WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

EFMD EQUIS ACCREDITED

Bachelor Thesis

Implementing a "polite" proxy for different Web Crawling Use Cases

Patrick Oliver Riemer

Date of Birth: 11.08.1995
Student ID: 1452119

Subject Area: Information Business

Studienkennzahl: J033/561

Supervisor: Prof. Dr. Axel Polleres, Dr. Jürgen Umbrich

Date of Submission: 10.September 2017

Department of Information Systems and Operations, Vienna University of Economics and Business, Welthandelsplatz 1, 1020 Vienna, Austria

DEPARTMENT FÜR INFORMATIONS-VERARBEITUNG UND PROZESS-MANAGEMENT DEPARTMENT OF INFORMATION SYSTEMS AND OPERATIONS

Contents

List of Figures

Abstract

The importance of web crawlers grows with the ongoing expansion of the World Wide Web. Without web crawlers search engines or web archives would not exist. Efficient crawling, however, is not possible without following best practices and obeying politeness rules. Not all robots crawling the web obey these rules leading to the exhaustive usage of network and computing resources which can be interpreted as Denial of Service attacks leading to the blocking of these crawlers. Our solution tries to overcome these issues by both, enforcing politeness through working as a proxy, and additionally offering services to check if a URL does not violate the constraints defined by the Robots Exclusion Protocol. We show a solution, implemented through a proxy, capable of handling a variety of clients and scenarios, enforce politeness and enable monitoring and caching without significantly increasing latency.. . .

1 Introduction

The World Wide Web has grown from a few hundred pages at its beginning to more than a trillion pages in 2008 and has not stopped growing since then. To make use of this unbelievable amount of data as a human being, we need help from search engines or nowadays we even rely on artificial intelligence. However these search engine must get these data from somewhere. The necessary exploration of the web and the extraction of information is done by programs called web crawlers, also known as robots, bots, wanderers and spiders. A web crawler starts at one page or from a set of pages and extracts all links to other pages from there. By following and saving all links on its way through the web a crawler discovers the internet completely automatically in an iterative and incremental fashion. Whereby a web crawler just navigates from link to link, the further processing of data is often done at a later point of time and by different software components.

With the ongoing expansion of the internet, in an exponential form or not depending on the source[9], the number and diversity of web crawlers is also growing. Web crawlers are used to collect data for statistical analysis of the web itself, to collect pages for archive like structures or for search engines and other indexing applications. With variety and increase in numbers of crawlers arise certain problems. Crawlers are visiting a variety of domains, each hosted on different servers and hardware. To prevent unknown crawlers from the internet from using domains' resources in an extensive way network owners undertake several measures. Therefore a crawler must adhere to certain rules to not get blocked or get hindered at its work. Small scripts or other naive implementations iterating through pages and links do not follow these rules and will be blocked if accessing content they are not allowed to access or by simply overloading domains by requesting hundreds of pages in seconds.

On the other side more sophisticated web crawlers respect these rules, are polite and respect *robots.txts* or sitemaps [3], but come with a different set of problems. Here scheduling and performance problems exist. To just grasp a small part of the web one crawler would need an unacceptable long timespan. Although using more than one crawler, maybe even at overlapping or identical domains needs scheduling solutions to avoid getting blocked while crawling efficiently at the same time. Each web crawler on its own might be polite. However this does not ensure politeness when several crawlers operate on the same domain because overall they are violating certain constraints.

At the Vienna University of Economics and Business there exist several kinds of crawlers and scenarios when politeness needs to be enforced and monitored. Students write simple scripts as part of lab courses or out of curiosity. Here the absent politeness and missing scheduling are quite urgent problems. Then there are clients like XQuery[1] demanding the sequential handling of HTTP requests during its operations. Other existing crawlers already implement scheduling and politeness mechanisms, but have to cope with more complex scheduling problems and the overall politeness of operating several independent crawlers. All these quite different implementations and the arising challenges have to be handled:

- Enforcement of politeness for all use cases
- Sequential handling of web requests where needed
- Enable caching for already fetched data
- Monitor performance and gather operational metrics
- Distributable operation of crawlers

As solution we suggest the implementation of a crawling service architecture. An implementation following an service oriented approach would support both naive crawlers and more sophisticated ones. Simple crawlers or even clients without any real crawling logic can use our solution in a proxy like way. Every request is routed through that proxy. Before each request is executed the proxy checks if any politeness constraints are violated. If not the request is executed and the client gets the respective response. Otherwise the proxy informs the client with a special error code.

More sophisticated crawlers might use our solution too, as a kind of proxy to ensure politeness. Furthermore the service will offer the possibility to submit a set of seed URLs. This list of URLs gets checked in terms of politeness violations and the URLs allowed to be crawled get returned to the client. This offers these kind of clients the possibility to ensure politeness, but to manage the crawling process on their own.

Ensuring politeness is not the only feature our solution is offering. There are issues common to all kinds of crawlers besides managing politeness. Caching is one of it. If there are independent implementations crawling on overlapping or identical domains it is possible to cache crawled results

[1]https://www.w3.org/XML/Query/

to make them accessible to future requests considering page updates. This can significantly reduce the time needed and the consumption of network resources.

Section 2 explains basic technological aspects related to our research efforts. In Section 3 we provide an overview of the current state of related work in literature and in commmercial use cases. In Section 4 we go more into detail about the underlying requirements and issues our implementation must offer a solutions for. We also summarise the theoretical background, ways to ensure politeness and related work done. In Section 5 we describe our technical solution to support a variety of clients and forms of requests and how our solution ensures politeness and enables caching. In Section 6 we examine the limitations of our solution and provide an outlook on further research and close by sharing our conclusions.

1.1 Research question

According to the various problems and tasks stated above the research question we are answering is:

"How can a solution, enabling the configurable and scalable enforcement of politeness for a diverse range of web crawling use cases, be implemented following a service oriented approach using Java?"

It is intended to implement a crawling service kind approach using the technologies and strategies described in the next section. It should support all the kinds of implementations as described before while simultaneously handling their implementation-specific weaknesses and enabling new features in a centralised and service oriented way. We believe that a service oriented approach is the best option here to handling various use cases at once by offering a uniform interface and proxy like behaviour.

1.2 Requirements

Based on the research question our service has to fulfil a set of requirements and a list of additional, not mandatory features which would improve the final solution, but are not part of the fundamental use cases. Both, requirements and additional features, can be organised in work packages. These work packages are going to be implemented in implementation cycles of our iterative prototyping approach. Work package 1 to 4 are mandatory features to support all three groups of clients in an efficient manner. The other work packages are additional features, improving performance via caching, enabling monitoring and statistics, and advanced politeness strategies and distributed crawling. The resulting list of work packages ordered by importance are:

1. Implementation of a proxy for simple and unsorted scripts (first use case)
2. Enforcement of politeness via configuration of crawl delay and Robots Exclusion Protocol
3. Enable sequential processing of responses (second use case)
4. Implementation of services auditing politeness (third use case)
5. Enable configuration at runtime
6. Implementation of caching
7. Improve enforcement of politeness via measuring the download rate of a domain using HEAD requests
8. Improve enforcement of politeness via page popularity and alternative strategies
9. Enable clustering and distributed crawling
10. Enable automatic provision of statistics

2 Preliminaries

Our research and the problems we are trying to solve in this work are both heavily related to the World Wide Web. This paper assumes basic knowledge about TCP and the internet in general. We also assume the reader to be familiar with the basic concepts of HTTP and Java. This section provides an overview over knowledge areas related to our work.

2.1 Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is the foundation of communication for the World Wide Web. Whenever a user opens a homepage and therefore looks up data in hypertext, HTTP is used. Hypertext is basically text with so called hyperlinks, references to other text which the reader can follow. A web crawler behaves more or less the same as the average internet user. A robot opens a page, looks through the hypertext and eventually follows the contained hyperlinks[18, p. 1].

HTTP is based on request-response conversations. A client sends a request to the server expecting an appropriate response containing the requested resource or an error message. A HTTP message consists of a header and a body (Listing 2 shows a sample response containing both header and body separated by an empty line). The message header contains meta information like request method, path, protocol, status code and an additional set of HTTP header fields. Whereas the body can be omitted, depending on the request method used (Listing 1 gives an example for a *GET* request without a body). The HTTP protocol defines a number of methods to define the action to be performed on the target resource. Following are the, for this paper, most important methods[18, pp. 51-55]:

- **GET** - Requests a representation of the target resource usually without mutation of any data.
- **HEAD** - Requests the same resource as *GET*, but without the response body, to only fetch the HTTP response headers.
- **POST** - Requests that the server stores the entity enclosed in the request body under the given URL.

```
GET /index.html HTTP/2
Host: twitter.com
Accept: */*
```

Listing 1: Sample HTTP request

```
HTTP/2 200
cache-control: no-cache, no-store
content-encoding: deflate
content-type: text/html;charset=utf-8
date: Thu, 15 Aug 2017 05:59:09 GMT
expires: Tue, 31 Mar 1981 05:00:00 GMT
last-modified: Thu, 15 Aug 2017 05:59:09 GMT

<!DOCTYPE html>
<html lang="de">
  <head>
  ...
  </head>
  <body>
  ...
  </body>
</html>
```

Listing 2: Sample HTTP response

The header part of a message does also contain a list of partly standardised HTTP headers, each defined in a new line, after the first to lines of a request, or respectively the first line of a response. These headers are used to transport additional meta information over HTTP. As the example response message given in Listing 2 shows, these fields are used to define the content type and the encoding of the message. However not all header fields are standardised and principally a message can contain any text as key-value pair [18, pp. 100-112].

2.2 Uniform Resource Locators

Uniform Resource Locators (URLs) are a specific type of Uniform Resource Identifiers (URIs) used to uniquely identify resources on the web. Each URL references a resource by specifying its location and the protocol to retrieve it. A typical URL has the form *http://www.google.com/index.html*, indicating the protocol (HTTP), the hostname (www.google.com) and the path (index.html). However the generic scheme of an URL is of the form *scheme:[//host[:port]][/path][?query][#fragment]*. The scheme in most crawling use cases is HTTP or HTTPS. The host may be an IP address or a hostname. The path has the same format as a typical filepath, compromising of a hierarchy separated by slashes. Non-hierarchical data can be sent through a query string, which is separated by a preceding question mark. A fragment is separated from the preceding party by a hash, identifying a specific secondary resource, like a heading of an HTML document[5, pp. 1-18][4, pp. 3-12].

Each resource on the web can be uniquely identified through an URL, yet different URLs may identify the same page. Two URLs for instance identify the same webpage although being different through their fragment or query part. These duplicate identifiers This would result in duplicates decreasing the effect of any caching efforts. Levene. et al provides a set of operations and steps to map different URLs onto canonical forms, which is illustrated by Figure 1. He also emphasises to follow the set of canonicalization rules consistently, otherwise they will not have the desired effect[26, pp. 159-160].

1. Convert the URL to lowercase.

2. Remove any trailing "/".

3. Remove the anchor tag of the URL.

4. Encode commonly used characters such as " " or German umlauts.[2]

5. Remove default starting pages of URLs such as *index.html* or *index.html*. These are commonly used as the entry point of a website.

