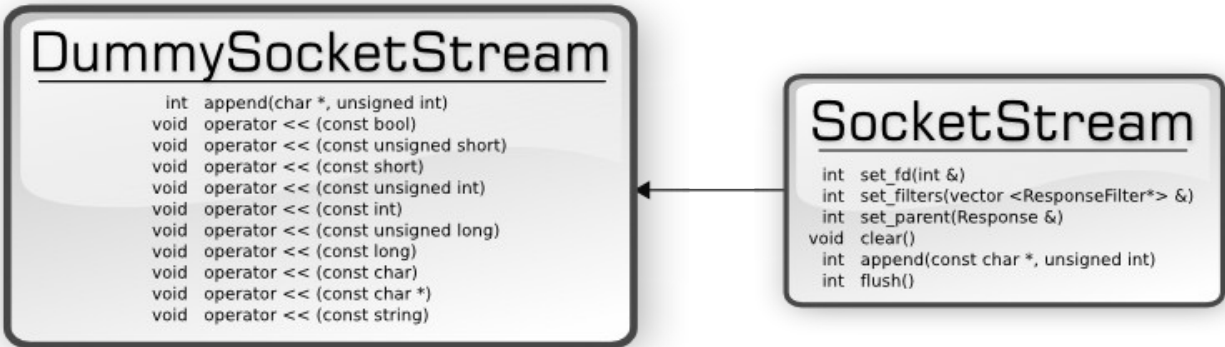


Figure 7. Socket Stream Objects

Oriven is designed to be multithreaded, and spawns several threads on startup. One thread is spawned to manage all the threads currently running. This thread's main purpose is to kill all threads when the server is being stopped or restarted. Another thread is started to bind to a socket and listen for incoming connections from clients. When a connection is made, a handler thread is spawned for each client that will handle HTTP requests from that client. The reason we spawn a single thread for each client is to keep things simple, however this limits us to only being able to handle as many connections as there are allowed to be threads. There are much better solutions and we would like to implement them at some point, specifically the ones mentioned in the C10K paper.

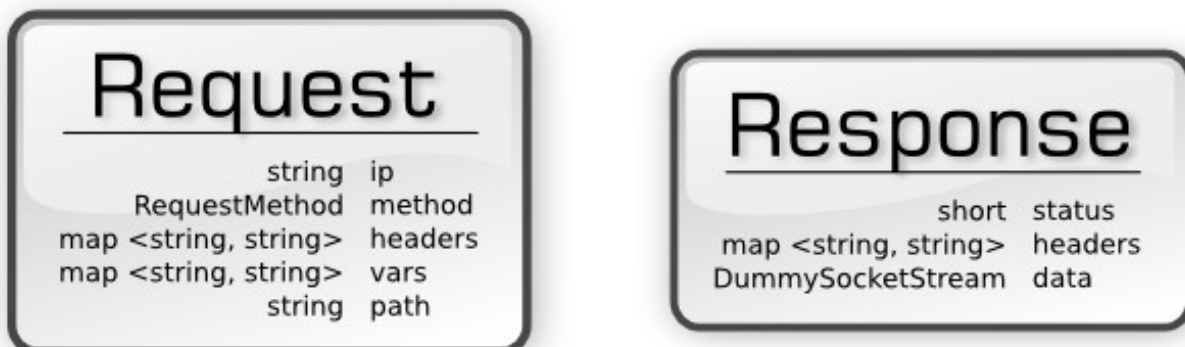


Figure 8. Request and Response objects

### *Module Implementations*

Because Oriven itself has no specific document handling capability in its core, at least one default module was required that would simply return the contents of files on the server. Before starting with such a module we decided to make one other module first to test the system, named `mod_hello`.

The purpose of `mod_hello` is just to return a simple “Hello, World!” message with proper headers no matter what is requested. It acts as a simple skeleton example for a document handler module. The handler simply sets a content-length header and sends the hard coded HTML to the client.

After Oriven was found to be working with the `mod_hello` module we decided to write the default file handler that would return file contents and set proper headers about the files being sent. When a request is made for a file on the server and `mod_default` is called, headers for the size and date of the file are set and the file is sent to the client through the response object by repeatedly reading into and sending a small buffer. This keeps memory usage down and allows us to send large files (as we set data on the Response object it is sent automatically in chunks using HTTP 1.1 chunking).

