Interim Report

Concurrent Thread-based Web Crawler

Michael Graham

ABSTRACT

This report discusses the early design and implementation of a piece of crawling software written in Java that has been designed to perform efficiently in a multi-threaded environment, its limitations and any future work required. Discussions of specific crawler functions are also mentioned, such as crawler policies, crawler techniques as well as a crawler's general architecture and its limitations.

Table of Contents

[1] Introduction

The ability to find specific information and websites is becoming increasingly essential for a typical end-user whilst browsing the web. This is essentially one of the main reasons as to why crawler software exists – to provide users a mirror of the web, either for archive purposes or simply to find relevant information using search keywords.

Essentially, a web crawler is a piece of software the scours the web, discovering links to new pages in an automated fashion. In general, crawlers are generally used to index pages ready for use by a search engine – however crawlers can also be design to discover and parse different resources on the web (examples are images, or even image and video meta-data).

With the web growing and changing significantly over the years, it is becoming increasingly more difficult to efficiently and accurately crawl the entire web at a respectable pace, which would therefore in turn provide users with in-accurate or outdated information. There is a lack of public information regarding these limitations and their solutions due to the competitive nature of the search engine indexing market (some big names include Google[1], Bing[2] and Internet Archive[1]).

[1] http://www.google.com - One of the leaders in search engine indexing
[2] http://www.bing.com - Microsoft's take on search engine indexing

The three biggest challenges a web crawler has to deal with are as follows:

1. A crawler must use as many resources as possible when crawling a specific site/server to allow the crawler to quickly and effectively deal with any content crawled and move onto other URLs or tasks. However at the same time, the crawler must not abuse a server hosting the crawled information by overloading it.

2. Crawlers like 'good pages'. Good pages can be defined as pages that are rich in information and hyperlinks. However, typically a crawler isn't going to know whether or not a page is 'good' before requesting the resource from the server and analyzing it. This relates to point (1) mentioned above – adding a bottleneck to when other URLS located within the same domain can be crawled and wasting resources crawling effectively 'useless' pages to the crawler.

3. Copies of resources / pages must be fresh and up-to-date. However, determining how often to re-visit pages and crawl them is another issue. If pages are re-visited too often and the page hasn't changed – resources are wasted. If pages aren't re-visited enough – pages that exist within the crawler are outdated. There is also the issue that a crawler must have the ability to discover new pages also – so re-visiting pages and discovering new pages require to be balanced effectively.

[1.2] Project Aims

This reports main goal is to demonstrate a simple working implementation of a concurrent thread-based web crawler written within the Java programming language. The initial simple implementation limited to simply crawling web pages in a concurrent environment, parsing web page hyperlinks and crawling to a specific depth. Possible improvement and further implementation for the next version of the crawler will also be discussed. Possible bottlenecks and limitations within traditional crawler architectures will also be explored, stating a range of previously researched techniques and algorithms for elements such as page freshness and revisit policies.

[2] Background

Web crawlers are typically a lot more complicated than one would expect with a small amount of background knowledge relating to them. In order to crawl the web successfully (by the term successfully, I mean quickly, efficiently and

[1] http://www.archive.org - An attempt to index the entire web across many years

minimal resource overhead) a great deal of architectural planning and structure is required.

[2.1] Existing Web Crawlers

When using the term "Web Crawler" most people would more than likely think of the most popular site on the web – Google[1]. However, there are also several other large-scale crawlers such as: Microsoft Bing, Internet Archive, Yahoo. There are also several open-source implementations for large-scale crawling such as: Apache Nutch[1], ABot[2] and Heritrix[3].

Majority of the larger-scale web crawlers are generally used as the background processing for search engines. Indexing and ranking pages based on their content quality and returning the correct information and results from search queries is a complicated and resource intensive task – hence why they're probably the most appreciated in terms of web crawling.

However not all crawlers are design to cover the entire web in a "general" fashion. Crawlers such as the Heritrix are designed to crawl the entire web and mirror exactly what it discovers – making it a crawler that is designed to download not only web-pages but other media types such as images and zip archives.

Although there are open-source implementations regarding web crawling, majority of the large-scale web crawler solutions are "business secrets" – making the competition to build an exceptional crawler all the more difficult.

