MIA - the Web Crawler

The project is implemented by

Bolette Ammitzbøll Madsen	19951231
Thomas Andreasen Ljungberg	19961591
Thomas Lindgaard	19972035
Kasper Dupont	19972183
Casper Lund Thomsen	19993647
Mads Laursen	19993727

Contents

1 Introduction

As part of the course "Algorithms for Web Indexing and Searching" we took part in the joint project of building a search engine. Our group had the task of designing and implementing a web crawler. The assignment required that the web crawler was to be restricted to the DK-domain and follow different politeness requirements such as not revisiting the same server too often and respecting the Robot Exclusion Protocol.

This report describes our implementation of the web crawler Mia[1], and we will cover our different design decisions and the experiences that we made during the project.

2 Problem

Before turning to our design, the tasks and the problems of a web crawler have to be explored. Simply put the task is the following:

- Pick a URL from a set.
- Fetch the web page.
- Parse the web page for links to other URLs.
- Filter unwanted links away.
- Normalize the URLs.
- Add these URLs to the set.

What differs between web crawlers is the algorithm used to pick the URL from the set; random, FIFO, LIFO; which filters are used etc. but what does not differ is the wanted result from any web crawl:

- The actual fetched pages in some file structure.
- A list of the URLs with references to their pages (or maybe just a list of IDs with references to both URLs and pages).
- A list of edges in the fetched web graph.

Although this *Pick-Fetch-Parse-Filter-Add* cycle sounds simple this is not the case. The problem is that although the request and interpretation of HTML is very standardized, people and servers do not follow these specifications. For instance the web server on `www.imf.au.dk` does not set the content length; `www.jubii.dk` does not respond with response code *404 Not Found* when asked for a non-existing web page; many web servers do not answer requests for robots.txt and the list continues.

Many people have written papers, books, etc. on the hows and whys of structuring and designing a web crawler. Some of them are good – see for instance [Mercator] and [Mercator01], but all that we have seen share the same lack of describing how people and servers actually behave on the World Wide Web. We will therefore move our focus to this area and let the interested reader consult the rest of the literature. But before we can do that, we will have to look at the standards in this field.

[1] The name Mia was suggested by Karsten Strandgaard Jørgensen and decided upon since it follows the tradition between computer scientists of using female names for programs and it is also a reference to the abbreviation of the Danish word for a billion.

2.1 Standards

This section is devoted to the different standards although we will not give a detailed description but only present the reader with some of the concepts.

2.1.1 HTML link structure

HTML linking uses URLs (see [RFC2396]) with the HTTP protocol. These URLs may be relative, either to the current page or the root of the webserver, or absolute and can contain several parts such as host, port, path, query and fragment. An absolute URL is the easy case, since it only needs to be parsed. A relative URL further needs to be combined with a base URL, which must be derived from the URL of the page containing the link. In both cases a simple algorithm (specified in [RFC2396]) must be applied to the URL, to remove path components such as "..".

The actual links may occur in several types of HTML tags. The most common links are the anchor, `a href`, variety, which are rendered as clickable links to other documents. Though anchors by far account for the majority of links, other forms of linking exist such as the link tag. These tags can describe typed links such as "index" or "table of content", which authors of websites can use to further describe the structure of their sites – and search engines to determine such things as the proper entrypoints to sites.

2.1.2 HTTP request

The HTTP Protocol, [RFC2616][2], is quite simple, and since there exists library implementations the crawler can easily be built on top of those. Some things that still need to be considered are matters of identifying the crawler and specifying how we obtained the link. The identification is quite important, since it allows server operators to notify us of problems we may be causing.

Server Response codes HTTP defines more than thirty different response codes, with which a server may answer a request. Each of these (see appendix A for the full list) must be handled properly by the client.

Hopefully we will get a code *200 OK* reply to most of our requests signifying that the server has the requested page and is transmitting it. Given the distributed nature of the web and the multitude of individual servers another code we must expect to see rather often is *404 Not Found*, which will most often be the result of outdated or faulty links.

Some of the more difficult response codes to handle are the 300 codes used for redirects. The permanent redirects can essentially cause several URLs to represent the same resource – not just copies of the same page, but the same actual file.[3]

2.1.3 Addressing

Another problem is the use of virtual web servers where many host names map to the same IP number. This is often used by companies with many names (`www.cola.com` and `www.coke.com`) or by mirrors. But it is also used to allow one IP number to host several host names, for instance Daimiposten and Madsfoek are both hosted on the same machine as Daimi. But how does one tell the difference?