6. Remove any path navigation operators such as "." or "..".

7. If the URL does not contain a special port remove port 80.

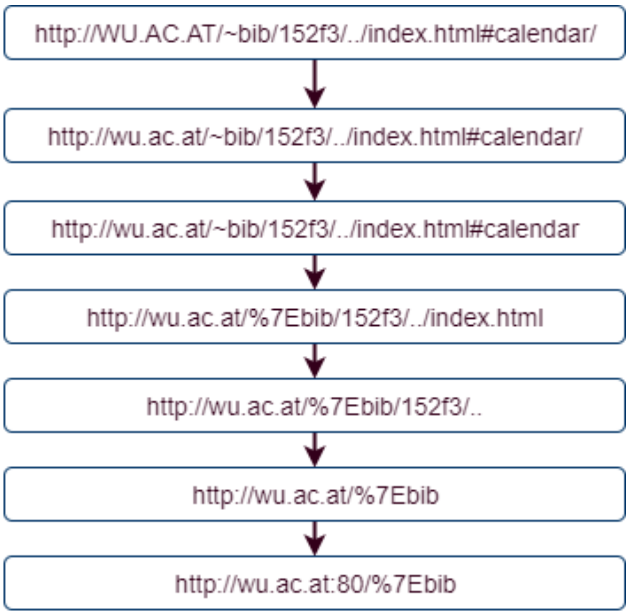
http://WU.AC.AT/~bib/152f3/../index.html#calendar/
http://wu.ac.at/~bib/152f3/../index.html#calendar/
http://wu.ac.at/~bib/152f3/../index.html#calendar
http://wu.ac.at/%7Ebib/152f3/../index.html
http://wu.ac.at/%7Ebib/152f3/..
http://wu.ac.at/%7Ebib
http://wu.ac.at:80/%7Ebib

Figure 1: How a URL is transformed through the steps of canonicalization.

2.3 Caching

Among the possibilities to improve the performance of information systems, almost nothing is as effective as the implementation of caching. A cache is a temporary storage of web objects, such as HTML pages and XML documents, for later retrieval. Caching is claimed to have several advantages[13, pp. 1-2][42, pp. 36-37][20, p. 91][33, p. 6]:

1. Reduction of consumed bandwidth because fewer requests and responses are sent over the underlying network infrastructure.

[2]`https://www.w3schools.com/tags/ref_urlencode.asp`

2. Reduction of the server load because duplicates are not fetched more than once, but are cached in beforehand resulting in fewer requests to handle.
3. Reduction of latency because cached objects are immediately available.
4. Higher Politeness due to reduced amount of traffic between source and target domain.

Using a cache also has some drawbacks, mainly the potential return of an already outdated object stored in the cache instead of obtaining the current version from the source. To avoid this inconsistency it must be defined which data is allowed to be cached and for how long it can be cached. Moreover, using a cache increases latency in case of a cache miss. At first every request is checked if it can be serviced directly from the cache, if the data is not cached already, it must be retrieved after the cache lookup. Therefore a high as possible hit rate is favourable to avoid the increased latency of a cache miss, whereas the hit rate is the ratio of requests satisfied by the cache [33, p. 6].

A client may want to explicitly request a freshly retrieved page bypassing eventual caching mechanisms. Normally this is done by including the header field *Cache-Control* in the HTTP request header. It can be used to specify directives for caches along the request/response chain. Allowing a client to specify the conditions under which he accepts a cached response or a server to declare if and how long a response is allowed to be cached. The field supports various different directives, each leading to different caching behaviour. However, as our solution takes the server's part when communicating with our crawling clients we will only consider the relevant Request Cache-Control directives [17, pp. 21-28]:

- **max-age=<seconds>** - The client only accepts a cached response whose lifetime does not exceed the specified number in seconds, except if max-stale is also given.
- **max-stale=<seconds>?** - The client is willing to accept a cached response whose freshness lifetime does not exceed the specified number in seconds, or generally accepts it if no number is specified.
- **min-fresh=<seconds>** - The client only accepts a cached response whose freshness lifetime is no less than its age plus the specified number in seconds.
- **no-cache** - The client only accepts a cached response if it is validated again beforehand.

- **no-store** - The cache must not store any information about the request or the response in non-volatile storage and make a best-effort to remove it from volatile storage afterwards.
- **only-if-cached** - The client only accepts a cached response, or a response with the 504 (Gateway Timeout) status code.

According to HTTP/1.1 the freshness lifetime of a response is defined as the time between its generation by the origin server and its expiration time. This expiration time can be specified through the *Expires* field in the response header containing a date in HTTP time stamp format (or as Java date and time pattern string: "EEE, dd MMM yyyy HH:mm:ss z"). If this date lies in the past, the response can be considered as already stale, although it can be cached because the expiration time is ignored if a client uses the *max-stale* directive. An alternative is the usage of the *max-age* directive in a response, here the specified number of seconds dictates the lifetime of a cached object[17, pp. 10-27].

3 Related work

The basic architecture of web crawlers has already been described in detail in a number of papers. Henrique et al. (2011) and Dixit and Sharma (2010) explain the basic structure of a web crawler and what the necessary components are. That a basic implementation consists of the crawler itself, an URL extractor or parser, an URL database and an interface for clients to retrieve collected data.

Bouras et al. (2005) and Cho and Garcia-Molina (2000) describe in brief the problems according to politeness and how to solve this issue by setting appropriate crawl delays. Crawl delays are the most common form to ensure politeness, also mentioned by Shkapenyuk and Suel (2002) and mentioned in the work of Dill et al. (2002) and Castillo et al. (2004). Whereas Thelwall and Stuart (2006) provide a brief overview of the four types of issues raised by web crawlers lacking politeness measures.

However most details about implementation and design of commercial web crawlers are not public. The most influential systems today, the systems used at Google, Microsoft or Amazon are among them. However there are some high sophisticated crawlers, whose structure has been made public. Like the work of Boldi et al. (2004) to implement the scalable and fully distributed web crawler called UbiCrawler. However the most popular crawling implementation in literature is the IRLbot by Lee at al. (2009), which is fully scalable up to 6 billions of web pages and beyond.

The objective of our research goes beyond the task of coordinated web crawling, be it distributed, scalable or not. There has little to nothing work done, regarding the enforcement of politeness not by the crawling system itself, but through a dedicated and transparent component in the form of a proxy or web services, depending on the use case.

4 Politeness

In this section we explain the term politeness and why it is necessary for the efficient operation of a web crawler. We also describe the constraints and dangers if crawling without being polite. There are several ways and protocols which are used to enforce politeness like the Robots Exclusion Protocol, defining rules and restrictions for web crawlers. Additionally we summarise other strategies and algorithms like PageRank to improve politeness. The section also deals with another important feature of our research, caching. The caching of retrieved web pages can improve performance and politeness significantly if used in an environment with a higher number of clients. However caching also comes with its own rules and directives to follow, defined by convention and RFCs, which are also described below. Last but not least we explain the different groups of use cases, our solution has to handle and how they differ from each other.

4.1 Politeness Issues

The never ending expansion of the internet does not only mean that the vast number of websites and domains is growing. It also results in much bigger domains. A crawler has to perform more requests to grasp all pages or to keep the already crawled data up to date. On the other side network resources and bandwidth are limited goods. Popular domains usually can cope with the amount of traffic generated by a greater number of users at the same time. They can also handle the additional traffic generated by web crawlers. However there are many domains out there not backed by this amount of costly infrastructure.

At the beginning of the history of web crawling, crawlers were written by computer science researchers, which were aware of network and computing characteristics and who were capable of the crawling impact. Today, however, crawlers are easily accessible for a wide range of people, and are not solely restricted to academic and research areas. Individuals and commercial users may not be aware of some issues coming with falsely configured and operating crawlers. Many small and private crawlers lead to medium use of network resources, but in higher number. Additionally there is no form of contract or convention possible to protect domains against abuses [40, p. 1774].

In general Thelwall and Stuart (2006) describe four types of issues raised by web crawlers, irrespectively if they are operated by research or commercial institutions or by private persons. These are:

1. Denial of Service
2. Cost
3. Privacy
4. Copyright

Denial of service, not be confused with the criminal pendant, describes the reduction of the availability of an entire information system or specific services. Servers that are busy processing incoming requests from web crawlers may not be able to respond to regular users because capacities are exhausted, or the experienced latency of regular users is much worse. Therefore it is possible for web crawlers, unintended or not, to financially harm target domains, or at least to harm the reputation of the owning institution. The second issue, cost, plays much less of a role now, than in the last decades. Missing bandwidth is not that a critical problem anymore. Almost every host offers plans with unlimited download traffic per month, lessens the impact additional traffic of web.

However privacy and copyright, the last two issues, are closely coupled and are currently gaining much more attention. Not every information on the Web can be used without consent of its owner. Most crawlers are saving data without checking for copyright issues or asking if the owner of a crawled domain for approval. Domain owners can take some counter measures to signal web crawlers to do not crawl a domain. In hindsight of changes throughout the last couple of years, we will concentrate on the two most important issues: copyright and denial of service [40, pp. 1773].

Even small scripts running on a common notebook can issue requests at an enormous rate. Simple loops can issue several thousand requests per minute. If these requests get not served by a cache, but result in database look ups or heavy computing, a crawling process can become a denial of service attack. Especially archives consist of a big amount of pages, while at the same time are not frequently used by a high number of users, making big infrastructure unnecessary. Poor network performance is not the only reason for crawlers to get blocked. Bigger institutions and companies protect their networks with Intrusion Detection Systems and Firewalls. These will block a source address automatically if it would issue hundreds of requests in parallel. Therefore it is quite common for such ill-behaving crawlers to get blocked by the target domain. This, of course, makes further crawling impossible for a specific time, ranging from a few hours to being blocked forever[11, p. 10][37, p.7].

This can be avoided by throttling the rate a crawler performs its requests with. However this can lead to starvation of the crawler itself. If the crawling rate gets throttled too much, the amount of URLs to be crawled for a specific domain can be overwhelming, leading to buffer and memory overflows due to congestion. Additionally a rate that slow would render every crawler useless. Such a crawler, performing one request per minute, would need over two months to crawl a domain like our university's (wu.ac.at) which has over 98,200 pages currently indexed by Google according to northcutt.com[3]. A crawler hindered by its own politeness restrictions is then faced with a decision to discard a fraction of its workload, ignore its restrictions or to suffer significant performance down break. Not a single one of these options is particularly desirable. [25, p. 3] Politeness, therefore, is basically a trade-off. The rate at which requests are done must be balanced between not exhausting the target domain and performing at an acceptable speed. Our solution has to consider this trade off to do not get blocked on the one side and to do not become a bottleneck for the respective clients on the other. Therefore strategies must be found, solving these issues.

4.2 Crawl delay

The most fundamental form of implementing politeness is the crawl delay. It is the time span between two requests, or between two requests issued to the same domain. This ensures that the resources of the target address are not used in an exhaustive way. The delay itself can be set taking several parameters into account, ranging from simple guessing to highly mathematical

[3]`https://northcutt.com/tools/free-seo-tools/google-indexed-pages-checker/`

approaches. Our implementation should support a variety of these strategies to be flexible enough to react to changed requirements or changes of whatever reason. To have basic politeness restrictions it should be possible to configure initial delays for domains and a default delay, if the delay for the current domain is not specified.