[2.2] General Overview of a Web Crawlers Architecture

Since the web is growing an increasingly fast rate, there are billions of web pages to process. However, the number of URL's pointing to these billions of web pages greatly exceeds the number of web pages that exist, which is why it is important to design a structurally efficient web crawler.

---

[1] http://nutch.apache.org/ - Apache Community open-source crawler
[2] http://code.google.com/p/abot/ - A C# open-source crawler
[3] https://webarchive.jira.com/wiki/display/Heritrix/Heritrix - Internet-Archives open-sourced crawler

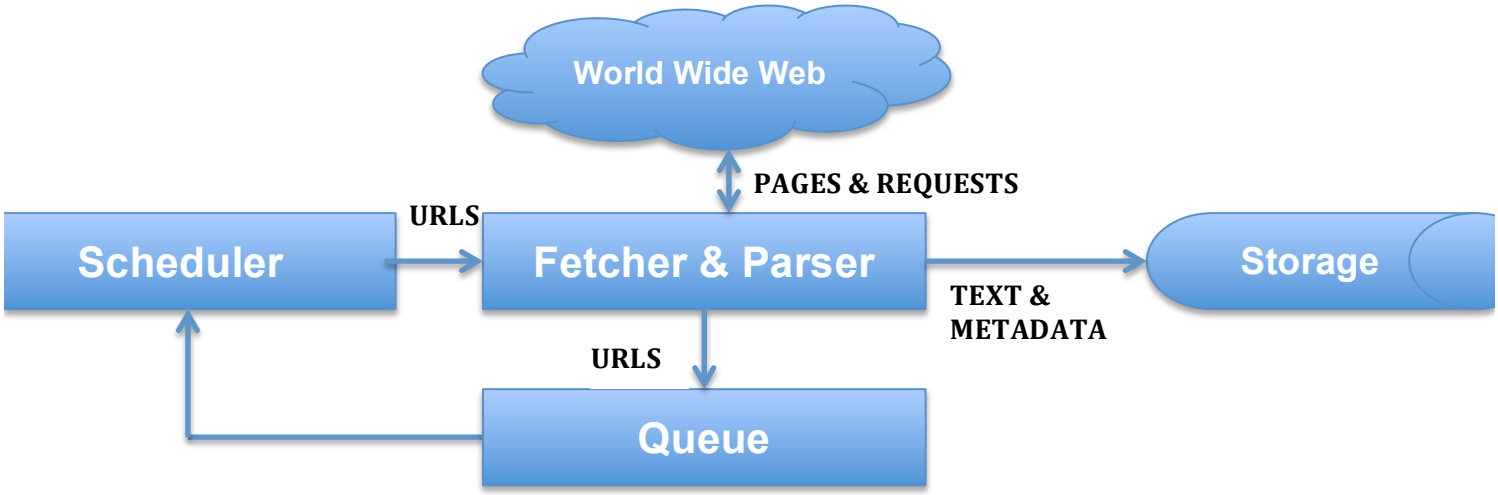

Figure [1] – A general high-level overview of a web crawler's architecture

As a general overview, a web crawler looks relatively simple. High-level components include:

- **Fetcher & Parser** – Downloads & parses downloaded content.
- **Storage** – Form of storage method for storing URLs and any specifically crawled content.
- **Queue** – a queue of URLs ready to be crawled by the crawler.
- **Scheduler** – a method of scheduling URLS (can either be based on content, timeout periods or other factors)

Although this is a simplistic view of initially how a crawler would operate, there are also many other details and complicated features required to be implemented within a real-world crawler. Other components required for a successful real world crawler include and are not limited to:

- **URL Frontier** – in reference to storage outlined in figure 1, this component is a method of storing URLS to be visited by the crawler. The URL frontier is also responsible for determining whether the URL should be scheduled, ready to be crawled – meaning implementations of politeness and freshness could become an aspect of the URL frontier component.
- **Downloader** – in reference to fetcher outlined in figure 1, this component is responsible for handling the multitude of content types encountered by visiting a URL and performing HTTP requests to download the encountered content.
- **Specialized Parser** – as stated in figure 1, a parser to extract content supplied by the downloader. This can vary depending on the goal of the

crawler. Examples include just parsing links, or even varied parsers that can crawl AJAX (Asynchronous JavaScript and XML) content and older websites that use framesets. Crucial crawler configurations such as robots.txt are required to be parsed also.