[2] See [HTTP] for further information on HTTP.

[3] Or other resource, as there is no guarantee that a URL points to a file.

2.1.4 Robot Exclusion Protocol

The Robot Exclusion Protocol [REP] is designed so that web masters can define which pages a given crawler should be allowed to crawl. The protocol is used by placing a file called robots.txt at the root of the web server. In this file you can specify per robot which credentials it has on this given domain.

2.1.5 Meta tag

If you don't have access to placing a robots.txt on the web server or don't want to, you can use a Meta tag that tells whether this page should be crawled or not. But this is kind of awkward to uphold because there is no way to decide beforehand whether or not the crawler is allowed to fetch the page. So the only way is to fetch the page and then discard it if it should not have been crawled. Furthermore we discovered that many crawlers do not support this Meta tag and everybody is encouraged to use the Robot Exclusion Protocol instead.

2.2 The human factor on the web

As already mentioned people misspell things – both web pages and configuration files. There are also a lot of people who break the various standards for any number of reasons.

2.2.1 HTML link structure

The common cause of badly formatted or erroneous links is the software used to create webpages[4] – and the browsers that have incorporated workarounds for these small and large errors. If the browsers were not equipped to handle these errors they would be fixed, simply because the creators of the pages could not use them. But since the browsers do handle the errors they are bound to exist in the wild. Examples of the errors include the use of backslashes instead of slashes, nonstandard use of the ".." construct and unescaped special characters.

2.2.2 HTTP request

The problems encountered with the HTTP protocol layer are small and mostly consists of people doing things allowed by the protocol – but inconvenient to us. Such things include sending very large documents, possibly without specifying the optional `length` header, and specifying wrong or invalid `content-type`. But most problems are with the response codes.

Server Response codes Although HTTP define in quite precise terms which response codes should be used in each case people manage to get it wrong. One such wrong response is the practice of replying *200 OK* instead of *404 Not Found* or some other relevant response when no page is available at the location specified. Servers suffering from this misconfiguration usually return a page detailing the error, but such a page may be returned along with a descriptive error code, rather than the *200 OK* code. Unfortunately crawlers or other automated agents cannot use error messages in the text, they need the response codes.

Another problem is scripts with poor or no input checking. Such scripts might return nonsensical result pages if an erroneous URL should be introduced in the crawler. Those result pages may then themselves contain bad URLs causing the

[4] Such programs as Microsoft FrontPage (TM) and HTML export functions available in numerous programs.

crawler to request further bad pages. This can obviously lead to serious problems – both for the crawler, the server and the Internet as a whole.[5]

2.2.3 Robot Exclusion Protocol

During the implementation of our crawler we have encountered several peculiarities with this protocol. We found that many web servers were not configured correctly. E.g. `www.jubii.dk`, where they didn't return a robots.txt but instead returned their standard error page and furthermore they responded *200 OK* to the request, and this seemed to be the pattern for many web servers. Another problem we found was that people couldn't write a proper robots.txt, e.g. they misspelled disallow or something like that.

We decided not to make a big effort to compensate for these things and instead to concentrate on normalizing links and other interesting parts of the crawler. So our parser is as straightforward as it can be and very simple.

3 Design

During the project we have been through many different design choices, too many to mention them all here, but on the way through all these designs choices, we made a radical change of design leading to a complete recoding of the entire system. We will describe both designs here.

3.1 Preliminary design

In our first design we basically tried to partition the crawler into three modules that could be worked on separately since this would allow us to split up into three subgroups. This resulted in the following – see figure 1:

'Fetcher' Given the URLs the fetcher is responsible for actually retrieving the pages while respecting the robot protocol and politeness.

'Graph Generator' Given the fetched pages, the graph generator is responsible for retrieving and passing on the links and generating the edges of the web graph.

'Sorter' Given the links from a number of pages, the sorter is responsible for sorting these and giving the URLs, that are not yet fetched, back to the fetcher. Also the retrieved URLs and the generated edges should be sorted.