The delay must be set with special care. As already described, a delay set too low can render the whole mechanism useless and a delay set too high can slow done every crawling procedure. The delay used in literature varies a lot. Some authors experienced no issued using a delay of only one second [16, p. 215][41, p. 12], others have used 30 seconds and more to do not run into any problems [1, p. 5]. However, even back in 2004 the anecdotal evidence from access logs showed that delays are varying between 20 seconds and several minutes. [10, p. 3] Therefore we will use a delay of 1 second here, simply because it seems like a suitable trade-off between speed and safety. This delay is taken into account if two requests will hit the same target domain, if these are different, we draw the conclusion that both domains are independent, making immediate crawling possible.

To cope with the never ending expansion of the Web itself, crawlers became heavily distributed in the past. This allows institutions like Google to open thousands of connections simultaneously. However not only should each part node in such a system should enforce a certain delay between two requests to the same host, such a system should never open more than one connection at a time to the same host in general [10, p. 4][8, p. 3][29, p.1][28, p. 1].

4.3 Robots Exclusion Protocol

Politeness does not only describes a crawler avoiding to use network resources in an exhaustive way. At some point of time a mechanism was needed to overcome the copyright issue described above, and to give domain owners the possibility to restrict page access and control web crawlers accessing their sites in general. This was the birth of the Robots Exclusion Protocol on 30 June 1994 on the robots mailing list, between the majority of robot authors and other people with an interest in robots. The protocol itself is not a official standard per se, it is also not owned by anyone and it is also not enforced. It is an informal agreement between domain owners on one side and web crawler owners on the other. The 'exclusion' part of the name also hints on the reason for the protocol. Back in 1993 and 1994 the number of web crawlers was increasing, whereby a part of crawled sites where they weren't welcome. So a solution had to be found to exclude robots from areas or in extreme cases from whole domains [34].

The proposed way to declare instructions for web crawlers is a simple file with media type "text/plain" accessible via HTTP under the root path of the domain that the instructions are to be applied to. A famous example for such a location would be *www.google.com/robots.txt*. If such a file can be found, the contained instructions must be parsed and followed accordingly. If such a file cannot be found a robot can principally visit every site of the domain it comes across, without any limitations (although a delay should be implemented, no matter the access restrictions). If the response code is indicating access restrictions (HTTP status code 401 or 403) the robot should regard access to the site as completely restricted. If the retrieval has failed because of a temporary failure the crawling process should be postponed unit the *robots.txt* can be retrieved [23, pp. 2-3].

According to the original Internet-Draft of December 4, 1996 the protocol consists of three different keywords, *User-agent*, *Allow* and *Disallow*. Whereas the file itself consists of several blocks, separated by *User-agent* directives. This allows the author to define directives only for web crawlers matching the given name, or any crawler if the wildcard character '*' is used. Every line of a *robots.txt* file has the same structure: <Field> ":" <value>. The Allow and Disallow lines indicate whether a robot is allowed or disallowed to access a URL matching the given path. Whereas a Allow directive is only needed if one wants to unblock a URL of a otherwise blocked parent directory [34] [23, pp. 4-6].

A *robots.txt* file is a simple text file, without any validation measures, compile time checks or similar. Therefore it is possible that more than one path would match, or both *Allow* and *Disallow* is defined for the same path. However the protocol states, that the most exclusive rule should take effect. Therefore our implementation has to parse and check all defined rules for a given path, until a *Disallowed* rule is found. Only if no such rule can be found and there is no rule left to evaluate, then the path is allowed to be crawled.

```
#Google Search Engine Robot
User-agent: Googlebot
Allow: /?_escaped_fragment_
```

```
Disallow: /search/realtime
Disallow: /search/users
Disallow: /*?
Disallow: /*/followers

# Every bot that might possibly read and respect this
  file.
User-agent: *
Allow: /*?lang=
Allow: /hashtag/*?src=
Allow: /search?q=\%23
Disallow: /search/realtime
Disallow: /search/users
Disallow: /search/*/grid

Disallow: /*?
Disallow: /*/followers
Disallow: /*/following

# Wait 1 second between successive requests. See
  ONBOARD-2698 for details.
Crawl-delay: 1

# Independent of user agent. Links in the sitemap are
  full URLs using https:// and need to match
# the protocol of the sitemap.
Sitemap: https://twitter.com/sitemap.xml
```

Listing 3: Excerpt of the robots.txt of www.twitter.com Accessed: 25.07.2017

The initial draft only declared three different directives. As years were going by this proved as not sufficient anymore, authors needed to share more information with web crawlers to improve efficiency. Listing 3 shows the most common protocol extensions additional to the original ones. The excerpt contains directives for Googlebot, the name of the robot used by Google, and different directives for all other web crawlers, each section coming with its own Disallow and Allow directives. Nowadays it is not uncommon to include comments (lines starting with '#') to simplify versioning and administration.

One way to overcome the issues of predicting the correct delay between requests while crawling is to use the one declared in the *robots.txt* if it exists. The *Crawl-delay* directive is also an extension of the protocol and it is not guaranteed that every crawler respects it. However it can be used to provide a hint for the desired crawling delay, given in seconds a crawler should wait between two requests. The *Crawl-delay* directive is not declared in the majority of cases, but if it is given it should be used. This also holds true for our proposed implementation of politeness enforcement [39, p. 1124].

A text file is not the only way to define directives for crawlers accessing a domain. One can use *<meta>* tags inside of web pages to control (misbehaving robots can ignore this too of course) if the respective page will be indexed. Such a line of HTML code has the following format: *<META NAME="ROBOTS" CONTENT="NOINDEX,NOFOLLOW">*. A polite crawler must parse these meta lines and use the more restricting directives if there are collisions with the *robots.txt* file. The format is not standardised. Therefore upper and lower case and also whitespaces should not make a significant difference. The name attribute identifies the target web crawler, or affects all web crawlers if *ROBOTS* is used. The content attribute specifies the commands. Some of the possible values are [21][38]:

- NOINDEX - prevents the page from being included in the index
- NOFOLLOW - prevents the links found on the page from being followed
- NOARCHIVE - prevents the page from being cached

To parse these *<meta>* tags it is necessary to read the relevant parts of fetched web pages. Our implementation will not perform any content or URL extraction. This is completely done on the client side. Content extraction would furthermore increase the experienced latency significantly. Because of these, and the spare usage of this way to declare directives our implementation will not obey to the rules defined directly in web pages.

4.4 Sitemaps

Another common used extension to the Robots Exclusion Protocol is the *Sitemap* directive. Such a directive points to the location of a file of the Sitemap Protocol. These XML files holds a list of URLs and optional additional meta data that a crawler may index. This is especially useful if sites of a domain are not linked together, like archives or newspapers, or if a domain becomes really big, then it can be possible that crawlers will miss updates or whole parts to index [12, pp. 691-692][35, pp. 991-993]. Google also made mentionable extensions here to improve the search quality. For e.g. Google News it is possible for newspapers to enrich their sitemaps with respective metadata to improve user experience. The *robots.txt* of the British newspaper BBC points to a dedicated *sitemap-uk-news-1.xml* sitemap[4], whose excerpt is given in Listing 4.

```
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
        xmlns:news="http://www.google.com/schemas/sitemap-news/0.9"
        xmlns:video="http://www.google.com/schemas/sitemap-video/1.1">
  <url>
    <loc>http://www.bbc.co.uk/news/world-europe-40713813</loc>
    <lastmod>2017-07-25T06:45:34Z</lastmod>
    <news:news>
      <news:publication>
        <news:name>BBC News</news:name>
        <news:language>en</news:language>
      </news:publication>
      <news:publication_date>2017-07-25T06:45:34Z</news:publication_date>
      <news:title>
        Pope Francis shuts off Vatican fountains amid Italy drought
      </news:title>
    </news:news>
  </url>
...
</urlset>
```

Listing 4: Excerpt of a sitemap of BBC dedicated to Google News *http://www.bbc.co.uk/sitemaps/sitemap-uk-news-1.xml* Accessed: 25.07.2017

4.5 Different strategies

A well-behaving robot respects the Robots Exclusion Protocol, also in its extended forms, by taking the *robots.txt* into account, and by using a reasonable crawl delay to do not abuse network resources of the target domain. There are several strategies to depict such a crawl delay. The dominant strategy here is to simply use the delay given in the *robots.txt*, it is not used that often, so one should have had a reason to write it in there. However there are also more or less sophisticated strategies mentioned in literature.

The most popular indicator used in the field of web crawling is the PageRank algorithm. Developed in the late 90's by Page et. al, this method is used to calculate the popularity and importance of a website. Websites are always linked to each other, these links are also used by web crawlers to identify further pages to crawl and navigate through the Web. The score of a page depends recursively upon the scores of the pages linking to it. Whereas the score itself depends on the number of links itself, and the number of different linking websites [32, pp. 1]. The assumption now is, the more important a page or domain is, the more often it changes, the more often it gets viewed, the faster it should be crawled. PageRank is used primarily to order and prioritise URL queues in crawling systems. The score defines the position of a URL inside such a queue. This score is also important for many applications performing incremental crawls, because it can give a hint about the change

[4]`http://www.bbc.co.uk/sitemaps/sitemap-uk-news-1.xml`

rate of such pages [22, p. 490][31, pp. 187-188][36, pp. 27-28][27, p. 49][2, pp. 895].

However the score calculated through PageRank can also used to assume the network capacity of a domain. The more popular a page is, the more request it will handle probably, because it has more users. This might not be true in all cases, although it may be a valid assumption. Therefore a crawler can use a smaller delay between requests the higher the score. For pages as popular as google.com or amazon.com, a crawler might even use no delay, because these domains deal with millions of users every day. The additional traffic generated through a web crawler will not make a real difference here. However to use PageRank as a method to depict crawling delays, a system has to calculate the relevant scores beforehand, rendering it only a useful strategy if the crawler is an incremental one, also improving its PageRank scores with each crawling cycle.

Not only the popularity of a page might be relevant for its network capacities. Also the download rate, or bandwidth can be an indicator for the overall capacity of a target domain. The download speed can easily be calculated by dividing the size of downloaded page through the time needed. Calculating transfer rates this way also includes network latency. So the rate also depends on the current state of the network and the current traffic, both influences latency. The download speed then can be used to classify domains and to use suitable crawl delays for each group. Furthermore it improves the speed of a crawler if fast loading sites are favoured and more frequently requested than sites from slow domains, enhancing the output of a crawler. A slightly different approach is it to take a crawl delay proportional times to the last measured downloading time as done by Najork et. al[29, p. 9-11]. This ensures that a crawler only uses a defined fraction of the target's server resources. Overall we assume the measured download speed to be a valid indicator for request handling capacity [31, pp. 187-188][15, pp. 292-293].

4.6 Caching

We expect that the implementation will be used by a higher number of independent crawlers potentially targeting the same domains at the same time, or at different points of time. Caching would improve politeness not only of clients, but also of the system itself. Politeness should also include sending as few requests as possible. If now two clients request the same page the second one can be served from the cache. This also reduces the overall traffic and hardware and network resources needed. The effectively of the cache itself completely depends on the hit rate. If the domains crawled do not overlap, then the cache will only serve a small number of requests or, in extreme cases, can be rendered useless when there is some kind of crawler side coordination.