- **Link Normalizer** – when parsing web pages for new fresh hyperlinks, it is important to "normalize" links. This effectively means determining whether or not the URL is encoded using HTML entities, as well as determining whether URLs are relative to the domain (e.g. domain "example.com" has a hyperlink defined as "<a href="/somedir/somepage.html">) – a crawler has to be able to determine whether or not the URL requires normalizing before passing over to the Downloader component.

The general lifetime process for a crawler can be defined within a few steps that appear simple; but contain some complex logic. The following pseudo-code defines the high-level logic of this cycle:

```
Var queue = getSeedUrls() // seed urls could be more than 1

While (queue length is not empty)
{
        var url = poll queue // pop the head of the queue
        var page = download page via downloader

        var childLinks[]= parse page links via link parser

        for each link in childLinks
        {
                // politeness implementation etc
                schedule link for crawl
        }
}
```

With a sheer amount of content available via the web, it is inevitable that writing a full-scale web crawler suddenly becomes very much a complicated and daunting process, dealing with different content types and link variations (also not forgetting different character set types within webpages, for example different character encodings for foreign languages) and ensuring pages are visited frequently enough and quickly enough – yet not too quick as to overload the domains webserver. Architecture design is critical – and in places parallelization is inevitable, which further complicates the architectural design by dealing with multiple processes executing simultaneously.

[2.3] Limitations of a Generic Web Crawler architecture

With the sheer size of the task a web crawler has to tackle, it's almost certain an architectural design will run into issues or limitations. In order to maintain an

index of URL's in vast quantities with an acceptable "freshness" (freshness determines how up-to-date a page is) – the crawler has to be able to download and process vast quantities of both requests and information per minute. In order to utilize multiple machines, concurrent concepts must be implemented. However, if a crawler that is downloading in parallel across multiple distributed machines, the crawler has to ensure that not too many requests are focused at one server / host simultaneously, otherwise the web server will become overloaded. This can be remedied by implementing a form of policy known in the crawling industry as a politeness policy (which is described in more detail later in [2.5.2]) to limit the amount of requests sent to a particular host / domain, taking into account server administrator wishes (robots.txt, explained later in [2.5.2]) and other factors dependent on the politeness policy algorithm used. At the same time, a web crawler has to attempt to utilize the maximum amount of resources available to it when requesting pages simultaneously from the same domain / server – have too strict a politeness policy and you will under-utilize resources available to the crawler, causing long-term catastrophes such as large over-heads of out-of-date pages and over-heads when re-queuing URLs within the crawlers scheduler and URL frontier.

Another critical issue web crawlers have to deal with daily is the ability to determine "good pages". What is a "good page"? A "good page" is a page that contains a saturated amount of hyperlinks or information (dependent on what the web crawler is crawling for) for the crawler to parse. However, it is difficult for a crawler to determine how "good" the page is to crawl, without first downloading the page and analyzing it. If a page contains useless data, for example is empty or contains un-related / useless information then resources have been wasted crawling the page. This relates closely to the crawler resource utilization mentioned above; crawling useless pages adds a bottleneck to the crawler's resources due to the nature of crawler politeness and respecting web administrator's server resources.

Determining how often and when a previously crawled URL should be visited to maintain page freshness is also another resource intensive and complex issue within the architectural design of a web crawler. If a URL that has previously been crawled isn't visited often enough, the freshness of the page will decrease meaning the indexed page will become out of date. This means in terms of a search engine, information relayed to the users keyword search becomes outdated – especially if the content nature of that page is dynamic or updates frequently. However, re-visit the page too often and you will waste valuable resources, which in the long run adds a bottleneck to how many pages can be crawled simultaneously (again relating to crawler politeness). A solution to this is known as a re-visit policy (and is described in further detail in [2.5.1]) – which helps aid in maintaining the balance between discovering new pages/information and re-visiting pages that are possibly outdates in order to re-new their freshness.

Finally, with the vast amount of URLs a concurrent crawler is likely to discover, a general queue data type will more than likely exhaust any memory within a

singular machine. Since the URL frontier typically handles any URLs that have been crawled or are required to be crawled, an effective and lightweight storage solution is required. Storage to a disk is plausible in a highly distributed crawler – but storage space is still limited and expensive to maintain / expand. To store within a database is also plausible – however this can still provide latency / efficiency issues. Take a scenario in a concurrent environment where one page is being crawled per second on a thread pool with a size of 100 – the database would be required to be polled 100times per second just to determine if a URL has already been crawled and indexed by the crawler. This would require a very robust and efficient database cluster – something that isn't easily achievable.

