Final Report

Concurrent Thread-based Web Crawler (“UniCrawler”)

Michael Graham

ABSTRACT

This final report discusses the final design and implementation of a piece of crawling software written entirely in Java called “UniCrawler”. The report will contain information regarding the implementation and design approach used for crawler components specific to this project, testing techniques carried out, and any issues that arose whilst implementing the typical generic crawler components. Finally, a brief analysis of the crawler and any future work or improvements that could be made to the final piece of software will be discussed.

Acknowledgements

I would like to thank my supervisor Professor David W. Walker, for taking the time to have regular meetings with me to discuss and guide me through this project.

I would also like to thank my moderator Professor Paul L. Rosin, for taking the time to critically evaluate my project hand-ins and provide valuable feedback.

Without the School of Computer Science & Informatics at Cardiff University this project would not have been made possible. I'd like to thank all faculty staff members for their time and dedication to their students.

Finally, I would like to dedicate this work to my parents and family, for supporting me throughout my life's education.

Table of Contents

[1] Introduction

Whilst web crawlers at a glance seem relatively straightforward from a logical point-of-view, designing and implementing an efficient and traditional crawler that can be compared to the likes of Google[1] and other large web corporations is a complex task. The objective of this report is to describe my design and implementation of a generic Web Crawler (named "UniCrawler") – which in terms of functionality and performance is very simplistic compared to other large-scale corporation web crawlers.

In terms of extensibility, UniCrawler has been designed to allow certain generic crawler components to be replaced by custom implementations (such as swapping the frontier component for a custom-made frontier component). Making use of various Object-Orientated programming techniques has made this possible; meaning components within the crawler can be re-implemented by other third parties.

In terms of scalability, the current implementation is very simplistic and has not been designed to scale up to the entire web. Although the multi-threaded nature of UniCrawler's architecture means it can discover new pages on the web at a significantly fast rate, the use of in-memory data structures means a crawl could be limited due to hardware constraints. This will be described in more detail under the limitations section of this report.

The primary motivation for this project was to gain a better understanding on web crawling architecture and to appreciate the complexity involved within writing an effective crawler. However other various motivations such as improving my java knowledge, improving my effectiveness of concurrent programming and providing my supervisor with the requested piece of software also apply.

The remainder of this final report is separated into appropriate sections. The next section (section [2]) describes the design choices chosen for UniCrawler for specific web crawler components. Section 3 compliments the previous section (crawler design) and describes the implementation stages of UniCrawler in a finer level of detail, referring to any concepts and ideas implemented throughout. Any problems that arose whilst implementing UniCrawler will also be discussed, as well as how those problems were initially overcome. Section 4 discusses the results of preliminary crawls using UniCrawler, as well as any tests carried out to ensure UniCrawler functions as expected. In this section, UniCrawler will also be critically evaluated based on any preliminary crawls and tests, commenting on any strengths or weaknesses of the initial implementation and design. Section 5 will comment and discuss any future work that is felt would be required to further improve upon the current implementation. Finally, section 6 provides any conclusions on the project overall.

[1] http://www.google.com/ - One of the worlds leaders in Crawling / Indexing

[2] Design

The overall aim for UniCrawler's design is to keep components relatively simple in terms of complexity and operation. Within the design of UniCrawler you can expect all of the typical crucial web crawler components, separated into their own respective Java packages within the project. This section describes the complete component overview design approach taken, and where appropriate comments on design alternatives will be included.

[2.1] User Requirements

From the offset it was necessary to identify specific user requirements for the intended crawler. The original project description regarding the implementation of a Web Crawler played an important role in determining the user requirements. The following project description for UniCrawler:

> "**In this project you will develop a multi-threaded Java program for crawling the Web that. Each web page encountered will be processed in some way that is dependent on its content. The software will offer options for constraining the search; for example, to just one web site.**"
>
> Upon studying the project description, I was able to determine several user requirements (in this case; for my supervisor) that was required to be included whilst designing UniCrawler:

1. The crawler software must be implemented using the Java programming language.
2. The crawler software must make use of concurrent programming techniques (Threading).
3. The crawler software must parse any downloaded content.
4. The crawler must incorporate different search constraints, which can be triggered or specified by the end-user.
5. The crawler must incorporate the typical crawler components and perform as a traditional web crawler.

[2.2] Standard Crawler Component Overview

Re-capping on the fundamental crawler components that are required for a functional crawler as stated within the Interim Report, the basic sequence executed by a crawler is to take a singular or multiple seed URL's and execute the following steps repeatedly: poll for a URL from a given storage method, download the document that is associated with the polled URL and parse the given document independently depending on its content (usually we want to extract any links within the document). For all of the links extracted from the document, each URL is required to be normalized to an absolute URL and then added to the stored URL list - unless the URL has been discovered previously.

Given this simple outline of a crawler, whilst designing UniCrawler it was necessary to identify crucial components required for basic functionality:

- Crawler component to store URLs (named "Frontier's" in UniCrawler)
- Crawler component(s) to download documents (named "Downloader's" in UniCrawler)
- Crawler component(s) to extract hyperlinks from downloaded documents (named "Parser's" in UniCrawler)

Although there are several other components related to UniCrawler that extend its functionality as a "basic" crawler, the components listed above are crucial for basic web crawler functionality. Other components will be described later within the report.

[2.3] UniCrawler Components

Figure 1 shows the main components of UniCrawler. As shown, each initial crawl is handled by multiple CrawlerWorker instances, where its thread number identifies each worker. The maximum amount of CrawlerWorker instances is determined within the crawler configuration by the end-user.

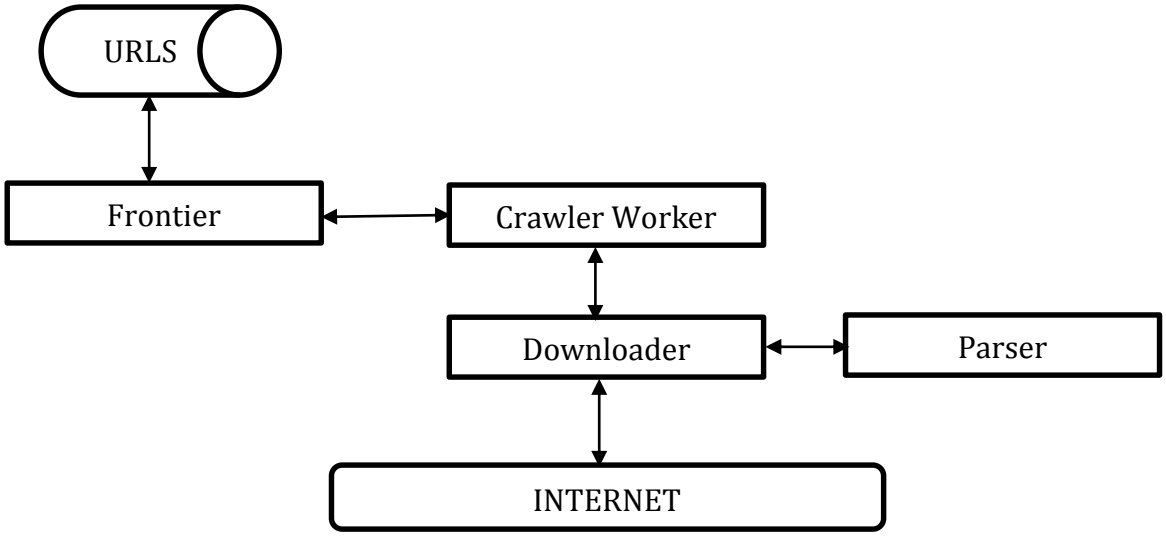

Figure [1] – UniCrawler's main components.