However this design quickly turned out to have a number of problems:

- If the fetcher and the graph generator are two separate modules, all pages will be fetched and saved and then read again and parsed – it seems to be extra work to look at all pages twice.
- If the URLs given to the fetcher are actually sorted, there seems to be a greater risk of blocking due to politeness.
- It seems, we need to 'carry' a lot of information around: When a link is found in the graph generator then we already have the page from which that link originates – but we cannot be sure that the destination exists until we have fetched the page and so we might end up carrying a lot of "dead links" around.

[5] There is no way to ensure that bad links are confined to the site generating them.

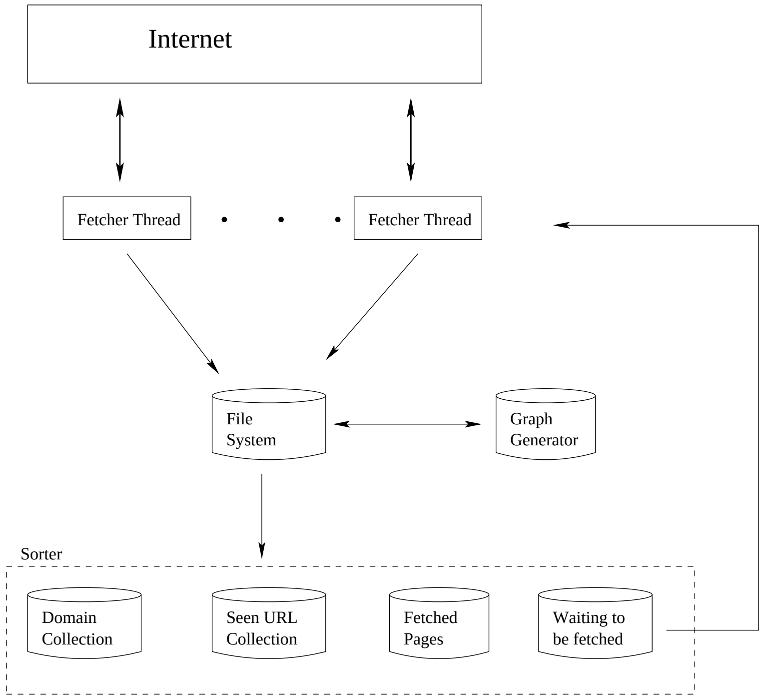

Figure 1: Preliminary design of the Mia Crawler.

Also if it was an internal link, we can add the IP address, so we avoid looking it up again. The sorter may then discover, that we've already fetched this page, and simply throw away everything (except the new edge). It may also discover, that we already had a link to this page from somewhere, and then we have to merge the two structures.

- Finally it seems we would have to have the fetcher and sorter running sequentially, but the fetcher phase would often take a long time to complete because of politeness and slow servers. In the last part of the phase little data would be downloaded.

3.2 Final Design

The design was changed (repeatedly), and finally ended up as can be seen in figure 2. Note that the graph generator is now just a part of the fetcher, while the sorter is still separate. Also the links are not parsed through the sorter. Instead the fetcher keeps the database of seen URLs and decides if a given link represents a new URL, that should be fetched. However it now outputs all possible edges, and the sorter then filters out the edges, that point to pages, which were not fetched.

This design also has a few problems:

- The fetcher needs to maintain some large datastructures which can slow down execution.
- The fetcher will need a far larger amount of memory.
- Checkpointing is harder to implement. This is related to the problem with slow servers in the previous design.