[2.4] Crawling Techniques

There are no 'set' crawling techniques – in fact in large-scale crawlers there is more than likely to be proprietary crawling techniques to gain a positive outcome depending on the type of service provided. However, there are two predominant fairly well documented crawling techniques that I will mention – Focused Crawling and Path-ascending crawling.

[2.4.1] Focused Crawling

The focused crawling technique (also known as tropical crawling) is a method of crawling that focuses directly on a specific topic or "keyword". The idea behind focused crawling is to only download and visit pages that are relevant to the topic or "keyword" supplied – making it an ideal technique for search engine crawlers. Essentially, pages are crawled based on their importance (importance being how relevant or similar the page is using a focused crawler algorithm). To predict the similarity / relevance of a page without physically downloading it is to analyze previously crawled page(s), paying extra attention to anchor links from hyperlinks to determine the similarity.

Given the example keyword / query, "holiday", previously crawled content will be analyzed to determine the similarity of the original search query and the pages/URLs that have yet to be crawled.

However, performance of focused crawling is highly dependent on the "quality" of a page. If a page contains little or no links, or even links that are dis-similar to the original query then this technique does not provide adequate results. An existing search engine is generally required to provide the crawler with seed URLs relating to the query to kick-start the crawling.

An example of a focused crawling is the "ARACHNID (Adaptive Retrieval Agents Choosing Heuristic Neighborhoods for Information Discovery)" algorithm by Filippo Menczer[2]. For more information regarding this algorithm, please refer to reference [2].

[2.4.2] Path-ascending Crawling

The Path-ascending crawling technique aim is to crawl and download as many resources from a given URL as possible. For each URL within the queue, the crawler will traverse through every possible path within the given URL until the root path is reached. This is typically the behavior that site Web Harvesters will use to ensure all resources are found and downloaded within a given URL. Path-ascending crawling has also been discovered to be useful for finding unknown / non-publically presented resources – such as pages or files that have no direct inbound link from external pages.

Given the example URL "http://domain.com/path1/path2/path3", a path-ascending crawler would traverse through the following URLs:

- http://domain.com/path1/path2/path3
- http://domain.com/path1/path2
- http://domain.com/path1
- http://domain.com

For more information regarding the performance of Path-ascending crawling, please see reference [3] by Viv Cothey.

[2.5] Web Crawler Policies

Due to the aggressive nature of a web crawler, there are several policies that is advised to be implemented to improve efficiency, reduce over-head within resources, and prevent abuse towards web administrators - as well as to crawl to gain the best possible results. The policies covered within this report are: page selection policy, page re-visit policy and the crawl politeness policy.

[2.5.1] Page re-visit policy

The page re-visit policy is fairly self-explanatory. Due to the web becoming increasingly dynamic (with server-side and client-side programming languages, in most cases pulling from a database), pages may change or become unavailable. It is important for a crawler to keep an up-to-date record of a page to provide accurate information – but at the same time not to re-visit the page too often. See Limitations of a Generic Web Crawler Architecture [2.3] for more information as to why a page re-visit policy is required to be implemented correctly.

A study carried out by Junghoo Cho and Hector Garcia-Molina[4] identified two possible re-visit policies. These two polices are known as "Uniform policy" – re-visiting every page within the crawler queue at the same frequency, ignoring the rate of change factors the page inherits, and "Proportional policy" – visiting specific pages more often than others due to their change behaviors (based upon an estimated age of the page using algorithms highlighted in reference [4]). The study concluded that "Uniform policy" is the best, as typically a crawler will

schedule too many crawls for frequently changing pages – hindering the performance of discovering new URLs and pages from pages that change less frequently.

For more in-depth reading regarding page re-visit polices, please refer to reference [4].

[2.5.2] Politeness policy

Politeness policy within a given domain / server is very important when crawling for new un-discovered pages. Hit a website with too many requests from a parallel downloader within a web crawler and either the server will overload under intense pressure, or server administrators will be contacting you. It's very important as a crawler not to abuse server administrator's resources.