The first stage of the crawl process is to poll for a URL from UniCrawler's URL Frontier component to be downloaded by the Downloader component. In the initial implementation, there is only support for downloading of webpages using the HTTP protocol – however the default downloader component (Webpage Downloader) inherits from an available interface allowing for customization, and more Downloader components to be integrated into UniCrawler in the future. The downloader component continues to download the requested URL and returns its result back to the Crawler Worker thread that invoked the original download request.

The second stage of the crawl process is to pass content downloaded from UniCrawler's downloader component onto the relevant parser component. In the initial implementation, there is only support for parsing of page hyperlinks, however much like the UniCrawler's downloader component each parser component inherits from an available interface allowing for further parsers to be implemented within the future.

The third and final stage of the crawl process consists of the CrawlerWorker traversing through the parsed results returned via the parser component. Here, several constraints are checked such as whether or not the URL has previously been discovered by UniCrawler, if the max crawling depth has been reached as set via the user configuration, or if the hyperlink discovered has been disallowed using the target hosts robot exclusion protocol (robots.txt)[1]. If all constraints are passed, then the traversed hyperlink is en-queued within UniCrawler's URL frontier, ready to be crawled by another crawler worker thread later on.

The above top-level overview of UniCrawler's design excludes explicit implementation details. Designing and making use of data structures that can handle large amounts of datasets efficiently consists of a wide range of engineering complexities. More detailed information regarding UniCrawler's use of data structures and finer design details can be found within the Implementation section of this report (Section 3).

[2.4] Configuration Design

There are many aspects of UniCrawler that can be configured to tailor an individual's needs when performing a crawl. Since the project requirements state that a crawl must have the ability to be constrained, there must be a simple and easy method of configuring these constraints as well as any other miscellaneous properties that will affect UniCrawler's operation.

[2.4.1] Crawler Configuration Properties

The 'Crawler properties' section of UniCrawler's properties file contains configurable properties relating specifically to how each crawler worker behaves. It allows for customization of what frontier component to use, as well as any search constraints to be applied to each crawler worker. Figure 2 contains a table listing each customizable property available.

Property Name	Data Type	Description
crawler.seedurl	String	The initial seed URL for the crawler to start the crawl. This must be in the form of a valid web address (starting with 'http://').
crawler.maxdepth	Integer	The maximum amount of 'hops' from the seed URL that should be crawled. If set to '-1', the maximum depth to be crawled is infinite.
crawler.seedonly	Boolean	Whether a crawl should be limited to the domain

		specified within the seed URL. If set to true, hyperlinks not matching the seed URL domain will be ignored.
crawler.frontier	String	Location of UniCrawler's frontier component class. If the class cannot be found, the default 'unicrawler.frontier.DefaultFrontier' component is used.
crawler.userobotsprotocol	Boolean	Whether a crawl should take into consideration a domains robots.txt. If set to false, the RobotsManager component is not used and robots.txt for subsequent hosts will not be downloaded and parsed.

Figure [2] – Table containing UniCrawler's Crawler-specific configurable properties.

[2.4.2] Concurrency Configuration Properties

The 'Concurrency properties' section of UniCrawler's properties file contains configurable properties relating to how the thread pool containing crawler workers behave. For a more in-depth description of these properties, please see reference [3]. Figure 3 contains a table listing each customizable property available.

Property Name	Data Type	Description
thread.poolsize	Integer	Number of threads to keep within the ThreadPool.
thread.pooldelay	Integer	The amount of time to wait before executing a crawler worker – if crawler worker has been set to execute after a specific delay (crawler politeness period).
thread.keepalive	Integer	The amount of time in milliseconds for threads to wait after executing a task to see if any more crawler worker tasks are en-queued.
thread.debug	Boolean	Whether information should be displayed for debugging purposes, showing active amount of crawler workers and various thread pool statistics.

Figure [3] – Table containing UniCrawler's concurrency properties.

[2.4.3] Downloader Configuration Properties

The 'Downloader properties' section of UniCrawler's properties file contains configurable properties regarding the downloader component. Figure 4 contains a table listing each customizable property available.

Property Name	Data Type	Description
downloader.useragent	String	The User-Agent used to identify the crawler. The User-Agent is used by some websites to display

		different information, or for server administrators to identify possible abuse within their access logs.
downloader.contenttypes	Comma Delimited String	Content types that UniCrawler should download when crawling. Multiple content types may be specified by adding a comma in-between content types e.g. 'text/html, application/pdf'.
downloader.followredirects	Boolean	Whether UniCrawler should follow redirects. If set to false, redirects will not be followed.
downloader.maxretries	Integer	The number of times a URL should be retried upon connection timeout or failure. Once this value has been reached, the downloader will not attempt to re-try the download.

Figure [4] – Table containing UniCrawler's downloader properties.

[2.4.4] Misc. Configuration Properties

The final section of UniCrawler's configuration design is the miscellaneous properties. The idea of this section is to limit output to the console window, which proves useful if debugging specific situations (such as only wanting to see pages / URL's that time-out or redirect). Figure 5 contains a table listing each customizable property available.

Property Name	Data Type	Description
log.showskipped	Boolean	Whether UniCrawler should output information to the console window about URL's that have been skipped for various reasons (e.g. in-appropriate content-type)
log.showresponseerrors	Boolean	Whether UniCrawler should output information to the console window regarding URL's that returned a HTTP error whilst attempting to download.
log.showmaxdepthwarn	Boolean	Whether UniCrawler should output information to the console window if the max depth has been reached whilst traversing through parsed hyperlinks. **(!WARNING! lots of 'spam' outputted to console when enabled)**
log.showseedonlywarn	Boolean	Whether UniCrawler should output information to the console if a parsed URL doesn't match the seed domain (only applies if crawler.seedonly property is set to TRUE).
log.showprevcrawledwarn	Boolean	Whether UniCrawler should output information to the console if a parsed URL has already been en-queued and crawled by UniCrawler.

Figure [5] – Table containing UniCrawler's miscellaneous properties.

[2.5] Design Justification

When the time came to design UniCrawler's components, there were initially two approaches to decide upon in terms of controlling the components and allowing for extensibility. The first approach being the simpler approach, as explained within section [2.3]. The second approach was slightly more complex, however allowed for a greater amount of extensibility. The idea was to implement a Controller/Manager for each component type (e.g. DownloaderController, ParserController etc.), which controlled instances of both default component implementations as well as any other instances of custom component implementations. This would mean custom components / specialized components could be instantiated for say, specific domains or content types by holding all instances within a collection object. Figure 6 provides a diagrammatic view of what this may have looked like if implemented.

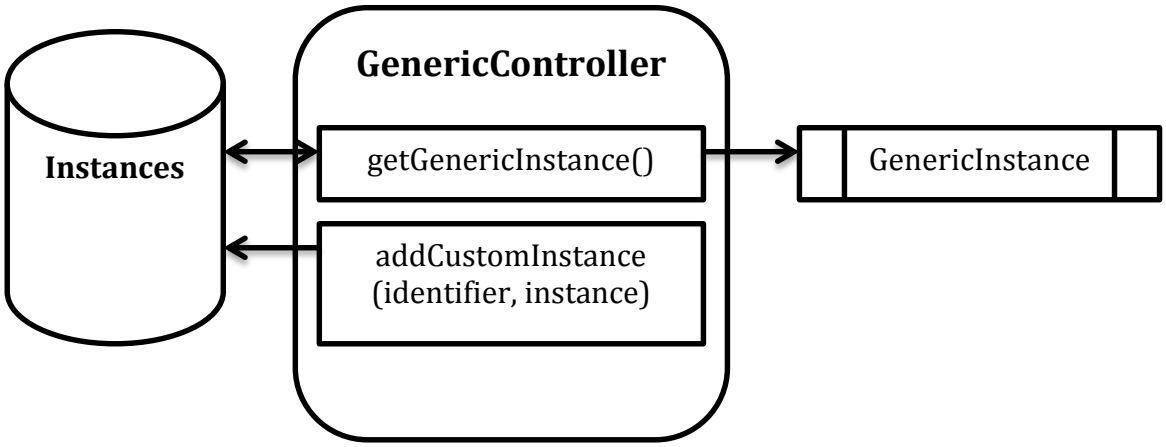

Figure [6] – Simple representation of how Component Controllers/Managers may have looked.