Although cache controlling is principally possible through HTTP header fields it is questionable if a web crawling system should respect all directives declared by RFC 7234. If a server response contains a *Expires* header field specifying a date lying in the past, or a *Cache-control* field containing *no-cache* and *no-store*, should a cache really discard such a response? Principally this decision should be made on the web crawler or client side. The client should decide how old a response is allowed to be until it is not accepted. The accepted age of cached pages is depending on the use case. However web crawling always just captures a snapshot of the part of the Web that is crawled. The interesting question here is, if a cache should discard directives disabling caching if two identical requests are received in a short time.

The decision whether a cached page is acceptable or not is the client's one. Therefore our solution should respect all standard Request Caching Directives, including *no-cache*. This will disable any caching efforts and may result in less overall throughput. However, a client relying on real time responses would otherwise not be served correctly. Regarding Response Caching Directives our cache will ignore the *no-cache* and *no-store* directives in favour of better performance and more politeness. Instead a response containing these directives will be considered fresh for 1 day, which we consider a good trade-off between freshness and performance, because a client relying on real time responses can use the *no-cache* directive.

4.7 A variety of use cases

Our solution should offer all these features described above. It should enable caching and also respect the specified header fields and directives defined by RFC 7234. Nonetheless the most important task is the enforcement of politeness. So we have to respect the Robots Exclusion Protocol and eventual domain specific politeness constraints. All these features must not only be implemented for one specific use case, developed against interfaces defined beforehand, but our solution has to handle a variety of use cases, significantly different from each other.

Generally the potential use cases can be divided into three groups, mainly based on the level of maturity in terms of web crawling, the user group and relevant special requirements, which is also illustrated by Table 1. This differentiation leads to the following groups:

- Simple and unsorted scripts
- Clients demanding sequential processing
- Clients demanding politeness services

4.7.1 Simple and unsorted scripts

The first group, simple or unsorted scripts, summarises all clients without special requirements and low maturity. Such clients may be implemented in almost every language and framework. Often they are just prototypes, proof of concepts or temporary endeavours. Mostly the crawling component itself is a loop structure, iterating over a set of URLs. However this simple implementations can cause the most critical problems, scaling with the amount of clients running in parallel and uncoordinated or unsorted. They, mostly unintended, consume server resources in an exhaustive way, because politeness is barely implemented paired with a potentially high request rate. In some cases it is even possible to send thousands of requests in a couple of seconds. This can and will in most cases be interpreted as denial of service attack and the crawler and it's whole domain will be blocked. Most of the domains out there have special Intrusion Detection Systems and also most Firewalls will automatically block such clients.

Clients of this first group are often run by students or users not aware of network and computing characteristics and which problems can arise if certain constraints are not considered. However, it is not possible to ensure that all clients inside a domain operate in a polite manner. Therefore politeness must be enforced in a central way before requests are able to reach its targets. This can be achieved by using a proxy. Requests are routed through our proposed implementation of a proxy enforcing politeness checking if a request violates eventual politeness restrictions. These restrictions can origin from the Robots Exclusion Protocol, are configured manually or calculated at runtime. If a request gets rejected, a special error code is returned, depending on the reason. If it fails because there are currently to many requests targeting the same domain the status code 429 *Too Many Requests* is returned, including information when to try it again. 403 *Forbidden* is returned for every request violating constraints defined by the Robots Exclusion Protocol. If the request passes all checks it will be forwarded to the target and the retrieved content is sent back to the client.

4.7.2 Sequential processing

Some implementations of crawlers require sequential processing. These clients issue a number of requests in parallel and expect to receive the responses in that exact order. However the serving time of a request is inherently unpredictable. The first request may take a few seconds to be served, while the response for the last request sent is instantly available. Such a scenario would lead to errors on the client side, which our implementation should avoid. This adds an additional complexity to the internal scheduling of our implementation, because it has to enforce a specific ordering of responses belonging to the same client. Furthermore the connections between client and server have to stay open a configurable amount of time. This timeout may be configured whether at application startup or by including a dedicated header field in the HTTP request.

One kind of clients belonging to this use case are scripts executed from clients implemented with XQuery and XSPARQL. XQuery is a query language applicable across many forms of XML structures [7]. Whereas XSPARQL is an extension of XQuery to query and format RDF (Resource Description Framework, which allows to represent data independently of the original language) structures [14, pp. 114-117][6, pp. 147-148]. These clients are used to crawl archives for open data and the semantic web. They also basically crawl by iterating over a list of URLs, but here it may be necessary that the responses come back in the same order as the requests were issued.

4.7.3 Politeness services

The last use case our solution has to handle are mature, specialised crawlers and crawling systems. These are already based on an internal client/server like architecture. Whereby the client sends a set of seed URLs to the server, or in some cases an internal queue. This queue internally manages the scheduling of the crawling process. Dependent on the requested domains and the requesting clients there are several processes active. Although, mimicking a client/server architecture, both sides are currently implemented altogether in one application. However this use case is the one coming closest to our proposed implementation of a central service handling requests of clients differing in their needs and implementation.

The already existing crawlers and crawling systems should be enhanced by politeness measures, executed on a client domain wide level. A proxy would not suit the special requirements because the request handling and

Characteristic	Unsorted	Sequential	Service
Maturity	low	middle	high
Users	average	prof.	prof.
Number of Clients	high	low	low
Response	Content	Content	URL
Politeness implemented	-	-	maybe
Order of processing	-	required	-
Batch	-	-	yes

Table 1: Overview of the characteristics for each group of crawlers

crawling should be done on the client side. Our proposed solution for this problem is to offer REST services for these clients to optionally check and enforce politeness. REST (Representational state transfer) services allow systems to communicate in a uniform, textual way over HTTP[19, pp. 76-106]. Before crawling the extracted links a crawler will send every URL, or a set of URLs to our implementation where each URL is checked if any politeness constraints are violated. The URLs passing all checks and obeying all rules will be sent back to the client. Additionally a client can request additional information about why an URLs is denied for crawling and when to crawl it. That allows for more efficient interaction between server and client, which can take different measures according to the additional data provided by the service.

5 Implementation

In this section we provide an architectural overview of our proposed solution and implementation. We explain why we have chosen a specific set of frameworks and libraries. The biggest and most complex parts of our solution are the modules handling politeness and caching. Therefore we go into very detail here. We also provide an overview of the work done to implement configuration at runtime and monitoring of the system and the various caches used. The complete source code can be found online on GitHub.[5]

[5]`https://github.com/PatrickRi/Zuul_proxy`

5.1 Architecture

Our solution has to handle various web crawling use cases, and therefore has to provide a number of endpoints and features. The implementation itself consists of various parts, each responsible for a specific task. Figure 2 provides an overview of the system's architecture. We aimed for a modular solution to make it easy to change single modules or include new ones. Figure 2 shows the three different use cases. Each one has different conversation schemes with differing requests and responses. Whereas the group of queues even has dedicated REST endpoints they talk to. The main system itself can be divided into three parts. The proxy component serves the first two use cases, unsorted scripts and clients requiring sequential processing. The proxy checks if incoming requests obey the relevant politeness constraints and returns the response of the destination. This is also the part utilising caching. The second part, here called *Services* consists of the various endpoints to submit a set of URLs which get checked. A client can choose how the URLs can be submitted, via request body or as query parameters.

Additionally there is a dedicated endpoint providing verbose output. Using it provides the requester with detailed information about why a URL violates politeness constraints and when to crawl a URL if it got rejected. The third part summarises the different REST and JMX (Java Management Extensions) endpoints to monitor the components. Most of these endpoints are provided out of the box by our chosen frameworks and libraries. Whereas the Java Management Extensions provide a way to directly communicate with dedicated Java objects, providing data and operations. These are usable through *jconsole* located in *JDK_HOME/bin* for instance, to visualise the metrics of the heavily instrumented JVM and additional providers [24, pp. 104-105].

Overall we are using four different caches, each serving a special purpose. The most obvious use case for a cache is the caching of server responses. Each successful response is cached in the *Web Page Cache*, together with all HTTP headers and additional metadata. To do not lose any configuration about crawling delays, whether they were configured manually, calculated or read from the *robots.txt*, we save it in a persistent cache. This *Configuration Cache* is loaded automatically at start, very much like a database. When crawling a domain by requesting hundreds of sites it would be superfluous to fetch the *robots.txt* every single time. Therefore every fetched *robots.txt* is saved for 24 hours in a cache, and after that it gets refetched on demand. The *Politeness Cache* does not primarily serves a storage purpose. We used

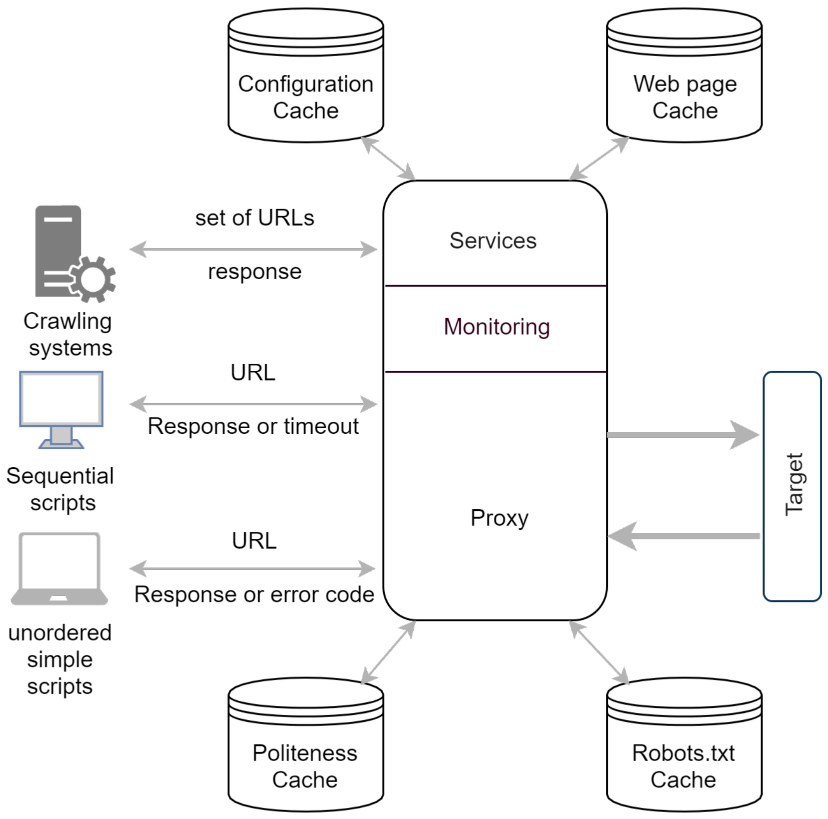

Figure 2: Architectural overview

the features of an Infinispan cache to determine if a request was issued to the same domain inside a time frame smaller than the currently active crawl delay for that domain.