There are several implications when executing a crawl, which include but aren't limited to:

- Bandwidth resources – hitting a server with multiple concurrent requests for different URLS use bandwidth resources, costing site administrators money

- Poor architecture – if a crawler has a poor architecture design, or elements are implemented in-efficiency (this includes policies outlined within this report, as well as physical crawler components such as the crawler queue) – not only will it waste resources for the crawler itself, but it will also have implications on the server administrator if requests are malformed or improperly executed.

One solution to prevent overloading servers is to include a form of intervals between requests, for example delaying the scheduling of a thread for X amount of milliseconds. Generally, a fixed delay is substantial enough per domain to not overload a webserver – however a delay could very well be calculated on how well the server responded to previous requests within the crawler queue (an adaptive policy).

Finally, another politeness method is abiding to site administrators robots.txt (Robot Exclusion Protocol). This simple text file can advise crawlers what resources to access, and what resources to exclude. This simple implementation can significantly reduce network and resource abuse – however using an adaptive or fixed request interval is generally considered more efficient and reliable.

[2.5.3] Page Selection policy

A selection policy is essentially a policy to restrict / prioritize pages relevant to a certain topic, instead of crawling aimlessly and downloading random areas of the

web. A study in 2005 by A. Gulli and A.Signorini stated that "we revise and update the estimated size of the indexable Web to at least 11.5 billion pages as of the end of January 2005"[5] – which is extremely outdated concerning the growth of the web today. As explained earlier, web crawlers wish to utilize as many resources as possible – without wasting or over-utilizing, hence why a it is desirable to request and download relevant pages.

There are several factors that determine a pages importance:

- Content quality
- The popularity of the page in a sense of amount of inbound links from other web pages
- The popularity of a page in a sense of how many hits the page receives
- Sometimes the URL and domain can be a deciding factor to determine page importance (e.g. domain is a well-used English word such as "holidays.com")

Designing a "real-world" selection policy is generally deemed fairly difficult, as a crawler has to work with partial data as the page hasn't been requested and downloaded (the crawler is determining whether or not to crawl the actual page based on the selection policy). Corporate crawlers (such as Google) selection policies are generally deemed to be proprietary (due to the competitive nature of other crawlers / search engine indexers).

[3] Approach

To compliment this report, I have written a simple web crawler within the Java Programming language (Java 7 to be precise) that performs a simple traversal through a queue, parsing any hyperlinks and then crawling to a specific depth (depth is number of "hops" after visiting a seed page). This is all executed within a concurrent, multi-threaded environment. It should also be noted that the usage of a 3rd party library named log4j[7] by the Apache Foundation is used to output logging information such as debug and status information.

[3.1] Simple implementation overview

To become familiar with the web crawling architecture, a RAD (Rapid Application Development) approach was taken – diving straight in and getting a working yet very limited (and in some places quite crude) implementation of a web crawler.

As outlined in the class diagram within appendices [1], there is not a very object orientated approach for the simple implementation. Currently, 8 classes exist, all of which are at very preliminary stages in an attempt to complete a concrete implementation.

- **CrawlerEnvironment** – includes main entry-point for Crawler application (void main()), as well as initialization of required classes and

concurrency.

- **CrawlerWorkerCounter** – thread-safe counter of how many active crawlers are within the used ScheduledThreadPoolExecutor.

- **CrawlerWorker** – a "worker" thread that performs the initial crawl. It's here where pages are processed and crawl depth and other search constraints are checked (such as whether to crawl outside the seed domain).

- **CrawlerMonitor** – a 'monitor' thread (or 'daemon') to overlook the status of the current ScheduledThreadPoolExecutor state. Handles shutdown and debug information regarding the Thread pool.

- **PropertiesFile** – a wrapper class for reading different data-types within a Java properties file (using Java's PropertyFile object).

- **CrawlerUrl** – a wrapper to contain the crawlers Urls. Has a 'crude' implementation of URL Normalization currently, as well as keeping track of depth and return a Java URL object.

- **WebpageDownloader** – a wrapper for the Java HttpURLConnection object to download pages. Currently very limited.

The use of an ScheduledThreadPoolExecutor (see reference [6] for the Javadocs) allows more than one concurrent CrawlerWorker to execute in parallel. The ScheduledThreadPoolExecutor also has the ability to delay threads from executing their main body – something that will be useful when integrating a good crawler politeness algorithm.