Another design justification taken into account was the design of UniCrawler's configuration. Since there is a relatively large number of configuration options for UniCrawler, using a properties file approach over passing a number of arguments via command line seemed more appropriate in terms of ease-of-use. The initial design of UniCrawler's configuration was passing one properties file via a command line argument, which contained both the Crawler's logging component (log4j)[2] configuration as well as UniCrawler specific configuration – however this was changed as log4j is a separate 3rd party library with separate configuration settings and seemed to clutter / complicate the configuration of the crawler. Since the average user would not want to change the log4j configuration, UniCrawler's configuration was moved to a separate properties file and two command line arguments are now passed to UniCrawler; the log4j library configuration file followed by UniCrawler's properties file.

[3] Implementation

This section of the report compliments the above section (Section 2, UniCrawler design) however provides more intricate details on the implementation of each of UniCrawler's components. Details such as data structures used and implementation issues for each component will be mentioned and justified where possible.

[3.1] UniCrawler's Environment

The idea of UniCrawler's environment class is to provide a base class where crucial component instances can be held and retrieved easily. The design pattern is similar to the factory design pattern, where functions can be called to return a specific instance of a component.

The environment class also handles instantiation of all UniCrawler components ensuring appropriate arguments are passed and all components instantiated without returning errors / exceptions.

One notable feature of the environment class is the ability to swap out the frontier component depending on the crawlers configuration file. The setupFrontier function makes use of Java's reflection techniques to construct an instance of a custom frontier implementation. If the class cannot be found using reflection, then the default frontier is automatically instantiated instead of terminating the environment prematurely. Once the frontier is constructed, the seed URL is then en-queued within the constructed frontier. Figure 7 demonstrates the reflection technique used to pass arguments to a custom frontier instance.

```
Constructor<?> frontierConstructor;

try {
	frontierConstructor =
Class.forName(this.config.getFrontierClassName()).getConstructor(CrawlerEnvi
ronment.class);

	this.mFrontier = (IFrontier)frontierConstructor.newInstance(this);

} catch (NoSuchMethodException | SecurityException | ClassNotFoundException
| InstantiationException | IllegalAccessException | IllegalArgumentException
| InvocationTargetException e) {
	logger.error("Unable to locate frontier class \"" +
this.config.getFrontierClassName() + "\" - using DefaultFrontier");

			// Unable to locate class ns specified in settings file
			this.mFrontier = new DefaultFrontier(this);
}
```

Figure [7] – Reflection technique used to instantiate and pass arguments to a custom frontier component.

Another important role of UniCrawler's environment class is to setup and create the thread pool used to hold and execute crawler worker threads (CrawlerWorker). The setupThreadPool method constructs a

ScheduledThreadPoolExecutor object, setting appropriate configuration flags from UniCrawler's properties file. The use of a ScheduledThreadPoolExecutor was decided based on how crawl politeness was going to function. Delaying the execution of a crawler worker was intended based on crawl politeness factors and to prevent overloading of a server. The schedule() method of the ScheduledThreadPoolExecutor allows for a delayed execution of a crawler worker, meaning a politeness time period could be implemented to prevent server overloading – whereas the execute() method instantly executed the crawler worker thread. The limitations of this choice will be explained in section 4.3.1, as well as an explanation in future work (section 5) as to why crawler politeness is not currently implemented.

Another note when setting up the ScheduledThreadPoolExecutor is the creation of a worker monitor thread. Within UniCrawler, this is identified as CrawlerMonitor. The idea behind this monitor thread is to track the statistics of the thread pool – such as how many completed threads, active threads and worker threads are waiting to be executed. This thread operates as a daemon and implements the runnable interface. Once a crawl has been completed (may be completed if crawl is limited to seed URL only or other search constraints), statistics are provided by the monitor thread as to how long the initial crawl was executing for, how many pages were discovered within the crawl and other statistics such as error counts and redirects.

[3.2] UniCrawler's Types

UniCrawler's types are essentially models to contain data. The two main types are wrappers for important objects within the crawling architecture. UniCrawler implements two custom objects – one for downloaded content (Page) and one for URL's to be processed by crawler worker threads (CrawlerURL).
The CrawlerURL class provides a method of retaining and dealing with constraints such as the URL's depth. It is a very simplistic approach, and only contains a string representation of the URL; its recorded crawl depth and a method for returning the string representation of the URL as a Java URL object.
The Page class provides a method of containing downloaded content information. Information such as the URL of the page downloaded, the downloaded pages content, HTTP response code returned by the downloader component, the content character-set for appropriately decoding the downloaded resources content and any HTTP headers extracted by the downloader component. All HTTP response headers are stored within an ArrayList data structure, as a response from the downloader component can contain an unknown number of headers (dependent on how the hosts webserver is configured), meaning it will need to grow in size. The Page object also contains a method for extracting the redirect URL from the HTTP headers collection – meaning if a HTTP redirect response code is encountered, the redirect URL can be extracted by the crawler worker thread by calling this method.

These types implemented within UniCrawler are crucial for operation, and without them the crawler worker would not be able to record and extract information such as crawl depth and other search constraints.

[3.3] Crawler Workers

Crawler Workers (known as CrawlerWorker within UniCrawler) are crucial to the operation of the initial crawl session. A Crawler Worker is essentially a thread that resides within the main ScheduledThreadPoolExecutor that performs the crawl.

Each worker is identified by its position within the thread pool. A separate class named CrawlerWorkerCounter has also been created containing an AtomicInteger to keep track of how many worker threads are currently active. The AtomicInteger doesn't necessarily mean each worker thread is active, this is due to constraints of the size of the thread pool – however it gives a good indication of how many crawls are waiting to be executed. Obtaining the number of active threads within the thread pool and comparing to the value of workers means the amount of crawls waiting to be executed can be calculated.

Each worker when executed communicates with crucial crawler components within UniCrawler. Whilst executing the run() method of the crawler worker, the assigned URL to the worker goes through a series of operations and conditional checks. Firstly, a conditional check to check if the URL matches the seed domain is performed – and providing the user has specified to constrain the crawl to the seed domain only the worker thread will be terminated and CrawlerWorkerCounter decremented. Next, the worker thread requests that the downloader component downloads the specified URL assigned to the worker thread. A series of conditional checks are check after the download to ensure the downloader provided a valid result – if the result from the downloader component is null then it is assumed the resource was either skipped or unsuccessfully downloaded.

The next stage of the worker thread is to check the HTTP response code obtained from the downloader component. It is important to check if the URL host server has issued a redirect or other HTTP error that means content is non-existent. If the downloader component returns a HTTP redirect (a HTTP code between 300 and 400) then the redirect URL is requested from the Page object by calling the getRedirectUrl() method. A conditional check ensures that the returned URL is not null and is a valid URL before en-queuing the new redirect URL and terminating the worker thread. If the downloader component returns a HTTP OK response (HTTP code 200), then the URL and Page object containing the downloaded content is added to the URL frontier's successful crawled pages collection (further explained in section [3.4]), and the parsePageLinks() method is called within the page object to start the parsing of downloaded content by UniCrawler's parser component(s). Whilst traversing through the results returned by the parser component(s) results, several conditional statements are checked such as whether or not the URL has been seen before by the frontier

component (and if it has, skip the current iteration) and whether the maximum crawl depth has been reached if specified within UniCrawler's configuration. Other conditionals such as checking if the parsed URL from the downloaded pages content is disallowed by robots.txt within the robots manager component, however this will be explained in more depth in section 3.6. Finally, if all conditions are met, the newly discovered URL is added to the frontier queue ready to be processed by another crawler worker at a later time. It should be noted that if any other response code is returned other than the codes mentioned above, the crawler worker terminates.