5.2 Technology

For the implementation we have chosen Java as programming language, because of our years long experience with it. To build the needed services and proxy functionality we have chosen Spring Boot.[6] We decided against similar frameworks and libraries like Apache Camel or Retrofit or standard TCP sockets. Spring Boot gives us the possibility to easily create stand-alone Spring based Applications. Spring itself is one of the biggest web frameworks and one of the biggest frameworks in the Java world itself.

It is completely based around the idea of inversion of control and aspect

[6]`https://projects.spring.io/spring-boot/`

oriented programming.[7] Both makes it easier to decouple the single parts of systems, leading to improved maintainability, expandability and flexibility. IoC, also known as dependency injection, is a process whereby dependencies like constructor arguments are not statically wrote inside the code, but an object just defined these dependencies. If now another object wants to use it, the Spring IoC container injects those dependencies when creating the desired object. Spring is shipped with an optional embedded server like Tomcat and Jetty, therefore a dedicated web server is not necessary, the server is started by the application itself. Bootstrapping of new projects is also much faster, due to the the various Maven 'starter' POMs, declaring the dependencies needed to work with the chosen Spring project. The starter *spring-boot-devtools* for instance provides us with all dependencies needed to automatically restart the server at classpath changes, live reload and remote debugging. Spring provides a large range of modular projects, very much like a construction kit. From security, over REST to monitoring, it is just needed to add the respective starter project to get the dependencies to implement the functionality.

Our proposed solution is based around the idea of a proxy and additional web services. One popular proxy solution is Zuul originated at Netflix. [8] Usually it is used as a gateway service, routing REST requests, based on different filters. The four filter types correspond to the lifecycle of a incoming HTTP request. *PRE* filters are executed before the routing of requests, *ROUTING* filters handle the routing itself. *POST* filters are executed after the routing, whereas *ERROR* filters handle errors which occurred during the process. Based on that, it is possible to implement a proxy using the idea of filters, applying these on incoming requests and enforce politeness that way. Netflix open sourced a number of their projects in the last years. Some of these projects were completely included in a dedicated Spring Project, Spring Cloud. This allows us to use the functionality of Zuul, together with the features of the Spring framework without any extra effort. Furthermore it is shipped together with a set of default filters for debugging, error handling and simple routing.

Based on the idea of handling incoming requests with filters, we wrote our own *ROUTING* filter, shown in Figure 3. It is responsible for caching, enforcement of politeness and response handling. For each incoming request

[7] `https://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html`

[8] `https://github.com/Netflix/zuul`

the filter checks if the destination of a request is the proxy itself. If this is the case a empty response is sent back to prevent endless loops because the request would be forwarded to itself, without any breaking condition. The next step is the check if the requested resource is already cached obeying eventual *Cache-Control* directives, if these checks are passed then the cached result is sent back with the headers of the original response. If a request can not or should not be served from the cache, before forwarding the request to its destination, it has to obey the constraints defined by the Robots Exclusion Protocol. And as last step it is checked if the target domain has not been crawled already in the respective crawl delay. The filter then forwards the request to the destination and handles the server's response. The fetched page gets cached and various metrics get updated throughout the process. At the end a dedicated filter provided by the Spring starter handles the processing of the response and return it back to the client.

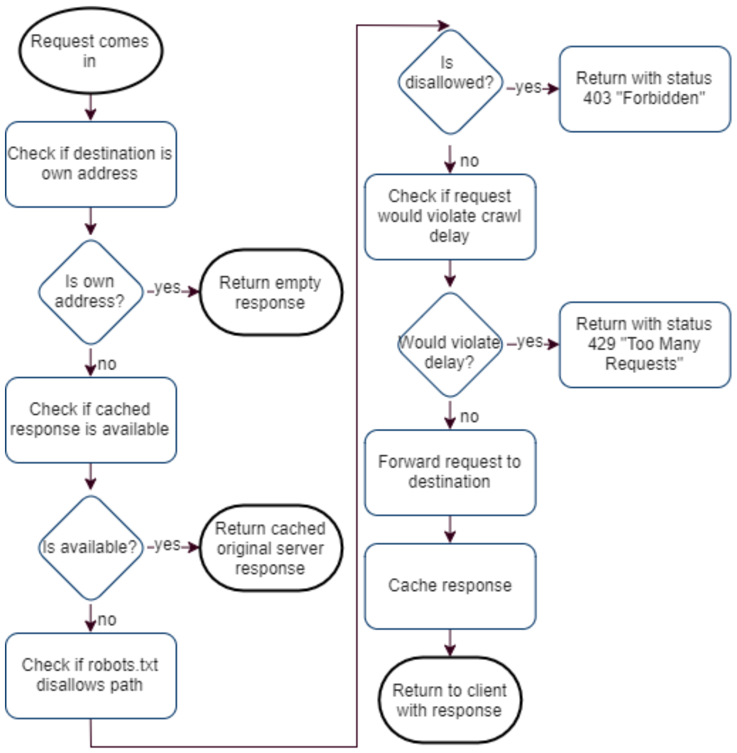
Request comes in
Check if destination is own address
Is own address?
yes
Return empty response
no
Check if cached response is available
Is available?
yes
Return cached original server response
no
Check if robots.txt disallows path
Is disallowed?
yes
Return with status 403 "Forbidden"
no
Check if request would violate crawl delay
Would violate delay?
yes
Return with status 429 "Too Many Requests"
no
Forward request to destination
Cache response
Return to client with response

Figure 3: Zuul filter managing politeness, caching and monitoring

5.3 Caching

We heavily rely on caches in various parts of our proof of concept. To reduce complexity and the effort needed we aimed at only using one cache implementation for all four caching use cases, as shown in Figure 2. Therefore we had quite diverse requirements on the caching implementations we evaluated. The requirements were as follows:

- Embeddable, without the need for a dedicated server
- Interoperatibility with Java or written in Java
- Persistable
- Distributable
- Automatic eviction and expiration

We aimed for a cache allowing both, to be run as dedicated server to share data between several instances, effectively being distributable, and also to be embedded to allow for fast prototyping. To keep the effort as small as possible the cache should be easily called and controlled by the Java code of our solution. To prevent data loss at each system restart the data should be persisted at shutdown or to prevent memory overflow. Furthermore the cache should offer a possibility to automatically expire entries when an entry exceeds its lifetime and should provide various metrics and statistics for monitoring. Looking at our requirements it was clear that database solutions like SQL databases or NoSQL and document based solutions like Cassandra or MongoDB do not suit our needs. This also holds true for Redis, which cannot be run in an embedded manner, it needs dedicated infrastructure to function properly. However there exist numerous other caching solutions in the Java ecosystem. During research the most popular ones seemed to be Ehcache[9], Caffeine (former Guava)[10], Infinispan[11] and the vanilla Java HashMap in various forms. We then evaluated this selection to get a best option to use for our implementation.

After adding the required dependencies and libraries all four alternatives are embeddable into a simple Java project without any need for a dedicated server. Moreover all alternatives are written in Java and therefore fully interoperable with it. Ehcache and Infinispan provide cache persistence out of the

[9]`http://www.ehcache.org/`

[10]`https://github.com/ben-manes/caffeine`

[11]`http://infinispan.org/`

Characteristic	Ehcache	Caffeine	Infinispan	HashMap
Embeddable	yes	yes	yes	yes
Java	yes	yes	yes	yes
Persistable	yes	manual	yes	no
Distributable	yes	no	yes	no
Eviction and Expiration	manual	manual	yes	no
Statistics	yes	yes	yes	no

Table 2: Overview of the characteristics for each group of crawlers

box by configuring the cache at creation time. This allows for overflow onto the hard drive and for reload from the persistence store after shutdown without losing data. HashMap does not provide such feature because data is hold in memory and for Caffeine one would need to write an extension to write entries into a file at eviction. Only Ehcache and Infinispan allow for easy distribution without much effort needed. However Infinispan does not need a dedicated server and can be operated fully distributed by using JGroups' discovery protocols to automatically discover neighbouring instance. For running Ehcache in a cluster, dedicated infrastructure is needed. Eviction refers to the process by which old or unused data can be dropped from the cache, or in our case can be written to a persistent cache storage. The only alternative fully offering eviction and entry based expiration if Infinispan. Ehcache and Caffeine would both provide similar features. However expiration would need to be implemented manually based on the data inside an entry. Infinispan allows to put entries into the cache altogether with an expiration time out of the box. To efficiently monitor and analyse the behaviour and performance of our solution the provision of statistics and metrics is mandatory. Except for HashMap, every alternative provides JMX beans and interfaces to fetch statistics like hit ratio, size and consumed memory. These findings are summarised in Table 2.

Based on our requirements and the assessment of four alternatives we have chosen Infinispan. Not only is it the only option supporting all of our requirements and we have previous experience using it. Distribution over bigger clusters can be easily achieved and eviction and cache persistence is also only a matter of a few lines of configuration. Statistics can be fetched via JMX and through predefined interfaces. The latter can be used to implement custom functionality or even extend the functionality of the monitoring of

Infinispan. Although benchmarks of the Caffeine developers [12] show that Infinispan does not provide superior performance, we will use it for all our use cases, because slightly better performance of other caches does not outweigh the offered features.

5.4 Web caching

After checking that the destination of an incoming request does not match the server's address, our implementation checks if a cached response can be returned. Caching improves the performance and reduces the workload for network resources, because requests can be served directly from memory. This also improves politeness, because each request not sent do not increase the current crawl delay for the target domain. However to correctly handle server responses and client requests our implementation has to respect the directives defined in the *Cache-Control*, *Content-Type* and *Expires* header fields. As already described above, these directives specify if a response is allowed to be cached, how long a response is valid, and what the client expects to retrieve and if a cached response will be accepted.

Every fetched response with status 200 *OK* will be cached. To respect the *Content-Type* directive each cache entry is uniquely identified by its URL and the content type it contains. If now a client requests a URL without specifying a content type it will receive a relevant cache entry without not associated with a content type. If a request contains a header field declaring a specific content type, it will only be served by the cache, if an entry associated with that specific content type is available. To improve hit rates we create various cache entries per response. One for the content type of the response, one for **/** and one without a content type. This is also considered at cache retrieval (Listing 5). If a content type was specified in the request only such a key will be looked up. Otherwise a cache lookup is done without a content type and if that fails the wildcard pendant gets returned, if available.

```
String url = new URL(requestURL).toString();
String header = request.getHeader(CONTENT_TYPE);
if (header == null || header.isEmpty()) {
  entry = cache.getEntry(new CacheKey(url));
  //Check if entry obeys Cache-Control directives
  if (!checkCacheControl(request, entry)) {
```

[12]https://github.com/ben-manes/caffeine/wiki/Benchmarks

```
    entry = cache.getEntry(new CacheKey(url,
      "*/*"));
  }
} else {
  entry = cache.getEntry(new CacheKey(url,
    header.replaceAll("\\s", "")));
}
//Check if entry obeys Cache-Control directives
if (checkCacheControl(request, entry)) {
  return entry;
}
```

Listing 5: Logic for cache retrieval

Every retrieved entry has to be checked if it obeys eventual constraints defined by the *Cache-Control* directives. Therefore we analyse the relevant header fields and extract the data to use it at cache retrieval. If a client request contains the *no-cache* directive the cache will be completely ignored. A cached entry may also be considered as irrelevant depending on the values of the *max-age*, *max-stale* or *min-fresh* directives, a request may contain. In such a case our implementation calculates the age or the freshness of the entry, dependent on the directive. However there are two ways to control the lifespan of a response. Both *Expires* and *max-age* can be used to specify when a page exceeds its lifespan and should not be considered at cache lookups anymore. We have chosen a restrictive strategy here and will take the more restrictive one if both are contained in the response to calculate age, staleness and freshness. If the age exceeds the *max-age* or the entry was stale for too long or the freshness time is less than requested the request cannot be served from the cache.