[3.2] Simple Implementation limitations

Since my implementation is a very basic and preliminary implementation of a web crawler, there are currently plenty of limitations. Current limitations are:

- **No abstraction** – there is currently no interfaces within the initial design. This means there is no room for expansion, e.g. when a downloader wishes to not download a web page, there is no inherited class to download other content types.

- **No crawler polices** – there are currently no web crawler polices implemented (politeness, page selection, re-visit), meaning the crawler only has limited behavior when discovering new links, updating of pages and scheduling new crawler worker tasks.

- **Limited / crude URL Normalization –** the current implementation / attempt at URL Normalization doesn't work for all scenarios. This means

the crawler is limited to crawling links discovered within downloaded web pages.

- **Limited parsing** – currently only hyperlinks are parsed.

- **Limited downloader** – content types, character sets and HTTP headers are either implemented very basically, or are currently not implemented.

- **No URL Frontier & storage** – there is currently no implementation of a URL frontier, meaning previously crawled URLS are re-crawled and are not stored.

[3.3] Future implementation work

As outlined within the limitations, there is a lot of future work to be carried out for the final implementation. The initial plan is to design interfaces for the downloader and parser components, to allow multiple content types and parsers without duplication of code. Other future work includes and isn't limited to:

- **Implementation of a crawler politeness policy** – an implementation of both the robots.txt protocol and the "Uniform" re-visit policy to abide to web server politeness.

- **Better Object Orientated design** – interfaces for downloader(s) and parser(s) to support multiple content types and loose coupling.

- **General overhaul of current components** – refactoring current components to suit new Object-Orientated design.

- **Design and implement a URL Frontier and storage method**

- **Different crawler techniques and policies** – if time permits, addition of crawler techniques (such as path-ascending) and polices (re-visit policy) will be implemented and analyzed.

[4] Conclusion

In this report I have covered almost every important aspect of the functionality of a web crawler. I have provided a simple implementation, of which has a lot of limitations, which have been discussed. For the final report, I hope to vastly improve my implementation, relating to the Future Work section within Approach [3.3].

Appendices

[1] Simple implementation Class Diagram - See uploaded item "classdiagram.png" under "Other Files" in PATS2.

[2] Simple implementation source code - See uploaded item "unicrawler.zip" under "Archive Files" in PATS2.

References

[1] Alexa. (2012). *Top 500 Global Sites.* Available: http://www.alexa.com/topsites/global. Last accessed 13th December 2012.

[2] Filippo Menczer. (1997). *ARACHNID (Adaptive Retrieval Agents Choosing Heuristic Neighborhoods for Information Discovery).*Available: http://informatics.indiana.edu/fil/Papers/ICML.ps. Last accessed 13th December 2012.

[3] Viv Cothey. (2004). Web-crawling reliability. Available: http://onlinelibrary.wiley.com/store/10.1002/asi.20078/asset/20078_ftp.pdf?v=1&t=haoas782&s=29ad99a260826b5ba64b356dc1a41e83a4026a1c&systemMessage=Wiley+Online+Library+will+be+disrupted+on+15+December. Last accessed 13th December 2012.

[4] Junghoo Cho and Hector Garcia-Molina. (2003). *Effective Page Refresh Policies for Web Crawlers.* Available: http://delivery.acm.org/10.1145/960000/958945/p390-cho.pdf?ip=131.251.253.21&acc=ACTIVE%20SERVICE&CFID=156660322&CFTOKEN=68960911&__acm__=1355430791_3eeefbb619427642f12490df6d6762c5. Last accessed 13th December 2012.

[5] A.Gulli, A.Signorini. (2005). *The indexable web is more than 11.5 billion pages.* Available: http://delivery.acm.org/10.1145/1070000/1062789/p902-gulli.pdf?ip=131.251.253.21&acc=ACTIVE%20SERVICE&CFID=156936930&CFTOKEN=68692141&__acm__=1355485489_84c0e28076b7d38d0b969c0b056b24e1. Last accessed 13th December 2012.

[6] Oracle. (). *Class ScheduledThreadPoolExecutor.* Available: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html. Last accessed 13th December 2012.

[7] Apache Foundation. (2012). *Apache log4j 1.2.* Available: http://logging.apache.org/log4j/1.2/. Last accessed 13th December 2012.