[3.4] The Frontier Component

The Frontier component of UniCrawler (also known as the URL Frontier) is one of the most crucial aspects of a crawler. Essentially, it is a component that contains all the URL's that are remaining to be processed by a CrawlerWorker (a crawl worker thread within a ThreadPool). UniCrawler's implementation is slightly different compared to traditional crawlers, as it supports storage for previously crawled URL's as well as storage for successful crawls.

Since UniCrawler performs a breadth-first traversal of the web (starting with the seed URL as supplied within the configuration), the Frontier component makes use of a FIFO (First-in, First-out) queue data structure for storing and managing URL's due to be crawled, meaning that URL's are de-queued in the order they were en-queued. The initial queue makes use of the abstract interface Queue[4] – which implements Java's LinkedList[5] data type. In terms of web crawling architecture, this is a very simple implementation. For further comments on the frontier's data structure limitations, please see section [4.3.2]. The data type of choice for storing successful crawls is a HashMap, as a copy of the crawled page is also stored within the collection with each subsequent map key being the absolute hyperlink to the crawled resource. Finally, the data type of choice for storing discovered URL's is an ArrayList. Since the discovered URL's are not required to be in any particular order, and the amount of discovered URL's within a crawl session is an unknown length (an array would not be suitable), an ArrayList seemed the most appropriate data structure to use.

The frontier component of UniCrawler is one of the "pluggable" components, meaning it can be swapped out for a completely different implementation. The class diagram representation of the frontier as shown in Figure 8 shows how UniCrawler's default frontier inherits from the IFrontier interface, as well as any fields and methods contained within the default implementation.

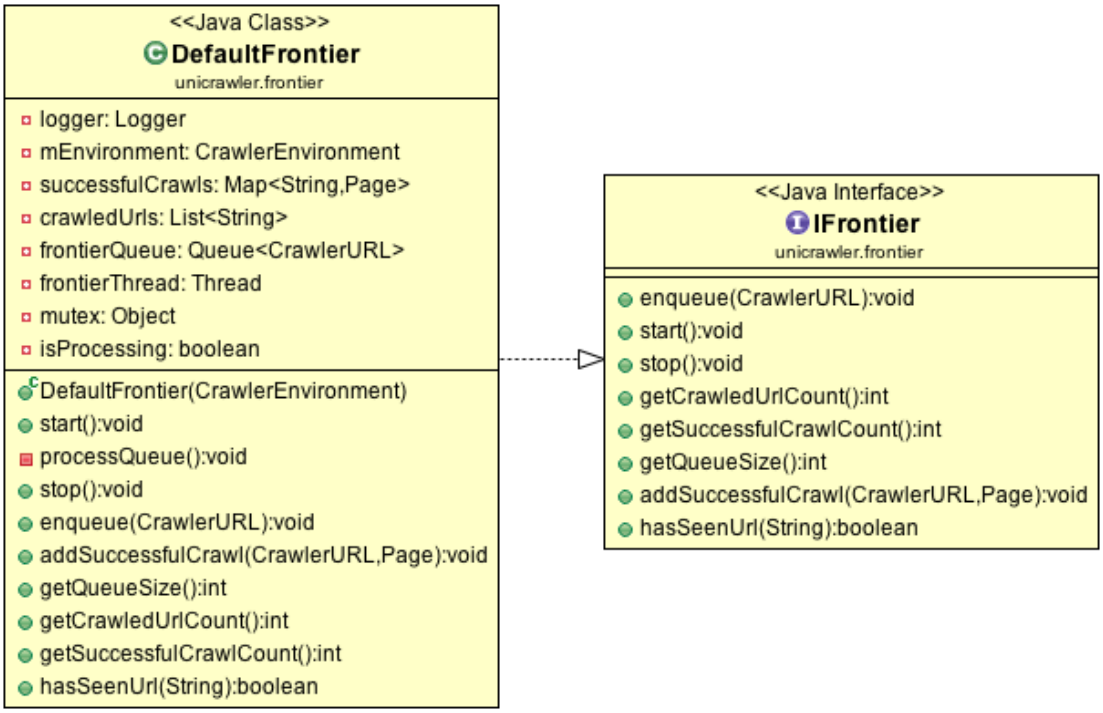

Figure [8] – Class diagram representation of UniCrawler's Frontier component.

The frontier queue is processed by making use of the Thread object (frontierThread) – which is initialized within the DefaultFrontier's constructor. Once the 'start' method is called, the Thread begins processing the queue, which takes place within the processQueue function. The use of the Thread object instead of explicitly allowing the class to implement the runnable interface is due to extensibility purposes; a 3rd party may wish to include several threads (for example, using multiple Queue's or other I/O operations) in order for the frontier to behave differently. All other functions are self-explanatory within this simplistic frontier component implementation.

There are several conditional checks in place within the frontier implementation. For example, when en-queuing a new URL, the frontier will check the crawledUrls collection to prevent the same resource being added to the active frontier queue more than once. Other conditional fail-safe checks include ensuring the queue is not empty before starting the frontiers Thread and therefore processing the queue, and ensuring when polling from the frontiers active URL queue that the object returned from the queue is not null.

[3.5] The Downloader Component

Much like the frontier component, UniCrawler's Downloader component(s) are crucial to crawler operation. This component handles downloading content as requested by the crawler worker threads, and returns the Page object (discussed in section [3.2]). The implementation provided within UniCrawler is a basic HTTP webpage downloader, capable of making HTTP requests to a web server to retrieve content.

The implementation provided by UniCrawler supports filtering multiple content types, a maximum number of download retries upon failure – as well as custom HTTP headers to be sent to the target host. Custom headers are handled by another object known as DownloaderRequest, which is essentially a container containing the target host URL and any custom headers within a collection to be sent to the target URL. Figure 9 displays the class representation of the HTTP downloader component and any other related objects.

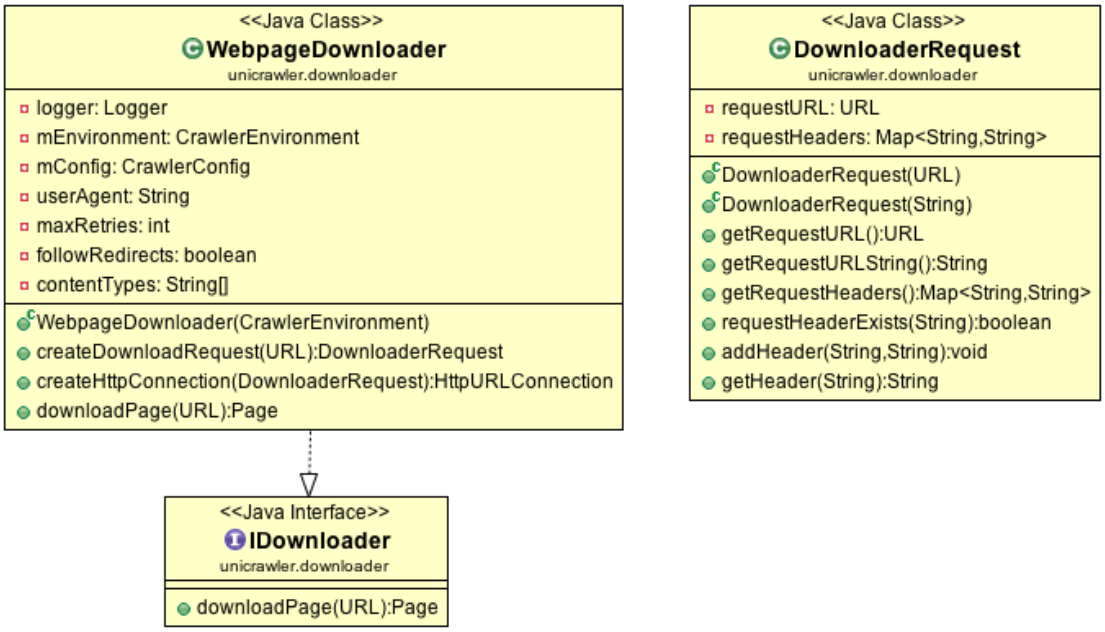

Figure [9] – Class representation of the downloader component and any associated components.