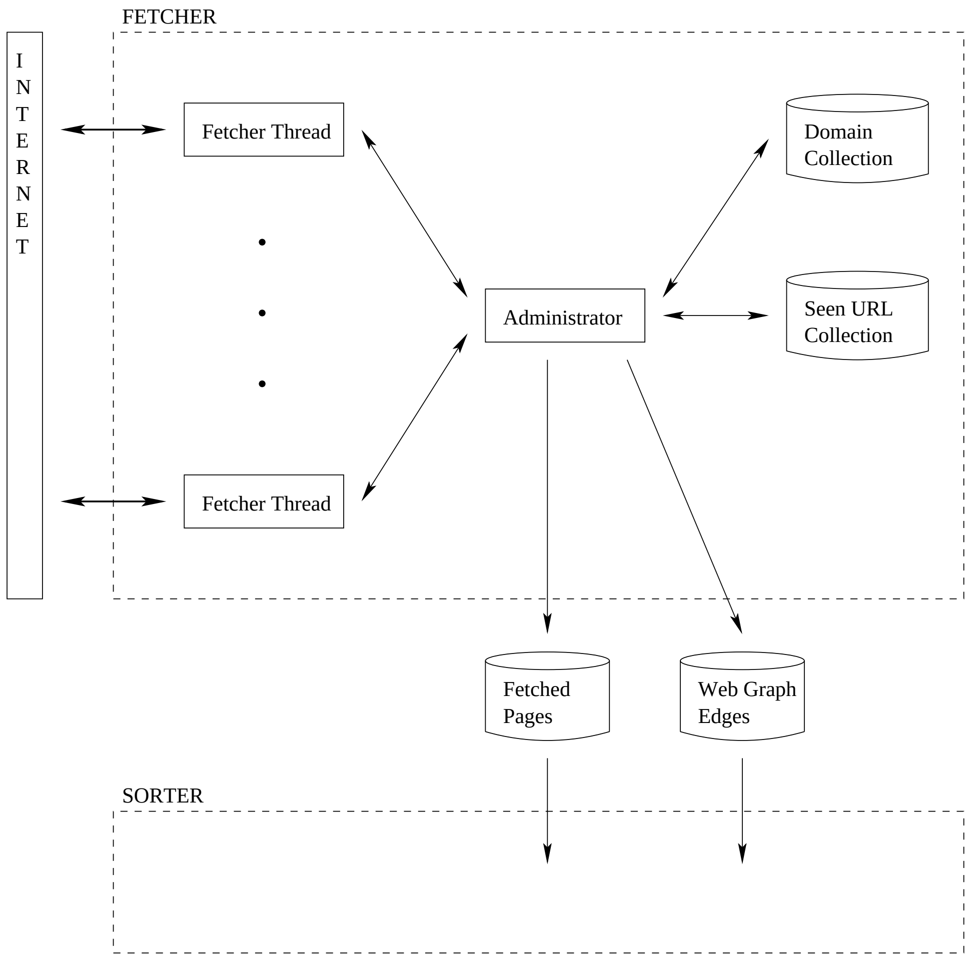

Figure 2: Final design of the Mia Crawler.

4 Implementation

We decided to implement the fetcher module in Java, as it seemed to offer a lot of the functionality, we would like – for instance the `java.net` package offered HTTP support; the `java.util` offered data structures and so on. The sorter module was implemented in C to boost performance.[6] We also created a web page describing the web crawler to the rest of the world.

The code is accessible from: `http://www.daimi.au.dk/~kasperd/mia/`.

4.1 The Fetcher

The fetcher consists of two classes – the `FetcherThread` and `AdminstratorThread`, and a bundle of helper classes. We will describe our implementation by describing how `FetcherThread` and `AdminstratorThread` jointly solves the task.

The `AdministratorThread` has three major tasks:

Next web page To pick the next web page to be fetched in such a way that the

[6] In retrospect it would also have been a good idea to consider performance tuning for the fetcher.

same server does not receive several requests from the crawler within 5 seconds, the so-called politeness requirement. It does this be maintaining a list of all known domains and in connection with this a list of all pages not yet fetched under this domain. When a `FetcherThread` wants a new page the `AdministratorThread` increments the pointer in the list of domains; asks the domain specific time stamp if the last page was fetched from that domain more than five seconds ago and if so removes the first available web page.

To boost performance the `AdministratorThread` maintains for each domain the IP address, port number and a cached copy of robots.txt.

Seen URL No web crawler wants to uncontrollably fetch the same pages and in our case we never want to fetch the same page more then once. Therefore we maintain a set of hashes of the URLs we have seen.

Generate the web graph When a `FetcherThread` adds new URLs to be fetched, it uses a structure which allows the `AdministratorThread` to see from which pages these URLs were seen. It splits this information into $(docId, docId)$ for each URL – actually the $docId$'s are MD5 hashes of the URLs.

The `FetcherThread` simply implements the formerly mentioned *pick-fetch-parse-filter-add* cycle.

Pick The thread asks `AdminstratorThread` for a URL to fetch.

Fetch Firstly the thread checks if we have tried to fetch a robots.txt from that domain according to the robot exclusion protocol. If this is not the case, it puts the URL back and tries to fetch the robots.txt instead, otherwise it tries to fetch the pages.