5.5 Robots Exclusion Protocol

One part of politeness enforcement is the Robots Exclusion Protocol. Each request which gets not served directly by the cache has to respect the Robots Exclusion Protocol. The relevant *robots.txt* file is fetched by taking the host part of the URL and appending */robots.txt*. If there is no file, every path of the domain is considered as allowed to be crawled. However if such a file can be fetched then each line is parsed into a directive which is then used later to check if a path is allowed to be accessed. Each *robots.txt* file is cached for one day to improve performance and avoid unnecessary traffic. After this timespan is has to be reloaded, to keep track of updates.

As Listing 3 shows, a file consists of several blocks, each starting with a line defining the user agent name (*User-agent*). Our proof of concept is named *wu-is-crawler*. Therefore our service just respects the block defining this name or, if no such block exists, the block defining the wildcard name ***. Each block then consists of a set of lines starting with either *Allow* or *Disallow*. A dedicated class extracts these lines and saves them for later retrieval to check if the requested path is allowed as defined by the constraints of the Robots Exclusion Protocol. This check is done by converting each extracted path to a regular expression. For performance reasons the extracted paths are ordered with the matching paths first followed by the most concrete ones via length of the specified path. Now we iterate over all matching blocks, and iterate again over the list of defined paths per block. This is done until a matching path is defined in a *Disallow* line or there are no paths left. The outcome is either true, which continues the process flow, or false, effectively leading to the abortion of the process.

5.6 Crawl delay

Besides the Robots Exclusion Protocol our politeness enforcement measures concentrate on crawl delays. Each domain is associated with a crawl delay specified in milliseconds. During that configured timespan it is not allowed to send requests targeting this domain. These delays are either specified by the Robots Exclusion Protocol via a *Crawl-delay* line, are configured manually or get calculated on the basis of various metrics. The delay can be configured manually in various ways as described below in the corresponding subsection. The configuration itself gets stored inside the dedicated, persist *Configuration Cache*. This prevents data loss at server shutdown and allows it to keep data for crawling delays over a longer timespan. Therefore it is possible to approximate delays also over several server restarts. The cache itself stores data objects holding the current delay, the corresponding domain, the manually configured delay and the delay specified in the *robots.txt*. We store these information in separate fields to hold track of the initially configured delay and eventual overrides of it.

To check if a request violates the crawl delay we use a handy feature offered by Infinispan. Infinispan allows for putting entries in it, altogether with an expiration time. In contrast to eviction, which means that unused entries are moved to persistent storage to prevent memory overflow, expiration means the removal of entries exceeding their lifetime, both from volatile and persistent storage. Initially we sketched complex solutions to keep track of outgoing requests and the relevant timestamps to control if a crawl delay constraint gets violated at every new request. This would have required complex data structures working with dedicated threads keeping track of storage eviction. The calculation of time series, searching the last relevant entry and calculating the difference to the current time in milliseconds, would also have been very complex.

However the dynamic expiration of cache entries offered by Infinispan allows for a simple yet effective solution. Instead of configuring a global expiration time for the whole cache we are looking up the currently configured delay for the destination domain and set it as expiration time for a new entry in the *Politeness Cache*. A dedicated expiration manager, shipped with Infinispan manages the removal of expiring entries in volatile and persistent storage and cluster wide if running in distributed mode, using dedicated threads. If now a request comes in we look up the target domain of the request in the *Politeness Cache*. If an entry with that domain as key can be found, we assume that the request would violate the constraints of the crawl delay and respond with the status code 429 *Too Many Request* and a *Retry* header mentioning the current delay for that domain, as defined by RFC 6585 [30, p.3].

5.7 Services

Not all use cases require a proxy based solution to handle caching and politeness. The third use case our implementation has to handle is the provision of politeness information to already existing crawling systems. Each of these systems may, or may not, be polite on its own. However if there is more than one of such a system running inside the same domain, then politeness has to be coordinated. To easily integrate politeness, these systems require an interface to fetch respective politeness information for a set of URLs. Our proposed solution offers three REST endpoints, each consuming and producing data in JSON format:

- *POST /politeness*
- *GET /politeness?urls=[url1,url2, ...]*
- *POST /politeness/verbose*

5.7.1 Filter services

The first two endpoints are identical, regarding their functionality. The only difference is that the first one reacts on POST requests, expecting a body with content type *application/json*. The second endpoint handles GET requests, retrieving a set of URLs from a query parameter named *urls* as comma separated list. Both respond with a simple list in JSON format containing the filtered previously submitted URLs, which are allowed to be crawled.

The more simple services are using the already implemented modules for our proxy solution enforcing politeness, which is shown in the code excerpt given in Listing 6. The algorithm iterates over the list of submitted URLs. The services return the URLs allowed to be crawled at the time the client gets the server response. Therefore we filter out every duplicate URL targeting a domain which is the destination of a URL which has already passed all checks. After this the URL has to obey all constraints as defined by the *robots.txt* and is not allowed to violate any constraints regarding the crawl delay. However it is possible that a request gets rejected by the proxy at actual crawling time, because a request arrived shortly before activated a crawl delay. These services just reflect the current situation regarding politeness constraints.

```
for (URL url : urls) {
  if (!visitedDomains.contains(url)) {
    if (robotsTxt.allows(url)) {
      if (cache.isAllowed(url.getHost())) {
        visitedDomains.add(url.getHost());
        result.add(url);
      }
    }
  }
}
```

Listing 6: Logic for checking URLs

5.7.2 Complex service

The *POST /verbose* endpoint offers a more detailed response. The list of URLs is submitted via JSON body similar to the first endpoint. However the response is not a simple list of strings or URLs, but an array of objects. Such a response object holds information about the URL and a Boolean value whether the URL can be crawled or not. If the URLs is not to be crawled then such an object also contains a error message and, if it violates the crawl delay but not the *robots.txt*, a *retryAfter* field specifying the milliseconds until the URL can be crawled. This endpoint can be used either for debugging purposes or human interaction, but also for more intelligent communication between server and client. A client may submit a list of URLs and then crawl the retrieved allowed URLs, omit the disallowed ones and start a delayed trigger for all response objects holding a valid value in the *retryAfter* field.

```
[
 "http://derstandard.at/Lifestyle",
 "http://derstandard.at/Karriere",
 "http://derstandard.at/suche/12312",
]
```

Listing 7: Example payload for calling complex service

If now a client submits a list of URLs like the one from Listing 7, to crawl the website of a popular Austrian newspaper, the service iterates over the list and checks every URL. A possible response would have a structure similar to the one illustrated by Listing 8. The first URL is allowed to be crawled which results in `allowed` being `true`. The destination of the second URL requested is equal to the one of the first URL. Therefore the attribute `allowed` is set to `false`. Furthermore the `retryAfter` field is set to the currently configured crawl delay of the domain and a respective error message is stored in the `error` field. The path of the third and last URL is disallowed by the Robots Exclusion Protocol which results into a dedicated error message and `retryAfter` to -1, which is the default value to signal the client that for a specific URL a delay does not have to be considered.

```
[{
 "url":"http://derstandard.at/Lifestyle",
 "error":null,
 "allowed":true,
 "retryAfter":-1
 },{
 "url":"http://derstandard.at/Karriere",
 "error":"There must be a delay of 1000
    milliseconds between each request.",
 "allowed":false,
 "retryAfter":1000
 },{
 "url":"http://derstandard.at/suche/12312",
 "error":"Blocked because URL is excluded from
    allowed URLs.",
 "allowed":false,
 "retryAfter":-1
}]
```

Listing 8: Example response of complex service

5.8 Configuration

SpringBoot allows for quite extensive configuration. It is possible to configure server ports, configuration sources, security settings and myriads of other things. Almost every Spring project has its own set of configuration parameters which can completely change the behaviour of the reconfigured system. The configuration itself can be done by various ways. However it is possible to overwrite configured values due to the way SpringBoot considers configuration properties, which is done in the following order: [13]

1. Updates via Spring Actuator
2. Command line arguments
3. Java System properties
4. OS environment variables
5. Application properties outside of packaged jar (application.yml)
6. Application properties inside of packaged jar (application.yml)

Besides the obligatory configuration of server port and logging levels we did some additional necessary configuration. Zuul per default works with a set routes, forwarding every incoming request to the target of the route (e.g. */service/external* to */service/internal/default*). To achieve a proxy like behaviour with Zuul we had to configure routes solely consisting of wildcards to match all incoming requests (*/*, */**, */***). However this lead to the problem that the set of custom service endpoints and the endpoints of Actuator got intercepted too. Therefore we had to either exclude it for the service endpoints or move the endpoints to a different port like the ones of Spring Actuator. Additionally we externalised the configuration for the different caches, making it possible to configure the memory size, thread count and the name of the JMX domain, all backed by reasonable defaults.

Configuration of the crawl delays is done according to the already described ways. Whereas the *application.yml* entries have the format shown in

[13] `https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html`

Listing 9. Basically it is a list of entries consisting of a *domain* field and a *delay* field controlling the crawl delay for the domain in milliseconds. It is also possible to configure a default domain, which provides a default crawl delay for all domains which are not explicitly configured.

Two of the use cases for our implementation are contradicting. It is not possible to provide a proxy intercepting requests and at the same time support the use case demanding a fixed order of responses with a timeout. Therefore a configuration parameter *crawler.politeness.timeout* determines the behaviour of the started instance of our implementation. If the timeout is set, then the proxy functionality is turned off and the configured value is used as timeout for halted responses to serve the respective use case and vice versa.

```
politeness:
  domains:
    - domain: derstandard.at
      delay: 1000
    - domain: orf.at
      delay: 5000
  default-domain:
    domain: default
    delay: 1500
```

Listing 9: Configuration of crawl delays for domains and the default value

5.9 Configuration at runtime

To now enable configuration at runtime we use Spring Actuator[14]. Which is a Spring project offering a number of additional features for monitoring and managing Spring applications. We use it for monitoring purposes as described below and for managing the configuration of our project at runtime. For this purpose we are using two of the many endpoints coming with Actuator. We use the endpoint *POST /env* to change configuration on the fly via REST. This is done through declaring a set of key value pairs in the request body. Spring recognises these properties, but only updates it when it receives a request at *POST /refresh*. The update is then achieved by creating a snapshot of the current system wide configuration. This snapshot is then compared to the new state and the changes then are published and set into

[14]`https://github.com/spring-projects/spring-boot/tree/master/spring-boot-actuator`

the according properties.