The downloadPage() method instinctively requests the given URL resource from a HTTP webserver. Information such as the HTTP response code after opening a HttpURLConnection to the resource is gathered, as well as HTTP headers returned by the hosts server. UniCrawler's basic implementation does have support for detecting whether or not a website is compressed using GZIP[6] – if GZIP is detected then the downloader component will continue to un-compress the requested content into a readable output, allowing parser components to parse the downloaded content properly. The downloader component also supports parsing of the resources character set – if unable to obtain the appropriate character set of the resource (in this instance the server may be misconfigured) the character set is set to UTF-8 by default. Finally, the Page object is returned (for clarification on what information is stored by UniCrawler's Page object, please see section 3.2).

There are several limitations with UniCrawler's downloader component, please see section 4.3.3 for more information.

[3.6] The Robot Exclusion Protocol Implementation

An advanced feature implemented into UniCrawler is the ability to abide to any web servers Robot Exclusion Protocol rules. The implementation is rather crude, however it works reasonably effectively and webmasters disallowed URL's are ignored by crawler worker threads.

The implementation of the Robot Exclusion Protocol in UniCrawler makes use of two objects. The RobotsDownloader class, which is responsible for requesting the robots.txt from the desired host and the RobotManager, which is responsible for managing entries for existing hosts previously parsed and parsing newly requested robots.txt resources. Figure 10 displays the class representation of these two objects.

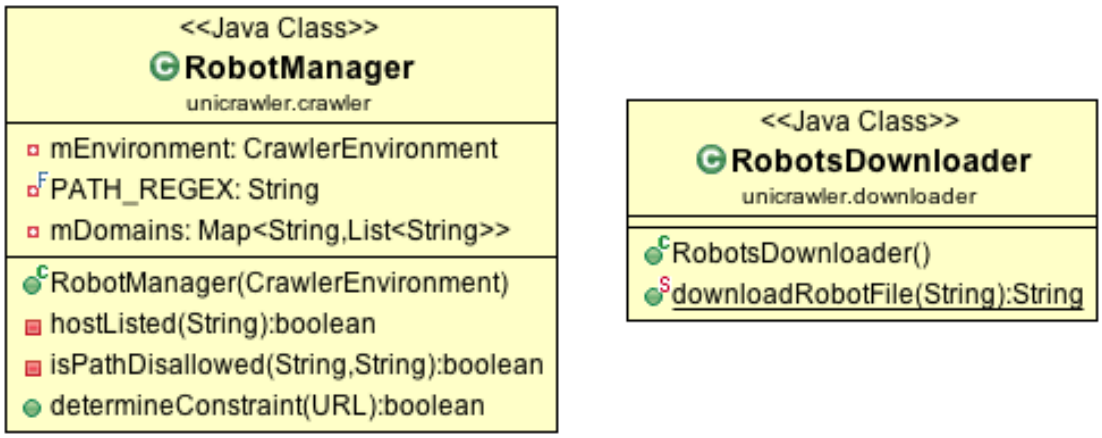

Figure [10] – Class representation of the Robot Exclusion Protocol components.

The RobotManager consists of a thread-safe HashMap collection to store disallowed hosts; the key representing the host domain, and the values being an ArrayList collection containing all disallowed URL's that the Webmaster has specified not to be crawled by web crawlers. To extract disallowed hosts from the robots.txt, a regex pattern is used to match any disallowed URL's that may be contained within the robots file and added to the collection. The RobotsDownloader implementation is very simplistic; making use of a BufferedReader and a stream from the URL object. A StringBuilder is then used to append each read line from the BufferedReader and returned to the RobotsManager for parsing.

[3.7] The Parser Implementation

The role of the parser component(s) in UniCrawler is to parse content downloaded in a way dependent on its content. UniCrawler by default only searches content for new hyperlinks and parses them accordingly. Figure 11 demonstrates a class representation of UniCrawler's hyperlink parser.

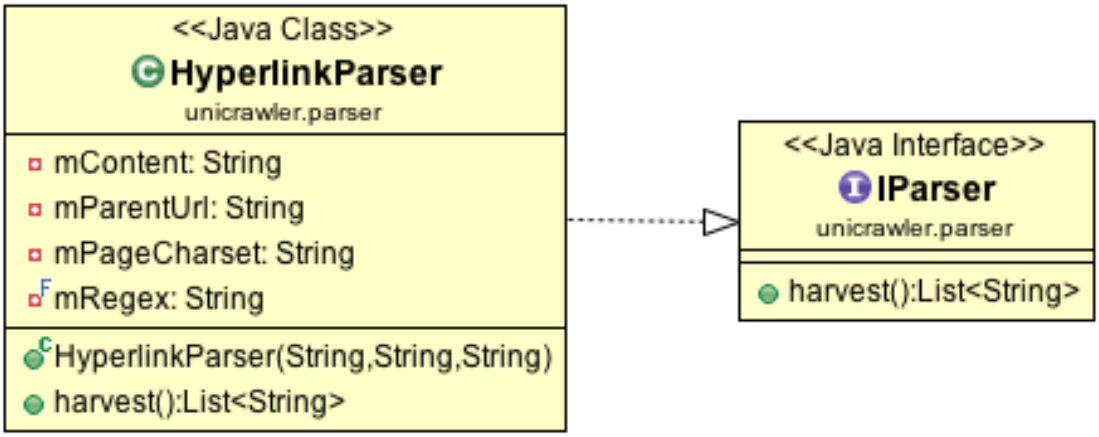

Figure [11] – Class representation of UniCrawler's parser component(s).

The default hyperlink parser within UniCrawler makes use of regex patterns to detect hyperlinks within HTML bodies. Once a match is found, the harvested hyperlink is then required to be normalized. Normalized links are generally referred to as "absolute URLs" – meaning they include the full domain and path, as well as document. In some cases poor HTML can mean URLs on a page are relative to their location, e.g. "../../page.html". This is ok when browsing a website as the browser can interpret the hyperlink. UniCrawler makes use of a 3rd party class to do this, and can be referred to by viewing reference 7. The hyperlink parser also makes an attempt at URL canonicalization (decoding characters within the URL based on the pages character set) by making use of Java's URLDecoder – however it is very limited. You can read more on the hyperlink parser limitations in section 4.3.4.

[3.8] The Configuration Implementation

The final UniCrawler component to discuss in detail is the configuration component. As stated earlier in Section 2.3, UniCrawler makes use of a properties file to load user-defined properties. These properties can be anything from search constraints to other miscellaneous crawler functionality changes.

Since the configuration is read from a properties file, a class has been created that utilizes Javas Properties object – which can be used to parse property key/value pairs within a properties file. This class can be passed a file location for any given properties file to be parsed, which is then in turn loaded with support of a FileInputStream object. This class contains methods for retrieving strings, integers and Booleans from the properties file – with the ability to specify default values if a property key cannot be located within the file.

A class for interacting with the properties file and holding parsed properties within variables has been designed and created, named CrawlerConfig. Within this class, properties are requested from the properties file using the PropertiesFile class mentioned above. This class is simple in functionality, with get/set methods for each subsequent crawler property implemented. Within the constructor, a few conditional statements to ensure properties required for

crawler functionality are present – as well as valid values for these required properties.