A number of things can happen while trying to fetch the page, but lets assume that it went well, e.g. response code *200 OK*.

Parse The thread now finds all anchor tags – the `a href` tags – and normalizes the links.

Filter As specified be the assignment we only crawl the DK-domain, so here we simply discard all links outside DK. The same goes for links to files not having an extension normally signifying a text file (ie. images, PDF files and others) – just looking at the extension may not be the best solution but it is good enough for our purpose.

Add All links that survived the filter is handed to the `AdminstratorThread` and the thread starts on a new *pick-fetch-parse-filter-add* cycle.

4.2 The Sorter

The sorting algorithm being used is a generic external memory sorting algorithm. This choice was made to be able to handle the large amounts of data we expected to be handling. The generic sorter was designed to handle elements of variable size. For these reasons the sorting algorithm is given a number of arguments specifying the actual sorting to be done. One argument specifies the required readahead to be able to compute the size of an element and compare the keys of two elements. Another argument specifies a function that can compute the size of an element. Finally there is an argument specifying a function to compare two elements. The sorter is designed in such a way that a function for merging two elements with same key into one element can easily be added, it did however turn out we didn't need this feature, so it was never implemented.

The internal sorting is implemented by sorting an array of pointers to elements. This approach was chosen to be able to handle elements of different sizes. The actual sorting is done using the Ansi C function `qsort()`. The main time usage of the entire sorting algorithm is in this function, but since the constant factor is only approximately 100 clockcycles we don't think this can be done much faster. And since the data downloaded in a few weeks can be processed in a couple of minutes there is no need for a faster algorithm.

The multiway merging is build from twoway merging. Rather than implementing the priority queue needed for a multiway merge sort we implemented a simple twoway merging which can easily be extended to do other tasks such as merging two elements with same key into a single element with the same key. The multiway merging is then done by parallel execution of multiple instances of the twoway merging function. We looked into different approaches to parallel execution. In C there is no language support for coprocessing, and portable approaches to coprocessing through the use of macros and static variables suggested by [CC] would not work in our case, since it assumes that there will never be more than one concurrent instance of the same function. Rather than learning how to use a specific nonportable coprocessing library[7] we chose to use standard POSIX processes communicating through pipes. There were some known advantages and disadvantages of this choice:

- It is a well known and reliable solution.
- It is quite easy to implement.
- It can take advantage of multiple CPUs.
- But it cannot be expected to be the most efficient solution.

If the inefficiency of this multiway merging algorithm would prove to be a problem there would still be ways to improve performance:

- Use buffering for pipe reads and writes in every process.
- Use threads instead of processes.
- Replace the kernel with a newer kernel with more efficient thread support.

However since it turned out we spent less time on merging than on the initial sort we did not implement any of these improvements.

The most tricky part of our multiway merging is the setup of the processes. Every process takes input from two files or other processes, which we for simplicity and intuition have chosen to make two children of the receiving process. In other words the processes must be organized as a binary tree with data being send from the bottom of the tree towards the top of the tree. We want to avoid having any process that merely copies one input to its output, so every process must be reading from two inputs which each must be either a pipe or a file. For this reason we cannot make a single input the base case, instead we have to make any number of inputs up to three a base case, and higher number of inputs is done by recursion. Notice in this case recursion doesn't mean the function calls itself, but rather that it starts two child processes to call itself, and the parent then calls the two way merge on

[7] Such as [coro].

the output from the children. To get the optimal time complexity the depth of the process tree must be logarithmic. So in the general case with more than three inputs we must give each child half the inputs to handle, or as close as possible in the cases where the number of inputs is odd. The process tree will now behave in a way slightly similar to a priority queue implemented as a heap. In the process tree elements bubble towards the top, and the control is not in the root but rather in the individual nodes pushing elements upwards as space becomes free higher up in the tree.