As the overview of configuration sources above indicates, there are various more ways to do configuration for our solution. However they all have some restrictions. Command line arguments and Java System properties can only be changed at start up of the server, respectively the Java Virtual Machine (JVM). OS environment variables can be updated at any time, however this is quite inconvenient to do. The most concise way is it to use the *application.yml* file and to do all configuration in one file. Although Spring does not offer a concise way to react on updates done in files at runtime we more or less copied the functionality of Spring Actuator and expand it to work with our configuration for crawl delays. If now the *application.yml* file gets updated Spring registers a change and our solution itself updates its configuration after calling Actuator's */refresh* endpoint.

5.10 Monitoring

Monitoring is absolutely essential to efficiently run an application. Whereas monitoring can be described as the observation and checking of the performance and availability of an information system. Every system should provide possibilities to view health and performance indicators. Without, it is not possible to identify errors or slow performance leading to poor quality of service. Therefore our implementation offers various ways to monitor the performance and availability of the system even down to the level of single components and allows responsible users to reason about the current behaviour of the system. For this reason our solution offers a number of endpoints:[15]

- *GET /env* - Exposes all configuration properties.
- *GET /health* - Shows information about status and disk space.
- *GET /metrics* - Provides an overview over a greater number of metrics.
- *GET /trace* - Displays trace information.
- *GET /delays* - Returns a list of maximum 2000 configured crawl delays.

Although the endpoints shipped with Actuator provide very rich information about application health and offer a message trace, we defined an

[15]`http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-endpoints`

additional endpoint *GET /delays* to expose currently configured crawl delays for monitoring purposes. However the respective cache can become quite big, leading to potential errors if one fetches all entries of it. Therefore the endpoint constraints the maximum result set to 2000, if the cache grows bigger than this threshold it returns an empty list.

Our implementation heavily utilises caches. As a result the performance and availability of these caches is critical for the system, making the monitoring of these caches essential. Infinispan directly provisions a large number of statistics and health indicators over JMX (Java Management Extensions).[16] Each running cache instance of a Java Virtual Machine provides its own statistics over dedicated JMX endpoints. Each Infinispan cache we are using owns a dedicated JMX domain. Under these domains Infinispan automatically provides statistics about memory consumption, hit ratios, misses, size and a number of other metrics. In addition JMX endpoints do not only expose information, but also operations on the underlying component. In the case of Infinispan it is possible to clear, shutdown or restart a single cache through the respective buttons.

However we found the provision of all these metrics through JMX endpoints not sufficient. It is cumbersome to collect these metrics by hand or to interact with JMX trough a programmatic way. Therefore we decided to extend the *GET /metrics* endpoint of Spring Actuator to display the relevant metrics *hits, size, misses, retrievals, hit ratio* and *memory consumption.* These metrics have the format *cache.*[*cache-name*].[*metric*]. `cache.page-cache.size` for instance, displays the current number of entries in the cache responsible for storing responses.

6 Evaluation

We evaluate the performance and functionality of our proof of concept in various ways. Of course one component was manual testing during development and implementation. As it is part of every development process to try out the implemented things, see if it works, fix errors and try it again. We also already assessed the functionality of the REST services, an excerpt of it can be seen in Listing 8. To check if the controlling of the domain specific crawling delay works correctly we implemented a small Java program. This program sends a request every 200ms to a fixed URL. For every request it

[16]http://infinispan.org/docs/stable/user_guide/user_guide.html#jmx:chapter

prints if the request succeeded or not. The result of this experiment is illustrated in Figure 4. Because the domain was configured with a delay of 1000ms, only every fifth request succeeds. The time window is big enough to let four and not five requests fail, because our concept measures the delay starting when a request is forwarded to its destination. Therefore one has to subtract the latency after that and the latency between the issue of a request till it arrives at the according check at the proxy.

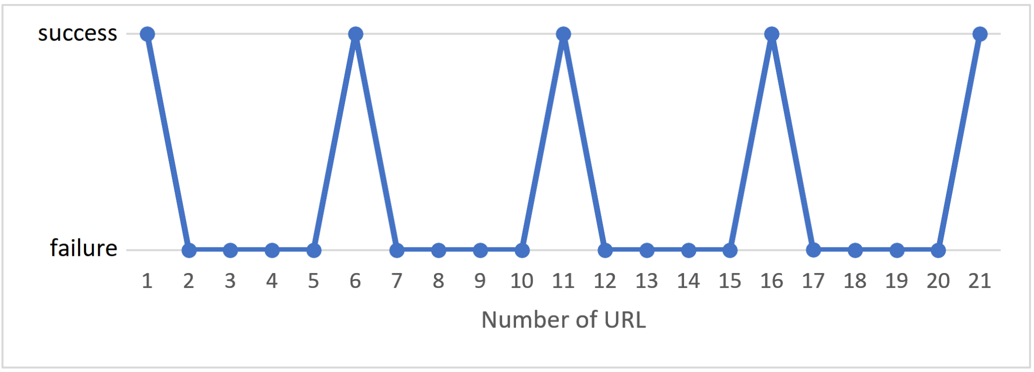

Figure 4: The effect of a delay on the crawling process

Although the correct functioning of our proposed solution is important, the performance of our implementation is equally important. A solution with performance significantly worse than sending request without a proxy in between would be rendered useless. Obviously it is not possible to implement a proxy based solution without increasing latency. However the increase must not affect the performance of clients in a critical way.

To evaluate this aspect of our solution we set up a second experiment. We defined a set of 20 URLs, all having the same target domain. Our test setup starts two clients at the same time with the same set of 20 URLs. Each client iterates over this set and issues a request for that URL, waits for the response and protocols the outcome and time needed. This is done for every URL in the set, with both clients running in separate threads and without sharing any resources. The only difference is, one client uses our proposed solution as proxy in between and the other client does not.

We executed this setup five times overall, after a warm up phase to avoid biased results from initialising code segments and components. Each request is sent with *Cache-Control: no-cache* in the header to disable any caching mechanisms. The resulting time series are illustrated by Figure 5. Each running cycle of a client is displayed as a line, showing the response time for each of the set of URLs. There are some outliers, identifiable by the response times significantly above average. The effect of outliers gets reduced by the increase of the set of data, here 100 data points per client. We assume that unpredictable network latency and varying bandwidth lead to this effects.

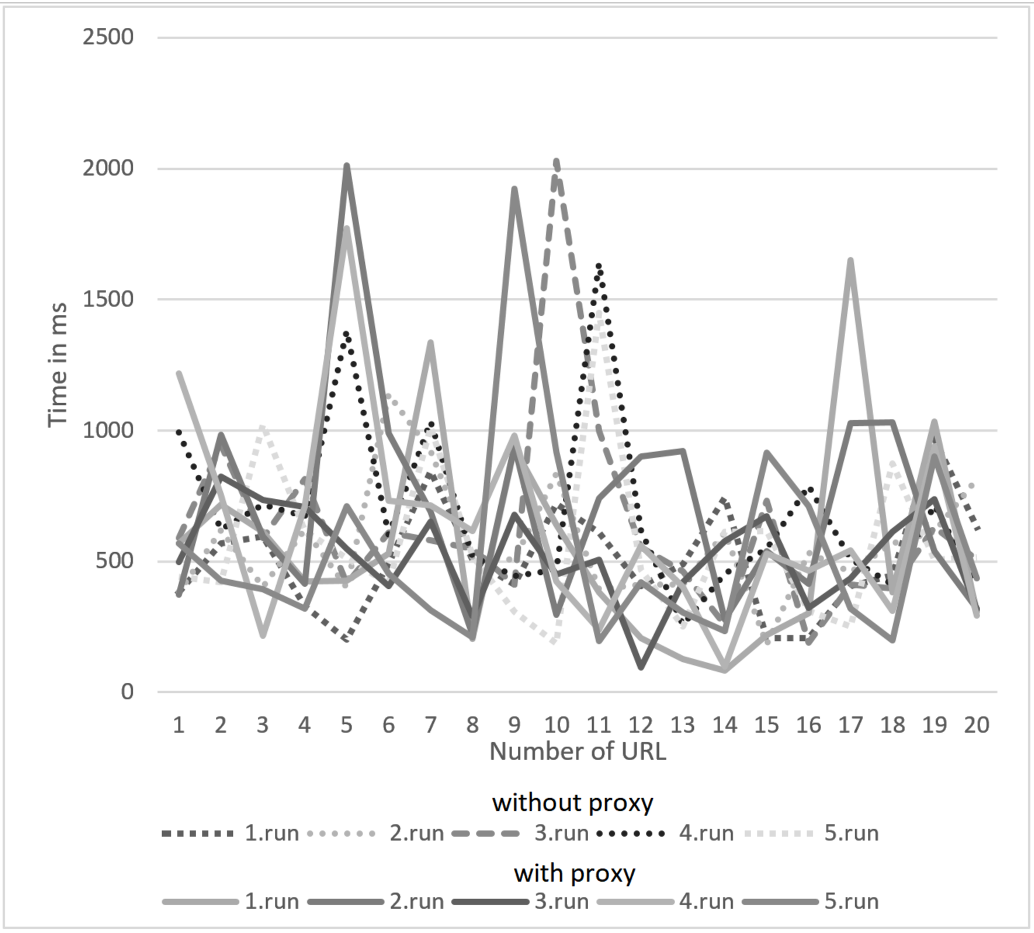

Figure 5: Request time on the y-axis per client, run cycle and URL on the x-axis

Solely based on Figure 5, one cannot predict the differences between two clients, and thus the difference our proxy makes. Therefore we aggregated the results using the average for all 5 data series per client to get more distinct results. As Figure 6 shows, the difference between the client working without a proxy and the client working with our proxy, is not significant. The overall average for all data points per client makes this way more clearer. We measured an average response time of 592,69ms for the client without a proxy. Our proxy based solution added a latency of nearly 10ms or 1.5%, resulting in an average response time of 601,76ms for the client using our solution.