### **Integrating Python into an Oriven Module**

Once the default module was finished I began work on integrating a high level programming language into Oriven through the use of a module to allow for dynamically generated web content. To do this I researched using PHP, Python, or Ruby and eventually chose to implement an interface to Python, due to my familiarity with the language and its internals, allowing us to write Python code to create server side applications.

### *Overview of Python*

Python is a powerful, byte-compiled, interpreted, object oriented, high level programming language with a very simple but powerful syntax. Python is in use all over the place these days, from national research labs to Google to government entities such as NASA. [x] Even Microsoft is

sponsoring a project to allow Python code to run in their .NET environment. [x] The language is very easy to learn and the code resembles easily readable pseudo-code. Python seems like a great language in which to write dynamic web applications. [8]

### *Basic Concepts*

In order to allow us to write Python code and have it run through the interpreter within Oriven I had to do several things. First, the Python interpreter must be initialized and setup to run programs within the handler module. Next, there must be some way for the Python code to have access to Oriven's internal objects (the Response object at the very least so that data can be sent to the client). I wrote Python objects to wrap around internal classes such as the ServerInfo, Request, and Response objects. This allows Python code to know about the server's configuration, information about the request and the client, and to send headers and data back to the client. One other change still needed to be made, though. Because we wanted to be able to use Python's built in "print" function, I wrote a new Python object that wraps the Response object's data stream and set it to override the sys.stdout object within the Python interpreter. This allows us to use the "print" function just like normal and have it automatically redirect the output to the Response object for a particular request, which will automatically handle sending headers and data in chunks to the client.

### *Results*

After a lot of work, the Python module was completed. It works well, and allows us to write dynamic web content. Everything works as you would hope it to in Python. The C++ classes are mapped to Python objects, vectors and lists are mapped to Python lists, maps are mapped to Python dictionary objects, etc. The print functionality works perfectly, however because of everything being automatic you must set headers before calling print for the first time. The code I wrote to test all the Python functionality is simple and clean, and it works well.

## Conclusion

In conclusion, I can say that Oriven was a successful project. We have created a working implementation of HTTP 1.1 (most of the required subset of functionality) in a modular, multithreaded, and modern way. The implementation took about three days after several days of planning, and a few more days to create the Python module.

To test the server I wrote another python script to spawn hundreds of threads that will all simultaneously request documents from Oriven. This allows us to stage high traffic conditions and monitor cpu, memory, and other usage statistics. The script can be run from many machines simultaneously to make the system more realistic.

Oriven can handle several hundred simultaneous connections (depending on operating system limits for thread count) and has very low cpu and memory usage. It starts up and can stop in several milliseconds. The core code has remained slim and does exactly what we wanted it to.

## Further Work

Much work can still be done on Oriven. Several modules that would be very useful can still be written, such as a module for on-the-fly response data compression, adding features to the `mod_rewrite` module that modifies requests based on regular expressions, a module to allow the use of PHP and/or Ruby code to dynamically generate content, and others. Internally, the `SocketStream` classes need to be cleaned up and some of the code needs to be audited for possible security conditions. Because Jens recently moved the socket stream implementation to automatically use chunking it hasn't been updated to use response filters, at this point only request filters and handlers are used. Also, threading could be re-thought and a different approach, such as one listed in the C10K paper could be implemented to allow handling more simultaneous connections in a sane way.

Oriven is a free software project released under the GNU General Public License so anyone can help contribute. The code is relatively clean and hopefully easy to understand after having read this document.

## **Bibliography**

- [0] Apache HTTP Server Project. <http://httpd.apache.org/>
- [1] Hypertext Transfer Protocol 1.1. <http://www.faqs.org/rfcs/rfc2616.html>
- [2] Hypertext Transfer Protocol. <http://en.wikipedia.org/wiki/Http>
- [3] Jigsaw – W3C's Server. <http://www.w3.org/Jigsaw/>
- [4] Lighttpd - Fly Light. <http://www.lighttpd.net/>
- [5] Module API Documentation. <http://modules.apache.org/reference>
- [6] Netcraft Web Server Survey. [http://news.netcraft.com/archives/web\\_server\\_survey.html](http://news.netcraft.com/archives/web_server_survey.html)
- [7] Oriven Server. <http://programmer-art.org/projects/oriven>
- [8] Python Programming Language. <http://python.org/>
- [9] The C10K Problem. <http://www.kegel.com/c10k.html>