4.3 Postprocessing

During changes of our design it was decided to use MD5 hash of URLs as preliminary ID of a page and assign new IDs as a postprocessing. This requires some sortings, scans, and minor conversions of the data. The exact processing is as follows:

- Each file with compressed documents is assigned an integer identifier.
- A record for each document is written to a stream. The record contains information about preliminary ID (hash of URL), ID of file, and information about offsets and sizes in this file.
- The stream is sorted by hash ID and written to a file.
- The edgelists are concatenated.
- The edgelists are sorted by source node.
- The source node hash IDs are replaced with integer IDs by performing a scan against the previously generated document index.
- The edgelists are sorted by destination node.
- The destination node hash IDs are replaced with integer IDs by performing a scan against the previously generated document index.
- The edgelist is converted from binary format to a textfile with each line containing a source destination pair.
- The edgelist is sorted by source node.
- The pageranks are computed from the edgelist.
- The pageranks are sorted by document.
- By scanning document index against pageranks all hash IDs are replaced with pagerank.
- The index is sorted again this time by pagerank.

By coincidence all binary sorts in the previous algorithm happened to use exactly the same comparison function.

5 Forthcoming

There are several obvious improvements waiting to be made – some of which are very simple and some of which require rewriting some of the Java Core Libraries.

Normalization of links is still not quite bug free, so in some special cases we construct URLs that are not quite correct.

Currently the crawler does not handle HTML frames since it only identifies the anchor tags (`<a href=...>`) in the HTML source. Therefore it will only fetch the page defining the frameset from frame based sites – ie. it will not fetch the pages pointed to by a `<frame src=...>` tag and these are the pages actually containing text on the given site. Similar extensions can be made for a number of other tags.

Improving the current implementation for finding the anchor tags could further extend the number of pages fetched by the crawler – not nearly all links on the World Wide Web follow the standards, so a lot of pages are lost to us because of faulty HTML, but they will still be available in browsers due to the tricks implemented in these.

Finally, performance improvements could be achieved by rewriting some of the Java Core Library code. The designers of the library code have tried to minimize the likelihood of race conditions in multithreaded programs by using excessive synchronization. A lot of profiling work has been made on the Mercator Web Crawler which is also written in Java, and it turned out that rewriting some of the library code was enough to triple the download rate of the crawler (see [JavaCore99]).

6 Conclusion

The greatest challenge by far in writing a web crawler has been to find ways around all the non-standard things in use on the World Wide Web. The list of violations of the standards and strange ways of configuring web servers that we came across while writing the MIA Web Crawler is endless!

When we began running our system, it turned out very early on that the code from the Java API was very slow and did not fit our needs (see the preceeding section), but due to the time constrains we did not have the possibility of changing to another programming language or implementing the needed functionality from scratch.

On another note we had some problems with threads that died along the way during a run of the crawler due to unhandled exceptions. The crawler also needed to hold several large data structures in memory in order to determine if an URL had been seen before among other things, so it turned out to be rather memory demanding – initially too memory demanding. But after having fixed those problems, the crawler has been running (more or less) smoothly, and in the final dataset there are just over 1,7 million pages or more than 5 GB compressed data.

References

[Mercator01] Marc Najork, and Allan Heydon. *High-Performance Web Crawling.* Compaq Systems Research Center, 2001.

[Mercator] Marc Najork, and Allan Heydon. *Mercator: A Scalable, Extensible Web Crawler.* Compaq Systems Research Center.

[JavaCore99] Marc Najork, and Allan Heydon *Performance Limitations of the Java Core Libraries.* Compaq Systems Research Center, 1999

[REP] The Robot Exclusion Protocol
`http://www.robotstxt.org/wc/robots.html`

[RFC2396] Uniform Resource Identifiers
`http://www.ietf.org/rfc/rfc2396.txt`

[HTTP] Hypertext Transfer Protocol Overview
`http://www.w3.org/Protocols/`

[RFC2616] Hypertext Transfer Protocol
`http://www.ietf.org/rfc/rfc2616.txt`

[CC] Coroutines in C by Simon Tatham
`http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html`

[coro] C Coroutines
`http://www.goron.de/~froese/coro/coro.html`

A Response Codes

2**	Successful
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
3**	Redirection
300	Multiple Choices
301	Moved Permantly
302	Found (temporary redirection)
303	See Other
304	Not Modified
305	Use Proxy
306	(Unused)
307	Temporary Redirect
4**	Client Error
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Request Entity Too Large
414	Request-URI Too Long
415	Unsupported Media Type
416	Requested Range Not Satisfiable
417	Expectation Failed
5**	Server Error
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported

Table 1: Response codes