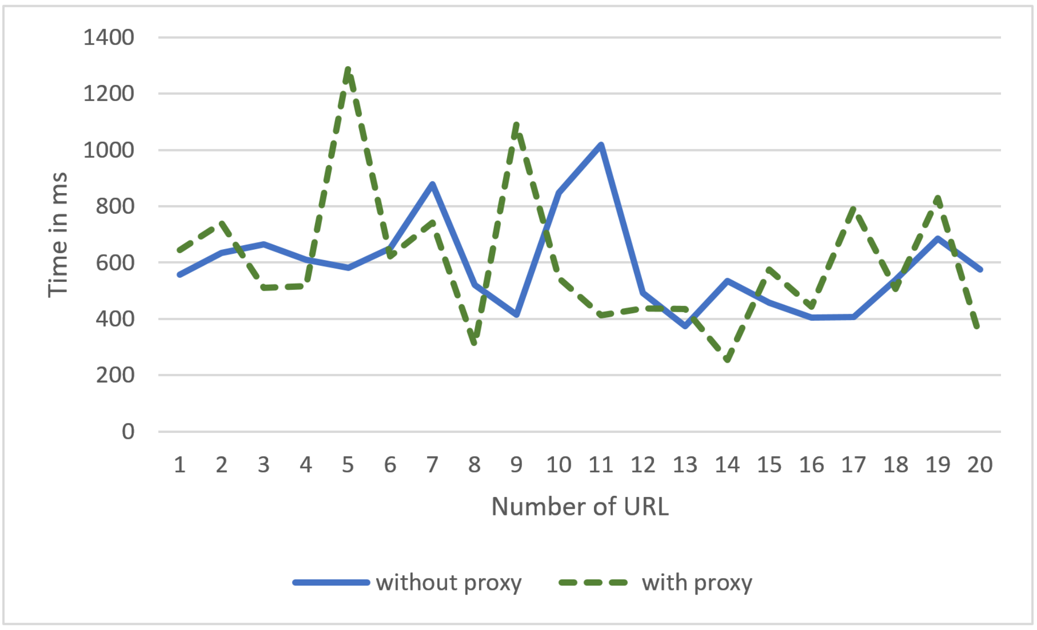

Figure 6: Average request time on the y-axis per client and URL on the x-axis

7 Conclusion and further research

The objective of this thesis is it to provide a solution for enforcing politeness for a variety of different use cases. Besides politeness our implementation should also provide caching and allow for extensive configuration and monitoring. We divided the use cases into three groups, each having different characteristics. These groups are simple or unsorted scripts, clients relying on sequential processing and clients requiring services to check politeness before crawling a URL. For our solution to function accordingly we divided it into two parts. A part implemented as proxy, working in between the client and its destination. The second part addresses the third use case, with clients having scheduling and crawling mechanisms. For these clients our solution offers a set of REST endpoints to check if the submitted URLs violate any politeness constraints.

We built our solution based on SpringBoot and a number of additional Spring projects for implementing monitoring, security and web services. Both parts, proxy and services, are using the same implemented components in background, which can further be broken down into more detail. One component handles the caching of retrieved web objects to improve performance and politeness. Another component handles the fetching of *robots.txt* files, the extractions of directives and the validation of a given URL. An additional component uses Infinispan, which is also used for caching and storing configuration properties, and its special ability to declare individual expiration times to handle crawl delays and ensures politeness that way.

Besides offering additional features like monitoring of all system components and configuration at runtime we showed that our solution is functioning properly. It automatically fetches *robots.txt* files, caches responses while respecting various *Cache-Control* directives, and aborts requests if they would violate crawling delay constraints. We also showed that the added latency of our solution is a negligible factor. Our experiment showed a increase in latency of 1,5%. We assume that to be a valuable trade off for the automatic handling of politeness and features like monitoring an caching.

Our proof of concept is far from complete tough. There are still a few open issues and some research has to be done. Currently our solution is prepared to run in a distributed manner and would be able to form a cluster. However according experiments has not been done yet, and considering that distribution always adds an additional layer of complexity and has the potential to raise problems.

The *robots.txt* file is not the only source for crawling directives. However at this state of our work, the proxy is not able to analyse the data fetched to extract eventual <*meta*> tags in the HTML code, containing directives of the Robots Exclusion Protocol. Our implementation also does not consider any *sitemap* directives. A service automatically fetching sitemaps of the destination domain, if existing, would be a valuable extension of our implementation.

Furthermore our solution currently only handles HTTP requests. Building a HTTPS proxy requires different methods, to handle tunnelling and various other security measures to prevent man in the middle attacks, exactly addressing what a proxy would do in our scenario.

Our future work will also concentrate on more advanced strategies for the estimation of crawl delays. Currently delays are configured manually or are determined by the Robots Exclusion Protocol. There are approaches for using algorithms like PageRank to calculate delays on the basis of domain popularity. Another approach would be the polling of delays by continuously decreasing the delay until the used IP address gets blocked by the destination host.

References

[1] Ricardo Baeza-yates and Carlos Castillo. Balancing Volume, Quality and Freshness in Web Crawling. In *In Soft Computing Systems - Design, Management and Applications*, pages 565–572. IOS Press, 2002.

[2] Ricardo Baeza-Yates, Carlos Castillo, Mauricio Marin, and Andrea Rodriguez. Crawling a Country: Better Strategies Than Breadth-first for Web Page Ordering. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 864–872, New York, NY, USA, 2005. ACM.

[3] J.D. Belfiore, I.M. Ellison-Taylor, S. Ramasubramanian, C.H. Chew, and S.E. Berkun. Method for downloading a sitemap from a server computer to a client computer in a web environment, February 2003.

[4] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform resource identifier (uri): Generic syntax. STD 66, RFC Editor, January 2005. `http://www.rfc-editor.org/rfc/rfc3986.txt`.

[5] Tim Berners-Lee, Larry Masinter, and Mark McCahill. Uniform resource locators (url). RFC 1738, RFC Editor, December 1994. `http://www.rfc-editor.org/rfc/rfc1738.txt`.

[6] Stefan Bischof, Stefan Decker, Thomas Krennwallner, Nuno Lopes, and Axel Polleres. Mapping between RDF and XML with XSPARQL. *Journal on Data Semantics*, 1(3):147–185, September 2012.

[7] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. XQuery 1.0: An XML Query Language (Second Edition; Revised 7 September 2015), September 2015.

[8] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Softw. Pract. Exper.*, 34(8):711–726, July 2004.

[9] Brian E. Brewington and George Cybenko. How Dynamic is the Web? In *Proceedings of the 9th International World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Netowrking*, pages 257–276, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

[10] Carlos Castillo, Mauricio Marin, Andrea Rodriguez, and Ricardo Baeza-Yates. Scheduling Algorithms for Web Crawling. In *Proceedings of the WebMedia & LA-Web 2004 Joint Conference 10th Brazilian Symposium on Multimedia and the Web 2Nd Latin American Web Congress*, LA-WEBMEDIA '04, pages 10–17, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Junghoo Cho and Hector Garcia-Molina. The Evolution of the Web and Implications for an Incremental Crawler. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 200–209, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[12] Richard Cyganiak, Holger Stenzhorn, Renaud Delbru, Stefan Decker, and Giovanni Tummarello. Semantic Sitemaps: Efficient and Flexible Access to Datasets on the Semantic Web. In *The Semantic Web: Research and Applications*, Lecture Notes in Computer Science, pages 690–704. Springer, Berlin, Heidelberg, June 2008.

[13] Brian D. Davison. A Survey of Proxy Cache Evaluation Techniques. *ResearchGate*, 24, February 2001.

[14] Daniele Dell'Aglio, Axel Polleres, Nuno Lopes, and Stefan Bischof. Querying the Web of Data with XSPARQL 1.1. In *Proceedings of the 2014 International Conference on Developers - Volume 1268*, ISWC-DEV'14, pages 113–118, Aachen, Germany, Germany, 2014. CEUR-WS.org.

[15] Michelangelo Diligenti, Marco Maggini, Filippo Maria Pucci, and Franco Scarselli. Design of a Crawler with Bounded Bandwidth. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters*, WWW Alt. '04, pages 292–293, New York, NY, USA, 2004. ACM.

[16] Stephen Dill, Ravi Kumar, Kevin S. Mccurley, Sridhar Rajagopalan, D. Sivakumar, and Andrew Tomkins. Self-similarity in the Web. *ACM Trans. Internet Technol.*, 2(3):205–223, August 2002.

[17] R. Fielding, M. Nottingham, and J. Reschke. Hypertext transfer protocol (http/1.1): Caching. RFC 7234, RFC Editor, June 2014. `http://www.rfc-editor.org/rfc/rfc7234.txt`.

[18] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, RFC Editor, June 1999. `http://www.rfc-editor.org/rfc/rfc2616.txt`.

[19] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, Irvine, 2000.

[20] Roy Friedman. Caching Web Services in Mobile Ad-hoc Networks: Opportunities and Challenges. In *Proceedings of the Second ACM International Workshop on Principles of Mobile Computing*, POMC '02, pages 90–96, New York, NY, USA, 2002. ACM.

[21] Google. Using the robots meta tag, May 2007. Accessed: July 25, 2017.

[22] Prashant and Raghuwanshi Janbandhu, Rashmi and Dahiwale. Analysis of web crawling algorithms (PDF Download Available). In *ResearchGate*, volume 2, pages 488–492, 2014.

[23] M. Koster. A Method for Web Robots Control, November 1996. Accessed: May 25, 2017.

[24] H. Kreger. Java Management Extensions for application management. *IBM Systems Journal*, 40(1):104–129, 2001.

[25] Hsin-Tsang Lee, Derek Leonard, Xiaoming Wang, and Dmitri Loguinov. IRLbot: Scaling to 6 Billion Pages and Beyond. *ACM Trans. Web*, 3(3):8:1–8:34, July 2009.

[26] Mark Levene and Alexandra Poulovassilis. *Web Dynamics: Adapting to Change in Content, Size, Topology and Use.* Springer Science & Business Media, March 2013. Google-Books-ID: WkmqCAAAQBAJ.

[27] Mauricio Marin, Rodrigo Paredes, and Carolina Bonacic. High-performance Priority Queues for Parallel Crawlers. In *Proceedings of the 10th ACM Workshop on Web Information and Data Management*, WIDM '08, pages 47–54, New York, NY, USA, 2008. ACM.

[28] Marc Najork. Web Crawler Architecture. In LING LIU and M. TAMER ÖZSU, editors, *Encyclopedia of Database Systems*, pages 3462–3465. Springer US, 2009. DOI: 10.1007/978-0-387-39940-9_457.

[29] Marc Najork and Allan Heydon. Handbook of Massive Data Sets. pages 25–45. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[30] M. Nottingham and R. Fielding. Additional http status codes. RFC 6585, RFC Editor, April 2012.

[31] Christopher Olston and Marc Najork. Web Crawling. *Foundations and Trends® in Information Retrieval*, 4(3):175–246, February 2010.

[32] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web., November 1999.

[33] Patrick Oliver Riemer. Model based Cache. Unpublished, March 2017.

[34] robotstxt.org. The Web Robots Pages, July 2017. Accessed: July 24, 2017.

[35] Uri Schonfeld and Narayanan Shivakumar. Sitemaps: Above and Beyond the Crawl of Duty. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 991–1000, New York, NY, USA, 2009. ACM.

[36] Sandeep Sharma and Ravinder (Guide) Kumar. *Web-Crawling Approaches in Search Engines.* Thesis, September 2008.

[37] Vladislav Shkapenyuk and Torsten Suel. Design and Implementation of a High-Performance Distributed Web Crawler. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE '02, pages 357–, Washington, DC, USA, 2002. IEEE Computer Society.

[38] Danny Sullivan. Meta Robots Tag 101: Blocking Spiders, Cached Pages & More, March 2007. Accessed: July 25, 2017.

[39] Yang Sun, Ziming Zhuang, and C. Lee Giles. A Large-scale Study of Robots.Txt. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 1123–1124, New York, NY, USA, 2007. ACM.

[40] Mike Thelwall and David Stuart. Web crawling ethics revisited: Cost, privacy, and denial of service. *Journal of the American Society for Information Science and Technology*, 57(13):1771–1779, November 2006.

[41] Jürgen Umbrich, Nina Mrzelj, and Axel Polleres. Towards capturing and preserving changes on the web of data. pages 50–65, 2015.

[42] Jia Wang. A Survey of Web Caching Schemes for the Internet. *SIGCOMM Comput. Commun. Rev.*, 29(5):36–46, October 1999.