[3.9] Implementation Issues

Throughout the development stages of UniCrawler I encountered several implementation issues. From a programmer's perspective, majority of these issues were relatively easy to overcome – however some did have a hindrance on the initial project plan and schedule.

One issue was with processing of the frontier queue. Running the thread consistently polling from the queue sometimes provided null results for the popped URL results. This was due to crawler workers not being able to put URL objects within the queue quick enough for the frontier thread to process. This was overcome by adding a simple conditional statement to check if the result is null before assigning a worker thread with the URL object.

Whilst developing the downloader component several implementation issues arose. Firstly, when requesting content from which the host's server compressed HTTP pages using the GZIP compression method – crawling seemed to terminate prematurely, with statistics only stating that 1 page was crawled and discovered (the original seed URL). This required some fairly in-depth debugging to determine that the downloaded content-type was not human readable, meaning that the regex pattern within the hyperlink parser component could not match any hyperlink elements within the HTML page. The fix was relatively simple and required use of Java's GZIPInputStream. A code-view of the GZIP implementation can be viewed in figure 11. Another issue that arose was parsing the downloaded resources character set. Using the HttpURLConnection getContentType() method does not always contain the character set – this could be due to a server misconfiguration on the requested resources webserver or that the character set is contained within the contents MIME (Multipurpose Internet Mail Extension) field. This was not overcome, and is explained in more detail within the limitations section (4.3.3) of this report.

```
// Do we have compressed data? usually the browser uncompresses this
for us
if (contentEncoding != null && contentEncoding.equals("gzip")) {
      content = IOUtils.toByteArray(new GZIPInputStream(is));
}
else {
      content = IOUtils.toByteArray(is);
}
```

Figure [11] – Implementation of GZIP decompression of pages that have been compressed by the host webserver.

Finally, probably the most time-consuming and biggest issue whilst implementing UniCrawler was the ability to normalize and canonicalization parsed URL's. At first, an implementation was attempted to be created to normalize all relative or non-absolute URL's discovered by UniCrawler via its own component. However, this proved very difficult due to the nature of

different URL formats across a wide range of different webpages – there is no standard that is consistent across the web. If a URL was poorly structured and the URL Normalizer could not normalize it to a required absolute URL by UniCrawler, then the page could not be crawled. Instead of wasting any more time on a concrete implementation, the component was scrapped and a 3rd party class was found that worked relatively well (much better than the original implementation). For a more detailed view of this library of this class, please see refer to reference 7. On the subject of URL normalization, URL canonicalization was also a relatively large issue. Since time constraints didn't allow for a concrete implementation, an URLDecoder object is used in conjunction with the parsed page character set in an attempt to decode any special characters. This simplistic approach seems to work relatively well; however does still fail in rare circumstances. Figure 12 demonstrates the handling of URL normalization and canonicalization, as well as ensuring the URL identifier starts with http.

```
// This is a pretty lame attempt at URL canonicalization by
specifying charset set by downloader headers
try {
      link = URLDecoder.decode(link, this.mPageCharset);
}
catch (UnsupportedEncodingException ex) {
      // if this happens, there's no fallback
}

// attempt to normalize url
link = UrlUtils.resolveUrl(this.mParentUrl, link);

// Check we have proper protocol, url normalization doesn't check for
rss://, feed:// etc
String identifier = link.substring(0, 5);

if ( ! identifier.equalsIgnoreCase("http:") ) {
      continue;
}
```

Figure [12] – Attempt implementation of URL normalization and canonicalization.

[4] Results and Evaluation

This section reports on the operational results of crawls by UniCrawler, with varying configurations. Comments on whether or not UniCrawler functions correctly will also be mentioned, as well as any testing techniques to ensure proper operation of each component. Limitations of UniCrawler's implementation/design will also be commented upon, justifying any improvements that could be made to the final implementation. The main intention of this section is not to provide an in-depth analysis of the web crawler's performance, but rather evidence UniCrawler is capable of performing a simple crawl of the web with user-defined constraints.

[4.1] Testing

Testing of UniCrawler was relatively limited. As each component was developed, outputting information such as variables and debug messages to the console window to ensure both conditional statements and other elements were working correctly was a form of testing I incorporated. This is a particularly bad method of testing, as it does not test every possible state that could be encountered by a user. It did however provide a quick insight as to whether blocks of code were performing as it should in various circumstances.

Another testing technique that was used throughout UniCrawler was the choice of IDE (Integrated Development Environment) named 'Eclipse'[8]. Making use of this IDE's debugging features such as code breakpoints, variable watches and thread inspector to inspect running worker threads proved very useful in spotting abnormal operation behavior. It also allowed to analyse the performance of the thread pool – how quickly the thread worker pool filled up and how many active worker threads within the pool were being used.

From the start of the project it was my intention to use a form of test-case tables in conjunction with a Java-based testing framework (such as JUnit[2]) for rigorous testing. However, lack of knowledge and previous use of such a framework in combination with time-constraints prevented me from doing this.

[4.2] Executed Crawls

To ensure functionality of UniCrawler's components, several test crawls have been carried out and demonstrated. Most crawls carried out were small crawls, and demonstrate the user-defined properties for constraining and limiting crawls by factors such as depth and seed host only.

[4.2.1] Crawl Limited to Seed URL Host

One of the crawl constraints provided by UniCrawler is the ability to limit crawling to the seed URL's host. This proves to work relatively well – however due to the nature of URL normalization, UniCrawler is unable to determine sub-domains to be of the same host as domains. This means if the seed URL contains a sub-domain (such as users.cs.cf.ac.uk) – it is determined to be of a different host without a sub-domain (such as cf.ac.uk). Implementing proper DNS lookups could solve this with domain resolution techniques – which will be explained more in section 4 (UniCrawler Limitations). Figure 13 demonstrates the result of a crawl that was carried out on my supervisor's homepage (without the frameset)[3], limiting to the seed domain only (users.cs.cf.ac.uk).

```
2013-05-01 20:10:26,239 [main] INFO  unicrawler.CrawlerEnvironment -
Starting crawl with seed URL
```

[2] http://junit.org/ - A programmer-orientated testing framework for Java.
[3] David Walker Main Page - http://users.cs.cf.ac.uk/David.W.Walker/head.html

```
"http://users.cs.cf.ac.uk/David.W.Walker/head.html"
2013-05-01 20:10:27,229 [Thread-0] INFO  unicrawler.crawler.CrawlerMonitor -
Discovered a total of 12 page(s)
2013-05-01 20:10:27,229 [Thread-0] INFO  unicrawler.crawler.CrawlerMonitor -
Successfully crawled 2/12 of these pages.
2013-05-01 20:10:27,230 [Thread-0] INFO  unicrawler.crawler.CrawlerMonitor -
Redirects: 0, Errors: 0, Skipped: 1

2013-05-01 20:10:27,240 [Thread-0] INFO  unicrawler.crawler.CrawlerMonitor -
Crawl finished in 00mins, 01secs and 1053ms
2013-05-01 20:10:27,240 [Thread-0] INFO  unicrawler.crawler.CrawlerMonitor -
Crawler ThreadPool is shutting down now...
```

Figure [13] – Crawl limiting the crawler workers to crawl URL's matching the seed URL's domain.

When a URL that does not match the seed domain is encountered, a warning is displayed to console providing the user configuration has been set. Figure 14 demonstrates this from the crawl above.

```
2013-05-01 20:48:29,647 [pool-1-thread-1] WARN
unicrawler.crawler.CrawlerWorker - Not crawling www.cardiffconnect.com - it
doesnt match seed domain
```

Figure [14] – Warning outputted regarding discovered hyperlink not matching seed domain.

[4.2.2] Crawl Limited by Depth

The other crawl constraint provided by UniCrawler is the ability to prevent crawling above a certain depth. Depth can be described as "hops" from the original seed URL, for example setting a maximum depth of 2 will mean harvesting a link from the seed domain allows the crawler to both crawl the harvested link from the seed domain, as well as links from the resource crawled from the seed URL. Figure 15 demonstrates a crawl of my supervisor's homepage with a maximum depth of 2.

```
2013-05-01 21:13:17,515 [main] INFO  unicrawler.CrawlerEnvironment - Loading
crawler configuration...
2013-05-01 21:13:17,531 [main] INFO  unicrawler.CrawlerEnvironment -
Starting crawl with seed URL
"http://users.cs.cf.ac.uk/David.W.Walker/head.html"
2013-05-01 21:13:55,573 [Thread-0] INFO  unicrawler.crawler.CrawlerMonitor -
Discovered a total of 351 page(s)
2013-05-01 21:13:55,574 [Thread-0] INFO  unicrawler.crawler.CrawlerMonitor -
Successfully crawled 304/351 of these pages.
2013-05-01 21:13:55,574 [Thread-0] INFO  unicrawler.crawler.CrawlerMonitor -
Redirects: 33, Errors: 8, Skipped: 3

2013-05-01 21:13:55,580 [Thread-0] INFO  unicrawler.crawler.CrawlerMonitor -
Crawl finished in 00mins, 38secs and 38060ms
2013-05-01 21:13:55,580 [Thread-0] INFO  unicrawler.crawler.CrawlerMonitor -
Crawler ThreadPool is shutting down now...
```

Figure [15] – A depth limited crawl within UniCrawler.

When the maximum depth has been reached, providing the end-user has enabled the logging of maximum depth level reached warning property within

UniCrawler's configuration file then a warning is issued to the console. Figure 16 demonstrates this warning.

```
2013-05-01 21:29:49,365 [pool-1-thread-8] WARN
unicrawler.crawler.CrawlerWorker - Max depth reached for
"http://socitm.govmetric.com/Home/Index"
2013-05-01 21:29:49,407 [pool-1-thread-22] WARN
unicrawler.crawler.CrawlerWorker - Max depth reached for
"https://www.gov.uk/#homepage"
```

Figure [16] – Maximum depth level reached warning messages.

[4.3] UniCrawler Limitations

Throughout UniCrawler, the simplistic nature of its design and implementation means there are many limitations in both an architecture perspective, as well as from a typical crawler's operational perspective. This section has been divided up into appropriate sub-sections outlining these limitations.

[4.3.1] Crawler Limitations

There are several crawler limitations that could not be implemented due to time constraints. This doesn't affect the initial functionality of UniCrawler, it could however hinder performance greatly.

UniCrawler behaves like a typical crawler – it does not support different crawling techniques such as path ascending crawling and focused crawling. This isn't necessarily a limitation of the crawler's behalf – however if a user wished to perform different crawling techniques – or simply implement their own, UniCrawler's design and architecture would not allow for it.

Another feature not implemented is a form of crawler politeness policy. The idea behind using a ScheduledThreadPoolExecutor was to implement a form of politeness policy – where the scheduling of crawler worker threads could be delayed by a set constant before executing the initial crawl. Within UniCrawler, crawler worker threads are executed instantaneously meaning the use of a ScheduledThreadPoolExecutor was not really necessary; and a normal thread pool could have been used in its place.

Finally, an advanced feature not implemented within UniCrawler and therefore is a limitation is a page revisit policy mechanism. Since URL's are checked to see if they have been discovered before by the frontier component, the current default implementation would not allow for such mechanism.

[4.3.2] Frontier Limitations

Although UniCrawler's frontier works relatively well for smaller-scale crawls, the use of in-memory data structures means the queue size is limited by computer memory constraints. With the significant rate of new URL's being discovered, a

form of I/O disk storage would be required. To do this originally, the use of Java's efficient NIO (New Input/Output, java.nio) was going to be used - however due to time constraints it was not possible to implement this feature.

As outlined within section 4.3.1, the use of a Queue data structure would not be appropriate if implementing crawler policies. This is because the queue data structure works in a first in first out policy, meaning URL's within the queue cannot be re-arranged depending on freshness / politeness efficiently. However, if the user wanted since there is an implementation to swap out the frontier component with a custom built frontier – meaning all of the above could be implemented if desired.

[4.3.3] Downloader Limitations

Although UniCrawler's HTTP downloader component provides a concrete implementation of fetching web pages, there are still limitations that could cause incorrect downloading, or improper requests of resources with special circumstances.

One limitation is the lack of support for SSL (Secure Socket Layer) connections to a webserver. UniCrawler only has support for URL's that provide a non-encrypted connection to a given host, meaning if a URL scheduled to be crawled was to contain the https protocol, the downloader component would be unable to properly process the download.

Another limitation is lack of ability to access pages that require a login. This could be either via cookies or a server .htaccess page. The downloader's components lack of ability to retain cookies or provide login credentials to possible protected resources means these pages would be unable to be crawled sufficiently.

Finally, if a pages character set cannot be parsed within the content-type header as set by the host's webserver then the character set is set to UTF-8 by default. This may not reflect the resources proper character set, meaning special characters located within the content (e.g. hyperlinks parsed) cannot be decoded properly. It may be a possibility to parse the resources content type within the MIME type as set by the host's webserver – however UniCrawler does not check for this currently.

[4.3.4] Parser Limitations

As mentioned in section 3.9 (implementation issues) there was initially lots of issues with the parser component of UniCrawler. Extraction performed by the parser component itself is not limited – and all hyperlinks are extracted successfully by making use of a regex pattern to match elements of the downloaded content.

The crucial limitation of the parser component is the ability to not be able to parse every URL encountered in the same way. Due to the nature of the web and varying HTML standards across a range of websites, it is largely difficult to re-build URL's to their absolute state – by absolute I mean including the full host, path and relative file. As it stands, UniCrawler's use of a 3rd party library works fairly well at normalizing these URL's – however at times error messages are displayed to the console stating that some URL's could not be parsed correctly. Figure 17 demonstrates an error message displayed by UniCrawler's parser component upon failed decoding / normalization of a URL. Another note would be the inability to determine that domains with sub-domains pre-appended may be of the same host. Thus, some form of DNS resolution would be required to determine if a sub-domain of a host is equal to the seed URL domain when constraining a crawl.

```
2013-05-02 14:20:41,639 [pool-1-thread-29] ERROR
unicrawler.crawler.CrawlerWorker - java.lang.IllegalArgumentException:
URLDecoder: Illegal hex characters in escape (%) pattern - For input string:
"= "
java.lang.IllegalArgumentException: URLDecoder: Illegal hex characters in
escape (%) pattern - For input string: "= "
       at java.net.URLDecoder.decode(URLDecoder.java:192)
       at unicrawler.parser.HyperlinkParser.harvest(HyperlinkParser.java:52)
       at unicrawler.types.Page.parsePageLinks(Page.java:44)
       at unicrawler.crawler.CrawlerWorker.run(CrawlerWorker.java:60)
       at
java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
       at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:334)
       at java.util.concurrent.FutureTask.run(FutureTask.java:166)
       at
java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.access$
201(ScheduledThreadPoolExecutor.java:178)
       at
java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(Sch
eduledThreadPoolExecutor.java:292)
       at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:11
10)
       at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:6
03)
       at java.lang.Thread.run(Thread.java:722)
```

Figure [17] – Error displayed by UniCrawler's hyperlink parser upon failure of decoding / normalization.

[4.4] Evaluation

When starting the UniCrawler project, I decided to take a RAD (Rapid Application Development) approach – meaning diving straight into the code, and adjusting the architecture of the crawler as development matured. Before initially writing any code, I had a rough idea as to how each component would communicate effectively with each other with a mindset of how to enable the crawler to become easily extendable.

The idea behind the design was to keep UniCrawler's architecture relatively simple. Over complication of the design posed a potential threat to the project

not being completed within the given and planned timescale. Overall I feel the simplistic design of the crawler works well, however I do not feel it is suitable for commercial use, but rather personal-gain or research purposes.

One functionality change I would make if starting UniCrawler again would be to not use a ScheduledThreadPoolExecutor. This choice as mentioned in section 4.3.1 (Crawler Limitations) was in light of being able to implement a politeness policy for crawling the same host. However, upon evaluating the use over a number of months – the thought occurred to me that if scheduling a crawler worker to execute with a delay, a slot within the thread-pool will be taken meaning other crawler workers (such as new URL's discovered) will not be able to be executed due to waiting for the scheduled worker to execute and finish. I was unable to find any reference as to this being the case, however logically it makes sense that the ScheduledThreadPoolExecutor would behave in this manner. Originally I would have liked to of implemented my own concurrent queues – however "re-inventing the wheel" did not seem appropriate due to time constraints within the project.

The use of the Java programming language was required by the project description, however I felt it was a good choice. Java provides all the libraries required to implement a successful web crawler – and also provides a wide range of different concurrency techniques. I already had prior experience with Java before carrying out this project, so picking up any new libraries or objects used within UniCrawler was not too difficult.

[5] Future Work

The development of UniCrawler is still in very early stages. There is a multitude of additional crawler functionality left to be implemented – as well as room for architecture improvements to make UniCrawler even more extendable. This section briefly explains any future work that either I or another developer could carry out to improve UniCrawler.

- **Testing Framework** – building test cases using a framework such as JUnit to ensure proper functionality. Testing manually took up a lot of time throughout the development of UniCrawler, and it is definitely something I would want to use for future large projects.

- **Better abstract architecture** – there are many architectural changes I would make based upon UniCrawler's simple implementation and architecture. Managers / controllers to control components and any customized components would allow for much greater extendibility and code re-usability.

- **Different crawler techniques** – different worker threads could be created to perform different types of crawls. This could consist of path ascending crawling, or even focused crawling if emphasizing on a search

engine indexer.

- **Crawler policy implementations** – the implementation of a crawler politeness policy would be crucial for large-scale crawls to prevent abuse of webmasters servers. This would require significant design changes – however I feel the benefit would outweigh the work required.

- **Disk storage for frontier** – storing partial queues to the disk using I/O within the frontier component would allow for much larger crawls. Since UniCrawler is limited to available memory on the computer executing the crawl, implementing this feature would provide a much better crawling solution. I had intentions of implementing this feature, however due to time constraints I was unable to.

- **Downloader SSL support** – supporting secure connections for discovered URL's requiring the https protocol. This could be implemented by using the HttpsURLConnection class within the java.net package. This could be implemented as a separate class (SecureWebpageDownloader).

- **Downloader authentication / cookie support** – supporting the ability to login to protected webpages would provide functionality to users who wish to crawl user-only areas.

- **Accurate URL normalization** – although the current URL normalization technique works well, a much more accurate and custom implementation would improve crawl results significantly. I felt that a URL normalizer is a project in itself – so perhaps would be a good side-project to UniCrawler due to the complex nature and variations of URL's encountered on the web.

Future work mentioned above is just the tip of the iceberg in terms of a commercial web crawler. Custom implementations of libraries instead of using Java's in-built libraries would be required to be in with a chance to even compete with large-scale crawlers such as Google and Bing. I feel a good starting point if someone else was to continue with UniCrawler would be to implement a crawler politeness policy to make use of the currently implemented ScheduledThreadPoolExecutor – I would personally be interested in finding out how well it worked for delaying crawler workers in terms of politeness, as well as its functionality as to whether or not a slot within the thread pool is reserved, preventing other worker threads waiting that fell outside the politeness period from executing.

[6] Conclusion

A web crawler is generally considered an important component of modern web services of today – however there is a lack of documentation on implementations and techniques used to build an efficient and scalable crawler. Since data collected by a crawler is generally too big to fit in modern-computers memory,

there can be performance issues when attempting to balance in-memory data and disk data. UniCrawler focused more on small-scale crawling, meaning it would not be fit for large-scale crawls in a commercial environment.

A crawler that has been built from scratch within the Java programming language, making use of in-built libraries has been provided. From the offset it was clear that UniCrawler had to support user-defined search constraints – which have been implemented by providing a user-defined, customizable configuration file. One of the requirements of the crawler was to execute within a concurrent environment – which has been achieved by making use of Java's available concurrency objects. A few small-scale crawls have been demonstrated, proving UniCrawler is in fact a functioning web crawler that can be used without too much hassle or configuration properties. An advanced feature that should be noted that I did have time to implement was the use of the Robots Exclusion Protocol – although a simplistic implementation it works well in conjunction with small-scale crawls.

Overall I feel UniCrawler has met its required specification. There are many advanced features I wanted to implement - however due to time constraints I was unable to. If items highlighted within the future work section was to be undertaken, there is no reason why UniCrawler could evolve into a something bigger and better than its current state. A full listing of code and diagrams can be found by referring to appendices 1.

[7] Reflection

The completion of this project has been a learning curve and an overall great method of exercising my planning and Java programming knowledge. The entire project allowed me to exercise many different skills such as; arranging meetings, managing time effectively, building an effective plan, designing and implementing a piece of software and writing an informative report to reflect the work that has been carried out.

I feel I have communicated effectively with my supervisor – arranging regular meetings and explaining any issues I had. I also felt I communicated effectively within these meetings, keeping my supervisor up-to-date with any progression towards the project and discussing any discrepancies I encountered.

I also felt I have improved my Java concurrency programming knowledge. I learnt new techniques for synchronizing collections in Java (such as using Collections.syncronizedMap) that I did not know about. Prior to this project, I had never used a thread pool, so deciding to use a thread pool within my project was entirely new to me.

Although I had a rough idea as to how web crawlers worked, I had never looked into web crawlers in the depth required for this project. I have learnt to appreciate the engineering complexities required to build and execute a large-scale crawl. I have also learnt valuable information about some existing crawler

architecture, as well as different crawling techniques and policies (such as page revisit policies). I felt my research on web crawlers was sufficient enough to build a working implementation and provide an in-depth analysis of crawlers within my interim report.

Overall, I have enjoyed the experience of researching and developing a web crawler, and would definitely enjoy further developing UniCrawler into something more mature in my spare time.

Appendices

[1] Full UniCrawler code archive and all class diagrams – See uploaded item "UniCrawler.zip" under "Archive Files" in PATS2.

References

[1] Wikipedia. (2013). Robots exclusion standard. Available: http://en.wikipedia.org/wiki/Robots_exclusion_standard. Last accessed 28th April 2013.

[2] Apache Foundation. (2012). *Apache log4j™ 1.2.* Available: http://logging.apache.org/log4j/1.2/. Last accessed 28th April 2013.

[3] Oracle. (Unknown). *Class ScheduledThreadPoolExecutor.* Available: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html. Last accessed 21st April 2013.

[4] Oracle. (Unknown). *Queue.* Available: http://docs.oracle.com/javase/7/docs/api/java/util/Queue.html. Last accessed 29th April 2013.

[5] Oracle. (Unknown). *LinkedList.* Available: http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html. Last accessed 29th April 2013.

[6] Jean-loup Gailly. (2003). *GZIP.* Available: http://www.gzip.org/. Last accessed 20th April 2013.

[7] HtmlUnit. (Unknown). HtmlUnit is a "GUI-Less browser for Java programs. Available: http://sourceforge.net/p/htmlunit/code/8165/tree/trunk/htmlunit/src/main/java/com/gargoylesoftware/htmlunit/util/UrlUtils.java. Last accessed 12th March 2013.

[8] Eclipse Foundation. (Unknown). Eclipse. Available: http://www.eclipse.org/. Last accessed 10th April 2013.